Incremental Compilation

The `incremental` package contains logic related to incremental compilation in ngtsc. Its goal is to ensure that the compiler's incremental performance is largely O(number of files changed in that iteration) instead of O(size of the program as a whole), by allowing the compiler to optimize away as much work as possible without sacrificing the correctness of its output.

An incremental compilation receives information about the prior compilation, including its `ts.Program` and the result of ngtsc's analyses of each class in that program. Depending on the nature of any changes made to files in the program between its prior and current versions, and on the semantic effect of those changes, ngtsc may perform 3 different optimizations as it processes the new build:

- It can reuse analysis work performed in the previous program

ngtsc receives the analyses of all decorated classes performed as part of the previous compilation, and can reuse that work for a class if it can prove that the results are not stale.

- It can skip emitting a file

Emitting a file is a very expensive operation in TypeScript, involving the execution of many internal TS transforms (downleveling, module system, etc) as well as the synthesis of a large text buffer for the final JS output. Skipping emit of a file is the most effective optimizations ngtsc can do. It's also one of the most challenging. Even if ngtsc's *analysis* of a specific file is not stale, that file may still need to be re-emitted if other changes in the program impact its semantics. For example, a change to a component selector affects other components which use that selector in their templates, even though no direct dependency exists between them.

- It can reuse template type-checking code

Template type-checking code is generated using semantic information extracted from the user's program. This generation can be expensive, and ngtsc attempts to reuse previous results as much as possible. This optimization can be thought of as a special case of the above re-emit optimization, since template type-checking code is a particular flavor of "emit" for a component.

Due to the way that template type-checking works (creation of a second `ts.Program` with `.ngtypecheck` files containing template type-checking blocks, or TCBs), reuse of template type-checking code is critical for good performance. Not only is generation of these TCBs expensive, but forcing TypeScript to re-parse and re-analyze every `.ngtypecheck` file on each incremental change would be costly as well.

The `incremental` package is dedicated to allowing ngtsc to make these important optimizations safely.

During an incremental compilation, the compiler begins with a process called "reconciliation", focused on understanding the differences between the incoming, new `ts.Program` and the last `ts.Program`. In TypeScript, an unchanged file will have its `ts.SourceFile` AST completely reused. Reconciliation therefore examines the `ts.SourceFile`s of both the old and new programs, and identifies files which have been added, removed, or changed. This information feeds in to the rest of the incremental compilation process.

Reuse of analysis results

Angular's process of understanding an individual component, directive, or other decorated class is known as "analysis". Analysis is always performed on a class-by-class basis, so the analysis of a component only takes into consideration information present in the `@Component` decorator, and not for example in the `@NgModule` which declares the component.

However, analysis *can* depend on information outside of the decorated class's file. This can happen in two ways:

- External resources, such as templates or stylesheets, are covered by analysis.
- The partial evaluation of expressions within a class's metadata may descend into symbols imported from other files.

For example, a directive's selector may be determined via an imported constant:

```
import {Directive} from '@angular/core';
import {DIR_SELECTOR} from './selectors';

@Directive({
  selector: DIR_SELECTOR,
})
export class Dir {}
```

The analysis of this directive *depends on* the value of `DIR_SELECTOR` from `selectors.ts`. Consequently, if `selectors.ts` changes, `Dir` needs to be re-analyzed, even if `dir.ts` has not changed.

The `incremental` system provides a mechanism which tracks such dependencies at the file level. The partial evaluation system records dependencies for any given evaluation operation when an import boundary is crossed, building up a file-to-file dependency graph. This graph is then transmitted to the next incremental compilation, where it can be used to determine, based on the set of files physically changed on disk, which files have *logically* changed and need to be re-analyzed.

Reuse of emit results

In plain TypeScript programs, the compiled JavaScript code for any given input file (e.g. `foo.ts`) depends only on the code within that input file. That is, only

the contents of `foo.ts` can affect the generated contents written to `foo.js`. The TypeScript compiler can therefore perform a very simple optimization, and avoid generating and emitting code for any input files which do not change. This is important for good incremental build performance, as emitting a file is a very expensive operation.

