打造首款React质量与性能分析工具

首先说一下为什么要写这款工具，由来是什么吧。其实我们在不同的公司里面，都会发现一个问题，框架这这么一款，官方文档介绍也很详细，为什么能够写出各种各样的代码，明明框架约束的很好，但是写出来的代码五花八门，要好好拜读别人的代码，才能知道其中的业务逻辑是什么，完全是事倍功半啊。如果有好的工具，必然能够提高公司的整体代码质量。

其实上面的问题，很早就被各样的牛人发现了，然后开发出了很好的工具来辅助开发，比如eslint，csslint等，帮助开发者矫正错误的代码风格，统一公司的代码风格。但是对于最近几年很火的React框架，是否有一些好的工具来完善react代码吗，答案肯定是有的，其实去网上搜，肯定也能找到一些，但是今天将要介绍的一款工具，react-perf工具，是从bable层的AST来分析语法，在编译阶段进行校正，最后在编译阶段没有问题后，再在运行阶段进行性能分析，给出性能建议，提高整体代码质量。

在给出方案前，让我们一起看看，在大家开发过程中，都有哪些不好的代码。

在componentWillMount设置setState?


```
componentWillMount(){
   this.setState({count:2})
}
```

这个问题是无话可说的，完全不应该存在。

直接赋值state？

```
clickEvent () {
    this.state.name = 'hello'
    this.state.age = 20
    this.setState(this.state)
}
```

背景：这段代码的历史背景是这样的，我们的开发者在使用setState的时候，发现this.setState是异步的，为了解决可以同步使用this.state,就用了这样的方式。问题是，现在好多开发者都是Ctrl+c/Ctrl+v,导致好多地方都是用这样的代码,在这里，我们也是坚决杜绝。

这里只是简单的列了一些代码，像这样的代码，其实还有很多很多。对于这些代码，完全可以在编译阶段进行杜绝。编译阶段怎么杜绝，那就只有靠babel来搞定了，接下来就是我们要说的主题了。

编译分析

babel这里就不说了，这里就直接进阶到babel的插件来分析了。如果大家不熟悉babel插件，那我简单说一下，babel插件就是通过遍历文件结构树，提供一种可以对AST进行增，修和查的机制。

插件具体实现步骤如下。

刷选react类

首先我们肯定是需要对react类进行处理，所以基于这一点，我们肯定是要对于类进行刷选，这里就引出来第一个配置项。

```
"needAddPerfRule": {
  "superClass": ["Component"]
}
```

而在babel插件里面，我们先对 `Class` 进行判断，如果不是继承 `Component` ,或者不是我们设置的父类，程序就返回，我们的实现如下：


```
Class: function Class(path, state) {
    var node = path.node;
    var superClass = node.superClass.name;
    if (superClass !== 'Component' && state.opts.needAddPerfRule
&& !state.opts.needAddPerfRule.superClass.includes(superClass)) {
   return;
 }
    path.traverse(classVisitor, { types: t, opts:state.opts })
}
```

进入类的方法进行分析

对于类方法进行分析，我们首先会对一些固定的方式进行分析，大家还记得开头的一些不良示范吗？对的，对于这些，我们就直接进行分析处理了。

分析this.state的直接赋值

细心的朋友应该发现，constructor被排除了，因为在它里面，是正常的赋值啦。

```
AssignmentExpression: function AssignmentExpression(path) {
  if (['constructor'].includes(method)) {
    return;
  }
```

```
    path.traverse({
      MemberExpression: function MemberExpression(path) {
        var node = path.node;
        if (node.property.name === 'state' && node.object.type ===
'ThisExpression' && path.parentPath.parentKey === 'left') {
          throw path.buildCodeFrameError('Do not assign value to
`state` directly');
        }
      }
    });
  }
```

分析无用的setState

明明可以在constructor里面直接赋值，为什么要用setState.

```
  if (['constructor', 'componentWillMount'].includes(method)) {
    path.traverse({
      MemberExpression: function MemberExpression(path) {
        var node = path.node;
        if (node.property.name === 'setState' && node.object.type
=== 'ThisExpression') {
          throw path.buildCodeFrameError('Please use `this.state`
in the constructor ');
        }
      }
    });
  }
```


