Clipcode™ Source Tour For Angular 12 [1] [2]

A detailed guided tour to the source trees of Angular and related projects

[1] ecuplxd 翻译，仅供参考。版本：2021-05-08
[2] 原文地址：https://www.clipcode.net/training/clipcode-source-tour.pdf

目录

前言

注意，Bazel 构建系统使用 JAVA 来构建 Angular ——但它在构建你自己的 Angular 应用时不是的必需的（除非你使用 Bazel，这对非常大的代码库来说是一个不错的选择）。注意 moment.js 是一个非常流行的日期和时间框架。我们看到它被 @angular/material-moment-adapter 包使用。

现代 Web 应用生态

Angular 生态由多个项目组成，这些项目协同工作，为开发现代 Web 应用提供全面的基础。基本上，所有这些项目都是用 TypeScript 编写的（甚至 TypeScript 编译器本身也是用 TypeScript 编写的）。除了 Bazel 主项目（使用 Java）、用于本地访问 macOS FSEvents 的 fsevents 以及 Jasmine 单元测试框架和 Karma 测试运行器（使用 JavaScript）。对其他项目的依赖各不相同。

请注意，基于 Java 的 Bazel 主项目用于构建 Angular 本身。构建常规 Angular 应用不需要 Bazel 和 Java。Bazel 非常适合增量和多核构建，因此在与大型代码库（例如 Angular 本身）一起使用时具有优势。未来，Bazel 很可能成为构建大型 Angular 应用的明智选项。

上面提到的那些项目都不依赖于 Visual Studio Code。这些项目只需要一个可以与 TypeScript tsc 编译器一起使用的代码编辑器——更多信息——请参见主页中间部分：

- http://typescriptlang.org

Visual Studio Code 就是一个非常好、开源、可免费用于 macOS、Linux 和 Windows 的编辑器，而且它也是用 TypeScript 编写的。

- https://code.visualstudio.com

当然你也可以选择使用任何编辑器来查阅这些项目的源码。

浏览 Markdown 文件

除了 TypeScript 中的源码之外，所有这些项目还包含 markdown 格式（*.md）的文件。Markdown 是一种领域特定语言（DSL），用于表达 HTML 文档。与 HTML 相比，更容易/更快进行手动编写。当转换为 HTML 时，它提供专业的文档。你可能遇到的一个问题是如何查看它们（作为 HTML，而不是作为 .md 文本）。将它们视为 html 的一种简单方法是查看 Github 上的 .md 文件，例如：

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

或者，在你自己的机器上，大多数文本编辑器要么直接支持 markdown，要么可以通过扩展获得。在检查带有 .md 文件的源码时，如果你的编辑器同时支持打开 markdown + 转换 markdown 为 HTML，这时就会很方便。StackOverflow 上有过提问：

- http://stackoverflow.com/questions/9843609/view-markdown-files-offline

比如，VSCode 本身就是支持 Markdown 的，如果你打开一个 .md 文件并按下 CTRL+SHIFT+V，会看到 markdown 预览：

- https://code.visualstudio.com/docs/languages/markdown

最后，如果你想学习 markdown，请访问：

- http://www.markdowntutorial.com

目标读者

本 Angular 源码教程面向希望深入了解 Angular 实际工作原理的中级到高级应用开发人员。要真正了解 Angular 的功能，应用开发人员应该阅读源码。是的，Angular 随附在 https://angular.io/docs 的开发人员文档（大部分就相当好了），但最好的方式还是结合源代码进行研究。

理解源码的好处

中高级开发人员有很多充分的理由需要熟悉 Angular 的源码，因为他们日常的应用开发都需要和 Angular 打交道。

增强理解——与许多软件包一样，关于概念设计和 API 的文档可能存在也可能不存在，如果存在也可能没有要求的那么全面，或多或少不能准确反映当前代码的真实意图——文档过时了，毕竟代码改动后，并不总是同步地更新文档，这适用于任何快速发展的框架。

高级调试——当出现问题时，因为它们肯定会出现（并且通常只是向潜在客户展示的一个重要展示），如果深入了解过源码，能更快地解决问题。

优化——对于具有高数据吞吐量的大规模生产应用，了解应用的基板实际如何工作对于决定如何/优化什么非常有用（例如，当应用开发人员真正了解 Angular 的 NgZone 的工作原理时，它是如何可选的（从 Angular v5 开始）以及它如何与 Angular 的变更检测交织在一起，然后他们可以将 CPU 密集型但非 UI 代码放置在同一线程内的不同区域中，这可以带来更好的性能）。

生产力——熟悉源码对于任何框架最大化生产力都是重要的，是开发人员积累更广泛的理解的一部分以及应用如何执行的基础。

良好实践——研究经过良好测试和详细 review 的大型代码库可以让开发人员养成良好编码习惯，并在自己的应用中实际运用。

获取源码

要研究源，你可以通过 Git（通常）或下载 zip 获取仓库副本然后进行浏览。有些软件包可能会提供关于获取源码的额外详细信息——例如，对于 Angular 项目，阅读 “Getting the Sources”：

- https://github.com/angular/angular/blob/master/docs/DEVELOPER.md

我们首先需要决定要使用哪个分支。对于 master，我们使用这个：

- https://github.com/angular/angular/tree/master

专门针对 Angular 主项目，额外的访问原阿门的方法是 Angular API 参考（https://angular.io/api），每个 Angular 类型的 API 页面在顶部有一个超链接关联源文件的页面（这解析为最新的稳定版本，可能会或可能与主站的源不同）。

保持最新

Angular 是一个快速迭代的项目。为了保持最新，阅读开发团队的每周会议记录，可在此处获取：

- http://g.co/ng/weekly-notes

GitHub 中的 ChangeLog 列出了新版本的功能：

- https://github.com/angular/angular/blob/master/CHANGELOG.md

1: Zone.js 内部

本篇探讨 Zone.js 的内部结构，以及 Angular 12 中是如何使用它的。

概览

Zone.js 项目提供了在单个 JavaScript 线程中运行多个异步执行上下文（即主浏览器 UI 线程或单个 web worker 中）。Zone 使用了将 JavaScript 事件循环分割单个线程的方式。单个 zone 不会跨越线程边界。考虑 zone 的良好方法是它们将 JavaScript 线程内的栈分割成多个小栈，并将 JavaScript 事件循环分划分为多个小的事件循环，这些循环似乎同时运行（但却共享相同的 VM 事件循环）。实际上，Zone.js 有助于你在单线程中编写多线程代码。

当你的应用程序加载 Zone.js 时，它会修补某些异步调用（例如 `setTimer`、`addEventListener`）来实现 zone 功能。Zone.js 通过向应用提供的回调函数添加包装器做到这点，当超时或检测到事件时，它首先运行包装器，然后运行回调。执行应用程序代码形成任务的块，每个任务在 zone 的上下文。Zones 按层次排列并提供有用的功能在错误处理、性能测量和执行配置等领域进入和离开 zone 时的工作项目（所有这些对自己实现现代 Web UI 框架中变更检测有参考价值）。

Zone.js 对应用代码大多是透明的。Zone.js 在后台运行，大部分“只是默默工作”。例如，Angular 使用 Zone.js，Angular 应用代码通常运行在 zone 内。成千上万的 Angular 应用开发人员甚至都没意识到这点。如果有需要，可以自定义 zone API 调用，手动管理 zone 的运行。有些应用开发人员可能希望采取某些步骤将非 UI 性能关键代码移出 Angular zone ——这时可以使用 NgZone 类。

项目信息

Zome.js 源码主页 + 根目录在 Angular 仓库中：

- https://github.com/angular/angular/tree/master/packages/zone.js

下面我们假设你已经将 <ZONE-MASTER> 目录下的 Zone.js 源码下载到了本地，任何将要使用到的路径名会以此作为相对路径。我们还将查看调用 Zone.js 的 Angular 代码，我们使用 <ANGULAR-MASTER> 作为该源码的根目录。

Zone.js 是用 TypeScript 编写的。它没有包依赖（它的 package.json 有这个条目：`"dependencies":`），虽然它有很多 devDependencies。这是一个相当小型源码，其大小（未压缩）约为 3 MB。

使用 Zone.js

在详细了解 Zone.js 自身是如何实现之前，我们首先了解在生产中如何使用它，这里以非常流行的 Angular 项目为例。

在 Angular 中使用 Zone.js 是可选的，不过却是默认开启。使用 zones 通常来说是个好主意，但存在某些场景（比如，使用 Angular Elements 构建 Web Compoenents）下是不必要的。

要在你的应用中使用 Zone.js，你需要加载它。在 package.json 中写入（如果你的项目是通过 Angular CLI 创建，会自动帮你添加进去）：

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

你应该在加载 core.js 之后加载 Zone.js（如果用到了它）。例如，如果你使用 Angular CLI 生成 Angular 应用（就像大多数生产应用一样），Angular CLI 将生成一个名为 <project-name>/src/polyfills.ts 的文件，它将包含：

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

Angular CLI 还会生成一个 angular.json 配置文件，该行设置 polyfills：

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

如果你的应用是用 TypeScript（推荐）编写的，您还需要配置声明访问。这些定义了 Zone.js API 并在以下位置提供：

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

（注意：这个文件有着良好的文档说明，值得那些了解 Zone.js 的人仔细学习）。不同于其它库的声明文件，zone.js.d.ts 根本不使用 `import` 或 `export`（那些命令甚至没有出现在那个文件中一次）。这意味着通常情况下，希望使用 zones 的应用代码不能只简单地导入其 .d.ts 文件。相反，应该使用 `///reference` 命令。这包括网站上的参考文件 `///reference` 包含文件中的引用。这种方法的好处是包含文件本身不必（但可以）使用 `import`，因此可能是一个脚本，相比与其必须是一个模块。使用 zones 并不强制应用程序使用模块（然而，大多数大型应用，包括所有 Angular 应用会使用）。最好通过一个例子来研究它是如何工作的，所以让我们看看 Angular 是如何包括 zone.d.ts。Angular 包含一个文件，types.d.ts 在包目录下（在它的模块目录和工具目录下有一个类似的）。

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

它包含如下内容：

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

在 Angular 中使用

编写 Angular 应用时，你的所有应用代码都在一个 zone 内运行，除非你采取具体步骤以确保其中一些不会。此外，大多数 Angular 框架代码本身在一个 zone 中运行。开始 Angular 应用时开发，你可以简单地忽略 zones，它们默认为你的应用配置好了，无需做任何特别的事情就能从中受益。文件 <ZONE-MASTER>/blob/master/MODULE.md 文件结尾结尾解释了 Angular 在哪里使用到了 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 是 Angular 启动变更检测的方式——当 zone 的小块栈是空时，发生变更检测。此外，zones 是 Angular 配置全局异常处理的方式。当任务发生错误时，其 zone 配置的错误处理程序被调用。Zone 提供了默认实现，应用可以通过依赖注入提供自定义实现。有关详细信息，请参见此处：

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

在该页面上，提供了自定义错误处理程序的代码示例：

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

Angular 提供了一个类 NgZone，它建立在 zone 上：

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

当你开始创建更高级的 Angular 应用程序时，特别是那些涉及不会在中途更改 UI 的计算密集型代码计算（但可能在最后），您会看到放置这样的 CPU 是可取的- 在单独的区域中进行密集的工作，您将为此使用自定义 NgZone。

在其他地方，我们将详细介绍 NgZone 和 Angular 中 zones 的使用一般当我们稍后探索主要 Angular 项目的源码时，但对于现在，请注意 NgZone 的源码在：

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

应用程序引导期间的 zone 设置位于：

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

当我们引导 Angular 应用程序时，我们要么使用 bootstrapModule<M>（使用动态编译器）或 bootstrapModuleFactory<M>（使用离线编译器）。这两个函数都在 application_ref.ts. bootstrapModule<M> 调用 Angular 编译器 1 然后调用 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)
  );
}
```

在 bootstrapModuleFactory 中，我们看到了如何为 Angular 初始化 zone：

```
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;
        });
      });
    });
  }
```

在 1 我们看到一个新的 NgZone 被创建，在 2 它的 run() 方法被调用，在 3 我们看到依赖注入请求了一个错误处理实现（除非应用提供自定义实现，否则将返回默认实现）在 4 处，我们看到错误处理程序将被用于配置新创建的 NgZone。最后在 5 处，我们看到对实际引导的调用。

所以总而言之，Angular 应用开发人员应该清楚地了解 zones，因为他们的应用代码将在其中的执行上下文运行。

API

Zone.js 在 <ZONE-MASTER>/dist/zone.js.d.ts 文件中暴露了应用可以使用的 API。

它提供了两个主要的类型：task 和 zones，还有一些辅助类型。zone 是一个（通常是命名的）异步执行上下文；一个任务是一个函数块功能（也可以被命名）。任务在 zone 的上下文中运行。

Zone.JS 还提供了一些 const 值，也称为 Zone 或者是 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;
```

回想一下，TypeScript 的声明空间区分为值和类型，因此 Zone 值与 Zone 类型是不同的。有关更多详细信息，请参阅 TypeScript 语言规范——第 2.3 节——Declarations：

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

除了用于定义 Zone 值外，ZoneType 不会进一步使用。

当你的应用代码想要找到当前 zone 只需要简单使用 Zone.current 即可，当它想要发现该 zone 内的当前任务时，使用 Zone.currentTask。如果你需要确定 Zone.js 是否可用于你的应用（它将用于 Angular 应用），只需确保 Zone 不是未定义的。如果我们查阅：

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

我们看到这恰好正是 Angular 的 NgZone.ts 所做的：

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

两个用来定义任务的简单辅助类分别是 TaskType 和 TaskData。TaskType 只是一个与任务相关联的可读性字符串。它通常设置为以下注释中提到的三种任务类型之一：

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

TaskData 包含一个布尔值（这个任务是周期性的，即要重复）和两个数字——在执行此任务前的 delay 和来自 setTimeout 的 handler id。

```
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;
}
```

一个任务是被声明为如下的接口：

```
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;
}
```

有三个标记接口继承自 Task：

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

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

注释很好地解释了他们的目的：

```
* - [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.
```

有三个辅助类被用来定义 Zone.HasTaskState，每个任务类型包含一个布尔值，一个字符串。

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

当一个 zone 希望将某些操作的执行方式委托给另一个 zone 时，使用 ZoneDelegate。所以对于 forcking（创建新任务）、调度、拦截、调用和错误处理时，可能会调用委托来执行操作。

```
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 是一个接口，允许实现在需要某些操作时声明应该拥有什么。它使用 ZoneDelegate 和当前 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;
}
```

Zone 类型的定义如下：

```
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;
}
```

Zone/ZoneSpec/ZoneDelegate 之间的接口关系

把 ZoneSpec 看成是控制一个 zone 如何工作的处理引擎。它是 Zone.fork() 方法的一个必要参数。

```
  // 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;
```

通常，当一个 zone 需要执行一个动作时，它会使用所提供的 ZoneSpec。你是否想录制长堆栈调用，任务追踪，使用 WTF（稍后进行讨论）或运行异步测试？对于每个情景提供不同的 ZoneSpec，每个都提供不同的功能，并具有不同的处理花销。Zone.js 提供 Zone 接口的一个实现，和多个实现 ZoneSpec 接口（在 <ZONE-MASTER>/lib/zone-spec）。具有专业需求的应用代码可以创建自定义 ZoneSpec。

一个应用程序可以建立一个 zones 的层次结构，有时一个 zone 需要调用更高层次的另一个 zone。为此，需要使用 ZoneDelegate。

目录结构

Zone.js 源代码树由一个根目录和若干文件以及以下直接子目录组成：

- dist
- doc
- example
- scripts
- tests

主要代码在 lib 下：

- lib

在编译过程中，源代码被构建到新创建的构建目录中。

根目录

该根目录包含以下 markdown 文档：

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

当我们检查 <ZONE-MASTER>/dist/ZONE.js.d.ts 时我们看到它实际上有非常好的文档记录，包含大量细节，能够帮助我们使用 Zone.js 启动和运行编写应用。从 DEVELOPER.md 文档中，我们可以看到 dist 的内容是自动生成（我们需要探索如何生成）。

该根目录包含以下 JSON 文件：

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

有多个文件以 tsconfig 开头——以下是主文件中的编译器选项：

```
  "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"
    ]
```

```
  }
```

package.json 文件包含一些元数据信息（包含 main 和 browser，根据是否将此包 1 加载到服务器中，提供其他入口点 [node] 或浏览器 2）。

```
{
  "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"
  },
```

和一系列脚本文件：

```
  "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
    ...
  },
```

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

- webpack.config.js

```
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'],
  },
};
```

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

- .gitignore

- gulpfile.js

dist

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

- 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

我们在稍后详细说明 <ZONE-MASTER>/lib/zone-spec 目录。

example

example 目录包含了一系列关于如何使用 Zone.js 的例子。

scripts

script 目录（以及其子目录）包含了一些执行脚本命令：

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

目录结构

Zone.js 的主要源码位于:

- <ZONE-MASTER>/lib

它包含一系列的子目录:

- browser
- closure
- common
- extra
- jasmine
- mix (a mix of browser and node)
- mocha
- node
- rxjs
- testing
- zone-spec

以及单一文件:

- zone.ts

它们的组织整理如下:

为了使 Zone.js 能用于函数，任何和异步代码执行有关的 JavaScript API 都需要打补丁——这些 API Zone.js 都有一个特定实现。所以对于诸如 `setTimeout` 或 `addEventListener` 之类的调用，Zone.js 都需要能够自己处理，这样，当超时、事件和 promise 触发，zone 代码就会首先运行。

Zone.js 支持对多种环境打补丁，browser、server(node)、Jasmine 都有一个子目录包含了补丁代码。通用补丁代码位于 common 目录。Zone.js API（不包括 `ZoneSpec`）的核心实现位于 zone.ts 文件。zone_spec 目录包含可配置逻辑可以添加到 zone 以更改其行为。有多种实现，应用可以创建自定义实现。

zone.ts

zone.ts 的前 600 行代码是注释良好的 Zone.js API 定义，最后会导出到 zone.d.ts。文件稍大的剩余部分是 Zone const 的实现：

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

_ZonePrivate 接口定义了管理每个 zone 的私有信息：

```
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;
  ..
```

```
}
```

当使用客户端 API 时是不会看到这些的，但当你调试 Zone.js 的实现时候，它会不时的出现。

管理微任务队列，需要如下变量：

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

_microTaskQueue 是一个微任务数组，这必须在我们放弃 VM 轮转之前执行。_isDrainingMicrotaskQueue 是一个布尔值，用于跟踪我们是否处于清空微任务队列的过程当一个任务运行在一个已经存在的任务时，它们是嵌套着，_nativeMicroTaskQueuePromise 用于获取一个原生微任务队列。以下两个函数用于管理微任务队列：

- scheduleMicroTask
- drainMicroTaskQueue

它还实现了三个类：

- Zone
- ZoneDelegate
- ZoneTask

该文件没有关于 ZoneSpec 的实现。他们在单独的 zone-spec 子目录中。

ZoneTask 是这些类中最简单的：

```
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';
  ..
}
```

构造函数只记录提供的参数并设置 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);
      };
    }
  }
```

这里有趣的活动是设置 invoke 函数。它递增 _numberOfNestedTaskFrames 计数器，调用 zone.runTask()，并在 finally 块中检查 _numberOfNestedTaskFrames 是否为 1，如果是，则调用独立函数 drainMicroTaskQueue()，然后递减 _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--;
    }
  }
```

一个自定义的 toString() 返回 data.handleId（如果可用）或者对象 toString() 的结果。

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

drainMicroTaskQueue() 定义如下：

```
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;
  }
}
```

_microTaskQueue 通过调用 scheduleMicroTask 来填充：

```
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.

`ZoneDelegate` 类必须处理八个场景：

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

它定义了变量来存储每个 `ZoneDelegate` 和 `ZoneSpec` 的值，这些值在构造函数中初始化。

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

`ZoneDelegate` 还声明了三个变量，用来保存 delegates zone 和 parent zone，以及表示任务的计数器（针对每种任务）：

```
  public zone: Zone;

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

  private _parentDelegate: ZoneDelegate | null;
```

在 `ZoneDelegate` 的构造函数中，`zone` 和 `parentDelegate` 字段从提供的参数中进行初始化，设置 `ZoneDelegate` 以及 8 个场景的 `ZoneSpec` 字段（使用 Typescript 的类型守护），要么是 `ZoneSpec`（如果不为 null），要么是父的 delegate：

```
  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);
    ..
  }
```

针对 8 个场景的 ZoneDelegate 方法只是将调用转发到选定的 ZoneSpec（父代理）并做一些内部处理。例如，invoke 方法检查是否定义了 _invokeZS，如果是，则调用其 onInvoke，否则直接调用给定的回调：

```
  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);
  }
```

scheduleTask 方法有点不同，它首先 1 尝试使用 _scheduleTaskZS（如果有定义），否则进入 else 分支进行一系列判断，2 尝试使用给定任务的 scheduleFn（如果有定义），否则 3 如果是微任务则调用 scheduleMicroTask)”，其它情况 4 抛出错误：

```
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;
    }
```

fork 方法是创建新 zone 的地方。如果定义了 _forkZS，则使用它，否则使用给定的 targetZone 和 zoneSpec 创建一个新 zone：

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

内部变量 _currentZoneFrame 被初始化为根 zone，_currentTask 初始化为 null：

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

2: Tsickle

概览

Tsickle 是一个小型实用程序，可以对 TypeScript 代码进行转换，使其适合作为 Closure 编译器的输入，而 Closure 编译器可以生成高度优化的 JavaScript 代码。

Tsickle 项目主页位于：

- https://github.com/angular/tsickle

将 Tsickle 介绍为：

> Tsickle 将 TypeScript 代码转换为 Closure Compiler 可接受的形式。这允许使用 TypeScript 来转换你的源代码，然后使用 Closure Compiler 来捆绑和优化它们，同时利用 Closure Compiler 中的类型信息。

要了解有关 Closure 的更多信息，请访问：

- https://github.com/google/closure-compiler

Tsickle 用于 Angular，但也可以用于非 Angular 项目。

目录结构

Tsickle 源码有以下子目录：

- src
- test
- test_files
- third_party

主目录有这些重要文件：

- readme.md
- package.json
- gulpfile.js
- tsconfig.json

readme.md 包含有关项目的有用信息，包括这个重要的 tsconfig.json 使用指南：

项目设置

Tsickle 通过包装 tsc 来工作。要使用它，你必须通过配置 tsconfig.json 中的设置来设置您的项目，以便在从命令行运行 tsc 时它可以正确构建。

如果您在构建文件（如 gulpfile 等）中有复杂的 tsc 命令行和标志，Tsickle 不会知道它。将所有内容都放在 tsconfig.json 中的另一个原因是你的编辑器也可以继承所有这些设置。

readme.md 包含有关项目的有用信息，包括有关使用 tsconfig.json 的重要指南：
package.json 文件包含：

