# 编写合约的编译脚本

之前的课程中，我们已经熟悉了智能合约的编译。编译是对合约进行部署和测试的前置步骤，编译步骤的目标是把源代码转成 ABI 和 Bytecode，并且能够处理编译时抛出的错误，确保不会在包含错误的源代码上进行编译。

开始我们的编译方式是用 solc 工具做命令行编译，这个过程中牵涉到大段内容的复制粘贴，很容易出错；之后在项目中引入 solc 模块，可以在 node 命令行中自动编译并读取结果内容。于是我们自然会想到，能不能将这个过程写成脚本，自动完成这些过程呢？这节课我们就来完成这个任务。

## 目录结构

首先新建一个项目目录，可以叫做 contract_workflow。

```
mkdir contract_workflow
cd contract_workflow
```

为了存放不同目的不同类型的文件，我们先在项目根目录下新建 4 个子目录：

```
mkdir contracts
mkdir scripts
mkdir compiled
mkdir tests
```

其中 contracts 目录存放合约源代码，scripts 目录存放编译脚本，complied 目录存放编译结果，tests 目录存放测试文件。

## 准备合约源码

为了简化工作，我们可以直接复制以前的 solidity 代码，也可以自己写一个简单的合约。比如，这里用到了我们最初写的简单合约 Car.sol：

```
pragma solidity ^0.4.22;
contract Car {
	string public brand;
	constructor(string initialBrand) public {
		brand = initialBrand;
	}
	function setBrand(string newBrand) public {
		brand = newBrand;
	}
}
```

将它放到 contracts 目录下。

## 准备编译工具

我们用 solc 作为编译的基础工具。用 npm 将 solc 安装到本地目录中：

```
npm install solc
```

## 开发编译脚本

我们已经熟悉了命令行编译的流程，现在我们试图将它脚本中。在 scripts 目录下新建文件 compile.js

```
const fs = require('fs');
const path = require('path');
const solc = require('solc');
const contractPath = path.resolve(__dirname, '../contracts',
'Car.sol');
const contractSource = fs.readFileSync(contractPath, 'utf8');
const result = solc.compile(contractSource, 1);
```

```
console.log(result);
```

我们把合约源码从文件中读出来，然后传给 solc 编译器，等待同步编译完成之后，把编译结果输出到控制台。

其中 solc.compile() 的第二个参数给 1，表示启用 solc 的编译优化器。

编译结果是一个嵌套的 js 对象，其中可以看到 contracts 属性包含了所有找到的合约（当然，我们的源码中只有一个 Car）。每个合约下面包含了 assembly、bytecode、interface、metadata、opcodes 等字段，我们最关心的当然是这两个：

- bytecode：字节码，部署合约到以太坊区块链上时需要使用；
- interface： 二进制应用接口（ABI），使用 web3 初始化智能合约交互实例的时候需要使用。

其中 interface 是被 JSON.stringify 过的字符串，我们用 JSON.parse 反解出来并格式化，就可以拿到合约的 abi 对象。

## 保存编译结果

让我们继续课程，现在将合约部署到区块链上。为此，你必须先通过传入 abi 定义来创建一个合约对象 VotingContract。然后用这个对象在链上部署并初始化合约。为了方便后续的部署和测试过程直接使用编译结果，需要把编译结果保存到文件系统中，在做改动之前，我们引入一个非常好用的小工具 fs-extra，在脚本中使用 fs-extra 直接替换到 fs，然后在脚本中加入以下代码：

```
Object.keys(result.contracts).forEach( name => {
        const contractName = name.replace(/^:/, '');
        const filePath = path.resolve(__dirname, '../compiled',
                                `${contractName}.json`);
        fs.outputJsonSync(filePath, result.contracts[name]);
        console.log(`save compiled contract ${contractName} to
                        ${filePath}`);
```

```
});
```

然后重新运行编译脚本，确保 complied 目录下包含了新生成的 Car.json。

类似于前端构建流程中的编译步骤，我们编译前通常需要把之前的结果清空，然后把最新的编译结果保存下来，这对保障一致性非常重要。所以继续对编译脚本做如下改动：

在脚本执行的开始加入清除编译结果的代码：

```
// cleanup
const compiledDir = path.resolve(__dirname, '../compiled');
fs.removeSync(compiledDir);
fs.ensureDirSync(compiledDir);
```

这里专门定义了 compiledDir，所以后面的 filePath 也可以改为：

```
const filePath =
      path.resolve(compiledDir, `${contractName}.json`);
```

新增的 cleanup 代码段的作用就是准备全新的目录，修改完之后，需要重新运行编译脚本，确保一切正常。

## 处理编译错误

现在的编译脚本只处理了最常见的情况，即 Solidity 源代码没问题，这个假设其实是不成立的。如果源代码有问题，我们在编译阶段就应该报出来，而不应该把错误的结果写入到文件系统，因为这样会导致后续步骤失败。 为了搞清楚编译器 solc 遇到错误时的行为，我们人为在源代码中引入错误（例如把 function 关键字写成 functio），看看脚本的表现如何。

重新运行编译脚本，发现它并没有报错，而是把错误作为输出内容打印出来，其中错误的可读性比较差。

所以我们对编译脚本稍作改动，在编译完成之后就检查 error，让它能够在出错时直接抛出错误：

```
// check errors
if (Array.isArray(result.errors) && result.errors.length) {
        throw new Error(result.errors[0]);
}
```

重新运行编译脚本，可以看到我们得到了可读性更好的错误提示。


## 最终版编译脚本

编译脚本的最终版如下：

```
const fs = require('fs-extra');
const path = require('path');
const solc = require('solc');

// cleanup
const compiledDir = path.resolve(__dirname, '../compiled');
fs.removeSync(compiledDir);
fs.ensureDirSync(compiledDir);
```

```
// compile const contractPath = path.resolve(__dirname,
                            '../contracts', 'Car.sol');
const contractSource = fs.readFileSync(contractPath, 'utf8');
const result = solc.compile(contractSource, 1);

// check errors
if (Array.isArray(result.errors) && result.errors.length) {
       throw new Error(result.errors[0]);
}

// save to disk
Object.keys(result.contracts).forEach(name => {
const contractName = name.replace(/^:/, '');
const filePath = path.resolve(compiledDir,
                            `${contractName}.json`);
fs.outputJsonSync(filePath, result.contracts[name]);
console.log(`save compiled contract ${contractName} to
             ${filePath}`); });
```