What is the 'core' package?

This package contains the core functionality of the Angular compiler. It provides APIs for the implementor of a TypeScript compiler to provide Angular compilation as well.

It supports the 'ngc' command-line tool and the Angular CLI (via the `NgtscProgram`), as well as an experimental integration with `tsc_wrapped` and the `ts_library` Bazel rule via `NgTscPlugin`.

Angular compilation

Angular compilation involves the translation of Angular decorators into static definition fields. At build time, this is done during the overall process of TypeScript compilation, where TypeScript code is type-checked and then downleveled to JavaScript code. Along the way, diagnostics specific to Angular can also be produced.

Compilation flow

Any use of the TypeScript compiler APIs follows a multi-step process:

1. A `ts.CompilerHost` is created.
2. That `ts.CompilerHost`, plus a set of "root files", is used to create a `ts.Program`.
3. The `ts.Program` is used to gather various kinds of diagnostics.
4. Eventually, the `ts.Program` is asked to `emit`, and JavaScript code is produced.

A compiler which integrates Angular compilation into this process follows a very similar flow, with a few extra steps:

1. A `ts.CompilerHost` is created.
2. That `ts.CompilerHost` is wrapped in an `NgCompilerHost`, which adds Angular specific files to the compilation.
3. A `ts.Program` is created from the `NgCompilerHost` and its augmented set of root files.
4. A `CompilationTicket` is created, optionally incorporating any state from a previous compilation run.
5. An `NgCompiler` is created using the `CompilationTicket`.
6. Diagnostics can be gathered from the `ts.Program` as normal, as well as from the `NgCompiler`.
7. Prior to `emit`, `NgCompiler.prepareEmit` is called to retrieve the Angular transformers which need to be fed to `ts.Program.emit`.
8. `emit` is called on the `ts.Program` with the Angular transformers from above, which produces JavaScript code with Angular extensions.

`NgCompiler` and incremental compilation

The Angular compiler is capable of incremental compilation, where information from a previous compilation is used to accelerate the next compilation. During compilation, the compiler produces two major kinds of information: local information (such as component and directive metadata) and global information (such as reified NgModule scopes). Incremental compilation is managed in two ways:

1. For most changes, a new `NgCompiler` can selectively inherit local information from a previous instance, and only needs to recompute it where an underlying TypeScript file has change. Global information is always recomputed from scratch in this case.
2. For specific changes, such as those in component resources, an `NgCompiler` can be reused in its entirety, and updated to incorporate the effects of such changes without needing to recompute any other information.

Note that these two modes differ in terms of whether a new `NgCompiler` instance is needed or whether a previous one can reused. To prevent leaking this implementation complexity and shield consumers from having to manage the lifecycle of `NgCompiler` so specifically, this process is abstracted via `CompilationTicket`s. Consumers first obtain a `CompilationTicket` (depending on the nature of the incoming change), and then use this ticket to retrieve an `NgCompiler` instance. In creating the `CompilationTicket`, the compiler can decide whether to reuse an old `NgCompiler` instance or to create a new one.

Asynchronous compilation

In some compilation environments (such as the Webpack-driven compilation inside the Angular CLI), various inputs to the compilation are only producible in an asynchronous fashion. For example, SASS compilation of `styleUrls` that link to SASS files requires spawning a child Webpack compilation. To support this, Angular has an asynchronous interface for loading such resources.

If this interface is used, an additional asynchronous step after `NgCompiler` creation is to call `NgCompiler.analyzeAsync` and await its `Promise`. After this operation completes, all resources have been loaded and the rest of the `NgCompiler` API can be used synchronously.

Wrapping the `ts.CompilerHost`

Angular compilation generates a number of synthetic files (files which did not exist originally as inputs), depending on configuration. Such files can include:

- `.ngfactory` shim files, if requested.
- `.ngsummary` shim files, if requested.
- A flat module index file, if requested.

- The `__ng_typecheck__.ts` file, which supports template type-checking code.

These files don't exist on disk, but need to appear as such to the `ts.Program`. This is accomplished by wrapping the `ts.CompilerHost` (which abstracts the outside world to the `ts.Program`) in an implementation which provides these synthetic files. This is the primary function of `NgCompilerHost`.

API definitions

The `core` package contains separate API definitions, which are used across the compiler. Of note is the interface `NgCompilerOptions`, which unifies various supported compilation options across Angular and TypeScript itself. It's assignable to `ts.CompilerOptions`, and implemented by the legacy `CompilerOptions` type in `//packages/compiler-cli/src/transformers/api.ts`.

The various types of options are split out into distinct interfaces according to their purpose and level of support.