# 经典案例-TodoMVC

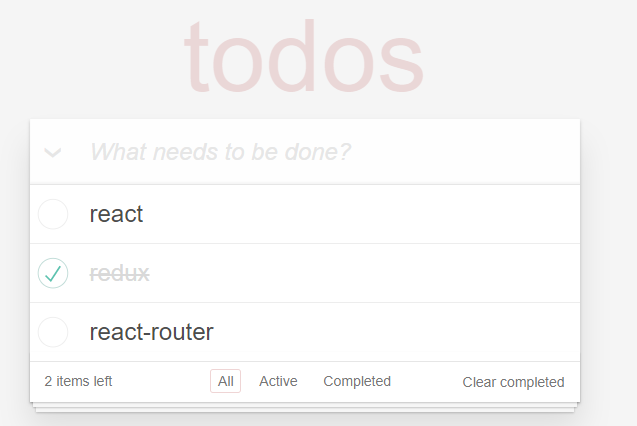
基本上任何一个前端的框架/库，都实现了这个todomv经典案例。

<http://todomvc.com/>



## 目标和分析

### 目标



### 分析

从三个角度来进行分析:

* 功能
* 组件划分
* 确定组件状态

**A.功能**

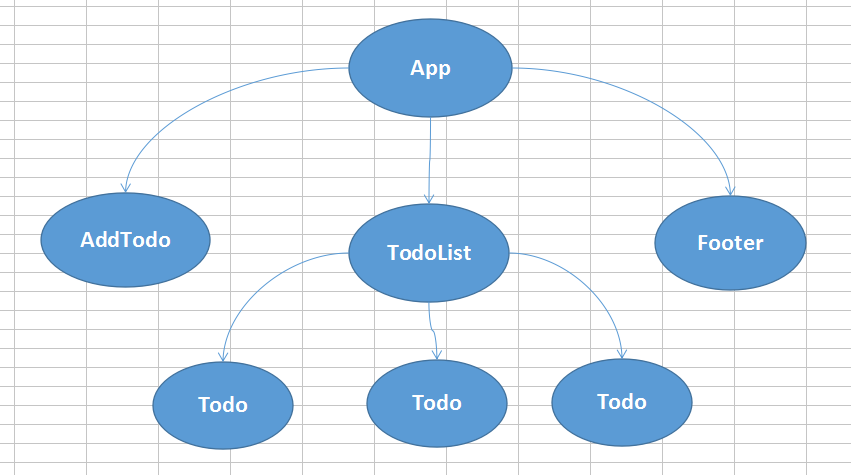
* 添加todo任务
* 显示todos任务列表
* 删除指定todo任务
* 切换指定todo任务的状态
* 批量切换todos任务的状态
* 统计未完成的任务
* 设置可见性显示对应的todos任务
* 清除已完成的todos任务
* 编辑指定的todo任务

**B.组件划分**

一共有5种组件

* App组件
* AddTodo组件
* TodoList组件
* Todo组件
* Footer组件

其关系如下：



**C.确定组件状态**

* todos：任务项，属于app组件的
* visibility：可见性，有all、active和completed三种值，属于app组件
* editing：是否处于编辑状态，属于todo组件的