```
"main": "built/src/tsickle.js",
"bin": "built/src/main.js",
```

gulpfile.js 文件包含以下 Gulp 任务：

- gulp format
- gulp test.check-format (formatting tests)
- gulp test.check-lint (run tslint)

src 子目录

src 子目录有如下源码文件:

- class_decorator_downlevel_transformer.ts
- cli_support.ts
- closure_externs.js
- decorator-annotator.ts
- decorators.ts
- es5processor.ts
- fileoverview_comment_transformer.ts
- jsdoc.ts
- main.ts
- modules_manifest.ts
- rewriter.ts
- source_map_utils.ts
- transformer_sourcemap.ts
- transformer_util.ts
- tsickle.ts
- type-translator.ts
- util.ts

main.ts 是对 tsickle 的调用开始执行的地方，tsickle.ts 是核心逻辑所在的地方——其他文件是辅助。
main.ts 底部的入口点调用 main 函数，将参数列表作为字符串数组传入。

```
function main(args: string[]): number {
  1 const { settings, tscArgs } = loadSettingsFromArgs(args);
  2 const config = loadTscConfig(tscArgs);
  if (config.errors.length) {
    console.error(tsickle.formatDiagnostics(config.errors));
    return 1;
  }

  if (config.options.module !== ts.ModuleKind.CommonJS) {
    // This is not an upstream TypeScript diagnostic, therefore it does not
    // go through the diagnostics array mechanism.
    console.error(
      'tsickle converts TypeScript modules to Closure modules via CommonJS internally. Set
      ↪  tsconfig.js "module": "commonjs"'
    );
    return 1;
  }

  // Run tsickle+TSC to convert inputs to Closure JS files.
  3 const result = toClosureJS(
    config.options,
    config.fileNames,
    settings,
    (filePath: string, contents: string) => {
      mkdirp.sync(path.dirname(filePath));
      4 fs.writeFileSync(filePath, contents, { encoding: 'utf-8' });
    }
  );
  if (result.diagnostics.length) {
    console.error(tsickle.formatDiagnostics(result.diagnostics));
    return 1;
  }

  5 if (settings.externsPath) {
    mkdirp.sync(path.dirname(settings.externsPath));
    fs.writeFileSync(
      settings.externsPath,
      tsickle.getGeneratedExterns(result.externs)
    );
  }
  return 0;
}

// CLI entry point
if (require.main === module) {
  process.exit(main(process.argv.splice(2)));
}
```

main 函数首先从 args 1 和 tsc 2 配置加载设置。然后它调用 `toClosureJs()` 3 函数，并将每个结果 JavaScript 文件输出到一个文件 4。如果在设置中设置了 externsPath，它们也会被写出到文件中 5。

`loadSettingsfromArgs()` 函数处理命令行参数，它可以是 tsickle 特定参数和常规 tsc 参数的混合。tsickle 特定的参数是–externs（生成 externs 文件）和–untyped（每个 TypeScript 类型都变成 Closure ? 类型）。

`toClosureJs()` 函数是转换发生的地方。它返回 1 转换文件内容的映射，可选地带有外部信息，它是这样配置的。

```
export function toClosureJS(
  options: ts.CompilerOptions,
  fileNames: string[],
  settings: Settings,
  writeFile?: ts.WriteFileCallback
): tsickle.EmitResult {
  1 const compilerHost = ts.createCompilerHost(options);
  2 const program = ts.createProgram(
    fileNames,
    options,
    compilerHost
  );

  3 const transformerHost: tsickle.TsickleHost = {
    shouldSkipTsickleProcessing: (fileName: string) => {
      return fileNames.indexOf(fileName) === -1;
    },
    shouldIgnoreWarningsForPath: (fileName: string) => false,
    pathToModuleName: cliSupport.pathToModuleName,
    fileNameToModuleId: (fileName) => fileName,
    es5Mode: true,
    googmodule: true,
    prelude: '',
    transformDecorators: true,
    transformTypesToClosure: true,
    typeBlackListPaths: new Set(),
    untyped: false,
    logWarning: (warning) =>
      console.error(tsickle.formatDiagnostics([warning])),
  };
  const diagnostics = ts.getPreEmitDiagnostics(program);
  if (diagnostics.length > 0) {
    return {
      diagnostics,
      modulesManifest: new ModulesManifest(),
      externs: {},
      emitSkipped: true,
      emittedFiles: [],
```

```
    };
  }
  4 return tsickle.emitWithTsickle(
    program,
    transformerHost,
    compilerHost,
    options,
    undefined,
    writeFile
  );
}
```

它首先根据提供的选项创建 1 一个编译器宿主，然后 2 它使用 TypeScript 的 createProgram 方法和原始程序源来确保它在语法上是正确的，并且任何错误消息都引用原始源，而不是修改后的源。然后它创建 3 一个 `tsickle.TsickleHost` 实例，它传递 4 给 `tsickle.emitWithTsickle()`。

`annotate` 是一个简单的函数：

```
export function annotate(
  typeChecker: ts.TypeChecker,
  file: ts.SourceFile,
  host: AnnotatorHost,
  tsHost?: ts.ModuleResolutionHost,
  tsOpts?: ts.CompilerOptions,
  sourceMapper?: SourceMapper
): { output: string; diagnostics: ts.Diagnostic[] } {
  return new Annotator(
    typeChecker,
    file,
    host,
    tsHost,
    tsOpts,
    sourceMapper
  ).annotate();
}
```

称为 rewriters 的类用于重写源码。rewriter.ts 文件具有 `Rewriter` 抽象类。一个重要的方法是 `maybeProcess()`。

```
/**
 * A Rewriter manages iterating through a ts.SourceFile, copying input
 * to output while letting the subclass potentially alter some nodes
 * along the way by implementing maybeProcess().
 */
export abstract class Rewriter {
  private output: string[] = [];
  /** Errors found while examining the code. */
  protected diagnostics: ts.Diagnostic[] = [];
  /** Current position in the output. */
```

```
  private position: SourcePosition = { line: 0, column: 0, position: 0 };
  /**
   * The current level of recursion through TypeScript Nodes.  Used in
formatting internal debug
   * print statements.
   */
  private indent = 0;
  /**
   * Skip emitting any code before the given offset.
   * E.g. used to avoid emitting @fileoverview
   * comments twice.
   */
  private skipCommentsUpToOffset = -1;
  constructor(
    public file: ts.SourceFile,
    private sourceMapper: SourceMapper = NOOP_SOURCE_MAPPER
  ) {}
  ..
}
```

根据此层次结构，tsickle.ts 有一些派生自 `Rewriter` 的类：

TODO：插图

`Annotator.maybeProcess()` 是实际重写发生的地方。

3: TS-API-Guardian

概览

Ts-api-guardian 是一个用于追踪包公有 API 的小工具。

用法

它用于在构建 Angular 时以检查 Angular 公有 API 的更改确保检测到对公共 API 的无意中更改。具体来说，你检查 Angular 主项目中的 gulpfile.ts：

- <ANGULAR-MASTER>/gulpfile.js

你会看到它有这样两行：

```
gulp.task('public-api:enforce', loadTask('public-api', 'enforce'));
gulp.task('public-api:update', ['build.sh'], loadTask('public-api', 'update'));
```

我们看向：

- <ANGULAR-MASTER>/tools/gulp-tasks/public-api.js

会看到名为'public-api:enforce' 和'public-api:update' 的两个任务，在这里我们会看到 ts-api-guardian 是如何被使用的。1 确保没有意外的改动，2 生成一个 “golden file” 代表 API：

```
    // Enforce that the public API matches the golden files
    // Note that these two commands work on built d.ts files
    // instead of the source
  1 enforce: (gulp) => (done) => {
    const platformScriptPath = require('./platform-script-path');
    const childProcess = require('child_process');
    const path = require('path');

    childProcess
      .spawn(
        path.join(
          __dirname,
          platformScriptPath(`../../node_modules/.bin/ts-api-guardian`)
        ),
```

```
      ['--verifyDir', path.normalize(publicApiDir)].concat(publicApiArgs),
      { stdio: 'inherit' }
    )
    .on('close', (errorCode) => {
      if (errorCode !== 0) {
        done(
          new Error(
            'Public API differs from golden file. Please run `gulp public-api:update`.'
          )
        );
      } else {
        done();
      }
    });
},
// Generate the public API golden files
update: (gulp) => (done) => {
  const platformScriptPath = require('./platform-script-path');
  const childProcess = require('child_process');
  const path = require('path');

  childProcess
    .spawn(
      path.join(
        __dirname,
        platformScriptPath(`../../node_modules/.bin/ts-api-guardian`)
      ),
      ['--outDir', path.normalize(publicApiDir)].concat(publicApiArgs),
      { stdio: 'inherit' }
    )
    .on('close', done);
},
```

目录结构

根目录包含这些文件:

- gulpfile.js
- package.json
- tsconfig.json
- tsd.json

和一级目录:

- bin
- lib
- test

gulpfile 中主要的任务为：

- compile
- test.compile
- test.unit
- watch

单元测试基于 mocha（不同于大多数其余的 Angular，它使用 jasmine）。

package.json 有如下 dependencies：

```
  "dependencies": {
    "chalk": "^1.1.3",
    "diff": "^2.2.3",
    "minimist": "^1.2.0",
    "typescript": "2.0.10"
  },
```

其中比较重要的是 diff 包，用来决定文本块之间的差异（https://www.npmjs.com/package/diff）。

Package.json 还列出了 ts-api-guardian 中的单个可调用程序：

```
  "bin": {
    "ts-api-guardian": "./bin/ts-api-guardian"
  },
```

bin

bin 只有一个文件，ts-api-guardian，其中只有这些行：

```
#!/usr/bin/env node
require('../build/lib/cli').startCli();
```

lib

lib 子目录包含这些文件：

- cli.ts ——命令行接口，处理参数列表和包装命令
- main.ts ——生成和验证 golden 文件主逻辑
- serializer.ts ——将代码序列化成 API（用于创建 golden 文件的内容）

golden 文件是 API 的文本表示，ts-api-guardian 的两个关键任务是根据提供的命令行参数创建或验证 golden 文件

Cli.ts 以一些关于如何调用 ts-api-guardian 的有用注释开始：

```
// # Generate one declaration file
// ts-api-guardian --out api_guard.d.ts index.d.ts
//
// # Generate multiple declaration files //#(output location like typescript)
// ts-api-guardian --outDir api_guard [--rootDir .] core/index.d.ts
```

```
core / testing.d.ts;
//
// # Print usage
// ts-api-guardian --help
//
// # Check against one declaration file
// ts-api-guardian --verify api_guard.d.ts index.d.ts
//
// # Check against multiple declaration files
// ts-api-guardian --verifyDir api_guard [--rootDir .] core/index.d.ts
core / testing.d.ts;
```

cli.ts 接受以下命令行选项：

```
--help                          Show this usage message
        --out <file>            Write golden output to file
        --outDir <dir>          Write golden file structure to directory
        --verify <file>         Read golden input from file
        --verifyDir <dir>       Read golden file structure from directory
        --rootDir <dir>         Specify the root directory of input files
        --stripExportPattern <regexp>
                                Do not output exports matching the pattern
        --allowModuleIdentifiers <identifier>
                                Whitelist identifier for "* as foo" imports
        --onStabilityMissing <warn|error|none>
                                Warn or error if an export has no stability annotation`);
```

Angular API 允许将注释附加到每个 API 上，表明它是稳定的、已弃用的还是实验性的。`onStabilityMissing` 选项指示如果缺少这样的注释需要什么操作。`startCli()` 函数解析命令行并初始化 `SerializationOptions` 的实例，然后对于生成模式调用 `SerializationOptions` 或对于验证模式调用 `verifyAgainstGoldenFile()` ——两者都在 main.ts 中，实际上是非常短的函数：

```
export function generateGoldenFile(
  entrypoint: string,
  outFile: string,
  options: SerializationOptions = {}
): void {
  const output = publicApi(entrypoint, options);
  ensureDirectory(path.dirname(outFile));
  fs.writeFileSync(outFile, output);
}
```

generateGoldenFile 调用 publicApi（来自 Serializer.ts）生成 golden 文件的内容，然后将其写入文件。VerifyAgainstGoldenFile() 也调用 publicApi 并将结果保存在名为 actual 的字符串中，然后将现有的 golden 文件数据加载到名为 expected 的字符串中，然后进行比较。如果不同，则调用 createPatch（来自 diff 包），以创建实际和预期 golden 文件之间差异的表示。

```
export function verifyAgainstGoldenFile(
  entrypoint: string,
  goldenFile: string,
  options: SerializationOptions = {}
): string {
  const actual = publicApi(entrypoint, options);
  const expected = fs.readFileSync(goldenFile).toString();

  if (actual === expected) {
    return '';
  } else {
    const patch = createPatch(
      goldenFile,
      expected,
      actual,
      'Golden file',
      'Generated API'
    );

    // Remove the header of the patch
    const start = patch.indexOf('\n', patch.indexOf('\n') + 1) + 1;

    return patch.substring(start);
  }
}
```

serializer.ts 定义了 SerializationOptions，它具有三个可选属性：

```
export interface SerializationOptions {
  /**
   * Removes all exports matching the regular expression.
   */
  stripExportPattern?: RegExp;
  /**
   * Whitelists these identifiers as modules in the output. For example,
   * ```
   * import * as angular from './angularjs';
   *
   * export class Foo extends angular.Bar {}
   * ```
   * will produce `export class Foo extends angular.Bar {}` and requires
   * whitelisting angular.
   */
  allowModuleIdentifiers?: string[];
  /**
   * Warns or errors if stability annotations are missing on an export.
   * Supports experimental, stable and deprecated.
```

```
   */
  onStabilityMissing?: string; // 'warn' | 'error' | 'none'
}
```

Serializer.ts 定义了一个公共 API 函数，它只调用 `publicApiInternal()`，后者又调用 `ResolvedDeclarationEmitter()`，这是一个 200 行的类，在其中执行实际工作。它具有三种执行序列化的方法：

- emit(): string
- private getResolvedSymbols(sourceFile: ts.SourceFile): ts.Symbol[]
- emitNode(node: ts.Node)

4: RxJS 6

简介

RxJS 是一个很棒的处理可观察流的框架。我们可以看到在 Angular 中像 event emitters、HTTP 响应以及服务都有应用。要成为一位优秀的 Angular 开发人员你首先不得不成为一位优秀的 RxJS 开发人员。这对于 Angular 的正确编程至关重要。RxJS 也是 NgRx 的基础基数，一个客户端 APP 非常流行的状态缓存引擎。

可观察对象是一个可枚举对象。使用可枚举，你的代码将调用 `.next()` 以重复获取序列中的下一项。相比之下，使用 observable，需提供一个函数来处理新数据项，一个可选的方法来处理错误，以及一个方法来处理完成事件（每一项结束）。当有新数据要推送的时候，observable 内部会调用你的代码。所以你可以想象一个 HTTP 请求被发送到服务器，一个数据块以多个数据包的形式返回。随着更多数据被返回，可观察对象将调用你代码中的回调函数来处理它们。

项目信息

RxJS 经历了多次迭代，最近又用 TypeScript 进行了重写。

项目主页地址是：http://reactivex.io/rxjs。

要开始使用源代码树，请访问 Github 上的仓库：

- https://github.com/ReactiveX/rxjs

API

在下面的链接中，我们假设 <RXJS> 指的是你 RxJS 的根目录位置。下面的链接实际上链接到 github 上的相关文档。

主要的导出在：

- <RXJS>/src/index.ts

有大量的操作符：

- <RXJS>/src/operators/index.ts

希望使用 RxJS 的应用开发人员大多会从这两个文件中导入他们需要的内容。在 src/internal 和 src/operators/internal 中有子目录，顾名思义，外部开发人员不应该直接导入（或以任何方式依赖于）这些，因为它们可能（将）从一个版本改变到另一个版本。

使用旧代码库并响应旧版本 RxJS 的开发人员应该在某个阶段升级到 RxJS 6 或更高版本。但是作为临时措施，可以使用兼容层来帮助旧代码针对 RxJS 6 运行。compat 包位于：

- <RXJS>/compat

如果你的代码库很大，立即升级到 RxJS 6 并不总是很方便，因此 compat 包可能非常有用。你应该考虑使用 compat 作为临时措施。随着时间的推移，你首先应该并且后来必须升级，因为预计 compat 包将在未来的某个时候消失。我们不在这里进一步讨论 compat 包。

<RXJS>/src/index.ts 导出的内容可以细分为以下几类：

- Observables
- Subjects (these are both observers and observables)
- Schedulers
- Subscription
- Notification
- Utils
- Error Types
- Static observable creation functions
- Two constants (EMPTY and NEVER)
- config

那么观察者在哪里导出？嗯……还有一个导出 `export *`：

```
1 export * from './internal/types';
```

如果我们看看：

- <RXJS>/src/types.ts

我们看到 `Observer` 被定义为一个接口，以及操作员接口、订阅接口、可观察接口和一系列其他观察者接口的定义——我们将很快探索它们。

目录结构

根目录包含这些子目录：

- .git
- github
- compat
- doc
- integration
- legacy-reexport

- migrations
- node-tests
- perf
- spec
- src
- tools
- tsconfig

根目录中的文件

有这些 JSON 文件：

- .dependency-cruiser.json
- esdoc.json
- package.json
- tsconfig.base.json
- tsconfig.json
- tsconfig.vscode.json
- tslint.json

有这些 .js 文件：

- .make-compat0package.js
- .make-helpers.js
- .make-packages.js
- .markdown-doctest-setup.js
- dangerfile.js
- index.js
- protractor.conf.js
- wallaby.js

有这些 . 文件：

- .editorconfig
- .eslintrc
- .gitignore
- .gitattributes
- .jshintrc
- .travis.yml
- .npmignore

有这些 markdown 文档：

- CHANGELOG.md

- CODE_OF_CONDUCT.md
- CONRIBUTING.md
- MIGRATION.md

两个 shell 脚本：

- publish.sh
- publish_docs.sh

还有一些其它文件：

- MAINTAINERS
- LICENSE.txt
- appveyor.yml

5: Platform-Server 包

概览

Platform-Server 表示应用在服务器上运行时的平台。

大多数情况下，Angular 应用在浏览器中运行并使用 Platform-Browser （使用离线模板编译器）或 Platform-Browser-Dynamic（使用运行时模板编译器）。在服务器上运行 Angular 应用是一种更专业的部署。它可能对搜索引擎优化和预渲染输出感兴趣，这些输出后来下载到浏览器（这可以加快向用户显示某些类型应用的内容的速度；还可以简化后端数据库的开发，这些数据库可能不是通过 REST API 暴露给浏览器代码）。Angular Universal 使用 Platform-Server 作为其平台模块。

在考虑 Platform-Server 时，好奇的软件工程师可能会思考两个问题。首先，我们想知道，如果浏览器有平台的动态和非动态版本，为什么服务器没有？这个问题的答案是，对于浏览器来说，希望支持网络连接较差的低端设备，因此减轻浏览器的负担很重要（因此仅使用离线模板编译器）；但是假设服务器有足够的磁盘空间和 RAM 以及少量额外的代码不是问题，所以捆绑两种服务器平台（一个使用离线模板编译器，另一个 - “动态” - 使用运行时模板编译器）在同一模块中简化了事情。

第二个问题是渲染如何在服务器（没有 DOM）上工作？答案是渲染输出通过 Angular 渲染器 API 写入 HTML 文件，然后可以将其下载到浏览器或提供给搜索引擎 - 我们需要探索这种渲染是如何工作的（但对于好奇的人，一个库使用称为 Domino，它为节点应用程序提供 DOM）。

Platform-Server API

index.ts 如下：

```
// This file is not used to build this module. It is only used during editing
// by the TypeScript language service and during build for verification.
// `ngc` replaces this file with production index.ts when it rewrites private
// symbol names.

export * from './public_api';
```

它的 public_api.ts 文件如下：

```
/**export {
 * @module
 * @description
 * Entry point for all public APIs of this package.
```

```
 */
export * from './src/platform-server';
// This file only reexports content of the `src` folder. Keep it that way.
```

它的 src/platfrom-server.ts 文件（这是一个连字符而不是下划线）如下：

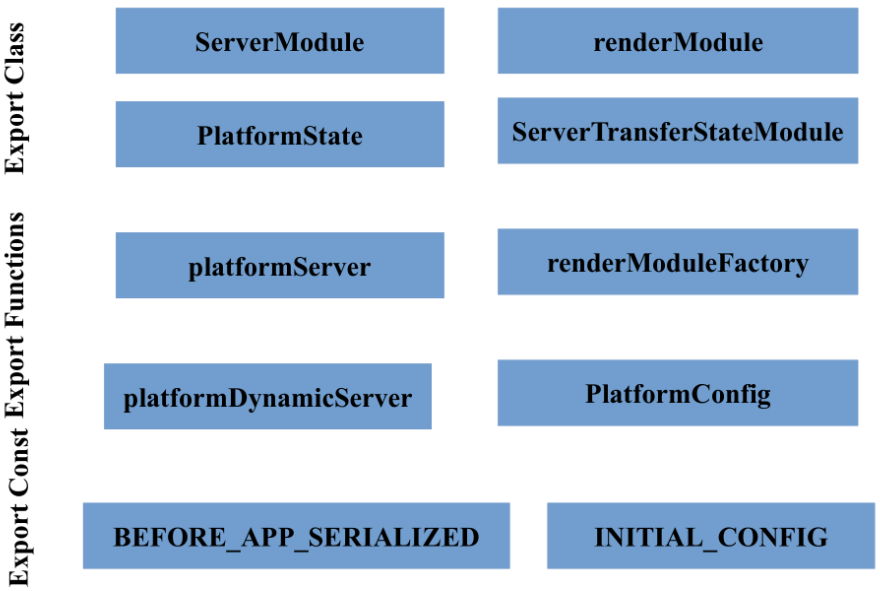
```
export { PlatformState } from './platform_state';
export { ServerModule, platformDynamicServer, platformServer } from './server';
export {
  BEFORE_APP_SERIALIZED,
  INITIAL_CONFIG,
  PlatformConfig,
} from './tokens';
export { ServerTransferStateModule } from './transfer_state';
export { renderModule, renderModuleFactory } from './utils';
export * from './private_export';
export { VERSION } from './version';
```

Platform-Server 主要的 src 目录也包含这个 private_export.ts 文件：

