James Garbutt

Software engineer. Front-end @crispthinking.

Creating a TypeScript Transformer

August 14, 2018 Edit Page

Last year TypeScript merged a pretty cool Pull Request which basically exposes the parts of the compiler API needed to make use of custom transformers.

These custom transformers can traverse the AST at compile time and do pretty useful things like add, replace and remove statements or particular nodes.

Soon after that PR landed, I figured its time to give it a go…

Where I began

I’ve worked pretty extensively on the Polymer Decorators in the past. We basically implement some decorators like so:

```
@customElement('x-foo')
class FooElement extends HTMLElement {}
```

Which, **at runtime** nicely results in a call to define the element:

```
customElements.define('x-foo', FooElement);
```

However, we soon end up with some messy output…

This is of course because we have to bundle in a polyfill for the decorators spec for use at run-time.

So here comes the next idea… what if we could do it at build time instead?

A little transformer

The documentation on this API is incredibly lacking and documentation on the compiler API as a whole, really. Not a great amount of it has been used extensively outside of TypeScript and not much has official documentation other than the type definitions.

After a lot of digging around GitHub for PRs containing these new transformers and a lot of reading through the compiler type definitions, I eventually managed a simple one:

```
function simpleTransformer<T extends ts.Node>(): ts.TransformerFactory<T> {
  return (context) => {
    const visit: ts.Visitor = (node) => {
      if (ts.isDecorator(node)) {
        return undefined;
      }
      return ts.visitEachChild(node, (child) => visit(child), context);
    };

    return (node) => ts.visitNode(node, visit);
  };
}
```

So what are we doing here?

We define our transformer function which should return a `TransformerFactory`, another function which, given a context, should return our transformer!

Now, a transformer as you can imagine is simply a function which takes a node and returns a node (maybe the same node, maybe not):

```
type Transformer<T extends Node> = (node: T) => T;
```

So our transformer is in fact this line:

```
(node) => ts.visitNode(node, visit)
```

The next step is to pair the transformer with a visitor. A visitor is a function which, given a node, returns a `VisitResult` (which is `Node|Node[]|undefined`):

```
const visit: ts.Visitor = (node) => {
  // ...
};
```

What we do in this function is the important part. This is where we will do all of our manipulation of the AST.

In my case, I just want to see a change, so I went ahead and removed all decorators:

```
if (ts.isDecorator(node)) {
  return undefined; // Replace the node with nothing
}
```

Here you see the first use of an `is*` method. Many of these exist to assert against the type of a node and thus narrow it down in further statements. These exist for almost all types of node (e.g. `isClassDeclaration`, `isFunction`, etc.).

They are cool little functions which make good use of the `node is T` return type such that TypeScript knows if this is truthy, it can infer a narrower type.

Importantly, the last part you may have noticed is:

```
return ts.visitEachChild(node, (child) => visit(child), context);
```

This is because a decorator could be at any depth and we start from the most outer scope of the file. For this reason, if we haven't matched a decorator, we want to keep traversing the tree in case there is one inside a deeper block of code.

That is exactly what this method is doing: run the visitor against each child of the node we are currently visiting.

Run it!

For quickly testing things out, I used the following code to compile some source with the transformer enabled:

```
let source = `
@foo
class MyClass {
  @bar
  public a: number;

  @baz
  public b: number;
}
`;
let result = ts.transpileModule(source, {
  compilerOptions: {module: ts.ModuleKind.CommonJS},
  transformers: {before: [simpleTransformer()]}
});
```

The result of which is roughly:

```
var MyClass = (function() {
  function MyClass() {}
  return MyClass;
})();
```

No decorators! No polyfills!

Getting more complex

The next thing was to have a look at how to do a more complex transform, such as adding a statement to the output.

Let’s take my `@customElement` example. Of course there is no point us having a run-time decorator execute the define call.

It should be pretty simple to instead replace it with the define call its self at build time.

I started by first trying to find classes with this and to strip it.

Here’s my input source:

```
@customElement('x-foo')
class XFoo {}
```

And the visitor:

```
if (ts.isClassDeclaration(node) && node.decorators && node.name) {
  const decorator = node.decorators.find((decorator) => {
    const decoratorExpr = decorator.expression;

    return ts.isCallExpression(decoratorExpr) &&
      decoratorExpr.expression.getText() === 'customElement';
  });

  if (decorator) {
    node.decorators = ts.createNodeArray(
      node.decorators.filter((d) => d !== decorator)
    );
  }

  return node;
}
```

Breaking it down…

We first try to filter down to class declarations which have decorators:

```
if (ts.isClassDeclaration(node) && node.decorators && node.name) {
```

We only care about class declarations because we should only allow `@customElement` on a class in our case.