## 构建项目结构

### 使用create-react-app快速构建项目

除了可以手动搭建之外，在实际开发的时候，一般推荐使用脚手架工具来快速搭建项目结构。

比较知名的脚手架工具有两个：

* **create-react-app （facebook官方使用的）**
* generator-react-webpack

使用create-react-app来快速构建项目结构

**第一步，需要全局安装**create-react-app

npm i create-react-app -g

**第二步，初始化项目**

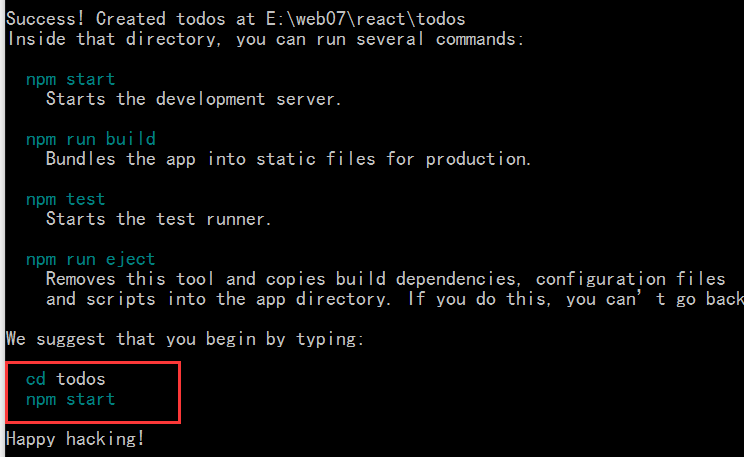
可以查看手册：



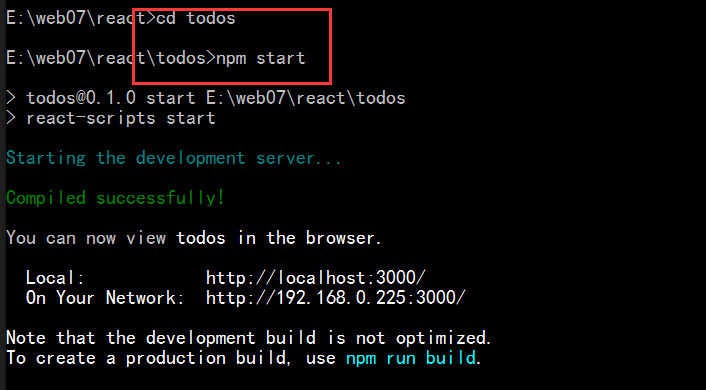
使用create-react-app命令初始化如下：

和vue不同的是，这个命令会自己去下载所有的模块和包，不需要我们手动执行安装命令。

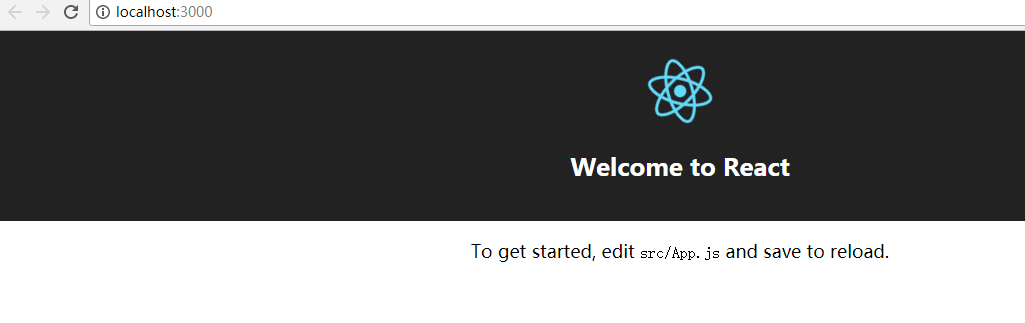
安装之后，看提示：



执行命令如下：

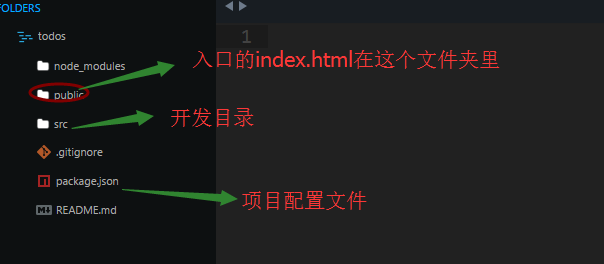


浏览如下：



### 项目结构分析

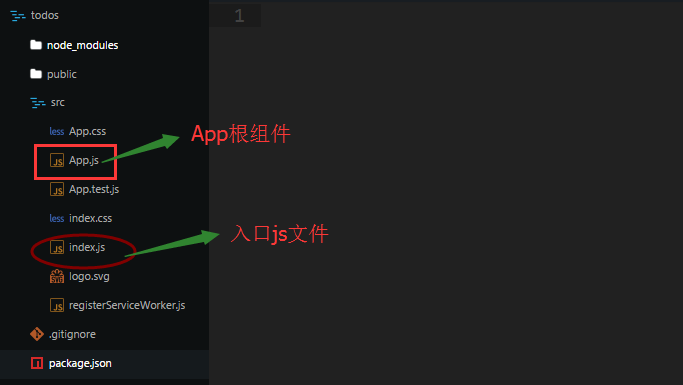
根目录如下：



Public目录如下：



Src目录如下：

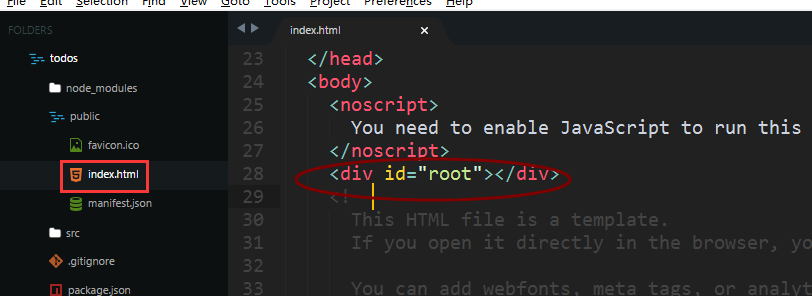


这个目录结构相对来说，有点乱。

我们在开发的时候，需要加以改造。

### 流程分析

**第一步，分析index.html文件**

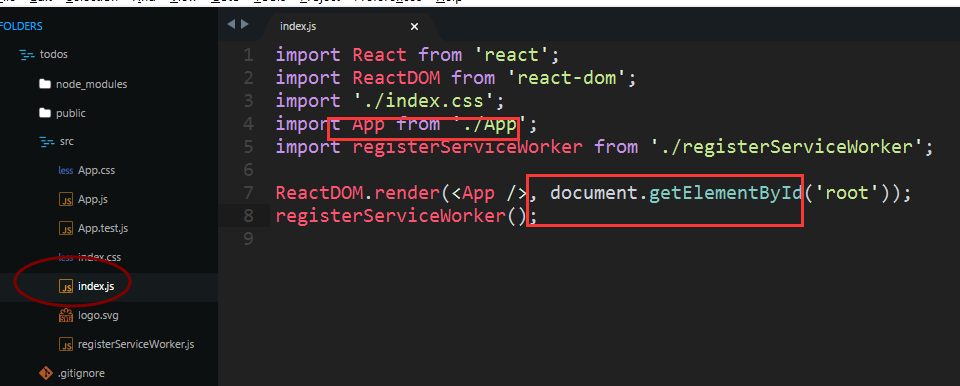


有一个挂载根组件的dom节点。

js会自动注入到这个页面中。

**第二步，分析src/index.js文件**

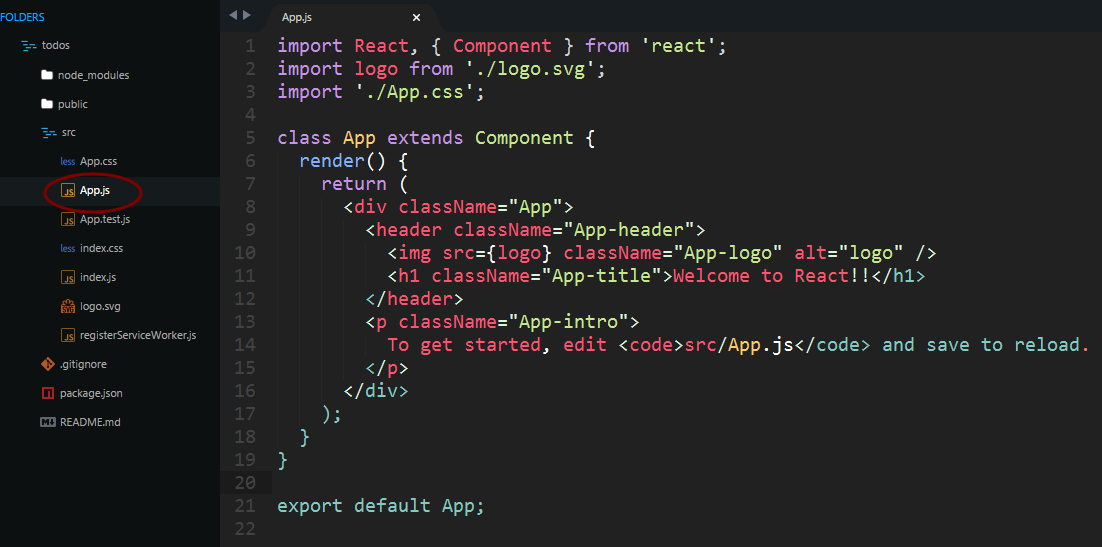
Js入口文件



主要做了下面几件事情：

* 引入react和react-dom
* 引入css文件
* 引入App组件
* 实例化App组件，渲染到root节点上

**第三步，分析App.js**



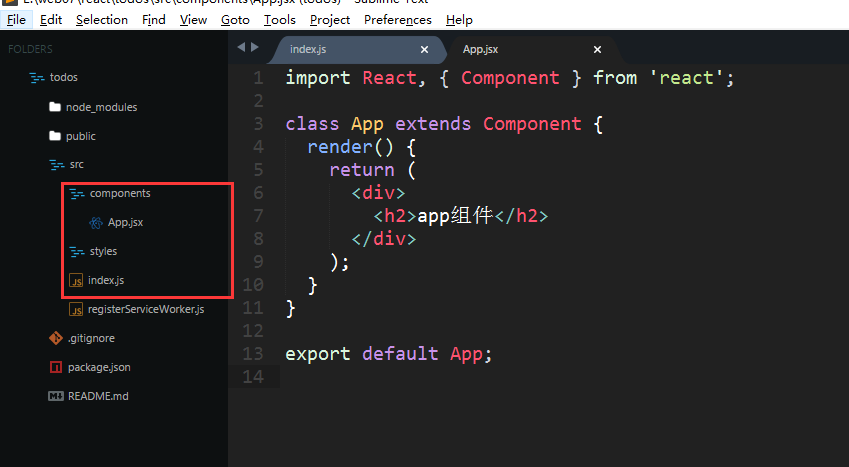
主要做了如下几件事：

* 引入React和Component
* 引入css
* 引入图片
* 定义组件
* 导出组件

## 定义组件并显示ui

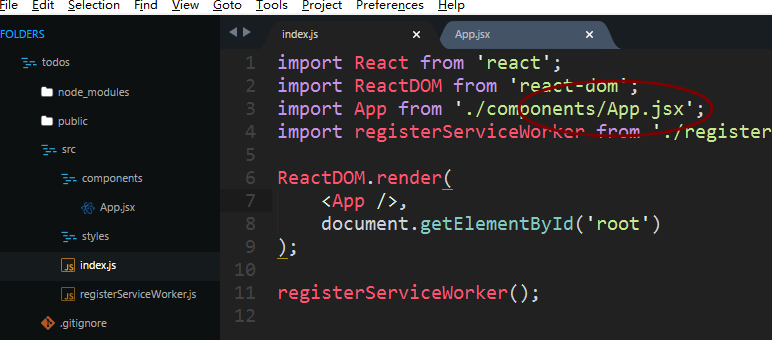
### 改造项目结构

去掉不需要的文件，改造如下：



在react开发时，组件的后缀名可以是js，也可以是jsx。

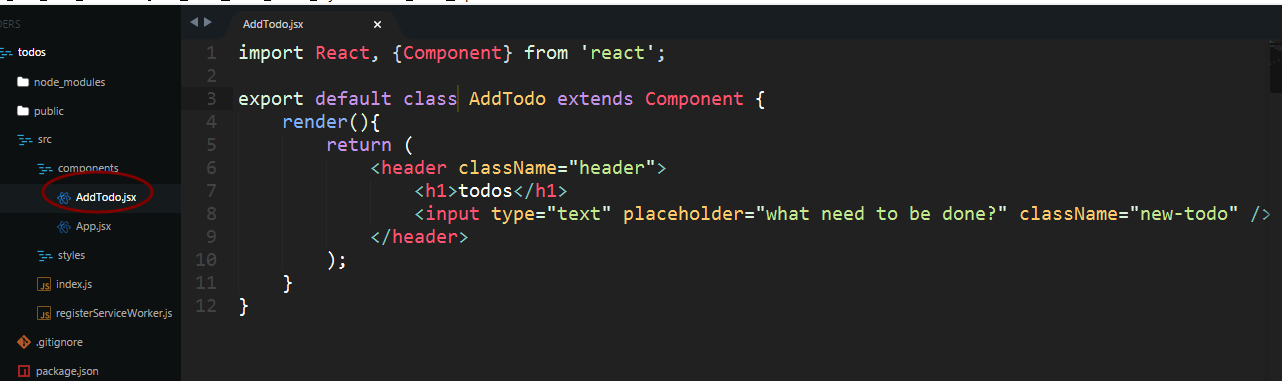
如果是jsx的话，在引入的时候，需要写上后缀。



然后需要分别定义组件并显示了。

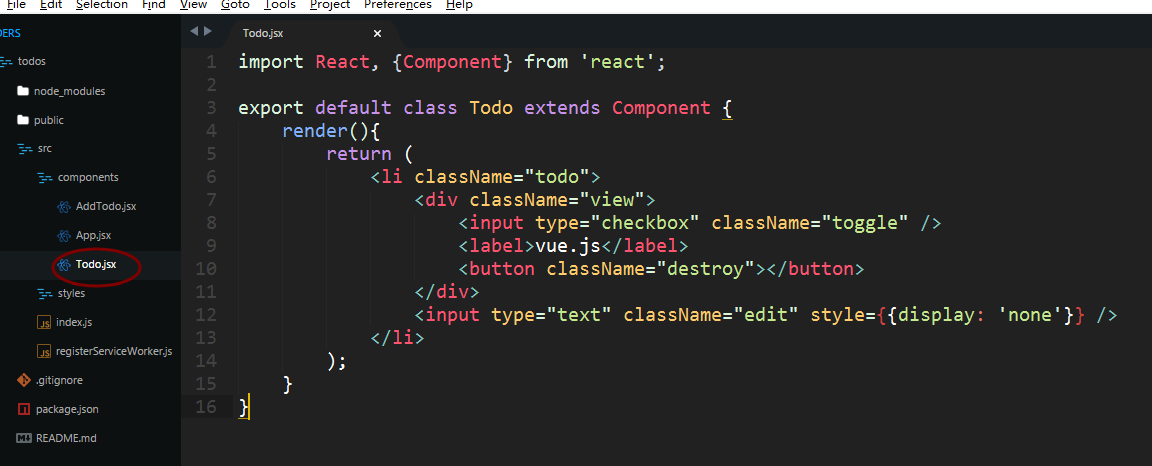
### 定义AddTodo组件

定义如下：



注意，在开发react的时候，通常分号还是需要加上的。（和vue进行对比）

### 定义Todo组件

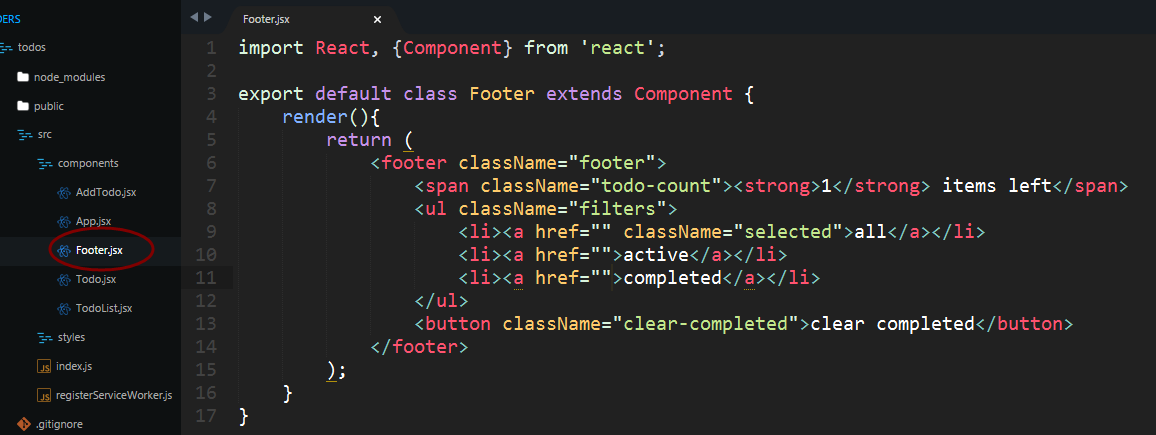


### 定义TodoList组件

定义如下：



### 定义Footer组件

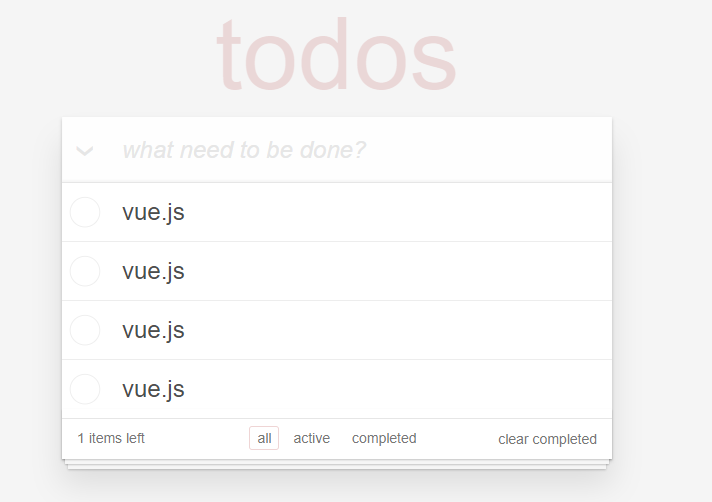


### 定义App组件

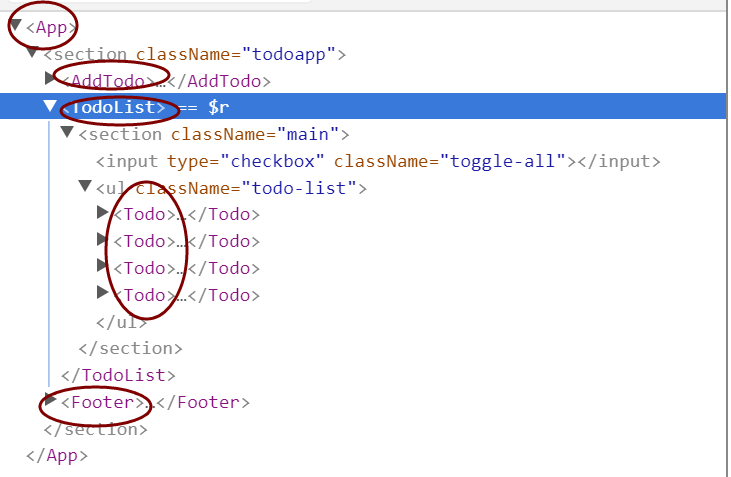
将AddTodo组件、TodoList组件和Footer组件组合，形成App组件，如下：