```
export {
  INTERNAL_SERVER_PLATFORM_PROVIDERS as ɵINTERNAL_SERVER_PLATFORM_PROVIDERS,
  SERVER_RENDER_PROVIDERS as ɵSERVER_RENDER_PROVIDERS,
} from './server';
export { ServerRendererFactory2 as ɵServerRendererFactory2 } from './server_renderer';
```

Platform-Server 包的导出 API 表示见图 5.1：

图 5.1: **Platform-Server Public API**



ServerModule 代表这个模块的 NgModule。platformServer() 使用离线编译器。plaformDynamicServer() 使用运行时编译器。两者都是对核心模块的 createPlatformFactory() 的调用。

源码结构

Platform-Server 包的源码包含以下目录：

- src
- test (unit tests in Jasmine)
- testing (test tooling)

以及这些文件：

- index.ts
- package.json
- public_api.ts

Package.json 的依赖项列为：

```
"peerDependencies": {
  "@angular/animations": "0.0.0-PLACEHOLDER",
  "@angular/common": "0.0.0-PLACEHOLDER",
  "@angular/compiler": "0.0.0-PLACEHOLDER",
  "@angular/core": "0.0.0-PLACEHOLDER",
  "@angular/platform-browser": "0.0.0-PLACEHOLDER",
  "@angular/platform-browser-dynamic": "0.0.0-PLACEHOLDER"
},
"dependencies": {
  "domino": "^2.1.2",
  "tslib": "^1.9.0",
  "xhr2": "^0.1.4"
},
```

domino 就是我们需要的服务器端 DOM 编辑包。

源码

platform-server/src

这些源文件存在：

- domino_adapter.ts
- http.ts
- location.ts
- platform_state.ts
- server_events.ts
- server_renderer.ts
- server.ts
- styles_host.ts

- tokens.ts
- transfer_state.ts
- utils.t
- version.ts

src 下没有子目录。

server.ts 文件声明了这个常量：

```
export const INTERNAL_SERVER_PLATFORM_PROVIDERS: StaticProvider[] = [
  { provide: DOCUMENT, useFactory: _document, deps: [Injector] },
  { provide: PLATFORM_ID, useValue: PLATFORM_SERVER_ID },
  {
    provide: PLATFORM_INITIALIZER,
    useFactory: initDominoAdapter,
    multi: true,
    deps: [Injector],
  },
  {
    provide: PlatformLocation,
    useClass: ServerPlatformLocation,
    deps: [DOCUMENT, [Optional, INITIAL_CONFIG]],
  },
  { provide: PlatformState, deps: [DOCUMENT] },
  // Add special provider that allows multiple instances of
  // platformServer* to be created.
  { provide: ALLOW_MULTIPLE_PLATFORMS, useValue: true },
];
```

工厂函数 platformServer 和 platformDynamicServer 分别创建使用离线模板编译器和运行时模板编译器的服务器平台。

它添加了两个额外的 provider 配置。首先是 PLATFORM_INITIALIZER，它是一个在引导前调用的初始化函数。在这里，我们看到它在调用本地函数 initDominoAdapter() 中初始化 domino 适配器，该函数为 DominoAdapter 调用 makeCurrent()：

```
function initDominoAdapter(injector: Injector) {
  return () => {
    DominoAdapter.makeCurrent();
  };
}
```

其次，它添加了 PlatformLocation，应用使用它与 location（URL）信息进行交互。它被设置为 location.ts 中定义的一个类，ServerPlatformLocation，它主要只是抛出异常：

```
/**
 * Server-side implementation of URL state. Implements `pathname`, `search`,
 * and `hash` but not the state stack.
 */
@Injectable()
```

```
export class ServerPlatformLocation implements PlatformLocation {
  public readonly href: string = '/';
  public readonly hostname: string = '/';
  public readonly protocol: string = '/';
  public readonly port: string = '/';
  public readonly pathname: string = '/';
  public readonly search: string = '';
  public readonly hash: string = '';
  private _hashUpdate = new Subject<LocationChangeEvent>();
  ..
  getBaseHrefFromDOM(): string {
    return getDOM().getBaseHref(this._doc)!;
  }

  onPopState(fn: LocationChangeListener): void {
    // No-op: a state stack is not implemented, so
    // no events will ever come.
  }

  onHashChange(fn: LocationChangeListener): void {
    this._hashUpdate.subscribe(fn);
  }

  get url(): string {
    return `${this.pathname}${this.search}${this.hash}`;
  }

  forward(): void {
    throw new Error('Not implemented');
  }
  back(): void {
    throw new Error('Not implemented');
  }

  // History API isn't available on server, therefore return undefined
  getState(): unknown {
    return undefined;
  }
}
```

server.ts 声明了两个导出函数：

```
export const platformServer = createPlatformFactory(
  platformCore,
  'server',
  INTERNAL_SERVER_PLATFORM_PROVIDERS
);
export const platformDynamicServer = createPlatformFactory(
```

```
  platformCoreDynamic,
  'serverDynamic',
  INTERNAL_SERVER_PLATFORM_PROVIDERS
);
```

我们之前看到 platformCore 定义在:

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

如下:

```
const _CORE_PLATFORM_PROVIDERS: StaticProvider[] = [
  // Set a default platform name for platforms that don't set it explicitly.
  { provide: PLATFORM_ID, useValue: 'unknown' },
  { provide: PlatformRef, deps: [Injector] },
  { provide: TestabilityRegistry, deps: [] },
  { provide: Console, deps: [] },
];

/**
 * This platform has to be included in any other platform
 *
 * @publicApi
 */
export const platformCore = createPlatformFactory(
  null,
  'core',
  _CORE_PLATFORM_PROVIDERS
);
```

platformCoreDynamic 向 platformCore 添加了额外的 provider 配置（用于动态编译器）并定义在:

- <ANGULAR-MASTER>/packages/platform-browser-dynamic/src/platfrom-browser-dynamic.ts

如下:

```
export const platformCoreDynamic = createPlatformFactory(
  platformCore,
  'coreDynamic',
  [
    { provide: COMPILER_OPTIONS, useValue: {}, multi: true },
    {
      provide: CompilerFactory,
      useClass: JitCompilerFactory,
      deps: [COMPILER_OPTIONS],
    },
  ]
);
```

createPlatformFactory() 定义在:

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

它使用给定的参数调用 Core 的 `createPlatform()`，从而构建一个新平台。

Platform-Server 的 server.ts 文件的其余部分要讨论的是名为 `ServerModule` 的 `NgModule` 的定义：

```
/**
 * The ng module for the server.
 *
 * @publicApi
 */
@NgModule({
  exports: [BrowserModule],
  imports: [HttpClientModule, NoopAnimationsModule],
  providers: [
    SERVER_RENDER_PROVIDERS,
    SERVER_HTTP_PROVIDERS,
    { provide: Testability, useValue: null },
    { provide: ViewportScroller, useClass: NullViewportScroller },
  ],
})
export class ServerModule {}

function _document(injector: Injector) {
  let config: PlatformConfig | null = injector.get(INITIAL_CONFIG, null);
  if (config && config.document) {
    return parseDocument(config.document, config.url);
  } else {
    return getDOM().createHtmlDocument();
  }
}
```

domino_adapter.ts 文件为 Domino 创建一个 DOM 适配器。Domino 库是在 Node.js 中运行的 HTML5 DOM 引擎。它的项目主页是：

- https://github.com/fgnass/domino

并指出：

> 顾名思义，*domino* 的目标是在 *Node.js* 中提供一个 *DOM*。

domino_adapter.ts 文件提供了一个适配器类 `DominoDomAdapter`，它基于 Domino 的序列化功能，它实现了一个适用于服务器环境的 `DomAdapter`。

domino adapter.ts 文件具有以下功能来解析和序列化文档：

```
/**
 * Parses a document string to a Document object.
 */
export function parseDocument(html: string, url = '/') {
  let window = domino.createWindow(html, url);
```

```
  let doc = window.document;
  return doc;
}

/**
 * Serializes a document to string.
 */
export function serializeDocument(doc: Document): string {
  return (doc as any).serialize();
}
```

我们看到 serializeDocument 被调用：

- <ANGULAR-MASTER>/packages/platform-server/src/platform_state.ts

如下：

```
@Injectable()
export class PlatformState {
  constructor(@Inject(DOCUMENT) private _doc: any) {}

  /**
   * Renders the current state of the platform to string.
   */
  renderToString(): string {
    return serializeDocument(this._doc);
  }

  /**
   * Returns the current DOM state.
   */
  getDocument(): any {
    return this._doc;
  }
}
```

BrowserDomAdapter 定义在：

- <ANGULAR-MASTER>/packages/platfrom-browser/src/browser/browser_adapter.ts

DominoAdapter 只是简单的继承 BrowserDomAdapter：

```
/**
 * DOM Adapter for the server platform based on
 * https://github.com/fgnass/domino.
 */
export class DominoAdapter extends BrowserDomAdapter {
  private static defaultDoc: Document;
```

```
  ..
}
```

它的静态 makeCurrent() 方法，我们看到 Universal Angular 用于服务器端渲染，初始化这三个变量，然后调用 setRootDomAdapter()：

```
  static makeCurrent() {
    setDomTypes();
    setRootDomAdapter(new DominoAdapter());
  }
```

回想一下 setRootDomAdapter() 定义在：

- <ANGULAR-MASTER>/packages/platform-browser/src/dom/dom_adapter.ts

如下：

```
let _DOM: DomAdapter = null!;

export function getDOM() {
  return _DOM;
}

export function setDOM(adapter: DomAdapter) {
  _DOM = adapter;
}

export function setRootDomAdapter(adapter: DomAdapter) {
  if (!_DOM) {
    _DOM = adapter;
  }
}
```

并且 getDOM() 被 DOMRenderer 使用。因此我们的 DominoAdapter 被连接到 DOM 渲染器中。

许多 DOM 适配器方法抛出异常，因为它们在服务器端没有意义：

```
  getHistory(): History {
    throw _notImplemented('getHistory');
  }
  getLocation(): Location {
    throw _notImplemented('getLocation');
  }
  getUserAgent(): string {
    return 'Fake user agent';
  }
```

6: Platform-WebWorker-Dynamic 包

概览

Platform-WebWorkers-Dynamic 是所有 Angular 包中最小的一个。它定义了一个常量，`platform-WorkerAppDynamic`，这是对 `createPlatformFactory()` 的调用。

将它作为一个单独的包管理的原因是 package.json 中的这一行：

```
1  "peerDependencies": {
2    ..,
3    "@angular/compiler": "0.0.0-PLACEHOLDER",
4    ..
5  },
```

它引入了相当大的运行时编译器。如果不使用它，我们希望避免这种情况（在使用离线编译器的非动态包中不需要它）。

Platform-WebWorker-Dynamic API

Platform-WebWorker-Dynamic 包导出的 API 表示见图 6.1：

图 6.1: **@Angular/Platform-WebWorker-Dynamic API**

源码结构

Platform-WebWorker-Dynamic 包源码树包含如下目录：

- src

当前，不包含测试或者测试子目录。它还包含了如下这些文件：

- index.ts
- package.json
- public_api.ts
- rollup.config.js
- tsconfig-build.json

index.ts 文件比较简单：

```
export * from './public_api';
```

public_api.ts 文件如下：

```
/**
 * @module
 * @description
 * Entry point for all public APIs of this package.
 */
export * from './src/platform-webworker-dynamic';
// This file only reexports content of the `src` folder. Keep it that way.
```

src/platform-webworker-dynamic.ts 文件如下：

```
// other imports
import {
  eResourceLoaderImpl as ResourceLoaderImpl,
  eplatformCoreDynamic as platformCoreDynamic,
} from '@angular/platform-browser-dynamic';

createPlatformFactory(platformCoreDynamic, 'workerAppDynamic', [
  {
    provide: COMPILER_OPTIONS,
    useValue: {
      providers: [
        { provide: ResourceLoader, useClass: ResourceLoaderImpl, deps: [] },
      ],
    },
    multi: true,
  },
  { provide: PLATFORM_ID, useValue: PLATFORM_WORKER_UI_ID },
]);
```

请注意，ResourceLoaderImpl 和 platformCoreDynamic 都是从 platform-browser-dynamic 包中导入的。

7: Router 包

概览

Angular Router 包提供如下能力：

- 基于 URLs 管理导航并对 URL 状态做出处理
- 管理应用程序状态和状态转换
- 根据需要进行热加载、懒加载和模块预加载

Router API

Angular Router 包导出的 API 结构见图 7.1、7.2 和 7.3。

图 7.1: Angular Router API (part 1)

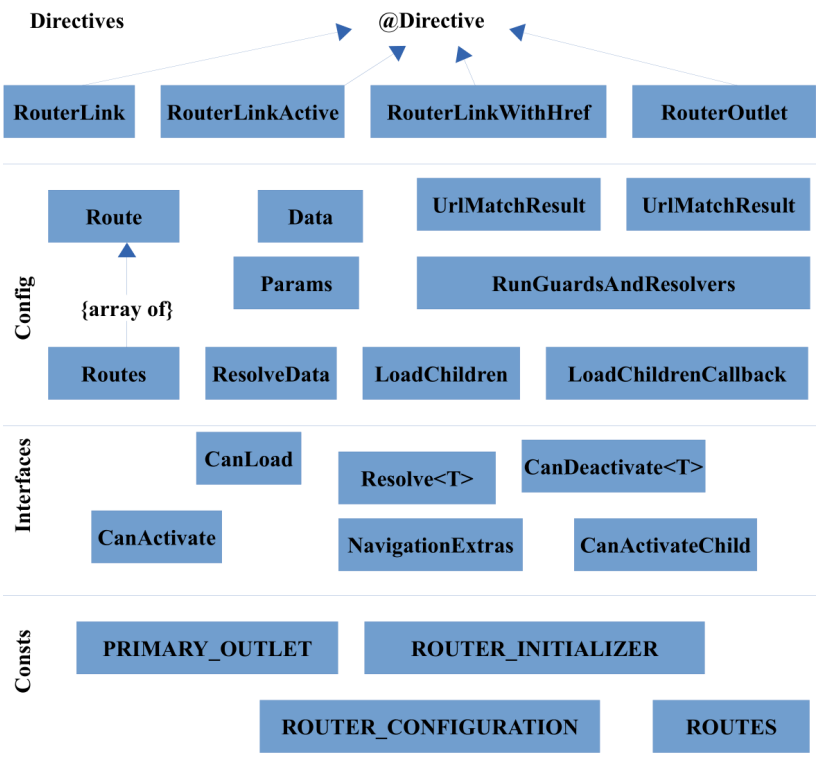


图 7.2: Angular Router API (part 2)

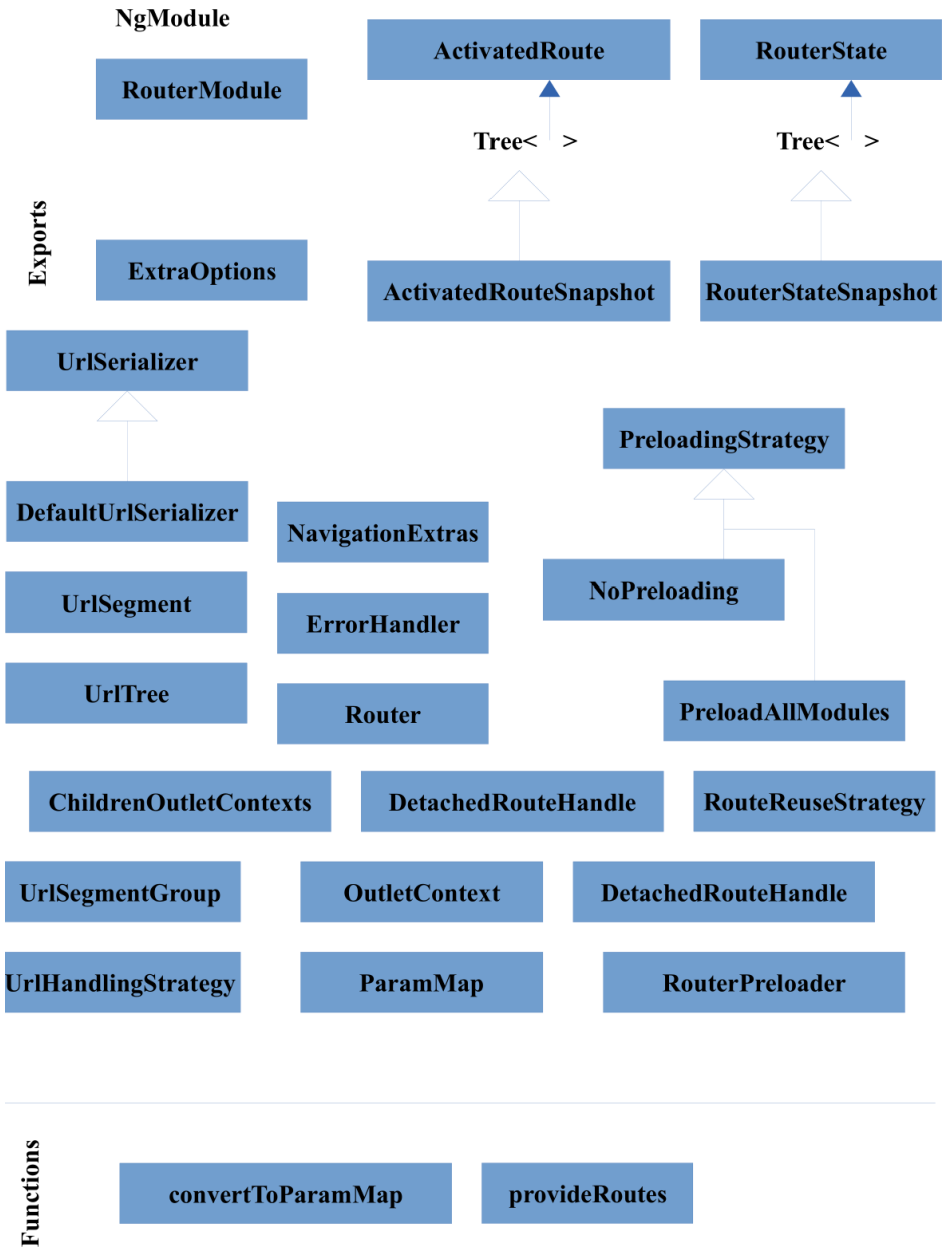
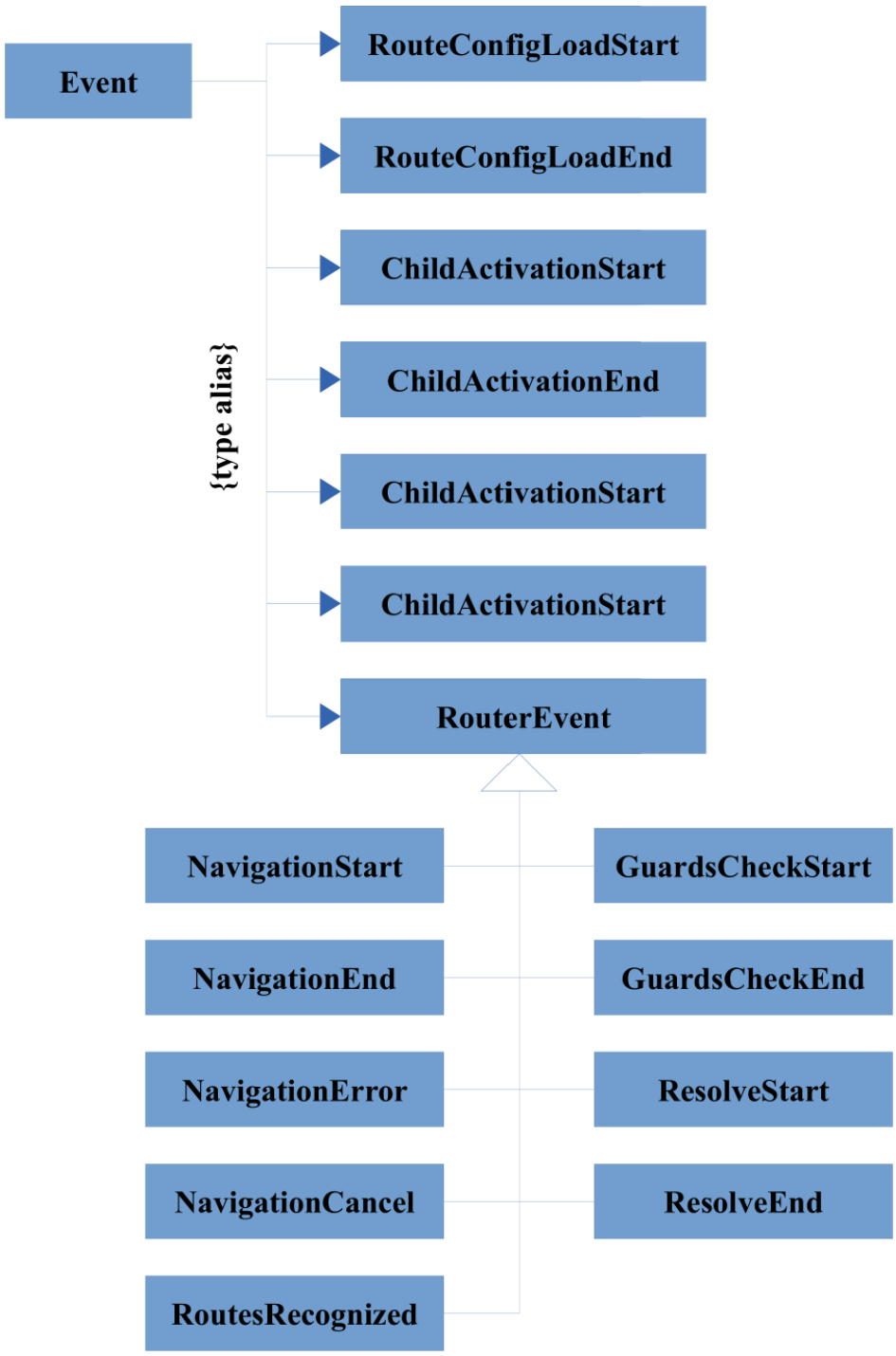


图 7.3: Angular Router API (part 2)



Angular Router 源码位于：

- <ANGULAR-MASTER>/packages/router

根目录包含 index.ts，它只是导出了 public_api.t 的内容，而后者又导出 .src/index.ts 的内容。这列出了 exports 内容并给出了路由器包大小的初步印象：