(in practice, the TypeScript feature of `const enum` declarations breaks this overly simple model)

In Angular applications, however, this optimization is not nearly so simple. The emit of a `.js` file in Angular is affected in four main ways:

- Just as in plain TS, it depends on the contents of the input `.ts` file.
- It can be affected by expressions that were statically evaluated during analysis of any decorated classes in the input, and these expressions can depend on other files.

For example, the directive with its selector specified via the imported `DIR_SELECTOR` constant above has compilation output which depends on the value of `DIR_SELECTOR`. Therefore, the `dir.js` file needs to be emitted whenever the value of the selector constant in `selectors.ts` changes, even if `dir.ts` itself is unchanged. The compiler therefore will re-emit `dir.js` if the `dir.ts` file is determined to have *logically* changed, using the same dependency graph that powers analysis reuse.

- Components can have external templates and CSS stylesheets which influence their compilation.

These are incorporated into a component's analysis dependencies.

- Components (and NgModules) are influenced by the NgModule graph, which controls which directives and pipes are "in scope" for each component's template.

This last relationship is the most difficult, as there is no import relationship between a component and the directives and pipes it uses in its template. That means that a component file can be logically unchanged, but still require re-emit if one of its dependencies has been updated in a way that influences the compilation of the component.

Example

For example, the output of a compiled component includes an array called `directiveDefs`, listing all of the directives and components actually used within the component's template. This array is built by combining the template (from analysis) with the "scope" of the component - the set of directives and pipes which are available for use in its template. This scope is synthesized from the analysis of not just the component's NgModule, but other NgModules which might be imported, and the components/directives that those NgModules export, and their analysis data as well.

These dependencies of a component on the directives/pipes it consumes, and the NgModule structures that made them visible, are not captured in the file-level dependency graph. This is due to the peculiar nature of NgModule and component relationships: NgModules import components, so there is never a reference from a component to its NgModule, or any of its directive or pipe dependencies.

In code, this looks like:

```
// dir.ts
@Directive({selector: '[dir]'})
export class Dir {}

// cmp.ts
@Component({
  selector: 'cmp',
  template: '<div dir></div>', // Matches the `[dir]` selector
})
export class Cmp {}

// mod.ts
import {Dir} from './dir';
import {Cmp} from './cmp';

@NgModule({declarations: [Dir, Cmp]})
export class Mod {}
```

Here, `Cmp` never directly imports or refers to `Dir`, but it *does* consume the directive in its template. During emit, `Cmp` would receive a `directiveDefs` array:

```
// cmp.js
import * as i1 from './dir';

export class Cmp {
  static cmp = defineComponent({
    ...
    directiveDefs: [i1.Dir],
  });
}
```

If `Dir`'s selector were to change to `[other]` in an incremental step, it might no longer match `Cmp`'s template, in which case `cmp.js` would need to be re-emitted.

SemanticSymbols

For each decorated class being processed, the compiler creates a `SemanticSymbol` representing the data regarding that class that's involved in these "indirect"

relationships. During the compiler's `resolve` phase, these `SemanticSymbol`s are connected together to form a "semantic dependency graph". Two classes of data are recorded:

- Information about the public shape API of the class.

For example, directives have a public API which includes their selector, any inputs or outputs, and their `exportAs` name if any.

- Information about the emit shape of the class, including any dependencies on other `SemanticSymbol`s.

This information allows the compiler to determine which classes have been semantically affected by other changes in the program (and therefore need to be re-emitted) according to a simple algorithm:

1. Determine the set of `SemanticSymbol`s which have had their public API changed.
2. For each `SemanticSymbol`, determine if its emit shape was affected by any of the public API changes (that is, if it depends on a symbol with public API changes).