显示如下：



可以通过react devtools来查看组件的结构，如下：

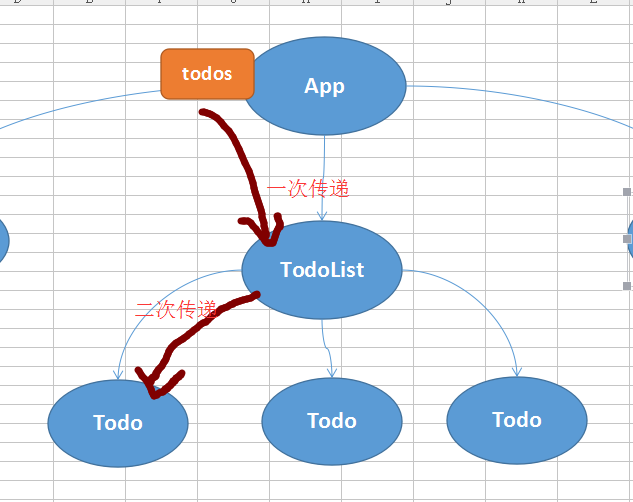


## 显示todos列表

### 分析

Todo任务最终是Todo组件来显示的，但是todos状态是定义在App中的。

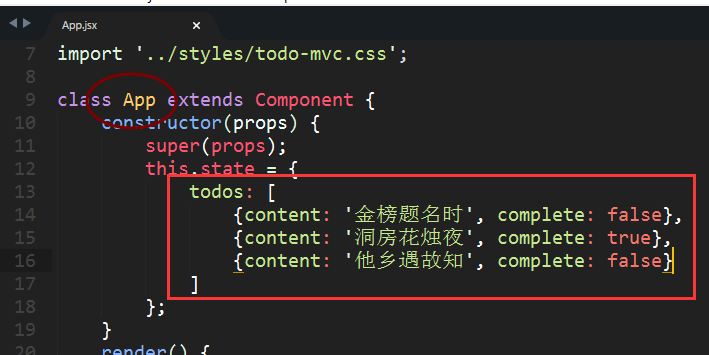
所以，需要进行两次属性传递，如图：



步骤如下：

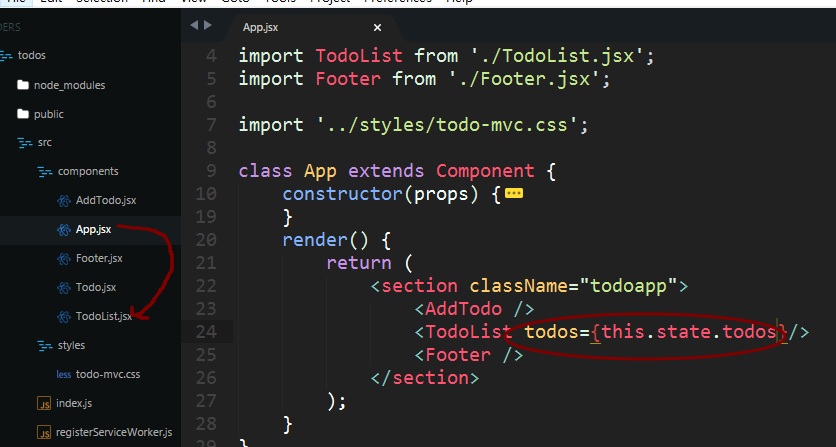
* 在App组件初始化状态
* 传递属性
* 在Todo组件中使用属性来输出

### 在App组件初始化状态

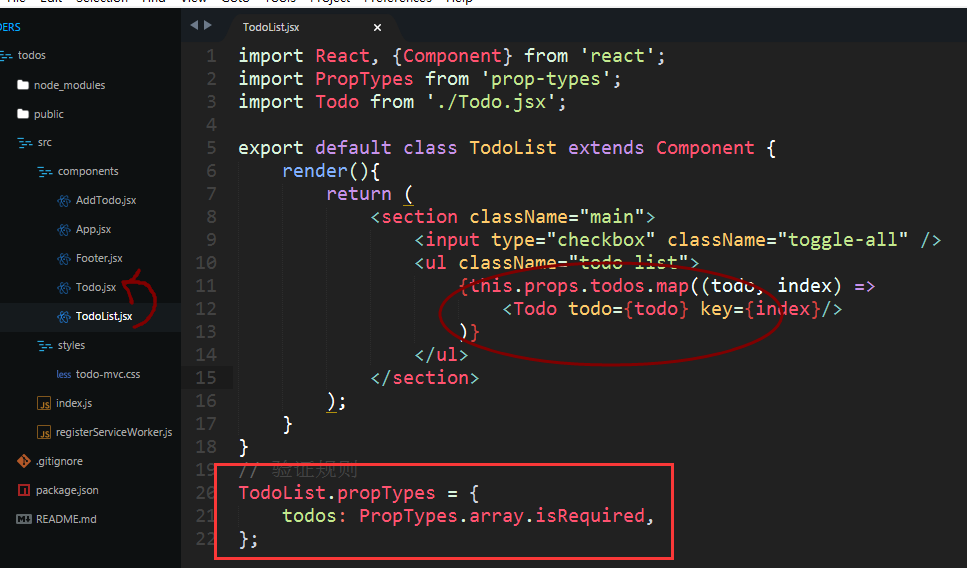


### 传递属性

第一次传递，如下：



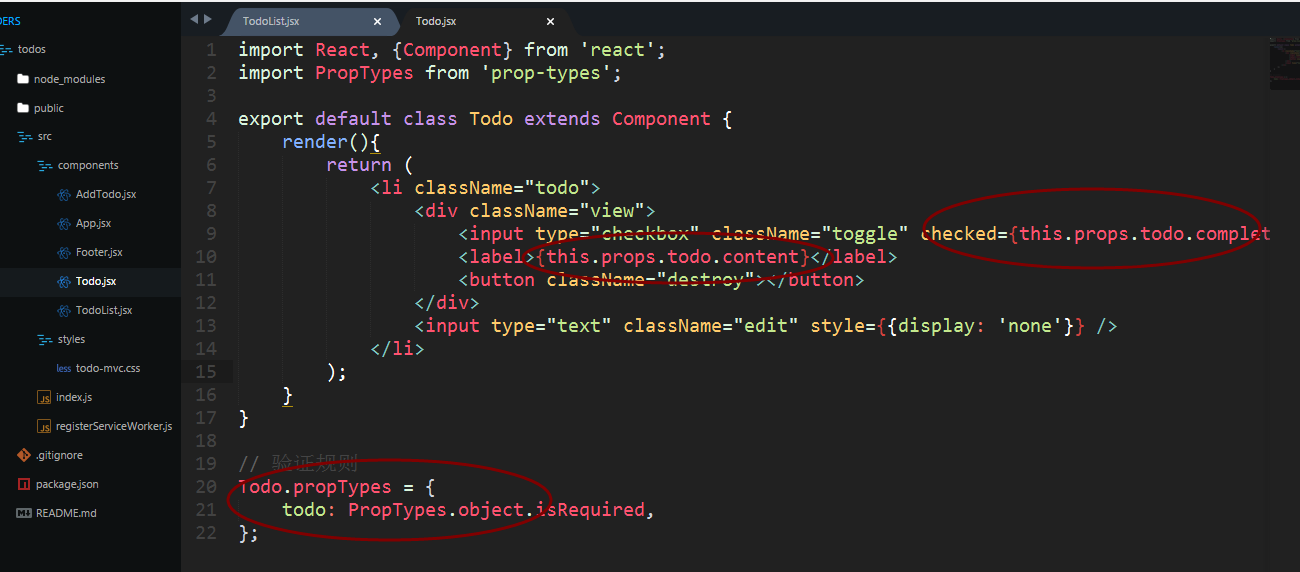
第二次属性传递，如下：



### 在Todo组件中使用属性来输出

在Todo组件中，就可以使用todo属性了。

编写代码如下：



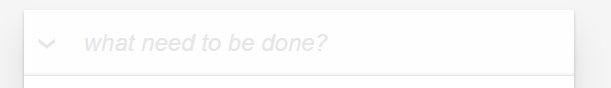
显示如下：



## 添加todo任务

### 目标

在input中，输入内容之后，点击enter键，就需要完成添加，如图：



### 分析

要点：

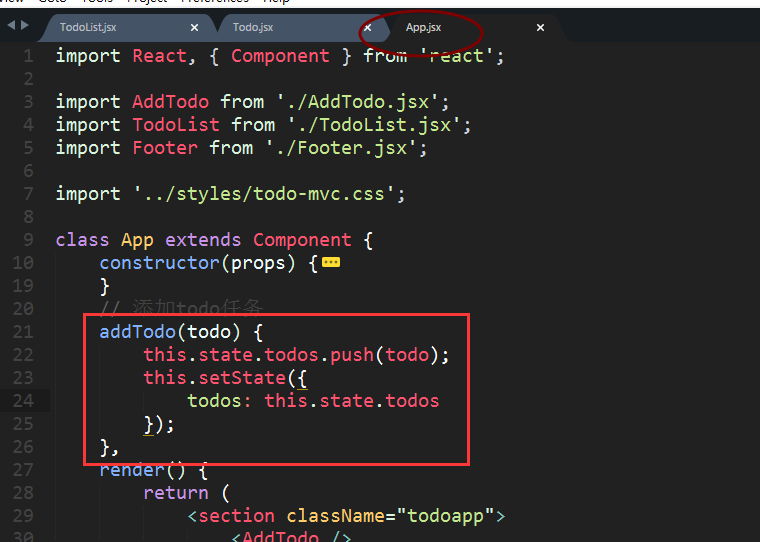
* 肯定要使用keyup事件，需要判断enter被按下了（keyCode === 13）
* 不能在子组件中修改父组件的状态，必须要传递修改的方法给子组件，如图：



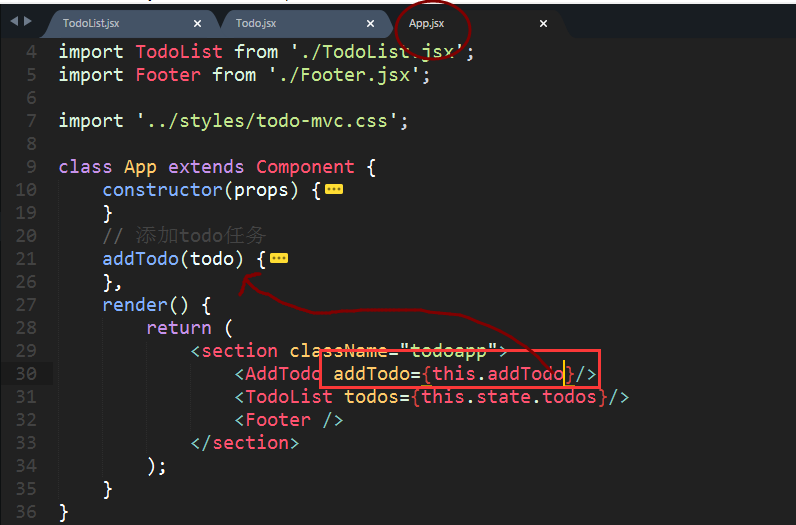
实现步骤如下：

* 在App组件中定义方法实现添加todo
* 将todo方法传递到AddTodo组件中
* 在AddTodo组件中，需要注册keyup事件，调用传递过来的添加方法实现添加功能