```
export {
  Data,
  LoadChildren,
  LoadChildrenCallback,
  ResolveData,
  Route,
  Routes,
  RunGuardsAndResolvers,
  UrlMatchResult,
  UrlMatcher,
} from './config';
export { RouterLink, RouterLinkWithHref } from './directives/router_link';
export { RouterLinkActive } from './directives/router_link_active';
export { RouterOutlet } from './directives/router_outlet';
export {
  ActivationEnd,
  ActivationStart,
  ChildActivationEnd,
  ChildActivationStart,
  Event,
  GuardsCheckEnd,
  GuardsCheckStart,
  NavigationCancel,
  NavigationEnd,
  NavigationError,
  NavigationStart,
  ResolveEnd,
  ResolveStart,
  RouteConfigLoadEnd,
  RouteConfigLoadStart,
  RouterEvent,
  RoutesRecognized,
} from './events';
export {
  CanActivate,
  CanActivateChild,
  CanDeactivate,
  CanLoad,
  Resolve,
} from './interfaces';
export {
```

```
  DetachedRouteHandle,
  RouteReuseStrategy,
} from './route_reuse_strategy';
export { NavigationExtras, Router } from './router';
export { ROUTES } from './router_config_loader';
export {
  ExtraOptions,
  ROUTER_CONFIGURATION,
  ROUTER_INITIALIZER,
  RouterModule,
  provideRoutes,
} from './router_module';
export { ChildrenOutletContexts, OutletContext } from './router_outlet_context';
export {
  NoPreloading,
  PreloadAllModules,
  PreloadingStrategy,
  RouterPreloader,
} from './router_preloader';
export {
  ActivatedRoute,
  ActivatedRouteSnapshot,
  RouterState,
  RouterStateSnapshot,
} from './router_state';
export { PRIMARY_OUTLET, ParamMap, Params, convertToParamMap } from './shared';
export { UrlHandlingStrategy } from './url_handling_strategy';
export {
  DefaultUrlSerializer,
  UrlSegment,
  UrlSegmentGroup,
  UrlSerializer,
  UrlTree,
} from './url_tree';
export { VERSION } from './version';
```

它还有一行：

```
export * from './private_export';
```

正如已经讨论过的，此类私有导出旨在用于 Angular 框架本身，而不应该由 Angular 应用直接使用。private_export.ts 文件具有这些导出（注意，它们的名称有“ɵ”字符）：

```
export { ROUTER_PROVIDERS as ɵROUTER_PROVIDERS } from './router_module';
export { flatten as ɵflatten } from './utils/collection';
```

目录结构

Router 包的源码树包含这些目录：

- scripts
- src
- test (unit tests in Jasmine)
- testing (testing tools)

Router 包根目录包含这些文件：

- BUILD.bazel
- index.ts
- karma-test-shim.ts
- karma.config.js
- LICENSE
- package.json
- public_api.ts
- README.md
- rollup.config.js
- tsconfig-build.json

源码

router/src

router/src 目录包含：

- apply_redirects.ts
- config.ts
- create_router_state.ts
- create_url_tree.ts
- events.ts
- interfaces.ts
- index.ts
- interfaces.ts
- pre_activation.ts
- private_export.ts
- recognize.ts
- resolve.ts
- route_reuse_strategy.ts
- router_config_loader.ts
- router_module.ts
- router_outlet_context.ts
- router_preloader.ts
- router_state.ts

- router.ts
- shared.ts
- url_handling_strategy.ts
- url_tree.ts
- version.ts

我们将先查看：

- <ANGULAR-MASTER>/packages/router/src/router_module.ts

它定义了 RouterModule 类和相关类型。
它定义了三个常量：

```
const ROUTER_DIRECTIVES = [
  RouterOutlet,
  RouterLink,
  RouterLinkWithHref,
  RouterLinkActive,
];
```

第一个，ROUTER_DIRECTIVES 是路由器指令的集合，可以出现在 Angular 模板中，定义路由内容在页面上的位置，以及如何显示用于路由的链接。ROUTER_DIRECTIVES 在 RouterModule 的 @NgModule 元数据的声明和导出中指定：

```
@NgModule({ declarations: ROUTER_DIRECTIVES, exports: ROUTER_DIRECTIVES })
export class RouterModule {
  ..
}
```

另外两个是 DI 的注入令牌：

```
export const ROUTER_CONFIGURATION = new InjectionToken<ExtraOptions>(
  'ROUTER_CONFIGURATION'
);
export const ROUTER_FORROOT_GUARD = new InjectionToken<void>(
  'ROUTER_FORROOT_GUARD'
);
```

它还定义了提供程序的 ROUTER_PROVIDERS 数组（仅用于 forRoot，不用于 forChild）：

```
export const ROUTER_PROVIDERS: Provider[] = [
  Location,
  { provide: UrlSerializer, useClass: DefaultUrlSerializer },
  {
    provide: Router,
    useFactory: setupRouter, ..
  },
  ChildrenOutletContexts,
  { provide: ActivatedRoute, useFactory: rootRoute, deps: [Router] },
  { provide: NgModuleFactoryLoader, useClass: SystemJsNgModuleLoader },
```

```
  RouterPreloader,
  NoPreloading,
  PreloadAllModules,
  { provide: ROUTER_CONFIGURATION, useValue: { enableTracing: false } },
];
```

一个重要的 provider 是 `Router`，这是实际的路由服务。这是在 DI 中设置以返回 `setupRouter` 工厂方法的结果。其缩写版本如下：

```
export function setupRouter(..) {
  const router = new Router(
    null,
    urlSerializer,
    contexts,
    location,
    injector,
    loader,
    compiler,
    flatten(config)
  );

  if (urlHandlingStrategy)
  if (routeReuseStrategy) ..
  if (opts.errorHandler) ..
  if (opts.enableTracing) ..
  if (opts.onSameUrlNavigation) ..
  if (opts.paramsInheritanceStrategy) ..
  ..
  return router;
}
```

它实例化路由服务，并为每个指定的选项/策略采取适当的行动。然后它返回新的路由服务实例。重要的是每个应用只有一个路由服务（Web 浏览器每个会话只有一个 URL），我们需要跟踪这是如何发生的。

RouterModule 定义如下：

```
@NgModule({ declarations: ROUTER_DIRECTIVES, exports: ROUTER_DIRECTIVES })
export class RouterModule {
  // Note: We are injecting the Router so it gets created eagerly...
  constructor(
    @Optional() @Inject(ROUTER_FORROOT_GUARD) guard: any,
    @Optional() router: Router
  ) {}
  static forRoot(routes: Routes, config?: ExtraOptions): ModuleWithProviders {
    ..
  }
  static forChild(routes: Routes): ModuleWithProviders {
    ..
```

```
  }
}
```

请注意，forRoot 和 forChild 都返回一个 ModuleWithProviders 实例。他们放进去的东西不一样。回想一下，这种类型定义在：

- <ANGULAR-MASTER>/packages/core/src/metadata/ng_module.ts

如下：

```
// wrapper around a module that also includes the providers.
export interface ModuleWithProviders {
  ngModule: Type<any>;
  providers?: Provider[];
}
```

ForChild 适用于除根路由模块之外的所有模块。它返回 ngModule 和一个仅包含调用 provideRoutes 结果的 providers 列表：

```
  static forChild(routes: Routes): ModuleWithProviders {
    return { ngModule: RouterModule, providers: [provideRoutes(routes)] };
  }
```

重要的是，它不包含 ROUTER_PROVIDERS。相比之下，forRoot 添加了这个和更多的 providers：

```
  static forRoot(routes: Routes, config?: ExtraOptions): ModuleWithProviders {
    return {
      ngModule: RouterModule,
      providers: [
        ROUTER_PROVIDERS,
        provideRoutes(routes),
        {
          provide: ROUTER_FORROOT_GUARD, ..
        },
        {
          provide: ROUTER_CONFIGURATION, ..
        },
        {
          provide: LocationStrategy, ..
        },
        {
          provide: PreloadingStrategy, ..
        },
        {
          provide: NgProbeToken, ..
        },
        provideRouterInitializer(),
      ],
```

```
    };
  }
```

ExtraOptions 是传递给 forRoot 的附加选项（它不与 forChild 一起使用）：

```
export interface ExtraOptions {
  enableTracing?: boolean;
  useHash?: boolean;
  initialNavigation?: InitialNavigation;
  errorHandler?: ErrorHandler;
  preloadingStrategy?: any;
  onSameUrlNavigation?: 'reload' | 'ignore';
  paramsInheritanceStrategy?: 'emptyOnly' | 'always';
}
```

例如，如果我们希望自定义预加载的工作方式，我们需要设置 preloadingStrategy 选项。provideRouterInitializer() 函数提供了一个 initializers 列表：

```
export function provideRouterInitializer() {
  return [
    RouterInitializer,
    {
      provide: APP_INITIALIZER,
      multi: true,
      useFactory: getAppInitializer,
      deps: [RouterInitializer],
    },
    {
      provide: ROUTER_INITIALIZER,
      useFactory: getBootstrapListener,
      deps: [RouterInitializer],
    },
    {
      provide: APP_BOOTSTRAP_LISTENER,
      multi: true,
      useExisting: ROUTER_INITIALIZER,
    },
  ];
}
```

这使用了 RouterInitializer 类，其目的最好通过代码中的此注释来解释：

```
/**
 * To initialize the router properly we need to do in two steps:
 *
 * We need to start the navigation in a APP_INITIALIZER to block the
 * bootstrap if a resolver or a guards executes asynchronously. Second, we
 * need to actually run activation in a BOOTSTRAP_LISTENER. We utilize the
 * afterPreactivation hook provided by the router to do that.
```

```
 *
 * The router navigation starts, reaches the point when preactivation is
 * done, and then pauses. It waits for the hook to be resolved. We then
 * resolve it only in a bootstrap listener.
 */
@Injectable()
export class RouterInitializer {
  ..
}
```

我们在检查 Core 包时看到它：

- <ANGULAR_MASTER>/packages/core/src/application_init.ts

将 APP_INITIALIZER 定义为：

```
// A function that will be executed when an application is initialized.
export const APP_INITIALIZER = new InjectionToken<Array<() => void>>(
  'Application Initializer'
);
```

Interfaces.ts 声明了许多有用的接口。

```
export interface CanActivate {
  canActivate(
    route: ActivatedRouteSnapshot,
    state: RouterStateSnapshot
  ): Observable<boolean> | Promise<boolean> | boolean;
}
export interface CanActivateChild {
  canActivateChild(
    childRoute: ActivatedRouteSnapshot,
    state: RouterStateSnapshot
  ): Observable<boolean> | Promise<boolean> | boolean;
}
export interface CanDeactivate<T> {
  canDeactivate(
    component: T,
    currentRoute: ActivatedRouteSnapshot,
    currentState: RouterStateSnapshot,
    nextState?: RouterStateSnapshot
  ): Observable<boolean> | Promise<boolean> | boolean;
}
export interface Resolve<T> {
  resolve(
    route: ActivatedRouteSnapshot,
    state: RouterStateSnapshot
  ): Observable<T> | Promise<T> | T;
}
export interface CanLoad {
```

```
  canLoad(route: Route): Observable<boolean> | Promise<boolean> | boolean;
}
```

router_config_loader.ts 定义了一个类——RouterConfigLoader ——和一个 opaque 令牌，经过一些簿记后，RouterConfigLoader 创建了 LoadedRouterConfig 的一个新实例：

```
export const ROUTES = new InjectionToken<Route[][]>('ROUTES');

export class RouterConfigLoader {
  constructor(
    private loader: NgModuleFactoryLoader,
    private compiler: Compiler,
    private onLoadStartListener?: (r: Route) => void,
    private onLoadEndListener?: (r: Route) => void
  ) {}

  load(parentInjector: Injector, route: Route): Observable<LoadedRouterConfig> {
    ..
    return new LoadedRouterConfig(flatten(module.injector.get(ROUTES)), module);
  }
  ..
}
```

router_state.ts 包含这些类（和一些辅助函数）：

- RouterState
- ActivatedRoute
- ActivatedRouteSnapshot
- RouterStateSnapshot

RouterState 定义为：

```
// RouterState is a tree of activated routes.
// Every node in this tree knows about the "consumed" URL
// segments, the extracted parameters, and the resolved data.
export class RouterState extends Tree<ActivatedRoute> {
  constructor(
    root: TreeNode<ActivatedRoute>,
    public snapshot: RouterStateSnapshot
  ) {
    super(root);
    setRouterState(<RouterState>this, root);
  }
}
```

RouterStateSnapshot 定义为：

```
/**
 * @whatItDoes Represents the state of the router at a moment in time.
 * RouterStateSnapshot is a tree of activated route snapshots. Every node in
 * this tree knows about the "consumed" URL segments, the extracted
 * parameters, and the resolved data.
 */
export class RouterStateSnapshot extends Tree<ActivatedRouteSnapshot> {
  constructor(public url: string, root: TreeNode<ActivatedRouteSnapshot>) {
    super(root);
    setRouterState(<RouterStateSnapshot>this, root);
  }
}
```

setRouterStateSnapshot() 函数定义为:

```
function setRouterState<U, T extends { _routerState: U }>(
  state: U,
  node: TreeNode<T>
): void {
  node.value._routerState = state;
  node.children.forEach((c) => setRouterState(state, c));
}
```

所以它为当前节点设置路由状态，然后递归调用 setRouterState() 为所有子节点设置它。
路由器出口指令使用 ActivatedRoute 类来描述它已加载的组件:

```
// Contains the information about a route associated with a component loaded
// in an outlet. An `ActivatedRoute` can also be used to traverse the
// router state tree
export class ActivatedRoute {
  ..
  constructor() { ..
    this._futureSnapshot = futureSnapshot;
  }

  /** The configuration used to match this route */
  get routeConfig(): Route | null {
    return this._futureSnapshot.routeConfig;
  }

  /** The root of the router state */
  get root(): ActivatedRoute {
    return this._routerState.root;
  }

  /** The parent of this route in the router state tree */
  get parent(): ActivatedRoute | null {
    return this._routerState.parent(this);
```

```
  }

  /** The first child of this route in the router state tree */
  get firstChild(): ActivatedRoute | null {
    return this._routerState.firstChild(this);
  }

  /** The children of this route in the router state tree */
  get children(): ActivatedRoute[] {
    return this._routerState.children(this);
  }

  /** The path from the root of the router state tree to this route */
  get pathFromRoot(): ActivatedRoute[] {
    return this._routerState.pathFromRoot(this);
  }

  get paramMap(): Observable<ParamMap> {
    ..
  }
  get queryParamMap(): Observable<ParamMap> {
    ..
  }
}
```

router/src/directives

该目录有如下文件：

- router_link.ts
- router_link_active.ts
- router_outlet.ts

router_link.ts 文件包含 RouterLink 指令：

```
@Directive({ selector: ':not(a)[routerLink]' })
export class RouterLink {
  @Input() queryParams: { [k: string]: any };
  @Input() fragment: string;
  @Input() queryParamsHandling: QueryParamsHandling;
  @Input() preserveFragment: boolean;
  @Input() skipLocationChange: boolean;
  @Input() replaceUrl: boolean;
  private commands: any[] = [];
  private preserve: boolean;

  constructor(
    private router: Router,
```

```
    private route: ActivatedRoute,
    @Attribute('tabindex') tabIndex: string,
    renderer: Renderer2,
    el: ElementRef
  ) {
    if (tabIndex == null) {
      renderer.setAttribute(el.nativeElement, 'tabindex', '0');
    }
  }
  ..
}
```

router link 命令通过以下方式设置：

```
  @Input()
  set routerLink(commands: any[] | string) {
    if (commands != null) {
      this.commands = Array.isArray(commands) ? commands : [commands];
    } else {
      this.commands = [];
    }
  }
```

当链接被点击时，会调用 onClick() 方法：

```
  @HostListener('click')
  onClick(): boolean {
    const extras = {
      skipLocationChange: attrBoolValue(this.skipLocationChange),
      replaceUrl: attrBoolValue(this.replaceUrl),
    };
    this.router.navigateByUrl(this.urlTree, extras);
    return true;
  }
```

urlTree getter 使用 Router.createlUrlTree()：

```
  get urlTree(): UrlTree {
    return this.router.createUrlTree(this.commands, {
      relativeTo: this.route,
      queryParams: this.queryParams,
      fragment: this.fragment,
      preserveQueryParams: attrBoolValue(this.preserve),
      queryParamsHandling: this.queryParamsHandling,
      preserveFragment: attrBoolValue(this.preserveFragment),
    });
  }
```

该文件还包含了 RouterLinkWithHref 指令：

```
@Directive({ selector: 'a[routerLink]' })
export class RouterLinkWithHref implements OnChanges, OnDestroy {
  ..
}
```

一个 href：

```
  // the url displayed on the anchor element.
  @HostBinding() href: string;
```

并将 urlTree 作为字段进行管理，并通过调用从构造函数设置它：

```
  private updateTargetUrlAndHref(): void {
    this.href = this.locationStrategy.prepareExternalUrl(
      this.router.serializeUrl(this.urlTree)
    );
  }
```

router_link_active.ts 文件包含 RouterLinkActive 指令：

```
@Directive({
  selector: '[routerLinkActive]',
  exportAs: 'routerLinkActive',
})
export class RouterLinkActive
  implements OnChanges, OnDestroy, AfterContentInit {
  ..
}
```

用于向活动路由元素添加 CSS 类。它的构造函数定义如下：

```
  constructor(
    private router: Router,
    private element: ElementRef,
    private renderer: Renderer2,
    private cdr: ChangeDetectorRef
  ) {
    this.subscription = router.events.subscribe((s) => {
      if (s instanceof NavigationEnd) {
        this.update();
      }
    });
  }
```

它的 update 方法使用配置的渲染器来设置元素的 class：

```
  private update(): void {
    if (!this.links || !this.linksWithHrefs || !this.router.navigated) return;
    Promise.resolve().then(() => {
      const hasActiveLinks = this.hasActiveLinks();
      if (this.isActive !== hasActiveLinks) {
```

```
        (this as any).isActive = hasActiveLinks;
        this.classes.forEach((c) => {
          if (hasActiveLinks) {
            this.renderer.addClass(this.element.nativeElement, c);
          } else {
            this.renderer.removeClass(this.element.nativeElement, c);
          }
        });
      }
    });
  }
```

router_outlet.ts 文件包含了 RouterOutlet 类：

```
// Acts as a placeholder that Angular dynamically fills based on the
// current router * state.
@Directive({ selector: 'router-outlet', exportAs: 'outlet' })
export class RouterOutlet implements OnDestroy, OnInit {
  private activated: ComponentRef<any> | null = null;
  private _activatedRoute: ActivatedRoute | null = null;
  private name: string;

  @Output('activate') activateEvents = new EventEmitter<any>();
  @Output('deactivate') deactivateEvents = new EventEmitter<any>();

  constructor(
    private parentContexts: ChildrenOutletContexts,
    1 private location: ViewContainerRef,
    2 private resolver: ComponentFactoryResolver,
    @Attribute('name') name: string,
    private changeDetector: ChangeDetectorRef
  ) {
    this.name = name || PRIMARY_OUTLET;
    parentContexts.onChildOutletCreated(this.name, this);
  }
  ..
}
```

这是生命周期取决于路由器的应用程序组件所在的位置。我们注意到构造函数的 ViewContainerRef 1 和 ComponentFactoryResolver 2 参数。

ngOnInit 将调用 attach 1 或 activateWith 2，具体取决于是否存在现有组件：

```
  ngOnInit(): void {
    if (!this.activated) {
      // If the outlet was not instantiated at the time the

      // route got activated we need to populate
      // the outlet when it is initialized (ie inside a NgIf)
```

```
    const context = this.parentContexts.getContext(this.name);
    if (context && context.route) {
      if (context.attachRef) {
        // `attachRef` is populated when there is an
        // existing component to mount
        this.attach(context.attachRef, context.route);
      } else {
        // otherwise the component defined in the configuration is created
        this.activateWith(
          context.route,
          context.resolver || null
        );
      }
    }
  }
}
```

attach 定义为：

```
// Called when the `RouteReuseStrategy` instructs to
// re-attach a previously detached subtree
attach(ref: ComponentRef<any>, activatedRoute: ActivatedRoute) {
  this.activated = ref;
  this._activatedRoute = activatedRoute;
  this.location.insert(ref.hostView);
}
```

当它的 activateWith 方法被调用时，resolver 将被要求为组件解析一个组件工厂：

```
activateWith(
  activatedRoute: ActivatedRoute,
  resolver: ComponentFactoryResolver | null
) {
  if (this.isActivated) {
    throw new Error('Cannot activate an already activated outlet');
  }
  this._activatedRoute = activatedRoute;
  const snapshot = activatedRoute._futureSnapshot;
  const component = <any>snapshot.routeConfig!.component;
  resolver = resolver || this.resolver;
  const factory = resolver.resolveComponentFactory(component);
  const childContexts = this.parentContexts.getOrCreateContext(
    this.name
  ).children;
  const injector = new OutletInjector(
    activatedRoute,
    childContexts,
    this.location.injector
  );
```

```
    this.activated = this.location.createComponent(
      factory,
      this.location.length,
      injector
    );

    // Calling `markForCheck` to make sure we will run the change
    // detection when the     // `RouterOutlet` is inside a
    //  `ChangeDetectionStrategy.OnPush` component.
    this.changeDetector.markForCheck();
    this.activateEvents.emit(this.activated.instance);
  }
```

location 字段的类型为 ViewContainerRef，我们看到它是在构造函数中设置的。ViewContainerRef 定义在：

- <ANGULAR-MASTER>/packages/core/src/linker/view_container_ref.ts