Determination of public API changes

The first step of this algorithm is to determine, for each `SemanticSymbol`, if its public API has been affected. Doing this requires knowing which `SemanticSymbol` in the previous program corresponds to the current version of the symbol. There are two ways that symbols can be "matched":

- The old and new symbols share the same `ts.ClassDeclaration`.

This is true whenever the `ts.SourceFile` declaring the class has not changed between the old and new programs. The public API of the symbol may still have changed (such as when a directive's selector is determined by a constant imported from another file, like in one of the examples above). But if the declaration file itself has not changed, then the previous symbol can be directly found this way.

- By its unique path and name.

If the file *has* changed, then symbols can be located by their declaration path plus their name, if they have a name that's guaranteed to be unique. Currently, this means that the classes are declared at the top level of the source file, so their names are in the module's scope. If this is the case, then a symbol can be matched to its ancestor even if the declaration itself has changed in the meantime. Note that there is no guarantee the symbol will be of the same type - an incremental step may change a directive into a component, or even into a pipe or injectable.

Once a previous symbol is located, its public API can be compared against the current version of the symbol. Symbols without a valid ancestor are assumed to have changed in their public API.

The compiler processes all `SemanticSymbol`s and determines the `Set` of them which have experienced public API changes. In the example above, this `Set` would include the `DirectiveSymbol` for `Dir`, since its selector would have changed.

Determination of emit requirements

For each potential output file, the compiler then looks at all declared `SemanticSymbol`s and uses their ancestor symbol (if present) as well as the `Set` of public API changes to make a determination if that file needs be emitted.

In the case of a `ComponentSymbol`, for example, the symbol tracks the dependencies of the component which will go into the `directiveDefs` array. If that array is different, the component needs to be re-emitted. Even if the same directives are referenced, if one of those directives has changed in its public API, the emitted output (especially when generating prelink library code) may be affected, and the component needs to be re-emitted.

`SemanticReferences`

`ComponentSymbol`s track their dependencies via an intermediate type, a `SemanticReference`. Such references track not only the `SemanticSymbol` of the dependency, but also the name by which it was imported previously. Even if a dependency's identity and public API remain the same, changes in how it was exported can affect the import which needs to be emitted within the component consuming it, and thus would require a re-emit.

Reuse of template type-checking results

Since type-checking block (TCB) generation for template type-checking is a form of emit, `SemanticSymbol`s also track the type-checking shape of decorated classes. This includes any data which is not public API, but upon which the TCB generation for components might depend. Such data includes:

- Type-checking API shape from any base classes, since TCB generation uses information from the full inheritance chain of a directive/pipe.
- The generic signature shape of the class.
- Private field names for `@Input`s and `@Output`s.

Using a similar algorithm to the `emit` optimization, the compiler can determine which files need their type-checking code regenerated, and which can continue to use TCB code from the previous program, even if some dependencies have unrelated changes.

Unsuccessful compilation attempts

Often, incremental compilations will fail. The user's input program may contain incomplete changes, typos, semantic errors, or other problems which prevent the compiler from fully analyzing or emitting it. Such errors create problems for

incremental build correctness, as the compiler relies on information extracted from the previous program to correctly optimize the next compilation. If the previous compilation failed, such information may be unreliable.

In theory, the compiler could simply not perform incremental compilation on top of a broken build, and assume that it must redo all analysis and re-emit all files, but this would result in devestatingly poor performance for common developer workflows that rely on automatically running builds and/or tests on every change. The compiler must deal with such scenarios more gracefully.

ngtsc solves this problem by always performing its incremental steps from a "last known good" compilation. Thus, if compilation A succeeds, and a subsequent compilation B fails, compilation C will begin using the state of compilation A as a starting point. This requires tracking of two important pieces of state:

- Reusable information, such as analysis results, from the last known good compilation.
- The accumulated set of files which have physically changed since the last known good compilation.

Using this information, ngtsc is able to "forget" about the intermediate failed attempts and begin each new compilation as if it were a single step from the last successful build. It can then ensure complete correctness of its reuse optimization, since it has reliable data extracted from the " previous" successful build.