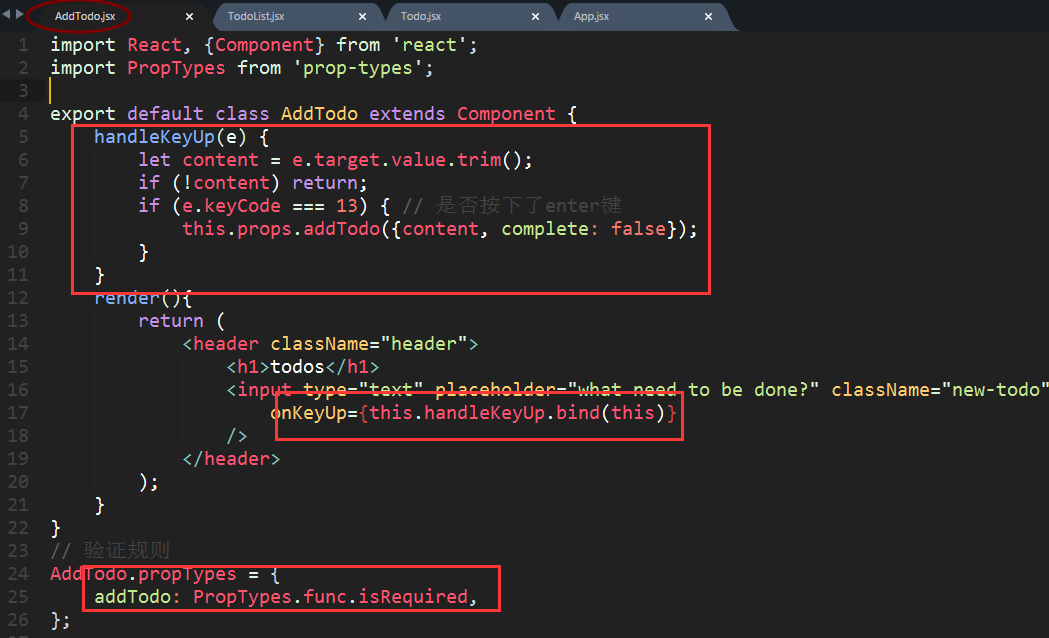
### 在App组件中定义方法实现添加todo



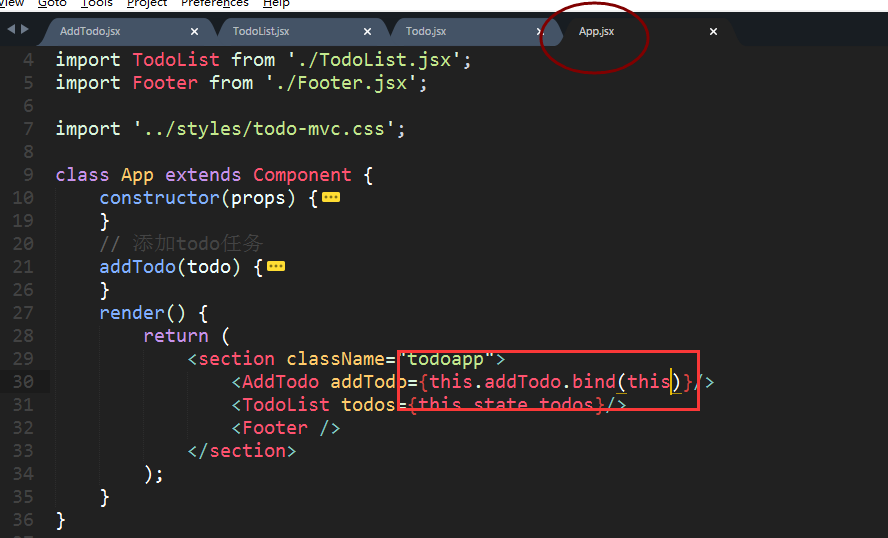
### 将todo方法传递到AddTodo组件中



### 在AddTodo组件中，注册事件，完成添加



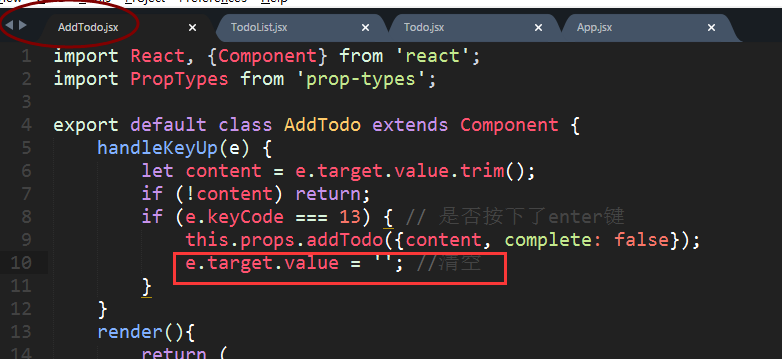
注意，同时需要在App组件传递属性时，绑定this，如下：



测试，如下：



添加完成之后，需要清空文本框，添加代码如下：



## 删除指定todo任务

### 目标

在每个todo项上，有一个x的按钮，

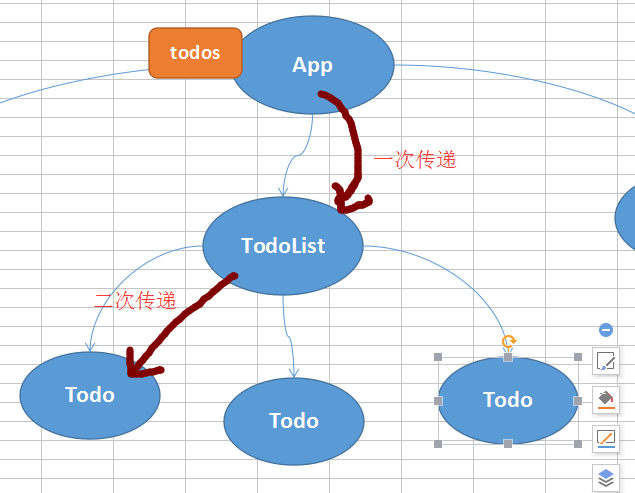


鼠标放到该todo项，就显示。

点击就应该删除该todo项

### 分析

需要在App组件中定义删除的方法，然后传递到Todo组件，如图：



步骤：

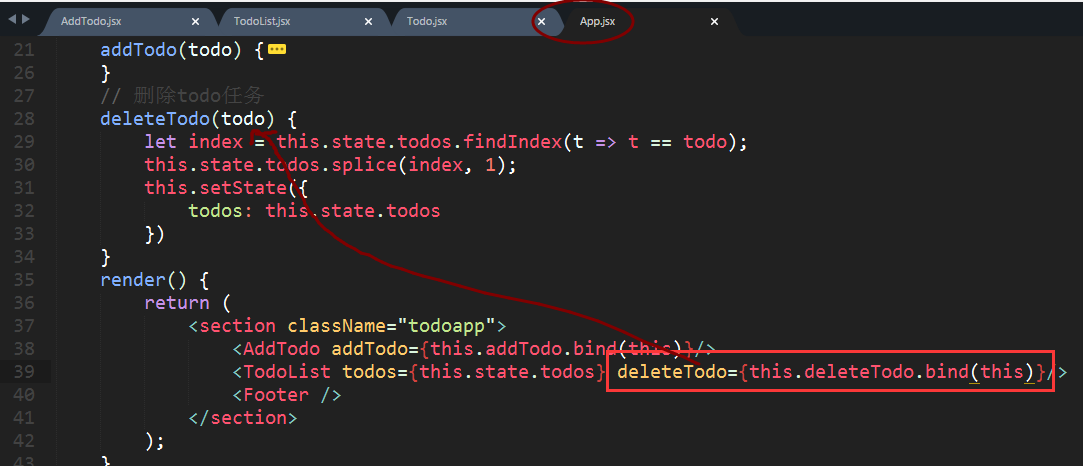
* 在App组件中定义删除方法
* 传递方法
* 在Todo组件中注册时间，调用方法删除之

### 在App组件中定义删除方法

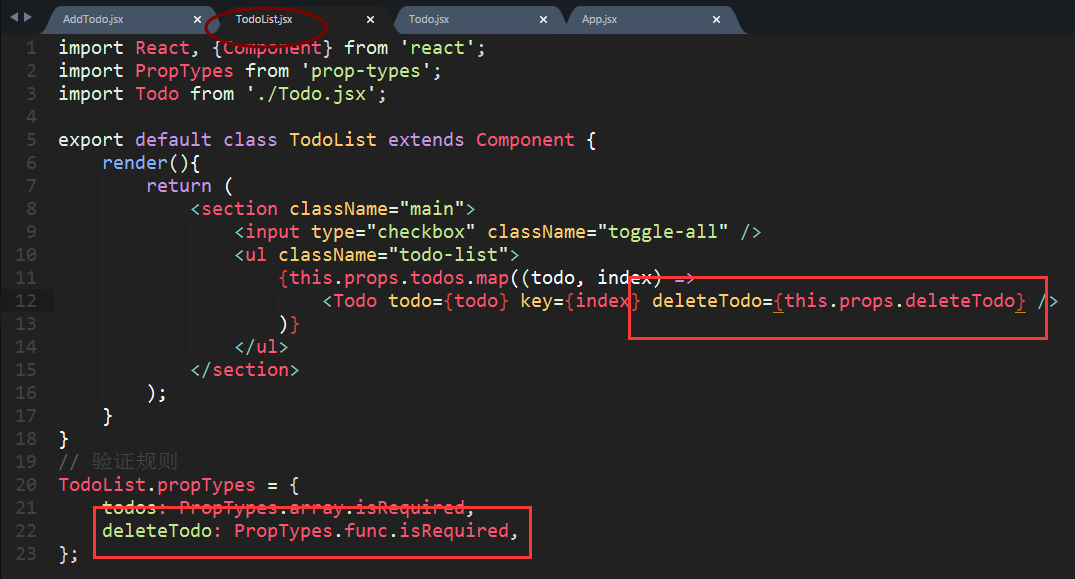


### 传递方法

第一次绑定，如下：

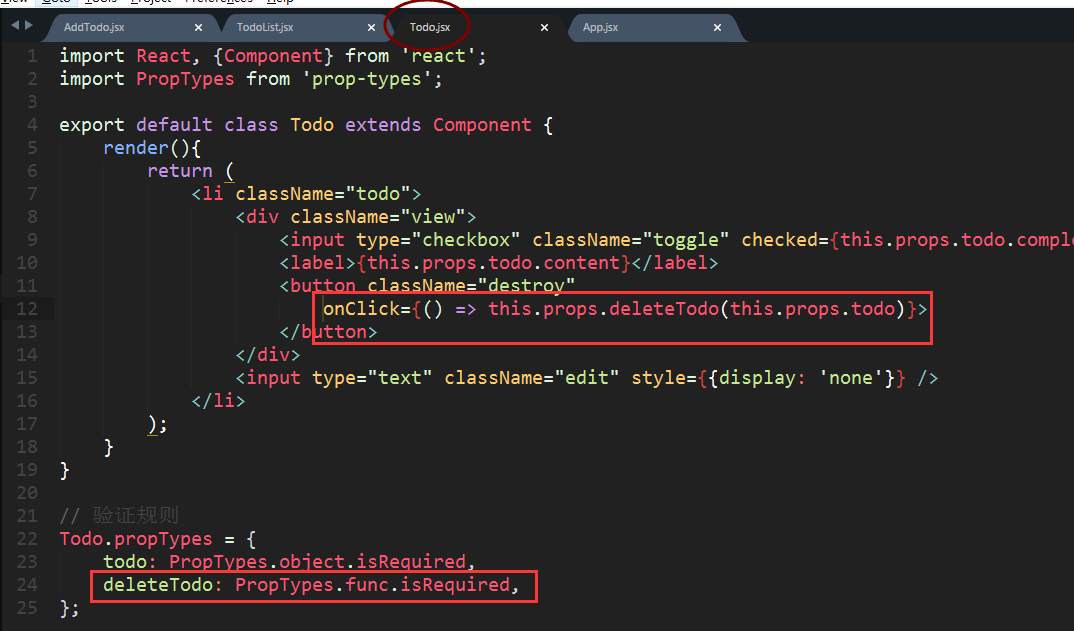


第二次绑定，如下：



### 在Todo组件中注册时间，调用方法删除之

编写代码如下：



测试，ok。

需要注意一点：**传递的方法如果是在事件中进行调用的。在定义方法的组件中，传递属性时，也需要使用bind来绑定一下。**

## 切换指定todo状态

### 目标

在每一个todo中，都有一个checkbox，可用于切换todo的状态

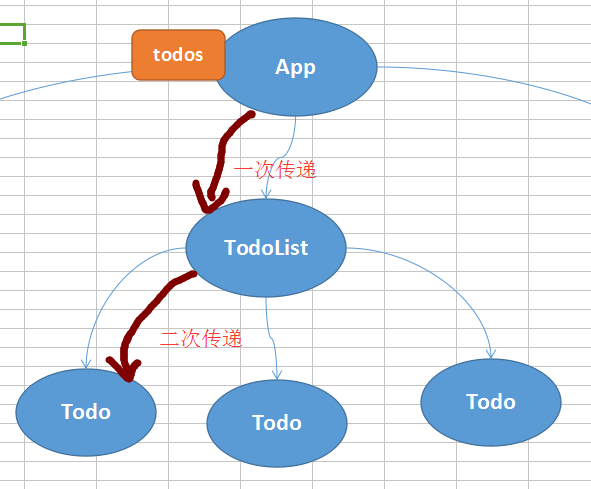


如果是已完成，样式如下：



### 分析

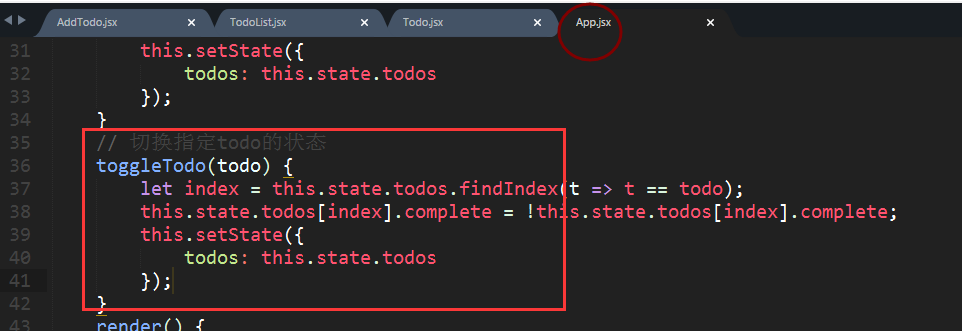
同样需要进行两次传递。



步骤如下：

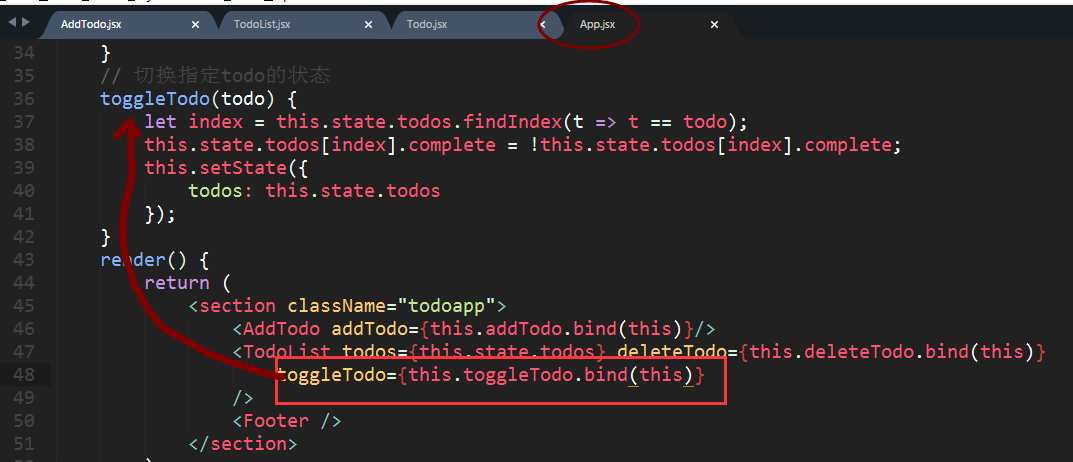
* 在App组件中定义方法，用于切换指定todo状态
* 传递方法
* 在Todo组件中注册事件，调用方法完成切换