```
export abstract class ViewContainerRef {
  // Returns the number of Views currently attached to this container.
  abstract get length(): number;

  // Instantiates a single Component and inserts its Host View into this
  // container at the specified `index`.
  // The component is instantiated using its ComponentFactory which can be
  // obtained via ComponentFactoryResolver.resolveComponentFactory
  // If `index` is not specified, the new View will be inserted as the last
  // View in the container. Returns the {@link ComponentRef} of the Host
  // View created for the newly instantiated Component.
  abstract createComponent<C>(
    componentFactory: ComponentFactory<C>,
    index?: number,
    injector?: Injector,
    projectableNodes?: any[][],
    ngModule?: NgModuleRef<any>
  ): ComponentRef<C>;
}
```

因此该组件被附加为 ViewContainer 中的最后一个条目。

8: Render3 (Ivy) in Angular

前置 - 路径和名称

本章探讨现代 Angular 源码中新的渲染引擎——“Ivy”。我们查看了主要的 Angular 仓库（以 ANGULAR-MASTER 为前缀的路径），并且还讨论了 Angular CLI 仓库（以 ANGULAR-CLI-MASTER 为前缀的路径）。你可能希望将这些仓库克隆到您的本地机器上，或者你可能希望在 Github 上参考它们。如果是后者，你应该将这些占位符替换为：

- ANGULAR-MASTER - https://github.com/angular/angular
- ANGUALR-CLI-MASTER - https://github.com/angular/angular-cli

我们更倾向于使用名称“Render3”（而不是带空格的“Render 3”），因为它有助于 Google 搜索等。它的代号是“Ivy”，我们将看到它在代码中的地方使用，并带有用于 Compiler-CLI 的新 `"enableIvy"` 选项，可以通过 Angular CLI 的 `"ng new"` 命令添加，如下所述：

- https://next.angular.io/guide/ivy

概览

Angular 的渲染正在进行进一步的进化。似乎还不久前 Angular2 的原始渲染架构演变为 Render2，现在是新的 Render3，一种完全不同的视图引擎方法。

主要变化可以用一个函数来总结(来自 <ANGULAR-MASTER>/packages/core/src/render3/interfaces/renderer.ts)：

```
export const domRendererFactory3: RendererFactory3 = {
  createRenderer: (
    hostElement: RElement | null,
    rendererType: RendererType2 | null
  ): Renderer3 => {
    return document;
  } 1,
};
```

默认情况下，新渲染引擎与标准 DOM 中的 document 对象兼容。其实，最好不要只说“兼容”，而要说“是”。当在浏览器 UI 线程中运行时，DOM 可用，然后是浏览器的原生 document 对象（由 Web 浏览器本身提供，而不是 Angular）真的是渲染器——它直接用于渲染内容。从字面上看，应用的执行速

度不会比这更快。无论你使用哪种框架以及它如何构建其渲染管道，最终都必须调用真实 DOM 中的这个 document 对象。那么为什么不在支持它的场景中直接调用它（即意味着在浏览器 UI 线程内）？这正是 Ivy 在可能的情况下所做的。对于其他渲染场景（Web Worker、服务器或更专业的，例如 WebRTC），则需要提供类似于 document 对象的内容。

对于来自 C++、C#、Java 或类似背景的读者来说，理解 TypeScript（和 JavaScript）使用 结构子类型 （也被亲切地称为“鸭子类型”）而不是那些使用的名义类型（命名类型）的其它语言非常重要。在 TypeScript 中，可以使用实现另一种类型字段的类型来代替它——没有必要实现公共接口或具有公共父类。因此，标准 DOM 中的 document 对象没有实现 Angular 的 Render3，但在函数中使用以返回这种类型（参见上面的 1）这一事实不是问题，只要它实现了 Render3 所需要的字段（它确实需要，我们很快就会发现）。

现在我们将更深入地探索当我们将 `"enableIvy"` 与 Angular CLI 结合使用时会发生什么，以及主要的 Angular 项目如何提供 Render3 功能。我们将看到涉及到它的三个包——Compiler-CLI、Compiler 和 Core。

公共文档

如果我们访问：

- https://next.angular.io/api?query=render

我们将会看到所有包含“render”关键字的公共 API 列表
所以目前，没有任何 Renderer3 功能的公共 API。

Angular CLI 和 enableIvy

希望使用 Ivy 的应用开发人员主要通过 Angular CLI 命令行选项：

- https://next.angular.io/guide/ivy

我们可以从 Angular CLI 8.2 源码中看到这个选项影响了基本代码的几个地方。Angular 应用选项的 schema 定义在此处：

- <ANGULAR-CLI-MASTER>/packages/schematics/angular/application/schema.json

它包含了 `enableIvy`：

```
{
  "$schema": "http://json-schema.org/schema",
  "id": "SchematicsAngularApp",
  "title": "Angular Application Options Schema",
  "type": "object",
  "description": "Generates a new basic app definition in the \"projects\" subfolder of the
  ↪  workspace.",
  "properties": {
    "enableIvy": {
```

```
      "description": "**EXPERIMENTAL** True to create a new app that uses the Ivy rendering
      ↪  engine.",
      "type": "boolean",
      "default": false,
      "x-user-analytics": 8
```

Angular 应用原理图的入口：

- <ANGULAR-CLI-MASTER>/packages/schematics/angular/application/index.ts

有如下代码：

```
const project = {
  root: normalize(projectRoot),
  sourceRoot,
  projectType: ProjectType.Application,
  prefix: options.prefix || 'app',
  schematics,
  targets: {
    build: {
      builder: Builders.Browser,
      options: {
        ..
        aot: !!options.enableIvy,
        ..
      }, ..
    }, ..
  }, ..
};
```

Angular NG New 选项的模式在此处定义：

- <ANGULAR-CLI-MASTER>/packages/schematics/angular/ng-new/schema.json

有如下条目：

```
{
  "$schema": "http://json-schema.org/schema",
  "id": "SchematicsAngularNgNew",
  "title": "Angular Ng New Options Schema",
  "type": "object",
  "properties": {
    "enableIvy": {
      "description": "When true, creates a new app that uses the Ivy rendering engine.",
      "type": "boolean",
      "default": false
    }
```

上面的描述字段最终在生成的文档中可以找到：

- https://next.angular.io/cli/new

ngnew" 入口：

- <ANGULAR-CLI-MASTER>/packages/schematics/angular/ng-new/index.ts

接受 enableIvy 参数，如下所示：

```
const applicationOptions: ApplicationOptions = {
  projectRoot: '',
  name: options.name,
  enableIvy: options.enableIvy,
  ...
  ..
};
```

如果存在的话，tsconfig.app.json 的模板会对 enableIvy 作出相应处理：

- <ANGULAR-CLI-MASTER>/packages/schematics/angular/application/files/tsconfig.app.json.template

做了如下处理：

```
<%\% if (enableIvy)  \%%>,
  "angularCompilerOptions": {
    "enableIvy": true
  }<%\%  \%%>
```

当启用 enableIvy 时，webpack 的 ngtools 功能对引导代码的配置略有不同。在下面文件：

- <ANGULAR-CLI-MASTER>/packages/ngtools/webpack/src/transformers/replace_bootstrap.ts

我们可以看到：

```
export function replaceBootstrap(
  ..,
  enableIvy?: boolean
): ts.TransformerFactory<ts.SourceFile> {
  ..
  if (!enableIvy) {
    className += 'NgFactory';
    modulePath += '.ngfactory';
    bootstrapIdentifier = 'bootstrapModuleFactory';
  }
  ..
}
```

所以当 enableIvy 不存在时，factory 构建产物的名称是不同的。从哪里调用 replaceBootstrap 呢？我们可以查看：

- <ANGULAR-CLI-MASTER>/packages/ngtools/webpack/src/angular_compiler_plugin.ts

我们看到 _maketransformers 方法如下：

```
private _makeTransformers() {
  ..
  if (this._platformTransformers !== null) {
    this._transformers.push(...this._platformTransformers);
  } else {
    if (this._platform === PLATFORM.Browser) {
      ..
      if (!this._JitMode) {
        // Replace bootstrap in browser AOT.
        this._transformers.push(
          replaceBootstrap(
            isAppPath,
            getEntryModule,
            getTypeChecker,
            !!this._compilerOptions.enableIvy
          )
        );
      }
    } else if (this._platform === PLATFORM.Server) {
      ..
    }
  } ..
}
```

我们还看到 Ivy 用于 _processlazyroutes：

```
// Process the lazy routes discovered, adding then to _lazyRoutes.
// TODO: find a way to remove lazy routes that don't exist anymore.
// This will require a registry of known references to a lazy route,
// removing it when no
// module references it anymore.
private _processLazyRoutes(discoveredLazyRoutes: LazyRouteMap) {
  Object.keys(discoveredLazyRoutes).forEach((lazyRouteKey) => {
    ..
    if (
      this._JitMode ||
      // When using Ivy and not using allowEmptyCodegenFiles,
      // factories are not generated.
      (this._compilerOptions.enableIvy &&
        !this._compilerOptions.allowEmptyCodegenFiles)
    ) {
      modulePath = lazyRouteTSFile;
      moduleKey = `${lazyRouteModule}${moduleName ? '#' + moduleName : ''}`;
    } else {
      ..
    }
  });
```

```
}
```

我们还看到 Ivy 对 _createorupdateprogram 的影响：

```
private async _createOrUpdateProgram() {
  ..
  if (!this.entryModule && !this._compilerOptions.enableIvy) {
    this._warnings.push(
      'Lazy routes discovery is not enabled. ' +
        'Because there is neither an entryModule nor a ' +
        'statically analyzable bootstrap code in the main file.'
    );
  }
  ..
}
```

enableIvy 对 Angular CLI 生成代码的影响

为了查看 "enableIvy" 选项对代码生成的不同，我们将创建两个新项目——一个启用，另一个不启用 "--enableIvy" 选项。为了节省时间，我们将使用 "--skipInstall" 选项，这意味着 npm install 不会下载依赖项。

```
ng new render3 --enableIvy --skipInstall
ng new render2 --skipInstall
```

在 render2 代码搜索不到 "ivy"，这符合预期。在 render3 代码中能搜到到 2 个 "ivy"。在 package.json 中，"scripts" 有额外的字段：

```
"scripts": {
  "postinstall": "ivy-ngcc",
  ..
},
```

在 tsconfig.app.json 中 "angularCompilerOptions"，有如下字段：

```
"angularCompilerOptions": {
  "enableIvy": true
}
```

现在我们已经看到 Angular CLI 是如何添加 enableIvy，我们已准备好继续探索 Compiler CLI 如何检测和对此作出反应。

Compiler-CLI 包中的 Render3

就像其名字一样，Compiler CLI 是命令行接口和 API（一个库）的一个集合。它有三个应用，我们可以看到是在定义：

- <ANGULAR-MASTER>/packages/compiler-cli/package.json

如下，列出了他们的入口点：

```
  "bin": {
    "ngc": "./src/main.js",
    "ivy-ngcc": "./src/ngcc/main-ngcc.js",
    "ng-xi18n": "./src/extract_i18n.js"
  },
```

Ngc 是主要的 Angular 模板编译器，ivy-ngcc 是 Angular 兼容性编译器（Angular Compatibility Compiler），ng-xi18n 用于国际化。

作为一个库，Compiler CLI 几乎导出了来自 Compiler 包中的类型，如 index.ts 文件所示，以及一些在 Compiler CLI 内定义的有用的诊断类型。其余的命名导出则是 Ivy-specific，但它们的内部结构受到 Ivy 使用的影响，我们很快就会看到。

在 src 根目录中，main.ts 和 perform_compile.ts 中有对 Ivy 的引用。

main.ts 有如下内容：

```
function createEmitCallback(
  options: api.CompilerOptions
): api.TsEmitCallback | undefined {
  const transformDecorators =
    !options.enableIvy && options.annotationsAs !== 'decorators';
  ..
}
```

和一个详细的错误报告：

```
function reportErrorsAndExit(
  allDiagnostics: Diagnostics,
  options?: api.CompilerOptions,
  consoleError: (s: string) => void = console.error
): number {
  const errorsAndWarnings = filterErrorsAndWarnings(allDiagnostics);
  if (errorsAndWarnings.length) {
    const formatHost = getFormatDiagnosticsHost(options);
    if (options && options.enableIvy === true) {
      const ngDiagnostics = errorsAndWarnings.filter(api.isNgDiagnostic);
      const tsDiagnostics = errorsAndWarnings.filter(api.isTsDiagnostic);
      consoleError(
        replaceTsWithNgInErrors(
          ts.formatDiagnosticsWithColorAndContext(tsDiagnostics, formatHost)
        )
      );
      consoleError(formatDiagnostics(ngDiagnostics, formatHost));
    } else {
      consoleError(formatDiagnostics(errorsAndWarnings, formatHost));
    }
  }
  return exitCodeFromResult(allDiagnostics);
}
```

perform_compile.ts 有如下内容：

```
export function createNgCompilerOptions(
  basePath: string,
  config: any,
  tsOptions: ts.CompilerOptions
): api.CompilerOptions {
  // enableIvy `ngtsc` is an alias for `true`.
  if (
    config.angularCompilerOptions &&
    config.angularCompilerOptions.enableIvy === 'ngtsc'
  ) {
    config.angularCompilerOptions.enableIvy = true;
  }
  return {
    ...tsOptions,
    ...config.angularCompilerOptions,
    genDir: basePath,
    basePath,
  };
}
```

CompilerOptions 接口定义在 transformers 子目录：

- <ANGULAR-MASTER>/packages/compiler-cli/src/transformers/api.ts

我们看到它定义了 CompilerOptions 接口，它继承自 ts.CompilerOptions，有一些 Angular 特殊字段。

```
export interface CompilerOptions extends ts.CompilerOptions {
  // NOTE: These comments and aio/content/guides/aot-compiler.md
  // should be kept in sync.
  ..
}
```

尽管有关于 aot-compiler.md 的注释，但 Markdown 文件实际上没有提及 Ivy。api.ts 中的 CompilerOptions 接口确实包含以下附加内容：

```
  /**
   * Tells the compiler to generate definitions using the Render3 style code
   * generation. This option defaults to `false`.
   *
   * Not all features are supported with this option enabled. It is only
   * supported for experimentation and testing of Render3 style code
   * generation.
   * Acceptable values are as follows:
   *
   * `false` - run ngc normally
   * `true` - run the ngtsc compiler instead of the normal ngc compiler
```

```
   * `ngtsc` - alias for `true`
   * `tsc` - behave like plain tsc as much as possible
   *         (used for testing JIT code)
   *
   * @publicApi
   */
  enableIvy?: boolean | 'ngtsc' | 'tsc';
}
```

所以我们看到除了正常的 tsc 编译器，还有一个新的 ngtsc 编译器，这个开关是用来选择一个的。我们很快就会探索 ngtsc 是如何工作的。

api.ts 文件还定义了 EmitFlags，这是我们感兴趣的（注意，default" 包含了代码生成）：

```
export enum EmitFlags {
  DTS = 1 << 0,
  JS = 1 << 1,
  Metadata = 1 << 2,
  I18nBundle = 1 << 3,
  Codegen = 1 << 4,
  Default = DTS | JS | Codegen,
  All = DTS | JS | Metadata | I18nBundle | Codegen,
}
```

tsc_pass_through.ts 文件定义了程序 API 的实现：

```
import { ivySwitchTransform } from '../ngtsc/switch';

/**
 * An implementation of the `Program` API which behaves similarly to plain `
 * tsc`. The only Angular specific behavior included in this `Program` is the
 * operation of the Ivy switch to turn on render3 behavior. This allows `ngc`
 * to behave like `tsc` in cases where JIT code needs to be tested.
 */
export class TscPassThroughProgram implements api.Program {
  ...
  emit(opts?: {
    ..
  }): ts.EmitResult {
    const emitCallback = (opts && opts.emitCallback) || defaultEmitCallback;
    const emitResult = emitCallback({
      ..
      customTransformers: { before: [ivySwitchTransform] },
    });
    return emitResult;
  }
}
```

我们看到 Ivy 以多种方式影响 program.ts 文件。

- <ANGULAR-MASTER>/packages/compiler-cli/src/transformers/program.ts

命令行选项通过 getaotCompilerOptions() 提取：

```
// Compute the AotCompiler options
function getAotCompilerOptions(options: CompilerOptions): AotCompilerOptions {
  ..
  return {
    ..
    enableIvy: options.enableIvy,
  };
}
```

Metadata 预期将是小写的：

```
// Fields to lower within metadata in render2 mode.
const LOWER_FIELDS =
                ['useValue', 'useFactory', 'data', 'id', 'loadChildren'];
// Fields to lower within metadata in render3 mode.
const R3_LOWER_FIELDS = [...LOWER_FIELDS, 'providers', 'imports', 'exports'];

class AngularCompilerProgram implements Program {
  constructor(
  ...
    this.loweringMetadataTransform =
                new LowerMetadataTransform(
                        options.enableIvy ? R3_LOWER_FIELDS : LOWER_FIELDS);
```

在 AngularCompilerProgram 中，我们还看到了对 Render3 使用的具象化装饰器：

```
  R3_REIFIED_DECORATORS = [
    'Component',
    'Directive',
    'Injectable',
    'NgModule',
    'Pipe',
  ];

  private get reifiedDecorators(): Set<StaticSymbol> {
    if (!this._reifiedDecorators) {
      const reflector = this.compiler.reflector;
      this._reifiedDecorators = new Set(
        R3_REIFIED_DECORATORS.map((name) =>
          reflector.findDeclaration('@angular/core', name)
        )
      );
    }
    return this._reifiedDecorators;
  }
```

emit 方法可以防止被 Ivy 无意中调用：

```
  emit(
    parameters: {
      emitFlags?: EmitFlags;
      cancellationToken?: ts.CancellationToken;
      customTransformers?: CustomTransformers;
      emitCallback?: TsEmitCallback;
      mergeEmitResultsCallback?: TsMergeEmitResultsCallback;
    } = {}
  ): ts.EmitResult {
    if (this.options.enableIvy) {
      throw new Error('Cannot run legacy compiler in ngtsc mode');
    }
    return this._emitRender2(parameters);
  }
```

createProgram 函数是决定使用哪个编译程序的地方；这受 enableIvy 值的影响：

```
export function createProgram({
  rootNames,
  options,
  host,
  oldProgram,
}: {
  rootNames: ReadonlyArray<string>;
  options: CompilerOptions;
  host: CompilerHost;
  oldProgram?: Program;
}): Program {
  if (options.enableIvy === true) {
    return new NgtscProgram(rootNames, options, host, oldProgram);
  } else if (options.enableIvy === 'tsc') {
    return new TscPassThroughProgram(rootNames, options, host, oldProgram);
  }
  return new AngularCompilerProgram(rootNames, options, host, oldProgram);
}
```

在离开 program.ts 之前，我们想提一下另外两个函数。该文件还包含 `defaultEmitCallback`：

```
const defaultEmitCallback: TsEmitCallback = ({
  program,
  targetSourceFile,
  writeFile,
  cancellationToken,
  emitOnlyDtsFiles,
  customTransformers,
}) =>
  1 program.emit(
    targetSourceFile,
```

```
    writeFile,
    cancellationToken,
    emitOnlyDtsFiles,
    customTransformers
  );
```

在 1 处我们可以看到编译器实际上是被一组通过参数传入的自定义转换器初始化的。

另一个重要的辅助函数是 `calculateTransforms()`，定义如下：

```
  private calculateTransforms(
    genFiles: Map<string, GeneratedFile> | undefined,
    partialModules: PartialModule[] | undefined,
    stripDecorators: Set<StaticSymbol> | undefined,
    customTransformers?: CustomTransformers
  ): ts.CustomTransformers {
    ...
    1 if (partialModules) {
      beforeTs.push(getAngularClassTransformerFactory(partialModules));
      ..
    } ..
  }
```

我们在 1 处看到这些 partial modules 是如何处理的。特别是，我们看到了新的 `getAngularClassTransformerFactory` 函数的使用。它定义在：

- <ANGULAR-MASTER>/packages/compiler-cli/src/transformers/r3_transform.ts

如下：

```
/**
 * Returns a transformer that adds the requested static methods
 * specified by modules.
 */
export function getAngularClassTransformerFactory(
  modules: PartialModule[]
): TransformerFactory {
  if (modules.length === 0) {
    // If no modules are specified, just return an identity transform.
    return () => (sf) => sf;
  }
  const moduleMap = new Map(
    modules.map<[string, PartialModule]>((m) => [m.fileName, m])
  );
  return function (context: ts.TransformationContext) {
    return function (sourceFile: ts.SourceFile): ts.SourceFile {
      const module = moduleMap.get(sourceFile.fileName);
      if (module) {
        const [newSourceFile] = updateSourceFile(sourceFile, module, context);
        return newSourceFile;
```

```
      }
      return sourceFile;
    };
  };
}
```

两个重要的类型也在 r_transform.ts 中定义，用以描述 Transformer 和 TransformerFactory：

```
export type Transformer = (sourceFile: ts.SourceFile) => ts.SourceFile;
export type TransformerFactory = (
  context: ts.TransformationContext
) => Transformer;
```

PartialModuleMetadataTransformer 定义在：

- <ANGULAR-MASTER>/packages/compiler-cli/src/transformers/r3_metadata_transform.ts

如下：

```
export class PartialModuleMetadataTransformer implements MetadataTransformer {
  private moduleMap: Map<string, PartialModule>;

  constructor(modules: PartialModule[]) {
    this.moduleMap = new Map(
      modules.map<[string, PartialModule]>((m) => [m.fileName, m])
    );
  }

  start(sourceFile: ts.SourceFile): ValueTransform | undefined {
    const partialModule = this.moduleMap.get(sourceFile.fileName);
    if (partialModule) {
      const classMap = new Map<string, ClassStmt>(
        partialModule.statements
          .filter(isClassStmt)
          .map<[string, ClassStmt]>((s) => [s.name, s])
      );
      if (classMap.size > 0) {
        return (value: MetadataValue, node: ts.Node): MetadataValue => {
          // For class metadata that is going to be transformed to have a
          // static method ensure the metadata contains a
          // static declaration the new static method.
          if (
            isClassMetadata(value) &&
            node.kind === ts.SyntaxKind.ClassDeclaration
          ) {
            const classDeclaration = node as ts.ClassDeclaration;
            if (classDeclaration.name) {
              const partialClass = classMap.get(classDeclaration.name.text);
              if (partialClass) {
```