错误演示

```
ERROR in ./src/State.js
Module build failed: SyntaxError: Please use `this.state` in the constructor

  31 |
  32 |   componentWillMount(){
> 33 |     this.setState({count:2})
     |     ^
  34 |   }
  35 |
```

分析render函数

我之前看到好多人在render里面写了好多原生元素，一看代码，直接就晕了，完全看不懂了，对于这点，我们将要引出另外一个配置项，用于检测最大的组件数目，保证代码的清晰和组件的颗粒度，提高react的diff效率。

```
  "maxRenderElements": 50
```

代码实现

```
function renderCheck (path, state) {
  let elementCount = 0
  path.traverse({
    JSXOpeningElement (path) {
      if (/^[^A-Z]+$/.test(path.node.name.name)) {
        elementCount ++
      }
    }
  })
  let maxRenderElements = state.opts.maxRenderElements ? state.opts.maxRenderElements : 50
  if (elementCount > maxRenderElements) {
    let opts = path.hub.file.opts
    let filepath = opts.filename.replace(opts.moduleRoot, '')
    let message = `${filepath}: render has ${elementCount} elements (maxRenderElements = ${maxRenderElements})\n`
    console.log('********************react perf warning********************\n'.bold.yellow)
    console.log(message.yellow.bold)
  }
}
```

自定义过滤

在不同企业，使用方式不同，对框架的理解也不同，有些时候，不能完全限制一些使用方式，需要自定化。我们将使用另一个配置属性，该选项的值都是字符串，而且是正则，通过符合正则的语句，来过滤不好的语法。

```
    "invalidStatements": [
      "this.refs.\\w+.value\\s*=\\s*\\S+"
    ]
```

相应代码实现部分，通过对表达式的分析，对于符合正则的语句，进行抛错。


```
  ExpressionStatement: function ExpressionStatement(path) {
    var invalidStatements = [];
    var file = void 0;
    var code = void 0;
    try {
      file = path.hub.file.code;
      code = file.slice(path.node.start, path.node.end);
      invalidStatements = state.opts.invalidStatements;
    } catch (e) {
      console.log(e);
    }
    for (var i = 0; i < invalidStatements.length; i++) {
      var reg = new RegExp(invalidStatements[i]);
      if (reg.test(code)) {
        throw path.buildCodeFrameError('Invalid Statement is
  checked by \'' + invalidStatements[i] + '\'');
      }
    }
  }
```

编译抛错演示

```
ERROR in ./src/State.js
Module build failed: SyntaxError: Invalid Statement is checked by 'this.refs.\w+.value\s*=\s*\S+'

  82 |   update(){
  83 |     this.refs.p.textContent = '11'
> 84 |     this.refs.button.value = '11'
     |     ^
```

运行分析

运行分析，是在编译没有出错后，在运行我们的应用程序时发现的问题，比如有超过指定时间渲染，或者有不必要的重新渲染操作，这些操作，在手机上运行web app，还是可以节省很多资源的。接下来就分析其本质。

其实对于分析工具，react就有react-addons-perf这个工具，但是有一个问题是，难道我们每次都要去引用相关的组件，再把相应的代码写到react的指定方法里面，测试好了之后，再去掉，完全不符合工业化和自动化的标准，所以本插件就是基于react-addons-perf这个工具，进行自动注入相应的组件和代码，方便用户去使用，省去一大堆麻烦，本插件还可以根据不同react版本，进行适配。

分析是否需要导入插件

首先去判断，该类是否是我们想要的类，如果是，那就自动导入perf类。


```
        ImportDeclaration (path, state) {
          let node = path.node
          let needAddPerf = node.specifiers.some((item) => {
            if (item.local.name === 'Component') {
              return true
            } else if (state.opts.needAddPerfRule && state.opts.needAddPerfRule.superClass.includes(item.local.name)) {
              return true
            }
          })
          if (needAddPerf) {
            importPerfClass(path, t)
          }
        },
```

```
function importPerfClass (path, types) {
  let nodes = []
  // add perf component
  let identifier = types.identifier('Perf')
  let importDefaultSpecifier = types.importDefaultSpecifier(identifier);
  let importDeclaration = types.importDeclaration([importDefaultSpecifier],types.stringLiteral('react-addons-perf'));
  nodes.push(importDeclaration)
  // add perf log
  identifier = types.identifier('perfLog')
  importDefaultSpecifier = types.importDefaultSpecifier(identifier);
  importDeclaration = types.importDeclaration([importDefaultSpecifier],types.stringLiteral('babel-plugin-react-perf/lib/log.js'))
  nodes.push(importDeclaration)
  path.insertAfter(nodes)
}
```