### 在App组件中定义方法，用于切换指定todo状态

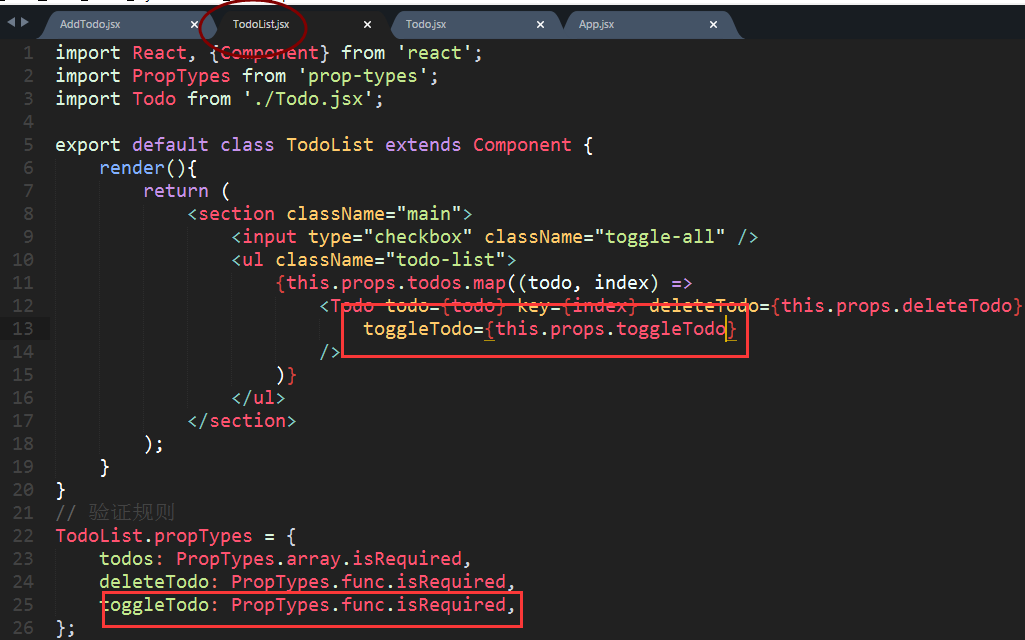


### 传递方法

第一次传递

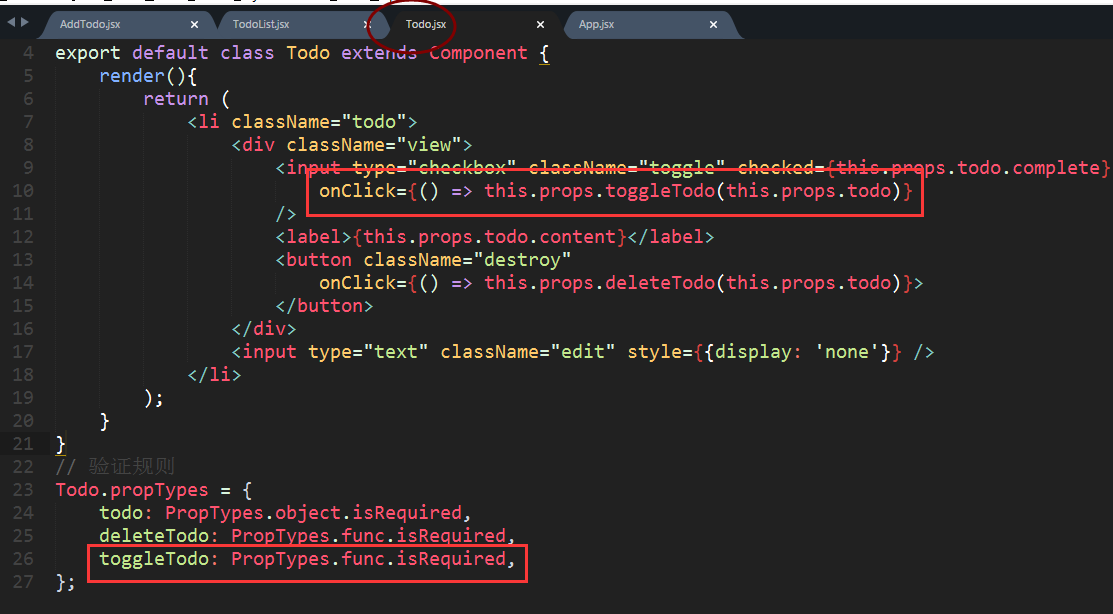


第二次传递



### 在Todo组件中注册事件，调用方法完成切换

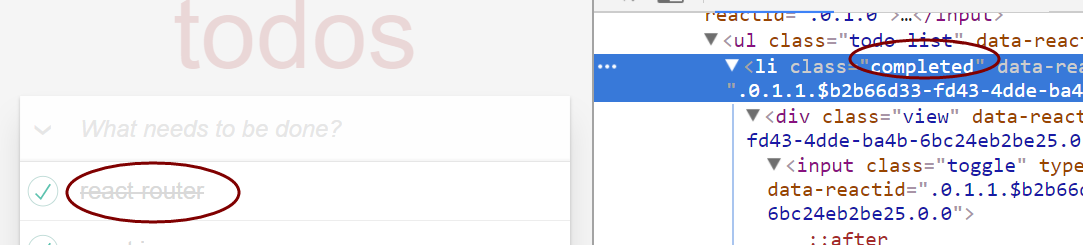
编写代码如下：



测试，ok。

### 添加样式

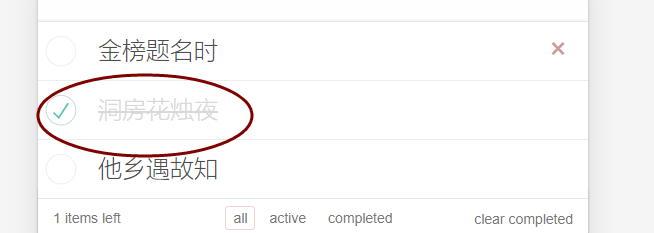
已完成的todo，是有一个删除线的。应该是有一个类的。如图：



所以，需要根据条件来显示是否有completed类



测试，如下：



## 批量切换todos状态

### 目标

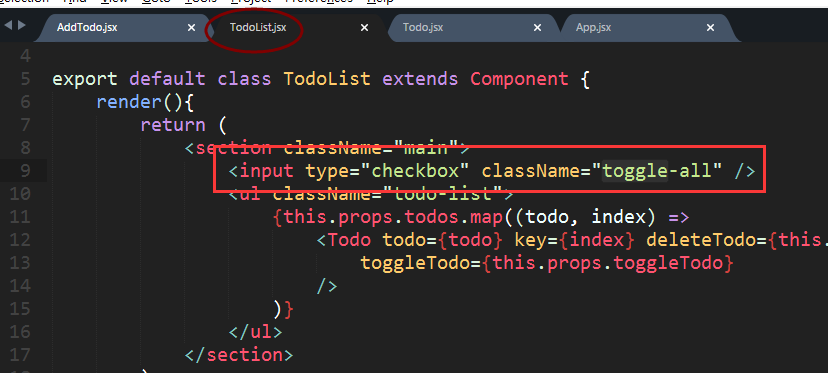
在文本框的左边，有一个全选的按钮，如图：



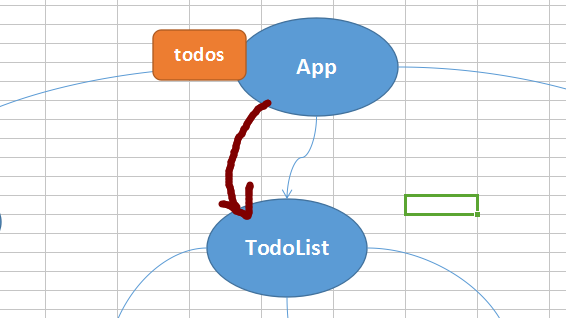
这里可以批量的切换todos任务的状态。

### 分析

注意，这个操作的checkbox，不在AddTodo组件中，是TodoList组件中的



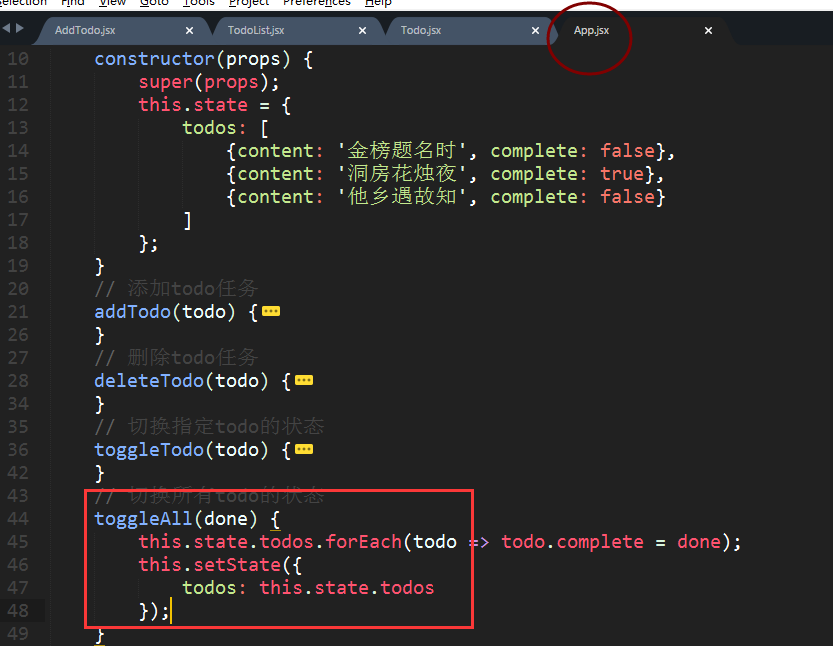
所以，需要从App组件向TodoList组件传递方法，如图：



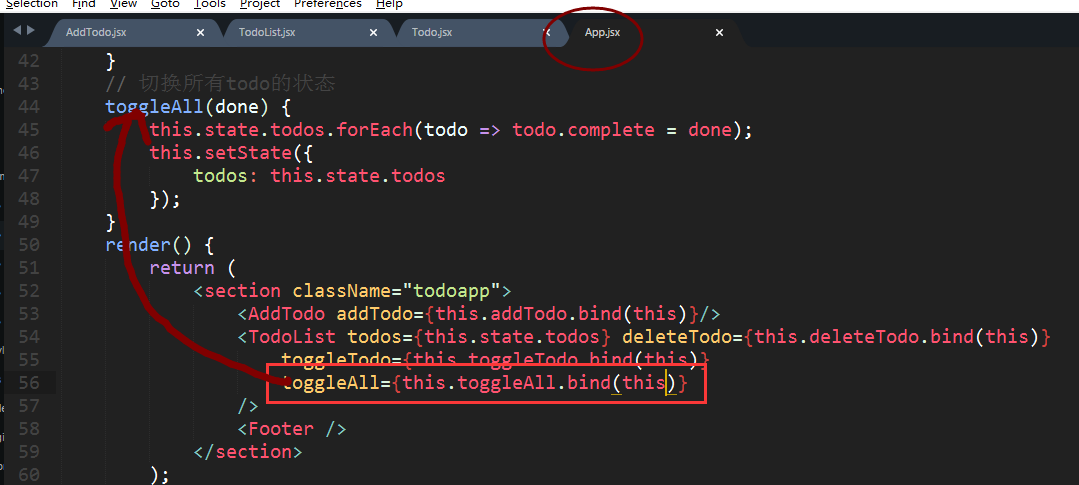
步骤：

* 在App组件中定义方法实现批量切换状态
* 传递方法
* 在TodoList组件中注册事件，调用方法，完成批量切换

### 在App组件中定义方法实现批量切换状态

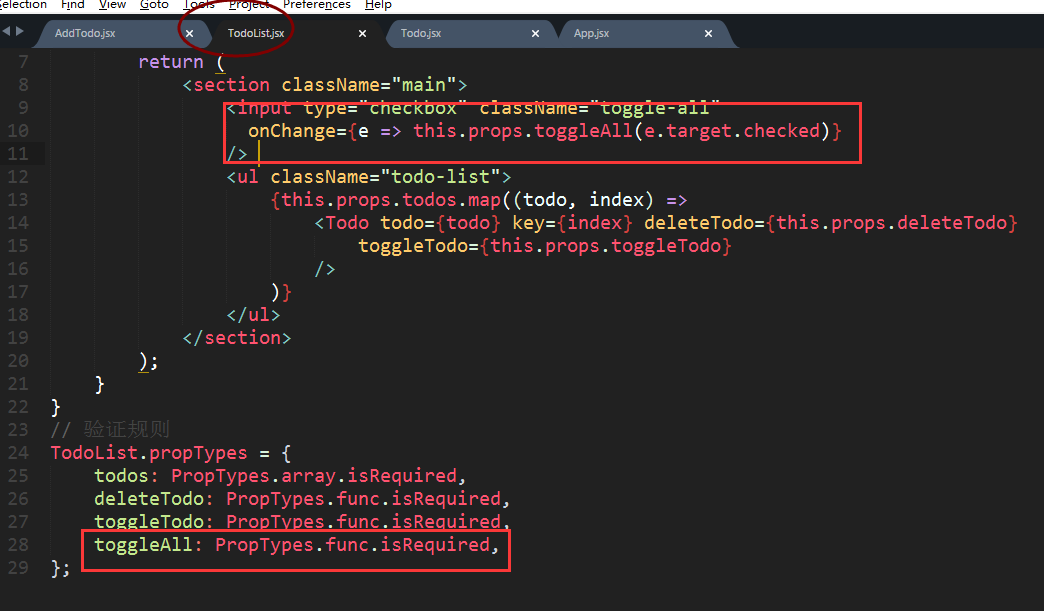


### 传递方法

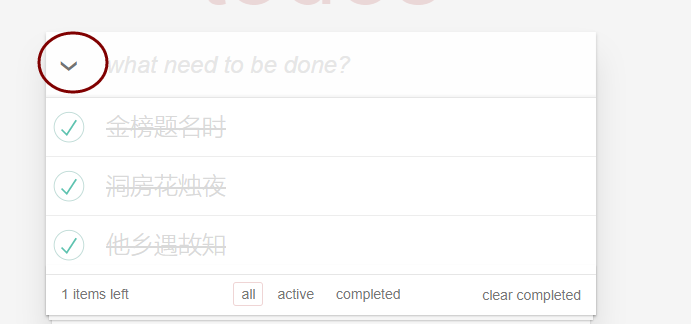


### 在TodoList组件中注册事件，调用方法，完成批量切换

编写代码如下：



测试，如下：

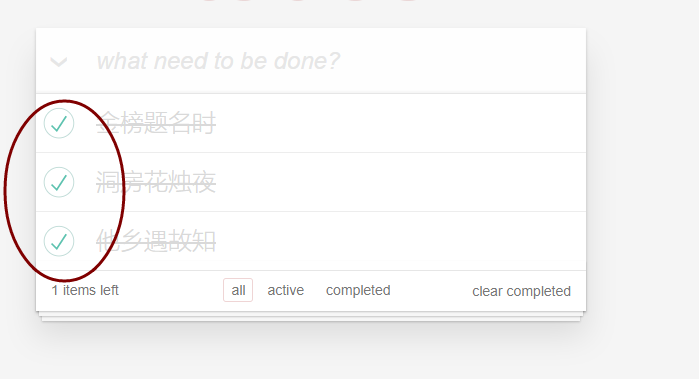




### 完善checkbox的同步

实际上，这个checkbox的全选按钮，同时也受下面的todo项的影响。

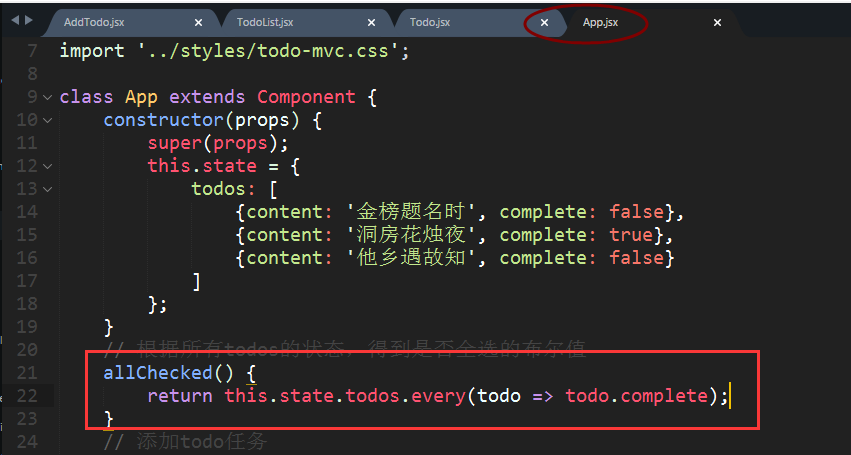
当前这个是有问题的，没有同步。如图：



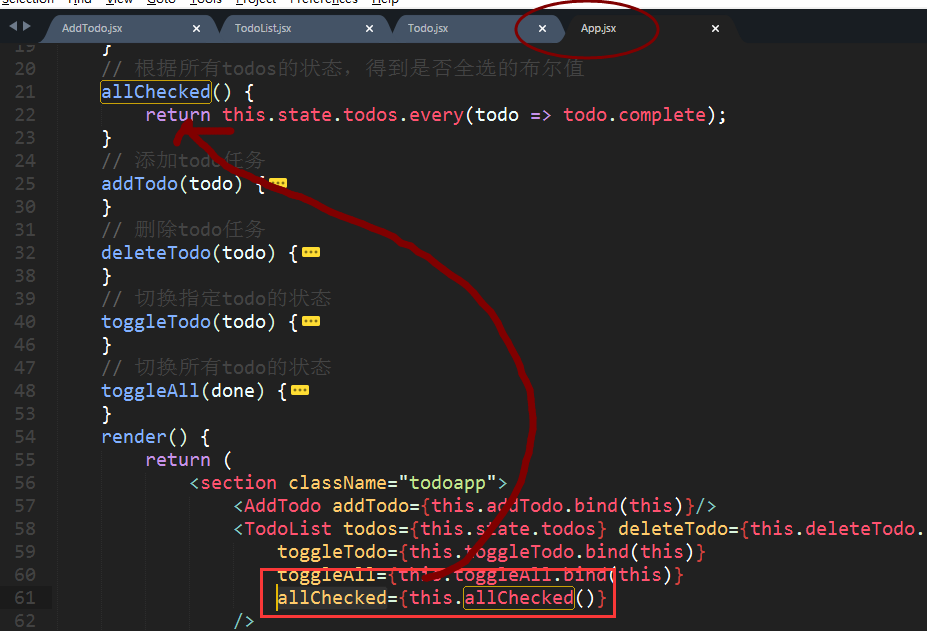
所以，这个全选按钮的状态，必须由所有todos的状态来决定。