```
                for (const field of partialClass.fields) {
                  if (
                    field.name &&
                    field.modifiers &&
                    field.modifiers.some(
                      (modifier) => modifier === StmtModifier.Static
                    )
                  ) {
                    value.statics = {
                      ...(value.statics || {}),
                      [field.name]: {},
                    };
                  }
                }
              }
            }
          }
          return value;
        };
      }
    }
  }
}
```

现在我们准备回过头去看一下 Compiler-CLI 的 program.ts 文件：

```
  private _emitRender3({
    1 emitFlags = EmitFlags.Default,
    cancellationToken,
    customTransformers,
    2 emitCallback = defaultEmitCallback,
  }: {
    emitFlags?: EmitFlags;
    cancellationToken?: ts.CancellationToken;
    customTransformers?: CustomTransformers;
    emitCallback?: TsEmitCallback; 3
  } = {}): ts.EmitResult {
    const emitStart = Date.now();

    // .. Check emitFlags
    // .. Set up code to emit partical modules
    // .. Set up code for file writing (not shown)
    // .. Build list of custom transformers
    // .. Make emitResult
    return emitResult;
  }
```

这需要一系列输入参数（注意 `emitFlags` 1 和 `emitCallback` 2），然后返回 `ts.EmitResult` 3 的实例。`emitFlags` 说明需要 emitted 什么——如果没有，那么我们立即返回：

```
if (
  (emitFlags &
    (EmitFlags.JS | EmitFlags.DTS | EmitFlags.Metadata | EmitFlags.Codegen)) ===
  0
) {
  return { emitSkipped: true, diagnostics: [], emittedFiles: [] };
}
```

partial modules 将被 Angular 编译器包中的 `emitAllPartialModules` 的这个重要调用进行处理（我们将很快对此进行详细研究）：

```
const modules = this.compiler.emitAllPartialModules(this.analyzedModules);
```

`this.analyzedModules` getter 定义如下：

```
  private get analyzedModules(): NgAnalyzedModules {
    if (!this._analyzedModules) {
      this.initSync();
    }
    return this._analyzedModules!; 1
  }
```

注意返回位置 1 处的!：这是 TypeScript 非空断言运算符，这是一个新特性，解释如下：

`_AnalyzedModules` 字段之前初始化为：

```
  private _analyzedModules: NgAnalyzedModules | undefined;
```

回到我们对 `_emitRender3` 的讨论——在调用 `this.compiler.emitAllPartialModules` 以 emit 模块之后，设置了 `writeTSFile` 和 `emitOnlyDtsFiles` 常量：

```
const writeTsFile: ts.WriteFileCallback = (
  outFileName,
  outData,
  writeByteOrderMark,
  onError?,
  sourceFiles?
) => {
  const sourceFile =
    sourceFiles && sourceFiles.length == 1 ? sourceFiles[0] : null;
  let genFile: GeneratedFile | undefined;
  this.writeFile(
    outFileName,
    outData,
    writeByteOrderMark,
    onError,
    undefined,
    sourceFiles
  );
};
```

```

const emitOnlyDtsFiles =
  (emitFlags & (EmitFlags.DTS | EmitFlags.JS)) == EmitFlags.DTS;
```

然后配置自定义转换器（请注意 partial modules 参数）：

```
const tsCustomTansformers = this.calculateTransforms(
  /* genFiles */ undefined,
  /* partialModules */ modules,
  customTransformers
);
```

最后，emitResult 像如下这样设置（注意那里的 customTransformers）然后返回：

```
const emitResult = emitCallback({
  program: this.tsProgram,
  host: this.host,
  options: this.options,
  writeFile: writeTsFile,
  emitOnlyDtsFiles,
  customTransformers: tsCustomTansformers,
});

return emitResult;
```

我们应该在 program.ts 中简要提及 _emitRender3 和 _emitRender2 之间的区别。与 _emitRender3 相比，后者是一个更大的函数：

```
  private _emitRender2({
    emitFlags = EmitFlags.Default,
    cancellationToken,
    customTransformers,
    emitCallback = defaultEmitCallback,
  }: {
    emitFlags?: EmitFlags;
    cancellationToken?: ts.CancellationToken;
    customTransformers?: CustomTransformers;
    emitCallback?: TsEmitCallback;
  } = {}): ts.EmitResult {
    ..
    let { genFiles, genDiags } = this.generateFilesForEmit(emitFlags);
    ..
  }
```

它还使用了一个不同的辅助函数 generateFilesForEmit 来对 Angular Compiler 进行不同的调用。所以重要的是，根据我们使用的渲染引擎，我们有两个单独的调用路径进入 Angular Compiler 包：

```
  private generateFilesForEmit(emitFlags: EmitFlags): {
    genFiles: GeneratedFile[];
    genDiags: ts.Diagnostic[];
```

```
  } {
    ..
    let genFiles = this.compiler
      .emitAllImpls(this.analyzedModules)
      .filter((genFile) => isInRootDir(genFile.genFileUrl, this.options));

    return { genFiles, genDiags: [] };
  }
```

此处调用的 ivy-ngcc 工具是来自 Complier-CLI 的 Angular Compatibility Compiler[1]，其描述在：

- <ANGULAR-MASTER>/packages/compiler-cli/src/ngcc

如下：

> 此编译器将使用 ngc 编译的 node_modules 转换为使用 ngtsc 编译的 node_modules。此转换将允许 Ivy 渲染引擎使用此类“遗留”包。

这里提到了 ngcc：

- https://next.angular.io/guide/ivy#ngcc

在后面探索 Compiler-Cli 时，我们会看看 ngcc。

Compiler 包中的 Render3

在 Angular Compiler 包内：

- <ANGULAR_MASTER>/packages/compiler

Render3 特性主要在四个文件中。他们是：

- <ANGULAR-MASTER>/packages/compiler/src/aot/partial_module.ts
- <ANGULAR-MASTER>/packages/compiler/src/aot/compiler.ts
- <ANGULAR-MASTER>/packages/compiler/src/render3/r3_identifiers.ts
- <ANGULAR-MASTER>/packages/compiler/src/render3/r3_view_compiler.ts

让我们从 partial_module.ts 开始。它只有几行，用以描述一个 partial module 的类型：

```
import * as o from '../output/output_ast';

export interface PartialModule {
  fileName: string;
  statements: o.Statement[];
}
```

我们从之前的 Compiler-CLI 中看到，它调用了 Compiler 包的 `emitAllPartialModules`。这可以在 src/compiler.ts 文件中找到，因此从 src/compiler.ts 的这一行导出（是的：相同的文件名，不同的目录）：

[1] 译者注：简称 ngcc

```
export * from './aot/compiler';
```

它被定义为：

```
emitAllPartialModules({
  ngModuleByPipeOrDirective,
  files,
}: NgAnalyzedModules): PartialModule[] {
  // Using reduce like this is a select
  // many pattern (where map is a select pattern)
  return files.reduce<PartialModule[]>((r, file) => {
    r.push(
      ...this._emitPartialModule(
        file.fileName,
        ngModuleByPipeOrDirective,
        file.directives,
        file.pipes,
        file.ngModules,
        file.injectables
      )
    );
    return r;
  }, []);
}
```

它调用内部的 _emitPartialModule 方法：

```
private _emitPartialModule(
  fileName: string,
  ngModuleByPipeOrDirective: Map<StaticSymbol, CompileNgModuleMetadata>,
  directives: StaticSymbol[],
  pipes: StaticSymbol[],
  ngModules: CompileNgModuleMetadata[],
  injectables: StaticSymbol[]
): PartialModule[] {
  const classes: o.ClassStmt[] = [];
  const context = this._createOutputContext(fileName);
  ..
}
```

在初始化上下文信息后，遍历 1 指令数组，如果找到一个组件 2，则调用 compileIvyComponent 3，否则调用 compileIvyDirective 4：

```
// Process all components and directives
1 directives.forEach((directiveType) => {
  const directiveMetadata =
    this._metadataResolver.getDirectiveMetadata(directiveType);

  if (directiveMetadata.isComponent) {
```

```
      ..
      const { template: parsedTemplate } = this._parseTemplate(
        directiveMetadata,
        module,
        module.transitiveModule.directives
      );
      2 compileIvyComponent(
        context,
        directiveMetadata,
        parsedTemplate,
        this._reflector
      );
    } else {
      3 compileIvyDirective(context, directiveMetadata, this._reflector);
    }
  });

  if (context.statements) {
    return [
      {
        fileName,
        statements: [...context.constantPool.statements, ...context.statements],
      },
    ];
  }
  return [];
```

我们注意到文件顶部的导入：

```
import {
  compileComponent as compileIvyComponent,
  compileDirective as compileIvyDirective,
} from '../render3/r3_view_compiler';
```

所以在 src/render3/r3_view_compiler.ts 让我们跟踪 `compileComponent` 和 `compileDirective`。

Compiler 包的 src 目录中的 render3 子目录 是 Render3 的新目录。它只包含两个文件，r3_view_compiler 和 r3_identifiers.ts。r3_identifiers.ts 通过这一行被导入到 r3_view_compiler.ts：

```
import { Identifiers as R3 } from './r3_identifiers';
```

所以在 r3_view_compiler.ts 的任何地方，我们看到 “R3” 被用于命名（超过 50 次），这意味着 r3_identifiers.ts 中的某些东西正在被使用。r3_identifiers.ts 未被编译器包中的任何其他地方引用。

r3_identifier.ts 包含用于各种指令的一长串外部引用标识符。这是一个例子（注意 “o” 是从 output_ast.ts 导入的）：

```
import * as o from '../output/output_ast';
const CORE = '@angular/core';
export class Identifiers {
  /* Methods */
```

```
  static NEW_METHOD = 'n';
  static HOST_BINDING_METHOD = 'h';

  /* Instructions */
  static createElement: o.ExternalReference = { name: 'eE', moduleName: CORE };
  static elementEnd: o.ExternalReference = { name: 'ee', moduleName: CORE };
  static text: o.ExternalReference = { name: 'eT', moduleName: CORE };
  static bind: o.ExternalReference = { name: 'eb', moduleName: CORE };
  static bind1: o.ExternalReference = { name: 'eb1', moduleName: CORE };
  static bind2: o.ExternalReference = { name: 'eb2', moduleName: CORE };
  static projection: o.ExternalReference = { name: 'eP', moduleName: CORE };
  static projectionDef: o.ExternalReference = { name: 'epD', moduleName: CORE };
  static injectElementRef: o.ExternalReference = {
    name: 'einjectElementRef',
    moduleName: CORE,
  };
  static injectTemplateRef: o.ExternalReference = {
    name: 'einjectTemplateRef',
    moduleName: CORE,
  };
  static defineComponent: o.ExternalReference = {
    name: 'edefineComponent',
    moduleName: CORE,
  };
  ..
}
```

compileDirective 函数实现在 src/render3/r3_view_compiler.ts：

```
export function compileDirective(
  outputCtx: OutputContext,
  directive: CompileDirectiveMetadata,
  reflector: CompileReflector
) {
  const definitionMapValues: {
    key: string;
    quoted: boolean;
    value: o.Expression;
  }[] = [];

  // e.g. 'type: MyDirective`
  definitionMapValues.push({
    key: 'type',
    value: outputCtx.importExpr(directive.type.reference),
    quoted: false,
  });

  // e.g. `factory: () => new MyApp(injectElementRef())`
```

```
  1 const templateFactory = createFactory(directive.type, outputCtx, reflector);
  2 definitionMapValues.push({
    key: 'factory',
    value: templateFactory,
    quoted: false,
  });

  // e.g 'inputs: {a: 'a'}`
  if (Object.getOwnPropertyNames(directive.inputs).length > 0) {
    definitionMapValues.push({
      key: 'inputs',
      quoted: false,
      value: mapToExpression(directive.inputs),
    });
  }

  const className = identifierName(directive.type)!;
  className || error(`Cannot resolver the name of ${directive.type}`);

  // Create the partial class to be merged with the actual class.
  3 outputCtx.statements.push(
    new o.ClassStmt(
      /* name */ className,
      /* parent */ null,
      /* fields */ [
        4 new o.ClassField(
          /* name */ 'ngDirectiveDef',
          /* type */ o.INFERRED_TYPE,
          /* modifiers */ [o.StmtModifier.Static],
          /* initializer */ 5 o
            .importExpr(R3.defineDirective)
            .callFn([o.literalMap(definitionMapValues)])
        ),
      ],
      /* getters */ [],
      /* constructorMethod */ new o.ClassMethod(null, [], []),
      /* methods */ []
    )
  );
}
```

它首先 1 创建一个模板工厂，然后将其添加到定义映射值数组上 2。然后它使用输出上下文 3 将新的 Class 语句添加到声明的数组上。我们注意到 initializer 设置为 R3.defineDirective 5。

compileComponent 函数（也在 r3_view_compiler.ts中）稍微复杂一些。让我们分阶段来看。它的签名是：

```
export function compileComponent(
  outputCtx: OutputContext,
  component: CompileDirectiveMetadata,
  template: TemplateAst[],
  reflector: CompileReflector
) {
  const definitionMapValues: {
    key: string;
    quoted: boolean;
    value: o.Expression;
  }[] = [];
  // e.g. `type: MyApp`
  definitionMapValues.push({
    key: 'type',
    value: outputCtx.importExpr(component.type.reference),
    quoted: false,
  });
  ...
  // some code regarding selectors (omitted)
  ..
}
```

然后它在定义映射值上设置模板函数表达式：

```
// e.g. `factory: function MyApp_Factory()
// { return new MyApp(injectElementRef()); }`
const templateFactory = 1 createFactory(
  component.type,
  outputCtx,
  reflector
);
definitionMapValues.push({
  key: 'factory',
  value: templateFactory,
  quoted: false,
});
```

我们注意到对 `createFactory` 函数 1 的调用，我们需要稍后跟进。然后它设置一个模板定义构建器并再次将其添加到定义映射值数组中：

```
// e.g. `template: function MyComponent_Template(_ctx, _cm) {...}`
const templateTypeName = component.type.reference.name;
const templateName = templateTypeName ? `${templateTypeName}_Template` : null;

const templateFunctionExpression = 1 new TemplateDefinitionBuilder(
  outputCtx,
  outputCtx.constantPool,
  reflector,
```

```
  CONTEXT_NAME,
  ROOT_SCOPE.nestedScope(),
  0,
  component.template!.ngContentSelectors,
  templateTypeName,
  templateName
)
  2
  .buildTemplateFunction(template, []);
definitionMapValues.push({
  key: 'template',
  value: templateFunctionExpression,
  quoted: false,
});
```

我们注意到 TemplateDefinitionBuilder 类的使用 1，以及对其 buildTemplateFunction 方法的调用 2，我们将很快检查这两个方法。然后它设置类名（并使用！非空断言运算符来确保它不为空）：

```
const className = identifierName(component.type)!;
```

最后它添加了新的 class 语句：

```
// Create the partial class to be merged with the actual class.
outputCtx.statements.push(
  new o.ClassStmt(
    /* name */ className,
    /* parent */ null,
    /* fields */ [
      new o.ClassField(
        /* name */ 'ngComponentDef',
        /* type */ o.INFERRED_TYPE,
        /* modifiers */ [o.StmtModifier.Static],
        /* initializer */ 1 o
          .importExpr(R3.defineComponent)
          .callFn([o.literalMap(definitionMapValues)])
      ),
    ],
    /* getters */ [],
    /* constructorMethod */ new o.ClassMethod(null, [], []),
    /* methods */ []
  )
);
```

我们注意到 initializer 设置为 R3.defineComponent 1。

createFactory 函数定义如下：

```
function createFactory(
  type: CompileTypeMetadata,
  outputCtx: OutputContext,
```

```
  reflector: CompileReflector
): o.FunctionExpr {
  let args: o.Expression[] = [];
  ..
}
```

它首先处理了三个反射器：

```
const elementRef = reflector.resolveExternalReference(Identifiers.ElementRef);
const templateRef = reflector.resolveExternalReference(Identifiers.TemplateRef);
const viewContainerRef = reflector.resolveExternalReference(
  Identifiers.ViewContainerRef
);
```

然后它循环遍历 type.diDeps 依赖项，并根据 token ref 添加相关的导入表达式：

```
  for (let dependency of type.diDeps) {
    if (dependency.isValue) {
      unsupported('value dependencies');
    }
    if (dependency.isHost) {
      unsupported('host dependencies');
    }
    const token = dependency.token;
    if (token) {
      const tokenRef = tokenReference(token);
      if (tokenRef === elementRef) {
        args.push(o.importExpr(R3.injectElementRef).callFn([]));
      } else if (tokenRef === templateRef) {
        args.push(o.importExpr(R3.injectTemplateRef).callFn([]));
      } else if (tokenRef === viewContainerRef) {
        args.push(o.importExpr(R3.injectViewContainerRef).callFn([]));
      } else {
        const value =
          token.identifier != null
            ? outputCtx.importExpr(tokenRef)
            : o.literal(tokenRef);
        args.push(o.importExpr(R3.inject).callFn([value]));
      }
    } else {
      unsupported('dependency without a token');
    }
  }

  return o.fn(
    [],
    [
      new o.ReturnStatement(
        new o.InstantiateExpr(outputCtx.importExpr(type.reference), args)
```

```
      ),
    ],
    o.INFERRED_TYPE,
    null,
    type.reference.name ? `${type.reference.name}_Factory` : null
  );
```

TemplateDefinitionBuilder 类（也位于 r3_view_compiler.ts）很大（350 行 +），可以认为是 Render3 编译的核心。它实现了 TemplateAstVisitor 接口。该接口定义在：

- <ANGULAR-MASTER>/packages/compiler/src/template_parser/template_ast.ts

内容如下：

```
// A visitor for {@link TemplateAst} trees that will process each node.
export interface TemplateAstVisitor {
  visit?(ast: TemplateAst, context: any): any;
  visitNgContent(ast: NgContentAst, context: any): any;
  visitEmbeddedTemplate(ast: EmbeddedTemplateAst, context: any): any;
  visitElement(ast: ElementAst, context: any): any;
  visitReference(ast: ReferenceAst, context: any): any;
  visitVariable(ast: VariableAst, context: any): any;
  visitEvent(ast: BoundEventAst, context: any): any;
  visitElementProperty(ast: BoundElementPropertyAst, context: any): any;
  visitAttr(ast: AttrAst, context: any): any;
  visitBoundText(ast: BoundTextAst, context: any): any;
  visitText(ast: TextAst, context: any): any;
  visitDirective(ast: DirectiveAst, context: any): any;
  visitDirectiveProperty(ast: BoundDirectivePropertyAst, context: any): any;
}
```

回到 r3_view_compiler.ts，TemplateDefinitionBuilder 的定义以如下内容开始：

```
class TemplateDefinitionBuilder implements TemplateAstVisitor, LocalResolver {
  constructor(
    private outputCtx: OutputContext,
    private constantPool: ConstantPool,
    private reflector: CompileReflector,
    private contextParameter: string,
    private bindingScope: BindingScope,
    private level = 0,
    private ngContentSelectors: string[],
    private contextName: string | null,
    private templateName: string | null
  ) {}
  ..
}
```

我们在 compileComponent 早期看到了对 buildTemplateFunction 的调用——它具有以下签名：

```
  buildTemplateFunction(
    asts: TemplateAst[],
    variables: VariableAst[]
  ): o.FunctionExpr {
    ..
  }
```

它返回 o.FunctionExpr 的实例。我们注意到文件顶部的导入：

```
import * as o from '../output/output_ast';
```

所以 o.FunctionExpr 意味着 FunctionExpr 类在：

- <ANGULAR-MASTER>/packages/compiler/src/output/output_ast.ts

该类定义如下：

```
export class FunctionExpr extends Expression {
  constructor(
    public params: FnParam[],
    public statements: Statement[],
    type?: Type | null,
    sourceSpan?: ParseSourceSpan | null,
    public name?: string | null
  ) {
    super(type, sourceSpan);
  }
  ...
}
```

当我们查看 output_ast.ts 时，我们看到了这个 fn 函数：

```
export function fn(
  params: FnParam[],
  body: Statement[],
  type?: Type | null,
  sourceSpan?: ParseSourceSpan | null,
  name?: string | null
): FunctionExpr {
  return new FunctionExpr(params, body, type, sourceSpan, name);
}
```

它只是从给定的参数中生成一个 FunctionExpr。

src/template_parser/template_ast.ts 中一个有趣的函数是 templateVisitAll：

```
/**
 * Visit every node in a list of {@link TemplateAst}s with the given
 * {@link TemplateAstVisitor}.
 */
export function templateVisitAll(
  visitor: TemplateAstVisitor,
```

```
  asts: TemplateAst[],
  context: any = null
): any[] {
  const result: any[] = [];
  const visit = visitor.visit
    ? (ast: TemplateAst) =>
        visitor.visit!(ast, context) || ast.visit(visitor, context)
    : (ast: TemplateAst) => ast.visit(visitor, context);
  asts.forEach((ast) => {
    const astResult = visit(ast);
    if (astResult) {
      result.push(astResult);
    }
  });
  return result;
}
```

现在让我们回到 r3_view_compiler.ts 中 TemplateDefinitionBuilder 的极其重要的 buildTemplateFunction 方法——它的定义总结如下：

```
  buildTemplateFunction(
    asts: TemplateAst[],
    variables: VariableAst[]
  ): o.FunctionExpr {
    // Create variable bindings
    ...
    // Collect content projections
    ...
    1 templateVisitAll(this, asts);
    ..
    2 return o.fn(
      [
        new o.FnParam(this.contextParameter, null),
        3
        new o.FnParam(CREATION_MODE_FLAG, o.BOOL_TYPE),
      ],
      [
        4 // Temporary variable declarations (i.e. let _t: any;)
        ...this._prefix,