We also only care if the class has a name and there are decorators as it is clear ours isn’t there otherwise.

Next, we try find our decorator:

```
const decorator = node.decorators.find((decorator) => {
  const decoratorExpr = decorator.expression;

  return ts.isCallExpression(decoratorExpr) &&
    decoratorExpr.expression.getText() === 'customElement';
});
```

We are trying to find a decorator which is a `CallExpression` and has our name: `customElement`.

So `@customElement()` would be a call expression as it is invoking the function. Whereas `@customElement` would be an identifier expression as it is nothing more than an identifier syntactically.

To understand this, I repeatedly (again, and again, and again) went back to ASTExplorer. Most of the properties and types roughly match up to what you see in ASTExplorer.

You can see the source I'm working with here for example.

Finally, we remove our decorator by setting the decorators node list to one without ours included:

```
if (decorator) {
  const filteredDecorators = node.decorators.filter((d) => d !== decorator);

  if (filteredDecorators.length > 0) {
    node.decorators = ts.createNodeArray(filteredDecorators);
  } else {
    node.decorators = undefined;
  }
}
```

Here you can see the first time we use one of TypeScript's `create*` methods. There are *many* of these and they allow you to create a node of any type you wish (e.g. `createBlock`, `createStatement`, `createFunctionExpression`, etc.).

In our case, we create a `NodeArray` as that is how decorators are represented on other nodes. The rest is clear, filter down by those which are not our decorator.

Importantly, we set the decorators to `undefined` if there are none left. This will prevent TypeScript from including the decorators polyfill just because we have an empty array of them.

Add a node!

The final piece of the puzzle in this example was to add a new node: the `customElements.define` call.

Now that we have our decorator and the class it is on, this becomes pretty simple:

```
if (decorator) {
  // ...

  const name = (decorator.expression as ts.CallExpression).arguments[0];
  const defineCall = ts.createStatement(ts.createCall(
    ts.createPropertyAccess(
      ts.createIdentifier('customElements'),
      ts.createIdentifier('define')
    ), undefined, [name, node.name]));

  return [node, defineCall];
}
```

Immediately you can see instead of returning the class node as is, we are now returning a new array of the class node and a new node: the define call.

Looking here, you can see the following structure:

```
ExpressionStatement {
  expression: CallExpression {
    expression: PropertyAccessExpression {
      expression: Identifier {
        text: 'customElements'
      }
      name: Identifier {
        text: 'define'
      }
    }
    arguments: [
      StringLiteral {
        text: 'x-foo'
      }
      Identifier {
        text: 'XFoo'
      }
    ]
  }
}
```

Translate this to API calls pretty much the same:

```
ts.createStatement(
  ts.createCall(
    ts.createPropertyAccess(
      ts.createIdentifier('customElements'),
      ts.createIdentifier('define')
    ),
    undefined, // type arguments, e.g. Foo<T>()
    [
      ts.createLiteral('x-foo'),
      ts.createIdentifier('XFoo')
    ]
  )
)
```

Replace the hard-coded literals with the ones we have picked up from our class and you're done.

Now we just return two nodes instead of one:

```
return [node, defineCall];
```

And we're finished!

The output now looks like this:

```
var MyClass = (function() {
  function MyClass() {
  }
  return MyClass;
})();
customElements.define('x-foo', XFoo);
```

Wrap up

There's probably a lot of flaky, insane things I'm doing around these parts. Picking up a fairly undocumented API isn't an easy thing to do, especially when there are very few examples floating around.

For example, I suspect I am not supposed to mutate the decorators node list. I am probably expected to return a new one some way. Who knows.

There's a lot to be learnt about this and probably a lot I can improve on. But maybe this quick look at what I went through to write my own transformer provides some guidance to people in a similar situation.

I relied heavily on the type definitions (most of them have helpful comments), ASTExplorer (most of the AST maps directly to method names in the API), and browsing GitHub related PRs for far too long.

Unfortunately, the TypeScript team does seem to have decided to *never ever* introduce a way to use transformers via your tsconfig, so you **will need a build script** to make use of one.

Anyhow, have fun!

Like 3 Share Tweet