需要根据当前的todos，得到一个是否全选的布尔值，然后传递到TodoList组件即可。

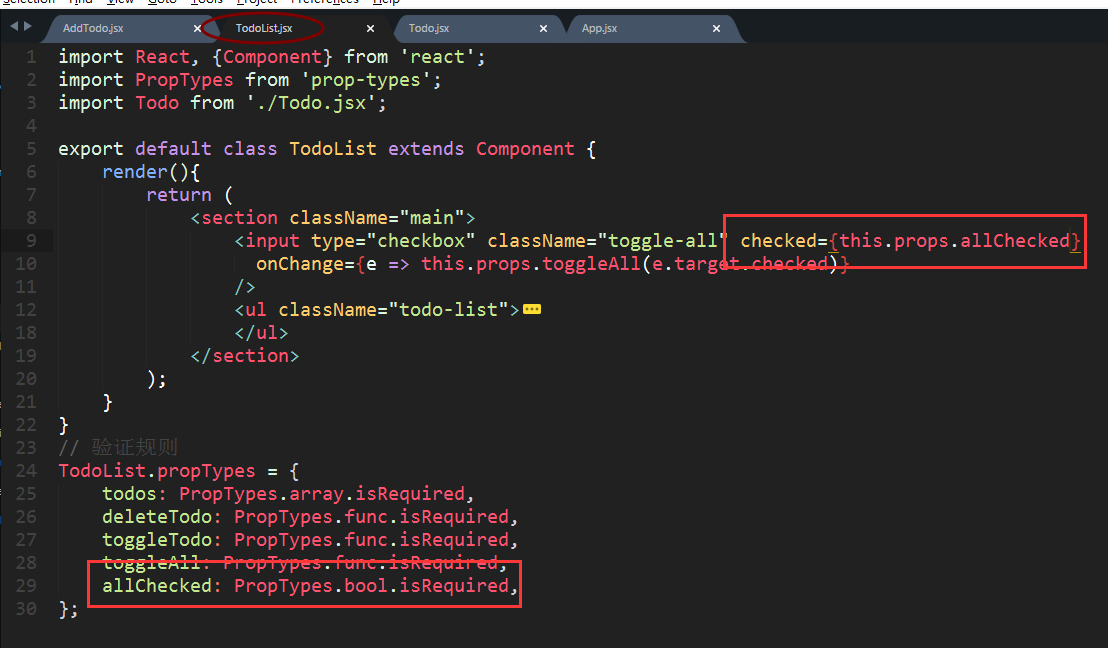
**第一步，定义方法如下：**



**第二步，调用并传递给TodoList组件**



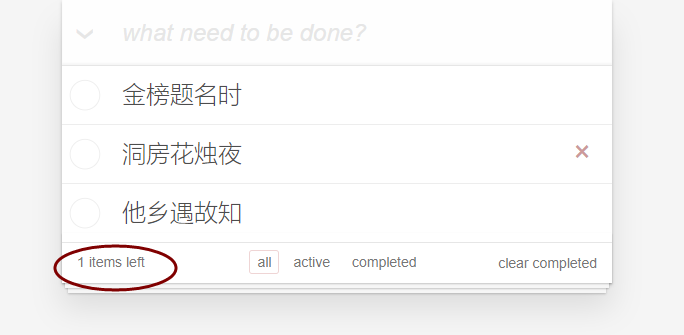
第三步，在TodoList组件中使用allChcked属性，如下：



测试，ok。

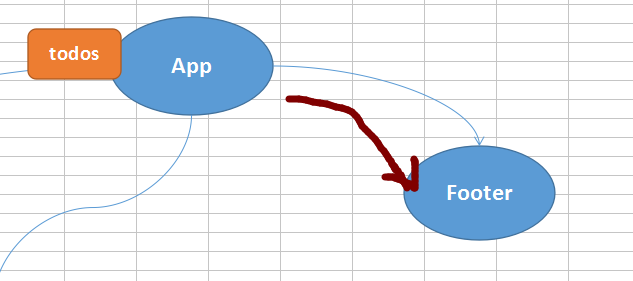
## 统计未完成的任务

### 目标



### 分析

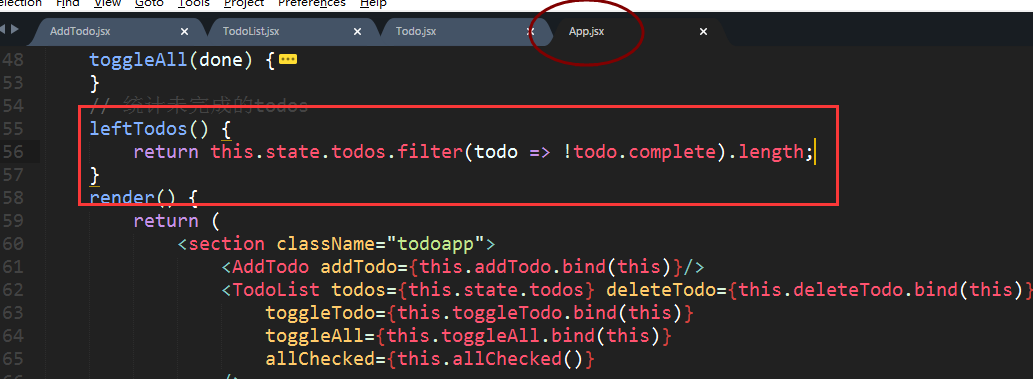
只需要一次传递即可，如图：



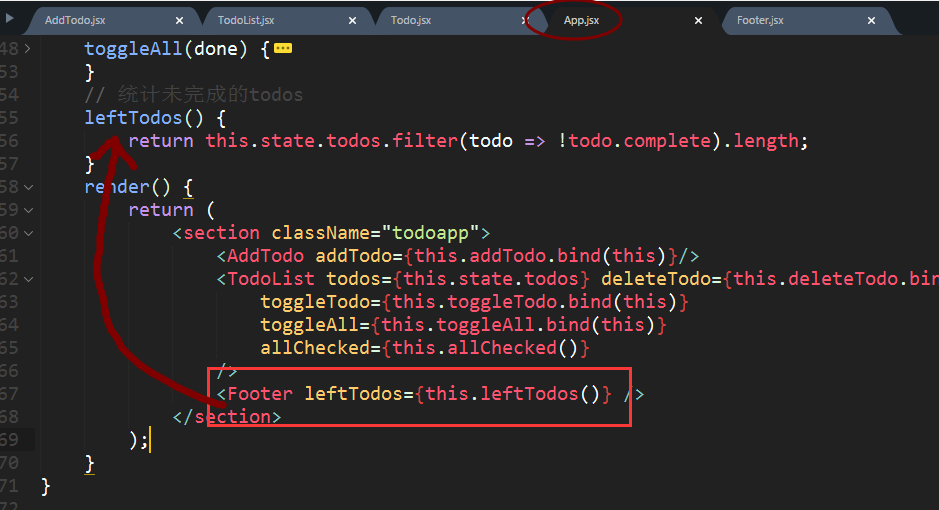
步骤：

* 在App组件中定义一个方法，用于统计未完成的todos
* 传递属性
* 在Footer组件中使用属性输出

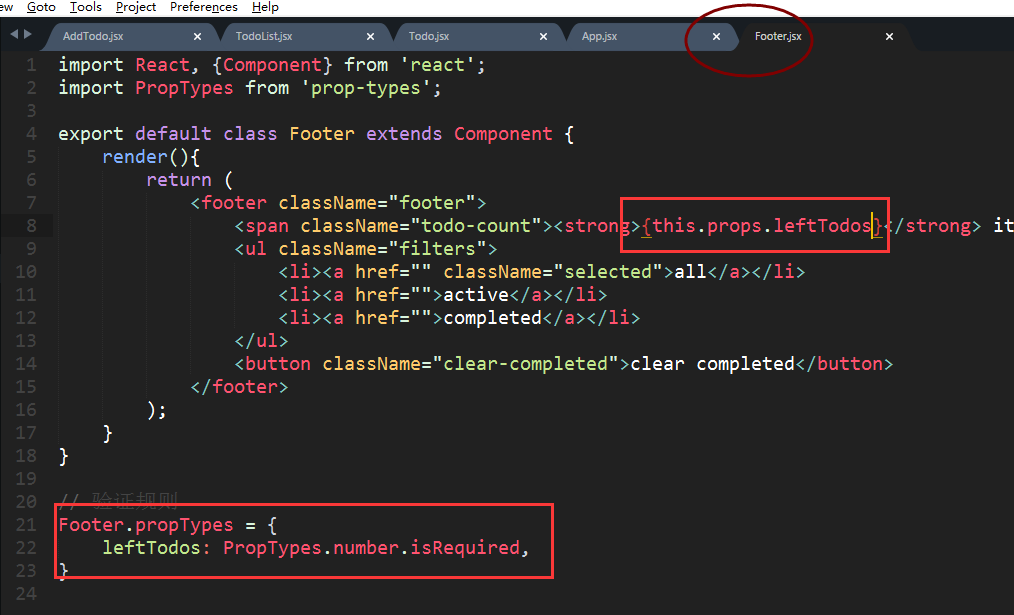
### 在App组件中定义一个方法，用于统计未完成的todos



### 传递属性



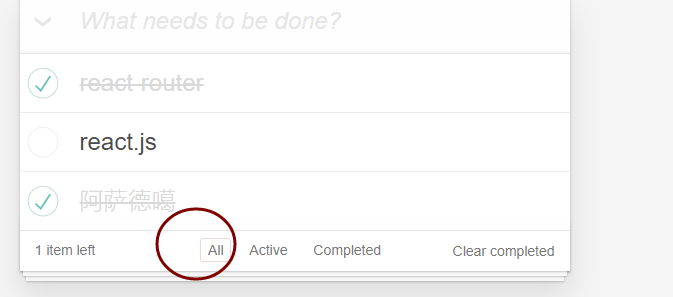
### 在Footer组件中使用属性输出

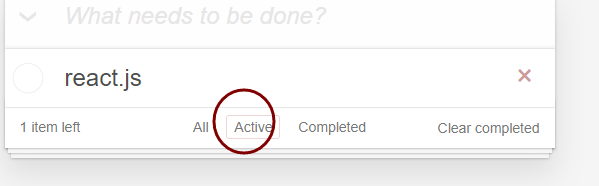


测试，ok。

## 设置可见性并显示对应的任务

### 目标







### 分析

在这个功能中，除了todos状态之外，还需要使用另外一个状态---visibility。

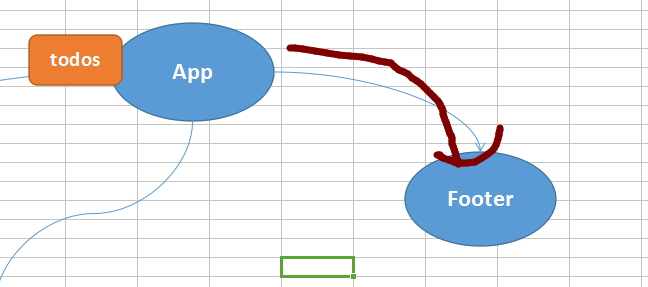
其中，这个visibility的取值有如下三个：

* all
* active
* completed

最终显示的todos列表，是由todos和visibility共同来决定的。

在单击超链接的时候，可以修改visibility的值。

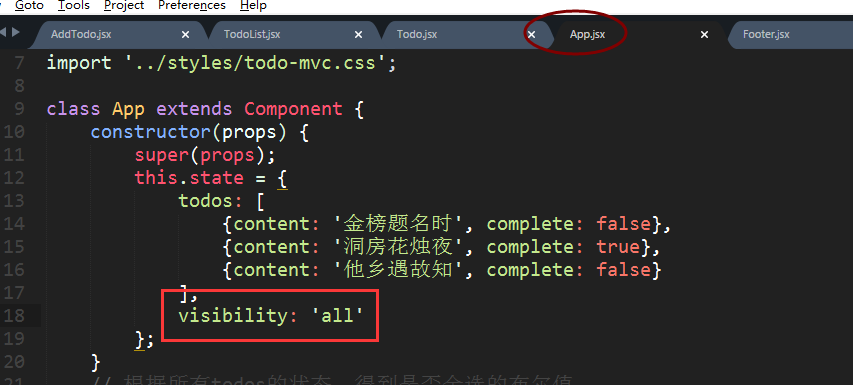
还是需要进行一次属性传递，如图：



步骤：

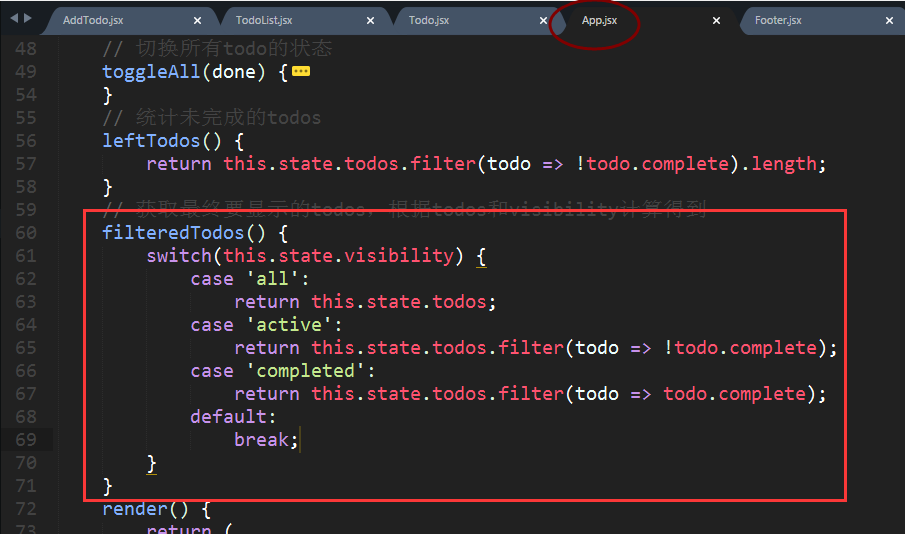
* 初始化visibility状态
* 根据visibility和todos，获取要显示的todos任务
* 需要定义方法修改visibility
* 将修改visibility的方法传递给Footer组件
* 在Footer组件中，注册事件，调用方法修改visibility