        // Creating mode (i.e. if (cm) { ... })
        ...creationMode,

        // Binding mode (i.e. ep(...))
        ...this._bindingMode,

        // Host mode (i.e. Comp.h(...))
        ...this._hostMode,
```

```
      // Refresh mode (i.e. Comp.r(...))
      ...this._refreshMode,

      // Nested templates (i.e. function CompTemplate() {})
      ...this._postfix,
    ],
    5 o.INFERRED_TYPE,
    null,
    this.templateName
  );
}
```

我们看到它首先访问模板树 1。然后它返回我们刚刚看到的 `fn` 函数调用 2 的结果，传入三个条目——3 一个 `FnParams` 数组 4、一个语句数组和 `o.INFERRED_TYPE` 5。这里发生的事情是模板树中的每个节点都被访问，并且在适当的情况下，使用正确的 Render3 指令将语句 emit 到输出语句数组。指令函数用于添加如下语句：

```
private instruction(
  statements: o.Statement[],
  span: ParseSourceSpan | null,
  reference: o.ExternalReference,
  ...params: o.Expression[]
) {
  statements.push(
    o.importExpr(reference, null, span).callFn(params, span).toStmt()
  );
}
```

例如，当访问文本节点时，应发出 Render3 文本指令（`R3.text`）。我们在 `visitText` 方法中看到了这种情况：

```
private _creationMode: o.Statement[] = [];

visitText(ast: TextAst) {
  // Text is defined in creation mode only.
  this.instruction(
    this._creationMode,
    ast.sourceSpan,
    R3.text,
    o.literal(this.allocateDataSlot()),
    o.literal(ast.value)
  );
}
```

元素有一个等效的方法 `visitElement`，它稍微复杂一些。在一些 setup 代码之后，它有这个：

```
// Generate the instruction create element instruction
this.instruction(
  this._creationMode,
  ast.sourceSpan,
  R3.createElement,
  ...parameters
);
```

还有一个 visitEmbeddedTemplate 方法，它 emit 许多 Render3 指令：

```
  visitEmbeddedTemplate(ast: EmbeddedTemplateAst) {
    ...
    // e.g. C(1, C1Template)
    this.instruction(
      this._creationMode,
      ast.sourceSpan,
      R3.containerCreate,
      o.literal(templateIndex),
      directivesArray,
      o.variable(templateName)
    );

    // e.g. Cr(1)
    this.instruction(
      this._refreshMode,
      ast.sourceSpan,
      R3.containerRefreshStart,
      o.literal(templateIndex)
    );

    // Generate directives
    this._visitDirectives(
      ast.directives,
      o.variable(this.contextParameter),
      templateIndex,
      directiveIndexMap
    );

    // e.g. cr();
    this.instruction(this._refreshMode, ast.sourceSpan, R3.containerRefreshEnd);

    // Create the template function
    const templateVisitor = new TemplateDefinitionBuilder(
      this.outputCtx,
      this.constantPool,
      this.reflector,
      templateContext,
```

```
      this.bindingScope.nestedScope(),
      this.level + 1,
      this.ngContentSelectors,
      contextName,
      templateName
    );
    const templateFunctionExpr = templateVisitor.buildTemplateFunction(
      ast.children,
      ast.variables
    );
    this._postfix.push(templateFunctionExpr.toDeclStmt(templateName, null));
  }
```

Core 包中的 Render3

API

Render3 没有公共 API。Core 包 包含 Render3（和 Render2）代码。它的 index.ts 文件只是导出 public_api.ts 的内容，而后者又导出了 ./src/core.ts 的内容。

关于公共 API，这有一个与渲染相关的行，导出：

```
export * from './render';
```

./src/render.ts 文件导出没有 Render3 API。它导出 Render2 API，如下：

```
export {
  RenderComponentType,
  Renderer,
  Renderer2,
  RendererFactory2,
  RendererStyleFlags2,
  RendererType2,
  RootRenderer,
} from './render/api';
```

请注意，Render2 只是少于 200 行代码的 ./src/render/api.ts 文件（core/src/render 子目录仅包含该文件）——它定义了上述类型但不包含执行。你可以在这里阅读全文：

- <ANGULAR-MASTER>/packages/core/src/render/api.ts

Render3 确实有私有 API。./src/core.ts 文件包含此行：

```
export * from './core_render3_private_export';
```

./src/core_render3_private_export.ts 文件有这个：

```
export {
  defineComponent as edefineComponent,
  detectChanges as edetectChanges,
  renderComponent as erenderComponent,
```

```
  ComponentType as ɵComponentType,
  C as ɵC,
  E as ɵE,
  L as ɵL,
  T as ɵT,
  V as ɵV,
  b as ɵb,
  b1 as ɵb1,
  b2 as ɵb2,
  b3 as ɵb3,
  b4 as ɵb4,
  b5 as ɵb5,
  b6 as ɵb6,
  b7 as ɵb7,
  b8 as ɵb8,
  bV as ɵbV,
  cR as ɵcR,
  cr as ɵcr,
  e as ɵe,
  p as ɵp,
  s as ɵs,
  t as ɵt,
  v as ɵv,
  r as ɵr,
} from './render3/index';
```

私有 API 适用于其他 Angular，而不是常规使用 Angular 的应用。因此，私有 API 前缀为希腊语 Theta 字符（“ɵ”），与 Angular 内的其他此类私有 API 共同。

许多非常短的类型名称的原因是 Angular 编译器将为你应用的 Angular 模板文件生成大量基于 Render3 的源代码，并且希望它尽可能紧凑，而不需要运行压缩器。通常没有人阅读这个生成的代码，所以需要紧凑而不是可读性。

如果我们查阅：

- <ANGULAR-MASTER>/packages/core/src/Render3/index.ts

我们看到它首先解释命名方案：

```
// Naming scheme:
// - Capital letters are for creating things:
// T(Text), E(Element), D(Directive), V(View),
// C(Container), L(Listener)
// - lower case letters are for binding: b(bind)
// - lower case letters are for binding target:
//     p(property), a(attribute), k(class), s(style), i(input)
// - lower case letters for guarding life cycle hooks: l(lifeCycle)
// - lower case for closing: c(containerEnd), e(elementEnd), v(viewEnd)
```

然后它有很长的指令导出列表，许多有缩写：

```
export {
  QUERY_READ_CONTAINER_REF,
  QUERY_READ_ELEMENT_REF,
  QUERY_READ_FROM_NODE,
  QUERY_READ_TEMPLATE_REF,
  InjectFlags,
  inject,
  injectElementRef,
  injectTemplateRef,
  injectViewContainerRef,
} from './di';
export {
  NO_CHANGE as NC,
  bind as b,
  bind1 as b1,
  bind2 as b2,
  bind3 as b3,
  bind4 as b4,
  bind5 as b5,
  bind6 as b6,
  bind7 as b7,
  bind8 as b8,
  bindV as bV,
  componentRefresh as r,
  container as C,
  containerRefreshStart as cR,
  containerRefreshEnd as cr,
  elementAttribute as a,
  elementClass as k,
  elementEnd as e,
  elementProperty as p,
  elementStart as E,
  elementStyle as s,
  listener as L,
  memory as m,
  projection as P,
  projectionDef as pD,
  text as T,
  textBinding as t,
  embeddedViewStart as V,
  embeddedViewEnd as v,
} from './instructions';
export {
  pipe as Pp,
  pipeBind1 as pb1,
  pipeBind2 as pb2,
```

```
  pipeBind3 as pb3,
  pipeBind4 as pb4,
  pipeBindV as pbV,
} from './pipe';
export { QueryList, query as Q, queryRefresh as qR } from './query';
export {
  objectLiteral1 as o1,
  objectLiteral2 as o2,
  objectLiteral3 as o3,
  objectLiteral4 as o4,
  objectLiteral5 as o5,
  objectLiteral6 as o6,
  objectLiteral7 as o7,
  objectLiteral8 as o8,
} from './object_literal';

export {
  ComponentDef,
  ComponentTemplate,
  ComponentType,
  DirectiveDef,
  DirectiveDefFlags,
  DirectiveType,
  NgOnChangesFeature,
  PublicFeature,
  defineComponent,
  defineDirective,
  definePipe,
};

export {
  createComponentRef,
  detectChanges,
  getHostElement,
  markDirty,
  renderComponent,
};

export { CssSelector } from './interfaces/projection';
```

每个一个或两个字母的导出对应于 .src/Render3/instructions.ts 中的一条指令。在下图中，我们给出了短导出名称和全名，这更清楚地说明了指令的意图。我们已经看到 compiler 和 compielr-cli 包如何使用 Core 的 Render3：

- <ANGULAR-MASTER>/packages/compiler/src/render3
- <ANGULAR-MASTER>/packages/compiler-cli/src/transformers/r3_transform.ts

Router 或 platform 包（使用 Render2）不使用它。

目录结构

Render3 功能的源码树直接包含这些源文件：

- assert.ts
- component.ts
- definition.ts
- di.ts
- hooks.ts
- index.ts
- instructions.ts
- ng_dev_mode.ts
- node_assert.ts
- node_manipulation.ts
- node_selector_matcher.ts
- object_literal.ts
- pipe.ts
- query.ts
- util.ts

以及这些文档：

- perf_notes.md
- TREE_SHAKING.md

一个 interfaces 子目录，包含这些文件：

- container.ts
- definition.ts
- injector.ts
- node.ts
- projection.ts
- query.ts
- renderer.ts
- view.ts

在某些 Render3 代码中，我们看到使用了“L”和“R”前缀。源码中进行了注释：

```
The "L" stands for "Logical" to differentiate between `RNodes` (actual
rendered DOM node) and our logical representation of DOM nodes, `LNodes`.
<ANGULAR-MASTER>/packages/core/src/render3/interfaces/node.ts
```

接口

当试图弄清楚 Render3 如何工作时，一个好的开始就是它的接口。让我们来看看这个：

- <ANGULAR-MASTER>/packages/core/src/render3/interfaces/renderer.ts

有一些简单的辅助接口描述节点，元素和文本节点。该节点定义为有三种方法用以插入，添加和删除一个子节点：

```
/** Subset of API needed for appending elements and text nodes. */
export interface RNode {
  removeChild(oldChild: RNode): void;

  // Insert a child node.
  // Used exclusively for adding View root nodes into ViewAnchor location.
  insertBefore(
    newChild: RNode,
    refChild: RNode | null,
    isViewRoot: boolean
  ): void;

  //Append a child node.
  //Used exclusively for building up DOM which are static (ie not View roots)
  appendChild(newChild: RNode): RNode;
}
```

该元素允许添加和删除侦听器，使用 attributes、properties 和样式配置——它定义为：

```
/**
 * Subset of API needed for writing attributes, properties, and setting up
 * listeners on Element.
 */
export interface RElement extends RNode {
  style: RCssStyleDeclaration;
  classList: RDomTokenList;
  setAttribute(name: string, value: string): void;
  removeAttribute(name: string): void;
  setAttributeNS(
    namespaceURI: string,
    qualifiedName: string,
    value: string
  ): void;
  addEventListener(
    type: string,
    listener: EventListener,
    useCapture?: boolean
  ): void;
  removeEventListener(
    type: string,
    listener?: EventListener,
    options?: boolean
  ): void;
```

```
  setProperty?(name: string, value: any): void;
}
```

文本节点添加了一个 textContent property：

```
export interface RText extends RNode {
  textContent: string | null;
}
```

它有这个工厂代码。请注意，createRenderer 的返回类型是 Render3 1 ——对于 domRendererFactory3 实现 2，这是正常的 DOM document

```
export interface RendererFactory3 {
  createRenderer(
    hostElement: RElement | null,
    rendererType: RendererType2 | null
  ): Renderer3 1;
  begin?(): void;
  end?(): void;
}
export const domRendererFactory3: RendererFactory3 = {
  createRenderer: (
    hostElement: RElement | null,
    rendererType: RendererType2 | null
  ): Renderer3 => {
    return document;
  } 2,
};
```

这是在主浏览器 UI 线程中运行的代码返回到常规 DOM 使用的关键，但允许其他地方的替代方案。Renderer3 是一个类型别名：

```
export type Renderer3 = ObjectOrientedRenderer3 | ProceduralRenderer3;
```

这表示支持的两种渲染器。注释中的粗体文本（4 - 6 行）突出显示了其中第一个的使用场景：

```
/**
 * Object Oriented style of API needed to create elements and text nodes.
 *
 * This is the native browser API style, e.g. operations are methods on
 * individual objects like HTMLElement. With this style, no additional
 * code is needed as a facade (reducing payload size).
 */
export interface ObjectOrientedRenderer3 {
  createElement(tagName: string): RElement;
  createTextNode(data: string): RText;
  querySelector(selectors: string): RElement | null;
}
```

ProceduralRender3 旨在用于 web workers 和服务器端：

```ts
/**
 * Procedural style of API needed to create elements and text nodes.
 *
 * In non-native browser environments (e.g. platforms such as web-workers),
 * this is the facade that enables element manipulation. This also
 * facilitates backwards compatibility with Renderer2.
 */
export interface ProceduralRenderer3 {
  destroy(): void;
  createElement(name: string, namespace?: string | null): RElement;
  createText(value: string): RText;
  destroyNode?: ((node: RNode) => void) | null;
  appendChild(parent: RElement, newChild: RNode): void;
  insertBefore(parent: RNode, newChild: RNode, refChild: RNode | null): void;
  removeChild(parent: RElement, oldChild: RNode): void;
  selectRootElement(selectorOrNode: string | any): RElement;
  setAttribute(
    el: RElement,
    name: string,
    value: string,
    namespace?: string | null
  ): void;
  removeAttribute(el: RElement, name: string, namespace?: string | null): void;
  addClass(el: RElement, name: string): void;
  removeClass(el: RElement, name: string): void;
  setStyle(
    el: RElement,
    style: string,
    value: any,
    flags?: RendererStyleFlags2 | RendererStyleFlags3
  ): void;
  removeStyle(
    el: RElement,
    style: string,
    flags?: RendererStyleFlags2 | RendererStyleFlags3
  ): void;
  setProperty(el: RElement, name: string, value: any): void;
  setValue(node: RText, value: string): void;
  listen(
    target: RNode,
    eventName: string,
    callback: (event: any) => boolean | void
  ): () => void;
}
```

现在让我们看看 view.ts。它包括以下用于处理静态数据的内容：

```
// The static data for an LView (shared between all templates of a
// given type). Stored on the template function as ngPrivateData.
export interface TView {
  data: TData;
  firstTemplatePass: boolean;
  initHooks: HookData | null;
  checkHooks: HookData | null;
  contentHooks: HookData | null;
  contentCheckHooks: HookData | null;
  viewHooks: HookData | null;
  viewCheckHooks: HookData | null;
  destroyHooks: HookData | null;
  objectLiterals: any[] | null;
}
```

TData 定义如下：

```
/**
 * Static data that corresponds to the instance-specific data array on an
 * Lview. Each node's static data is stored in tData at the same index that
 * it's stored in the data array. Each directive's definition is stored here
 * at the same index as its directive instance in the data array. Any nodes
 * that do not have static data store a null value in tData to avoid a
 * sparse array.
 */
export type TData = (TNode | DirectiveDef<any> | null)[];
```

将需要 LViews 或 LContainers 树，因此该类型是层次结构中的一个节点：

```
/** Interface necessary to work with view tree traversal */
export interface LViewOrLContainer {
  next: LView | LContainer | null;
  child?: LView | LContainer | null;
  views?: LViewNode[];
  parent: LView | null;
}
```

LView 存储与处理视图指令相关的信息。每个字段的详细注释（此处未显示）都在 这里。

```
/**
 * `LView` stores all of the information needed to process the instructions
 * as they are invoked from the template. Each embedded view and component
 * view has its own `LView`. When processing a particular view, we set the
 * `currentView` to that `LView`. When that view is done processing, the
 * `currentView` is set back to whatever the original `currentView` was
 * before (the parent `LView`).
 * Keeping separate state for each view facilities view insertion / deletion,
 * so we don't have to edit the data array based on which views are present.
 */
```

```
export interface LView {
  creationMode: boolean;
  readonly parent: LView | null;
  readonly node: LViewNode | LElementNode;
  readonly id: number;
  readonly renderer: Renderer3;
  bindingStartIndex: number | null;
  cleanup: any[] | null;
  lifecycleStage: LifecycleStage;
  child: LView | LContainer | null;
  tail: LView | LContainer | null;
  next: LView | LContainer | null;
  readonly data: any[];
  tView: TView;
  template: ComponentTemplate<{}> | null;
  context: {} | null;
  dynamicViewCount: number;
  queries: LQueries | null;
}
```

container.ts 文件查看 containers，它们是视图和子 containers 的集合。它导出一种类型，TContainer

```
/**
 * The static equivalent of LContainer, used in TContainerNode.
 *
 * The container needs to store static data for each of its embedded views
 * (TViews). Otherwise, nodes in embedded views with the same index as nodes
 * in their parent views will overwrite each other, as they are in
 * the same template.
 *
 * Each index in this array corresponds to the static data for a certain
 * view. So if you had V(0) and V(1) in a container, you might have:
 *
 * [
 *   [{tagName: 'div', attrs: ...}, null],     // V(0) TView
 *   [{tagName: 'button', attrs ...}, null]    // V(1) TView
 * ]
 */
export type TContainer = TView[];
```

和一个接口，LContainer：（注 源文件 包含每个字段的详细注释）：

```
/** The state associated with an LContainer */
export interface LContainer {
  nextIndex: number;
  next: LView | LContainer | null;
  parent: LView | null;
  readonly views: LViewNode[];
  renderParent: LElementNode | null;
```

```
  readonly template: ComponentTemplate<any> | null;
  dynamicViewCount: number;
  queries: LQueries | null;
}
```

query.ts 文件包含 QueryReadType 类：

```
export class QueryReadType<T> {
  private defeatStructuralTyping: any;
}
```

以及 LQuery 接口：

```
/** Used for tracking queries (e.g. ViewChild, ContentChild). */
export interface LQueries {
  child(): LQueries | null;
  addNode(node: LNode): void;
  container(): LQueries | null;
  enterView(newViewIndex: number): LQueries | null;
  removeView(removeIndex: number): void;
  track<T>(
    queryList: QueryList<T>,
    predicate: Type<any> | string[],
    descend?: boolean,
    read?: QueryReadType<T> | Type<T>
  ): void;
}
```

projection.ts 文件定义 LProjection 如下：

```
// Linked list of projected nodes (using the pNextOrParent property).
export interface LProjection {
  head: LElementNode | LTextNode | LContainerNode | null;
  tail: LElementNode | LTextNode | LContainerNode | null;
}
```

injector.ts 文件有这个：

```
export interface LInjector {
  readonly parent: LInjector | null;
  readonly node: LElementNode | LContainerNode;
  bf0: number;
  bf1: number;
  bf2: number;
  bf3: number;
  cbf0: number;
  cbf1: number;
  cbf2: number;
  cbf3: number;
  injector: Injector | null;
```

```
  /** Stores the TemplateRef so subsequent injections of the TemplateRef get
the same instance. */
  templateRef: TemplateRef<any> | null;
  /** Stores the ViewContainerRef so subsequent injections of the
ViewContainerRef get the same
   * instance. */
  viewContainerRef: ViewContainerRef | null;
  /** Stores the ElementRef so subsequent injections of the ElementRef get
the same instance. */
  elementRef: ElementRef | null;
}
```

node.ts 文件很大，包含节点相关类型的层次结构。

根 LNode 定义（缩写）为：

```
/**
 * LNode is an internal data structure which is used for the incremental DOM
 * algorithm. The "L" stands for "Logical" to differentiate between `RNodes`
 * (actual rendered DOM node) and our logical representation of DOM nodes,
 * `Lnodes`. The data structure is optimized for speed and size.
 *
 * In order to be fast, all subtypes of `LNode` should have the same shape.
 * Because size of the `LNode` matters, many fields have multiple roles
 * depending on the `LNode` subtype.
 */
export interface LNode {
  flags: LNodeFlags;
  readonly native: RElement | RText | null | undefined;
  readonly parent: LNode | null;
  child: LNode | null;
  next: LNode | null;
  readonly data: LView | LContainer | LProjection | null;
  readonly view: LView;
  nodeInjector: LInjector | null;
  queries: LQueries | null;
  pNextOrParent: LNode | null;
  tNode: TNode | null;
}
```

其他每种类型都添加了一些额外的字段来表示该节点类型。

9: 表单包

概览

表单模块提供了管理 web 表单的能力。

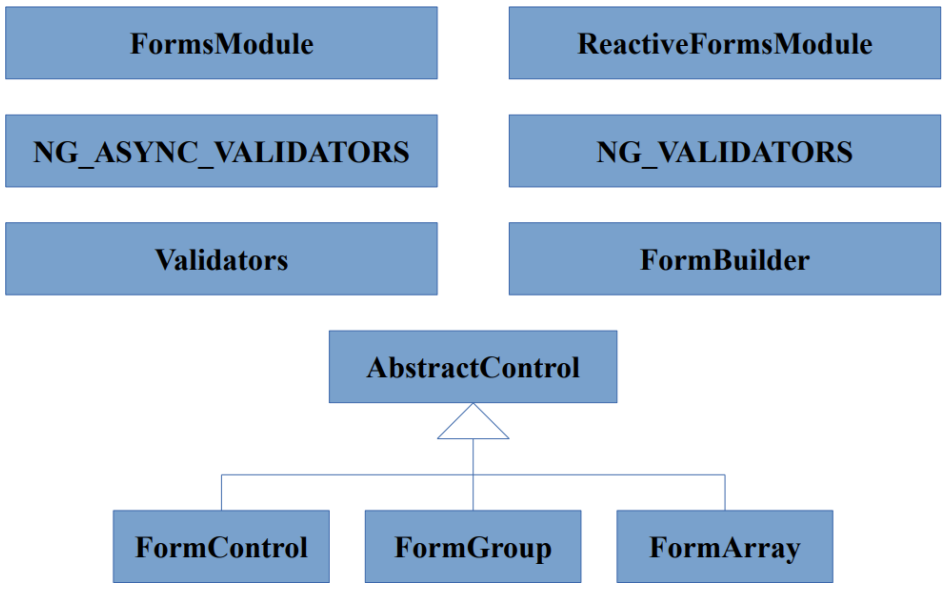
主要提供了以下功能：验证、提交、数据建模、附加指令以及用于动态表单的构建器。

API

@Angular/Forms 暴露的 API 接口大致可以分为四组：main、directives、accessors 和 validator directives。

main 大致结构见图 9.1。

图 9.1: @Angular/Forms API (main)



它的 index.ts 文件内容如下：