对类的非生命周期方法进行自动性能代码注入

```
function addPerfResult (path, state) {
  let t = state.types
  let nodes = []
  let memberExpression
  let callExpression
  let expressionStatement

  // add perf.start
  callExpression = t.callExpression(t.identifier('perfLog'), [t.identifier('Perf'), t.identifier(JSON.stringify(state.opts))])
  expressionStatement = t.expressionStatement(callExpression)
  nodes.push(expressionStatement)

  path.insertAfter(nodes)
}
```

输出监控信息

这里完全是正常的js语法了，没有用babel的语法，很简单，一看就知道，但是这里是有缺陷的，大家可以思考一下，缺陷是什么？

```
module.exports = function(Perf, config) {
  Perf.start()
  setTimeout(function () {
    Perf.stop()
    let lastMeasurements = Perf.getLastMeasurements()
    let wastedReport = Perf.getWasted(lastMeasurements)
    if (wastedReport && wastedReport.length) {
      console.info('%c react-Perf: wasted report', 'color:red;font-size:20px')
      console.table(wastedReport)
    }
    if (Perf.getInclusive) {
      let inclusivedReport = Perf.getInclusive(lastMeasurements)
      let maxExecuteLimit = config.maxExecuteLimit ? config.maxExecuteLimit : 5
      inclusivedReport = inclusivedReport.filter((item) => {
        return item.inclusiveRenderDuration > maxExecuteLimit
      })
      if (inclusivedReport && inclusivedReport.length) {
        console.info(`%c react-Perf: inclusive report (maxExecuteLimit = ${maxExecuteLimit}ms)`, 'color:red;font-size:20px')
        console.table(inclusivedReport)
      }
    } else {
      Perf.printInclusive(lastMeasurements)
    }
  }, 200)
}
```


运行演示：

1. 超过运行配置时间的（maxExecuteLimit配置）。

react-Perf: inclusive report (maxExecuteLimit = 5ms) log.js:18

log.js:19

(index)	key	instanceCount	inclusiveRenderDura…	renderCount
0	"App > State"	1	5.910000000000764	1

▶ Array[1]

1. 有不必要渲染的。

react-Perf: wasted report log.js:8

log.js:9

(index)	key	instanceCount	inclusiveRenderDura…	renderCount
0	"App > State"	1	4.5900000000001455	1
1	"State > Props"	1	0.04500000000007276	1
2	"State > Wang"	1	0.03500000000076397…	1
3	"Props > Mine"	1	0.02000000000043655…	1

▶ Array[4]

配置

对于配置的话，其实我们有两种方式。

1. babelrc里面进行设置，这样的话，完全会在开发，测试，线上等环境生效，其实对于这个插件，我们有时只要在开发上用就行。

2. 可以在webpack的babel-loader里面配置，这样可以只在开发的webpack配置进行运行这个插件。

配置文件结构

```
"plugins": [
  ["react-perf",{
      "maxExecuteLimit": 5,
      "maxRenderElements": 50,
      "needAddPerfRule": {
        "superClass": ["Component"]
      },
      "invalidStatements": [
        "this.refs.\\w+.value\\s*=\\s*\\S+"
      ]
    }
  ]
]
```


插件git地址

地址：https://github.com/vicwang163/babel-plugin-react-perf

结束

其实对于这个插件，未来还有很多可以开发的地方，更多是要大家多多提意见，把东西做好，服务好大家。

题外话：大家对自己写的react代码自信的话，可以用该插件跑一下，偷偷告诉大家，作者自己用了该插件测试了一下自己曾经写的react代码，心里阴影那么大。