### 初始化visibility状态



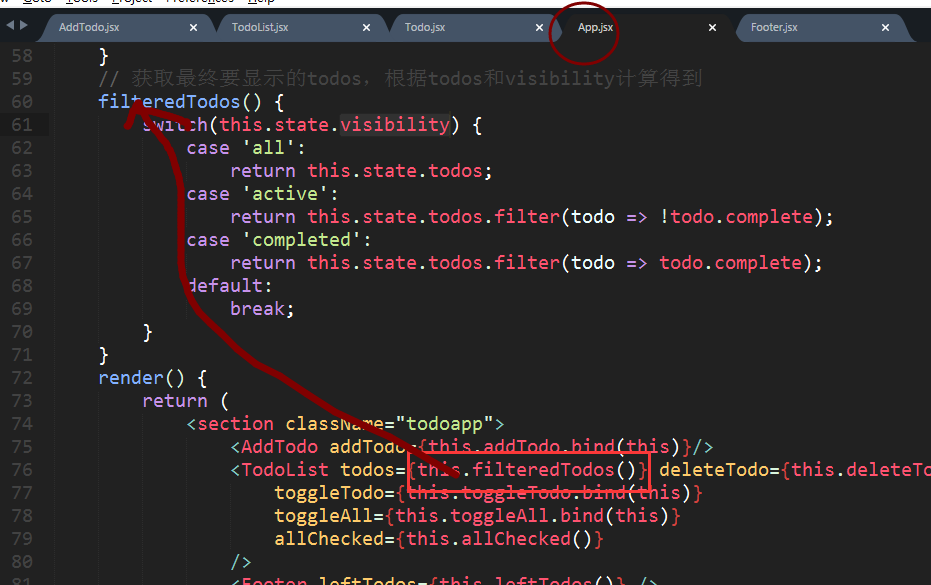
### 根据visibility和todos，获取要显示的todos任务

定义一个方法如下：



这就好比是vue中的计算属性。

然后调用该方法，获取要显示的todos，传递给TodoList组件，如下：



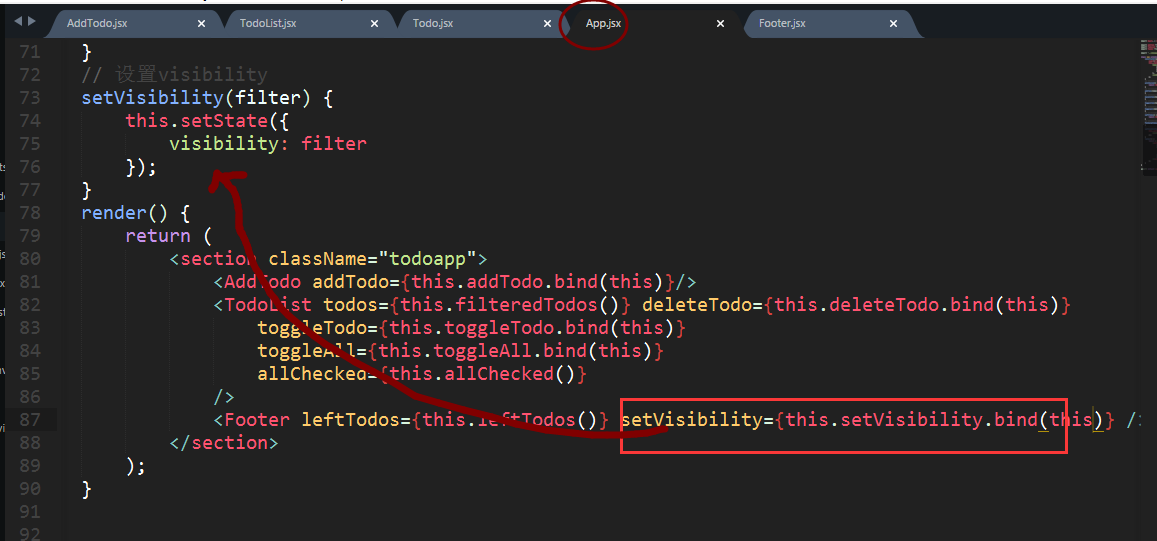
测试，ok

提示：可以手动修改visibility的值，测试结果。

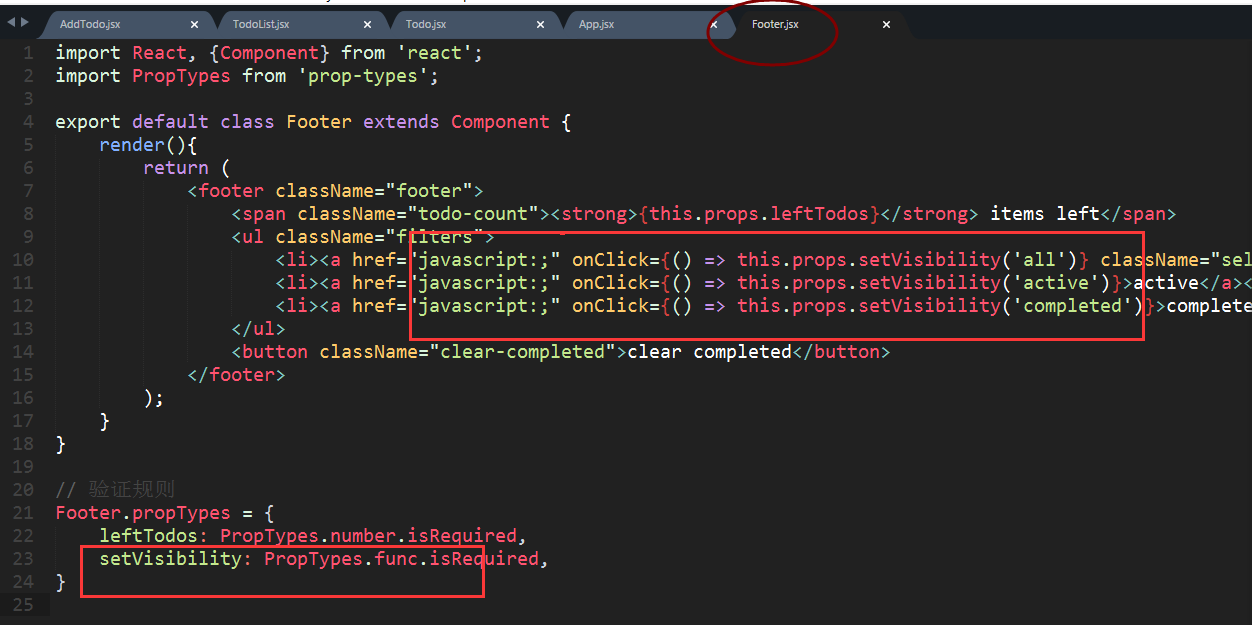
### 需要定义方法修改visibility



### 将修改visibility的方法传递给Footer组件



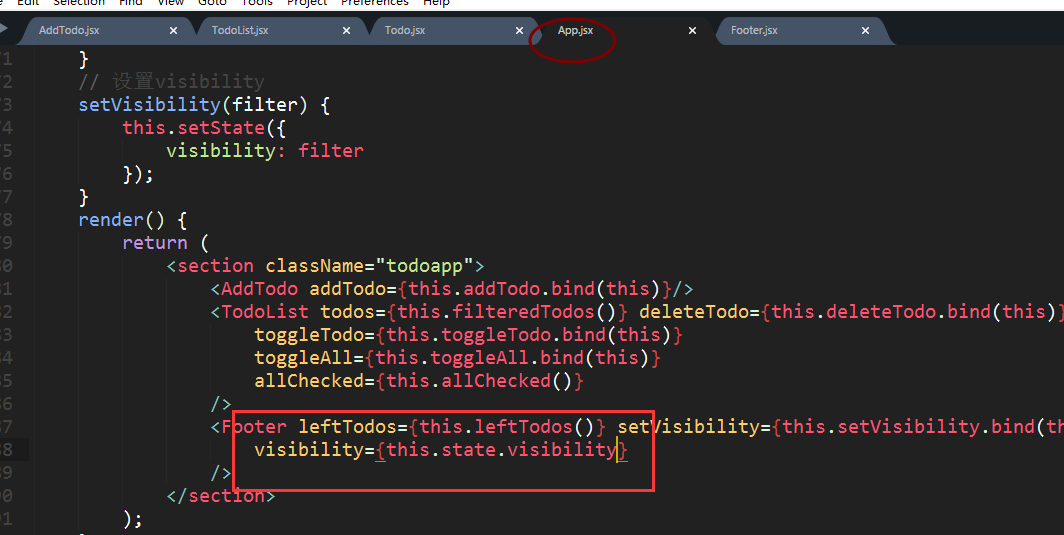
### 在Footer组件中，注册事件，调用方法修改visibility



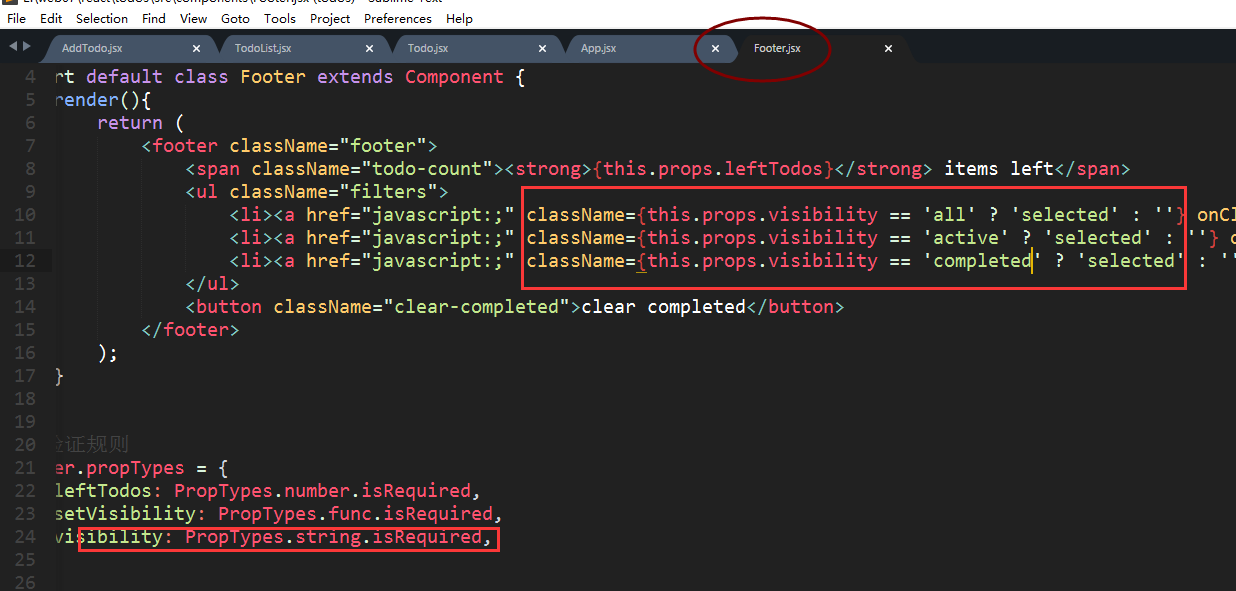
测试，ok。

### 完善-样式的同步

将visibility的值传递给Footer组件，根据visibility来进行判断即可。



然后在Footer组件中，使用属性，进行判断即可，

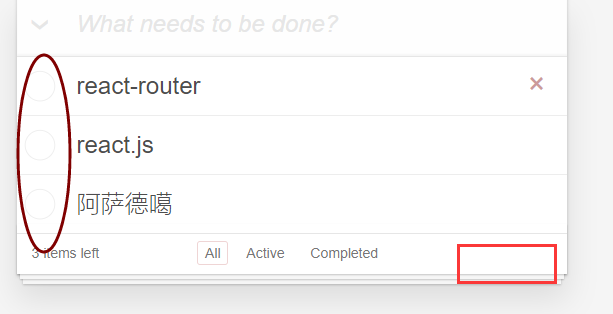


测试，ok。

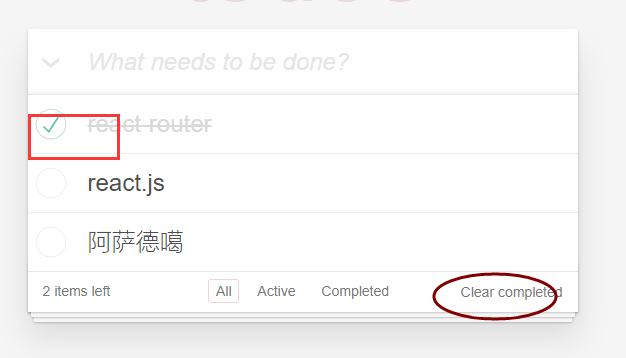
## 清除已完成的todos

### 目标

如果没有已完成的任务，就不显示这个清除的按钮

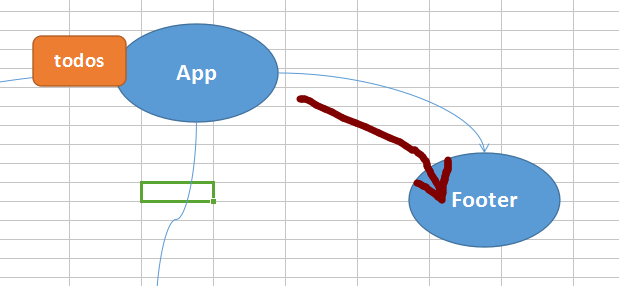


当至少有一个是已完成的，就显示



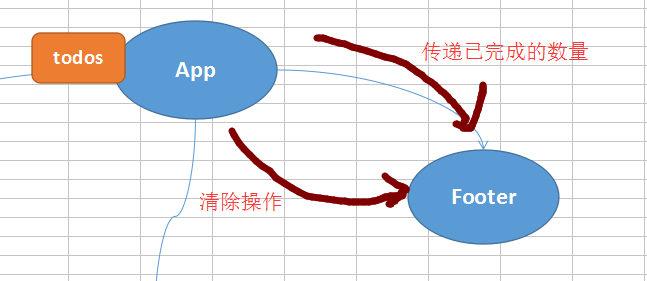