```
1 export * from './src/forms';
```

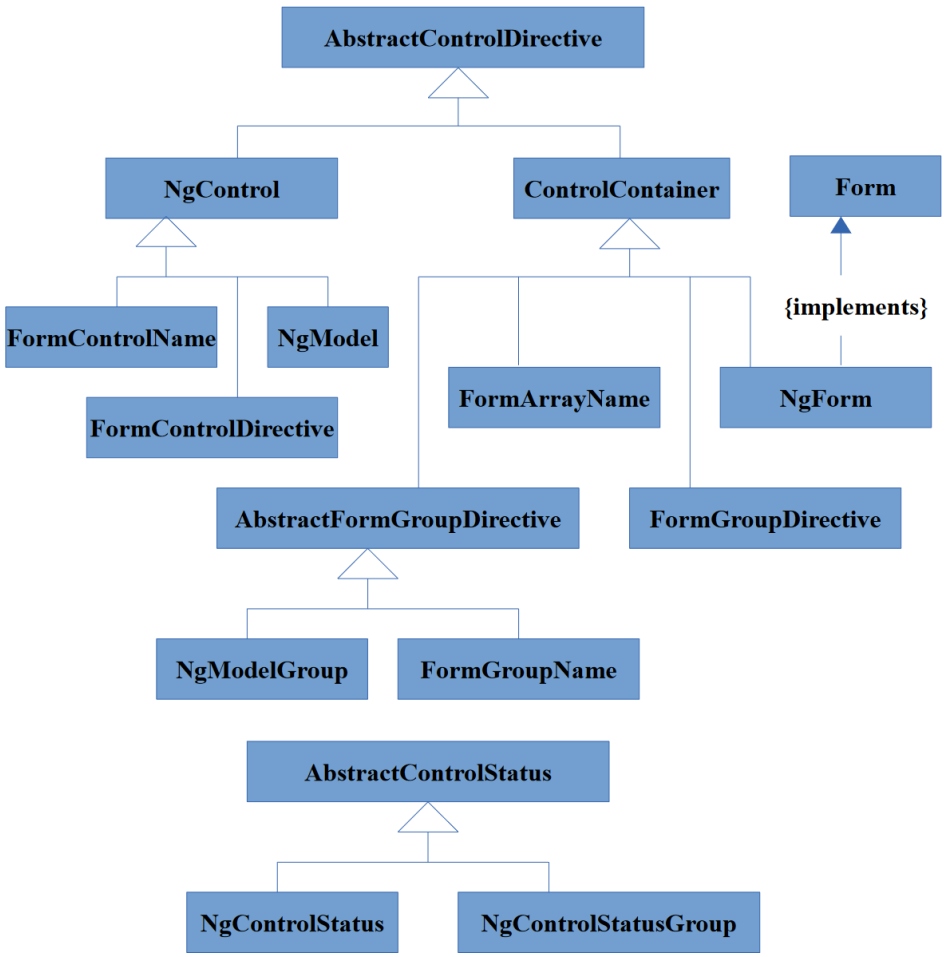
form.ts 中与上述内容相关的导出是：

```
export { FormBuilder } from './form_builder';
export { AbstractControl, FormArray, FormControl, FormGroup } from './model';
export { NG_ASYNC_VALIDATORS, NG_VALIDATORS, Validators } from './validators';
export * from './form_providers';
```

还提供了两个 NgModules，一个是 FormsModule，用于普通表单，另一个是 ReactiveFormsModule，用于响应式表单。表单控件层次结构从根开始，FormControl 用于实际的控件，以及两种控件组合，一种是表单组（固定大小），另一种用于数组（动态大小）

为两种类型的表单提供了一个大的指令继承结构（见图 9.2）。

图 9.2: @Angular/Forms API (directives)



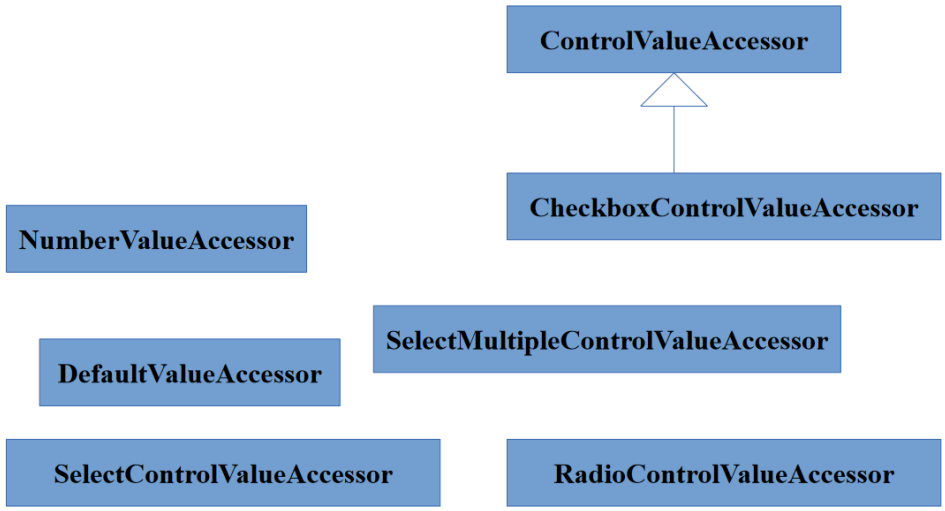
forms.ts 中与指令导出相关的部分是：

```
export { AbstractControlDirective } from './directives/abstract_control_directive';
export { AbstractFormGroupDirective } from './directives/abstract_form_group_directive';
export { ControlContainer } from './directives/control_container';
export { Form } from './directives/form_interface';
export { NgControl } from './directives/ng_control';
export {
  NgControlStatus,
```

```
  NgControlStatusGroup,
} from './directives/ng_control_status';
export { NgForm } from './directives/ng_form';
export { NgModel } from './directives/ng_model';
export { NgModelGroup } from './directives/ng_model_group';
export { FormControlDirective } from './directives/reactive_directives/form_control_directive';
export { FormControlName } from './directives/reactive_directives/form_control_name';
export { FormGroupDirective } from './directives/reactive_directives/form_group_directive';
export { FormArrayName } from './directives/reactive_directives/form_group_name';
export { FormGroupName } from './directives/reactive_directives/form_group_name';
```

访问器继承关系表示见图 9.3。

图 9.3: @Angular/Forms API (accessors)



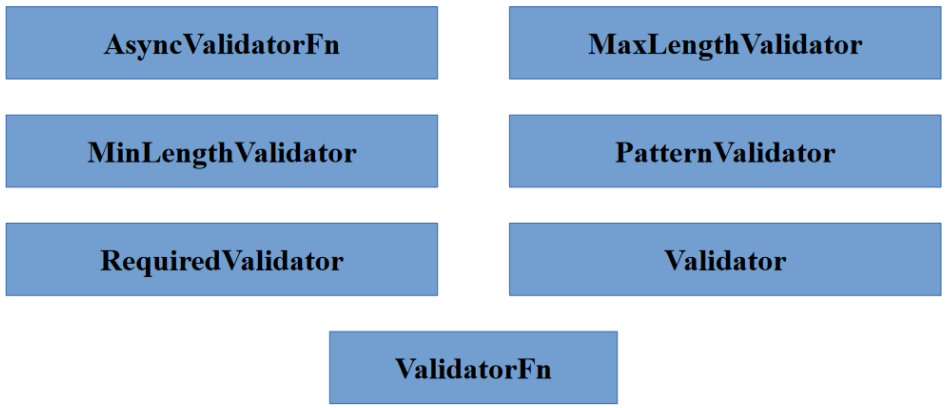
form.ts 中与访问器相关的部分是：

```
export { CheckboxControlValueAccessor } from './directives/checkbox_value_accessor';
export {
  ControlValueAccessor,
  NG_VALUE_ACCESSOR,
} from './directives/control_value_accessor';
export { DefaultValueAccessor } from './directives/default_value_accessor';
export {
  NgSelectOption,
  SelectControlValueAccessor,
} from './directives/select_control_value_accessor';
export { SelectMultipleControlValueAccessor } from
↪  './directives/select_multiple_control_value_accessor';
```

和验证指令相关的部分是（见图 9.4）：

```
export {
  AsyncValidatorFn,
  MaxLengthValidator,
```

图 9.4: @Angular/Forms API (validator directives)




```
  MinLengthValidator,
  PatternValidator,
  RequiredValidator,
  Validator,
  ValidatorFn,
} from './directives/validators';
```

目录结构

Forms 包源码包含如下目录：

- src
- test (unit tests in Jasmine)

请注意，与大多数其他 @Angular 模块不同，Forms 没有测试目录。
Forms 主目录包含以下文件：

- index.ts
- package.json
- rollup.config.js
- tsconfig.json

源码

forms/src

@angular/forms/src 目录包含如下文件：

- directives.ts
- form_builder.ts
- form_providers.ts
- forms.ts

- model.ts
- validators.ts

forms_providers.ts 定义了 FormsModule 和 ReactiveFormsModule 两个 NgModules，内容如下：

```
@NgModule({
  declarations: TEMPLATE_DRIVEN_DIRECTIVES,
  providers: [RadioControlRegistry],
  exports: [InternalFormsSharedModule, TEMPLATE_DRIVEN_DIRECTIVES],
})
export class FormsModule {}
@NgModule({
  declarations: [REACTIVE_DRIVEN_DIRECTIVES],
  providers: [FormBuilder, RadioControlRegistry],
  exports: [InternalFormsSharedModule, REACTIVE_DRIVEN_DIRECTIVES],
})
export class ReactiveFormsModule {}
```

我们注意到 FormsModule 和 ReactiveFormsModule 两者的主要区别是：ReactiveFormsModule 有一个额外的 FormBuilder provider 配置，FormModule 的导出包含了 TEMPLATE_DRIVEN_DIRECTIVES，而 ReactiveFormsModule 的导出则包含 REACTIVE_DRIVEN_DIRECTIVES。

directives.ts 定义如下：

```
export const SHARED_FORM_DIRECTIVES: Type<any>[] = [
  NgSelectOption,
  NgSelectMultipleOption,
  DefaultValueAccessor,
  NumberValueAccessor,
  CheckboxControlValueAccessor,
  SelectControlValueAccessor,
  SelectMultipleControlValueAccessor,
  RadioControlValueAccessor,
  NgControlStatus,
  NgControlStatusGroup,
  RequiredValidator,
  MinLengthValidator,
  MaxLengthValidator,
  PatternValidator,
];
export const TEMPLATE_DRIVEN_DIRECTIVES: Type<any>[] = [
  NgModel,
  NgModelGroup,
  NgForm,
];
export const REACTIVE_DRIVEN_DIRECTIVES: Type<any>[] = [
  FormControlDirective,
  FormGroupDirective,
  FormControlName,
```

```
  FormGroupName,
  FormArrayName,
];
export const FORM_DIRECTIVES: Type<any>[][] = [
  TEMPLATE_DRIVEN_DIRECTIVES,
  SHARED_FORM_DIRECTIVES,
];
export const REACTIVE_FORM_DIRECTIVES: Type<any>[][] = [
  REACTIVE_DRIVEN_DIRECTIVES,
  SHARED_FORM_DIRECTIVES,
];

@NgModule({
  declarations: SHARED_FORM_DIRECTIVES,
  exports: SHARED_FORM_DIRECTIVES,
})
export class InternalFormsSharedModule {}
```

一个比较好的关于如何使用 REACTIVE_FORM_DIRECTIVES 创建动态表单的讨论：

- https://angular.io/docs/ts/latest/cookbook/dynamic-form.html

form_builder.ts 文件定义了可注入的 FormsBuilder 类，可以通过 group()、array() 或 control() 方法动态构建相应的 FormGroup、FormArray 以及 FormControl。方法定义如下：

```
  group(
    controlsConfig: { [key: string]: any },
    extra: { [key: string]: any } = null
  ): FormGroup {
    const controls = this._reduceControls(controlsConfig);
    const validator: ValidatorFn = isPresent(extra)
      ? StringMapWrapper.get(extra, 'validator')
      : null;
    const asyncValidator: AsyncValidatorFn = isPresent(extra)
      ? StringMapWrapper.get(extra, 'asyncValidator')
      : null;
    return new FormGroup(controls, validator, asyncValidator);
  }

  control(
    formState: Object,
    validator: ValidatorFn | ValidatorFn[] = null,
    asyncValidator: AsyncValidatorFn | AsyncValidatorFn[] = null
  ): FormControl {
    return new FormControl(formState, validator, asyncValidator);
  }

  array(
```

```
    controlsConfig: any[],
    validator: ValidatorFn = null,
    asyncValidator: AsyncValidatorFn = null
  ): FormArray {
    var controls = controlsConfig.map((c) => this._createControl(c));
    return new FormArray(controls, validator, asyncValidator);
  }
```

validators.ts 文件首先声明了两个用于依赖注入的 opaque tokens：

```
export const NG_VALIDATORS: OpaqueToken = new OpaqueToken('NgValidators');
export const NG_ASYNC_VALIDATORS: OpaqueToken = new OpaqueToken(
  'NgAsyncValidators'
);
```

它还定义了 Validators 类：

```
// A validator is a function that processes a FormControl or
// collection of controls and returns a map of errors.
//
// A null map means that validation has passed.

export class Validators {
  static required(control: AbstractControl): { [key: string]: boolean } {}
  static minLength(minLength: number): ValidatorFn {}
  static maxLength(maxLength: number): ValidatorFn {}
  static pattern(pattern: string): ValidatorFn {}
}
```

其中一个校验器的实现：

```
  static required(control: AbstractControl): { [key: string]: boolean } { 1
    return 2 isBlank(control.value) ||
    (isString(control.value) && control.value == '')
      ? 3 { required: true }
      : 4 null;
  }
```

返回值 1 是一个 key 为 string，value 为 bool 的对象。如果 2 处成立，则返回 `{'required': true}` 3，否则返回 null 4。

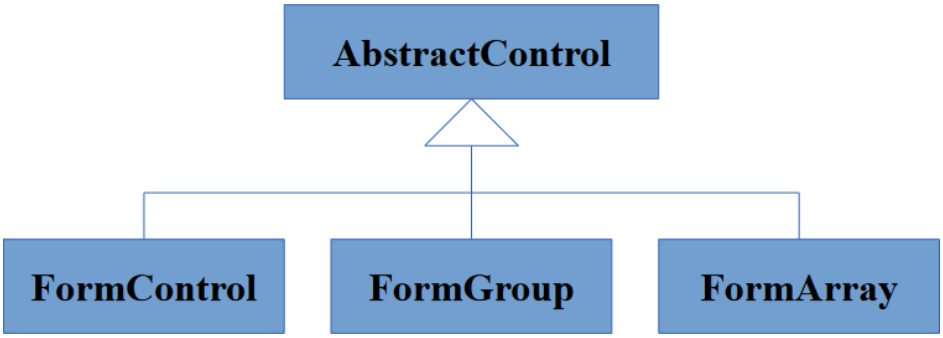
model.ts 文件比较大，定义了表单控件继承关系（见图 9.5）。

表单控件的状态是以下之一：

```
// Indicates that a FormControl is valid,
// i.e. that no errors exist in the input value
export const VALID = 'VALID';

// Indicates that a FormControl is invalid,
// i.e. that an error exists in the input value.
export const INVALID = 'INVALID';
```

图 9.5: Form Control Hierarchy




```

// Indicates that a FormControl is pending, i.e. that async validation is
// occurring and errors are not yet available for the input value.
export const PENDING = 'PENDING';

// Indicates that a FormControl is disabled, i.e. that the control is
// exempt from ancestor calculations of validity or value.
export const DISABLED = 'DISABLED';
```

AbstractControl 类构造器接收一个 validator 和 async validator 函数。还定义了一系列的 getters 用于映射私有字段。value 字段代表我们想要存储在表单控件内的数据：

```
  get value(): any {
    return this._value;
  }
```

_status 字段表示 validator 状态：

```
  get status(): string {
    return this._status;
  }
  get valid(): boolean {
    return this._status === VALID;
  }
  get invalid(): boolean {
    return this._status === INVALID;
  }
  get pending(): boolean {
    return this._status == PENDING;
  }
```

_error 字段返回错误映射（如果有）：

```
  get errors(): { [key: string]: any } {
    return this._errors;
  }
```

_pristine 字段代表控件数据是否改变了——pristine() 为 true 则表示未变化，dirty() 为 true 则表示变化了：

```
get pristine(): boolean {
  return this._pristine;
}
get dirty(): boolean {
  return !this.pristine;
}
```

_touched 代表控件是否被用户访问（但这并不意味着控件值发生了变化）

```
get touched(): boolean {
  return this._touched;
}
get untouched(): boolean {
  return !this._touched;
}
```

还有两个 xxChanges() getter，value changes 和 status changes，返回 observables：

```
get valueChanges(): Observable<any> {
  return this._valueChanges;
}
get statusChanges(): Observable<any> {
  return this._statusChanges;
}
```

通过 event emitters 进行初始化：

```
_initObservables() {
  this._valueChanges = new EventEmitter();
  this._statusChanges = new EventEmitter();
}
```

AbstractControl 还声明了一个函数：

```
abstract _anyControls(condition: Function): boolean;
```

它对控件及其子控件执行条件函数并返回一个布尔值。_anyControls 函数在许多辅助函数中用于确定有关控件的信息，例如：

```
_anyControlsHaveStatus(status: string): boolean {
  return this._anyControls(
    (control: AbstractControl) => control.status == status
  );
}
```

它有一个 parent 字段：

```
  private _parent: FormGroup | FormArray;
```

当控件的状态正在更新时使用。

```
  markAsDirty({ onlySelf }: { onlySelf?: boolean } = {}): void {
    onlySelf = normalizeBool(onlySelf);
    this._pristine = false;
    if (isPresent(this._parent) && !onlySelf) {
      this._parent.markAsDirty({ onlySelf: onlySelf });
    }
  }
```

它通过以下方式设置：

```
  setParent(parent: FormGroup | FormArray): void {
    this._parent = parent;
  }
```

它是继承的，用于查找根：

```
  get root(): AbstractControl {
    let x: AbstractControl = this;
    while (isPresent(x._parent)) {
      x = x._parent;
    }
    return x;
  }
```

FormControl 类是为原子控件（不包含任何子控件）提供的。

```
// By default, a `FormControl` is created for every `<input>` or
// other form component.
export class FormControl extends AbstractControl {
  _onChange: Function[] = [];

  // Register a listener for change events.
  registerOnChange(fn: Function): void {
    this._onChange.push(fn);
  }
  ..
}
```

它的 _value 字段是通过 setValue() 方法设置的，该方法根据给定的四个可选布尔值做出相应改动：

```
  setValue(
    value: any,
    {
      onlySelf,
      emitEvent,
      emitModelToViewChange,
```

```
      emitViewToModelChange,
    }: {
      onlySelf?: boolean;
      emitEvent?: boolean;
      emitModelToViewChange?: boolean;
      emitViewToModelChange?: boolean;
    } = {}
  ): void {
    emitModelToViewChange = isPresent(emitModelToViewChange)
      ? emitModelToViewChange
      : true;
    emitViewToModelChange = isPresent(emitViewToModelChange)
      ? emitViewToModelChange
      : true;

    this._value = value;
    if (this._onChange.length && emitModelToViewChange) {
      this._onChange.forEach((changeFn) =>
        changeFn(this._value, emitViewToModelChange)
      );
    }
    this.updateValueAndValidity({ onlySelf: onlySelf, emitEvent: emitEvent });
  }
```

reset() 方法用于重置控件数据：

```
  reset(
    formState: any = null,
    { onlySelf }: { onlySelf?: boolean } = {}
  ): void {
    this._applyFormState(formState);
    this.markAsPristine({ onlySelf });
    this.markAsUntouched({ onlySelf });
    this.setValue(this._value, { onlySelf });
  }
```

FormGroup 类继承自 AbstractControl：

```
export class FormGroup extends AbstractControl {
  ..
}
```

它的构造函数的第一个参数定义了一个控件关联映射（与 FormArray 形成对比）：

```
  constructor(
    public controls: { [key: string]: AbstractControl },
    validator: ValidatorFn = null,
    asyncValidator: AsyncValidatorFn = null
  ) {
    super(validator, asyncValidator);
```

```
    this._initObservables();
    this._setParentForControls();
    this.updateValueAndValidity({ onlySelf: true, emitEvent: false });
  }
```

可以通过以下方式向 FormGroup 注册控件：

```
  // Register a control with the group's list of controls.
  registerControl(name: string, control: AbstractControl): AbstractControl {
    if (this.controls[name]) return this.controls[name];
    this.controls[name] = control;
    control.setParent(this);
    return control;
  }
```

group 中所有控件的值可以通过以下方式设置：

```
  setValue(
    value: { [key: string]: any },
    { onlySelf }: { onlySelf?: boolean } = {}
  ): void {
    this._checkAllValuesPresent(value);
    StringMapWrapper.forEach(value, (newValue: any, name: string) => {
      this._throwIfControlMissing(name); 1
      this.controls[name].setValue(newValue, { onlySelf: true });
    });
    this.updateValueAndValidity({ onlySelf: onlySelf });
  }
```

请注意，如果缺少任何控件，它会引发异常 1。

FormArray 类继承自 AbstractControl：

```
export class FormArray extends AbstractControl {
  ..
}
```

它的构造函数的第一个参数只是一个数组：

```
  constructor(
    public controls: AbstractControl[],
    validator: ValidatorFn = null,
    asyncValidator: AsyncValidatorFn = null
  ) {
    super(validator, asyncValidator);
    this._initObservables();
    this._setParentForControls();
    this.updateValueAndValidity({ onlySelf: true, emitEvent: false });
  }
```

允许你在数组的末尾或给定位置进行插入，删除：

```
  // Insert a new {@link AbstractControl} at the end of the array.
  push(control: AbstractControl): void {
    this.controls.push(control);
    control.setParent(this);
    this.updateValueAndValidity();
  }

  // Insert a new {@link AbstractControl} at the given `index` in the array.
  insert(index: number, control: AbstractControl): void {
    ListWrapper.insert(this.controls, index, control);
    control.setParent(this);
    this.updateValueAndValidity();
  }

  // Remove the control at the given `index` in the array.
  removeAt(index: number): void {
    ListWrapper.removeAt(this.controls, index);
    this.updateValueAndValidity();
  }
```