### 分析

针对清除按钮的显示，需要获取已完成的todos任务数量。需要在App组件中获取，然后传递给Footer组件，如图：



针对清除操作，需要在App组件中定义好方法，然后传递给Footer组件，在Footer组件中注册事件并调用。

所以，此处需要传递一个数值和一个方法，如图：

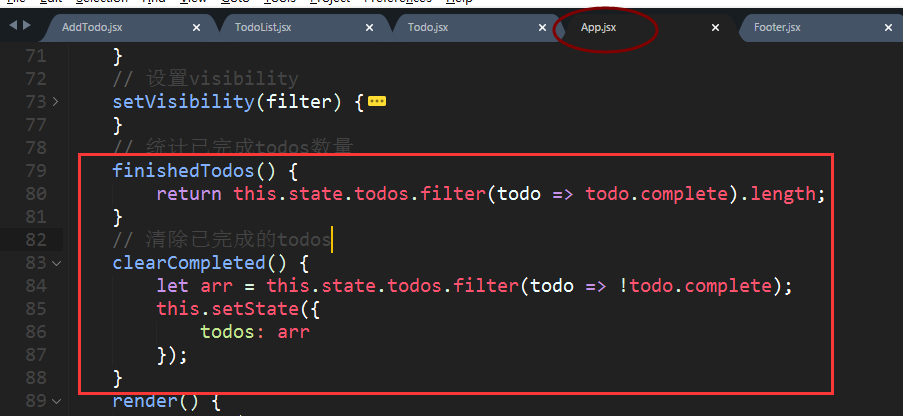


步骤如下：

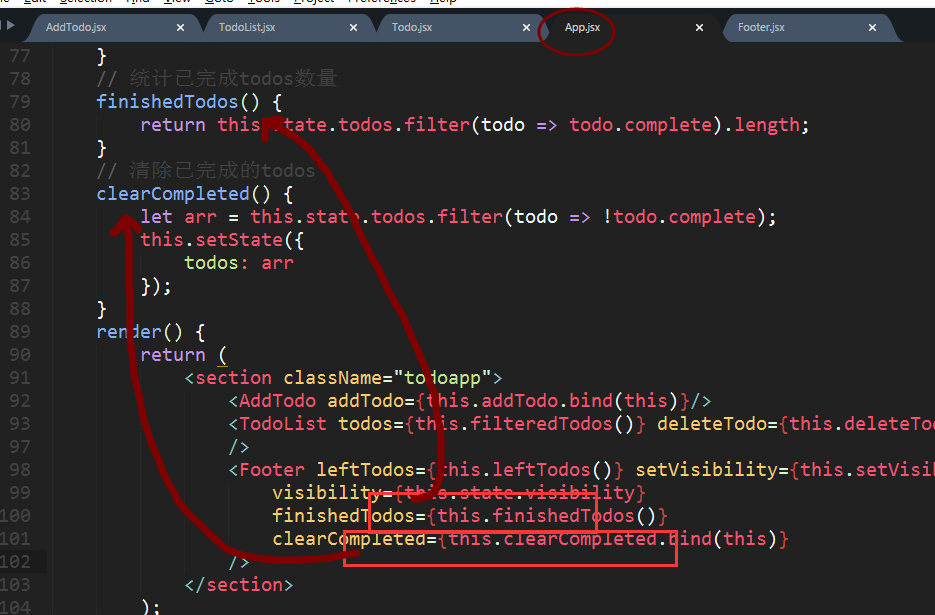
* 在App组件中，定义两个方法，获取已完成的数量及清除已完成的todos
* 进行属性传递
* 在Footer组件中根据已完成数量决定是否显示清除按钮
* 在Footer组件中注册事件，调用方法，完成清除

### 在App组件中，定义两个方法，获取已完成的数量及清除已完成的todos

定义方法如下：

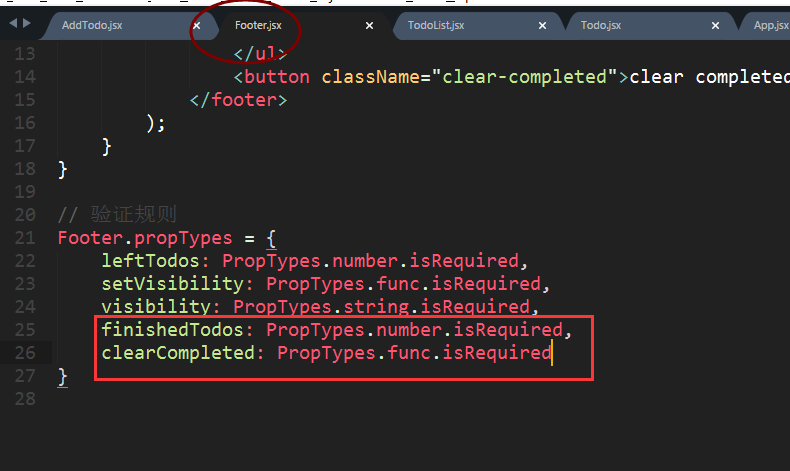


### 进行属性传递

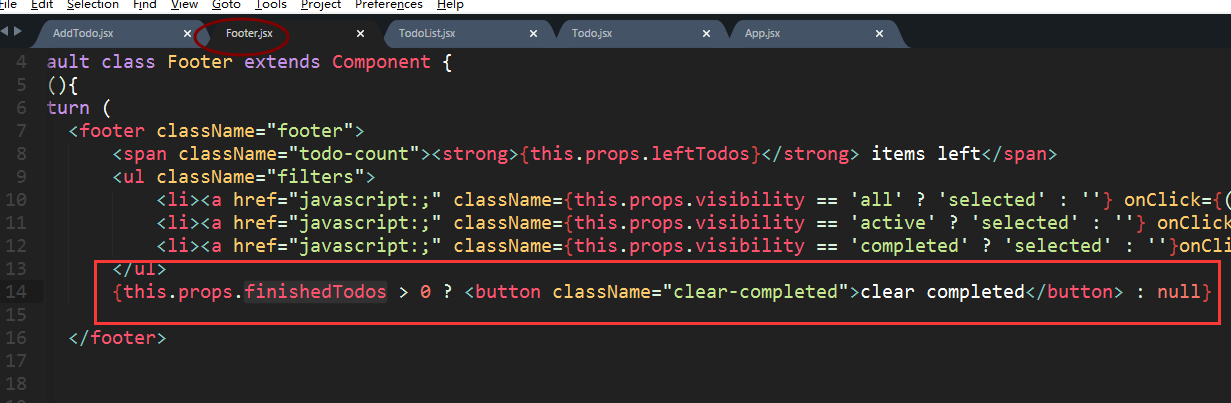


### 在Footer组件中根据已完成数量决定是否显示清除按钮

首先，在Footer组件中定义好验证规则，如下：



然后根据条件来显示button，如下：



测试，ok。

### 在Footer组件中注册事件，调用方法，完成清除

代码如下：

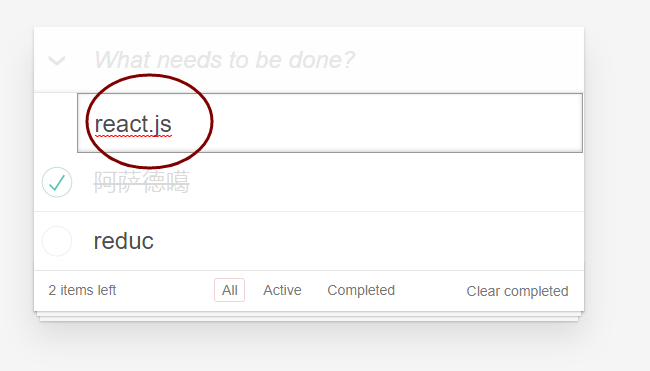


测试，ok。

## 编辑todo（难点）

### 目标

针对todo项，可以双击，进入编辑状态



可以在编辑状态下面修改todo内容。

修改之后，有三种操作：

* 按下enter键，完成修改
* 按下 esc键，放弃修改
* 失去焦点，完成修改

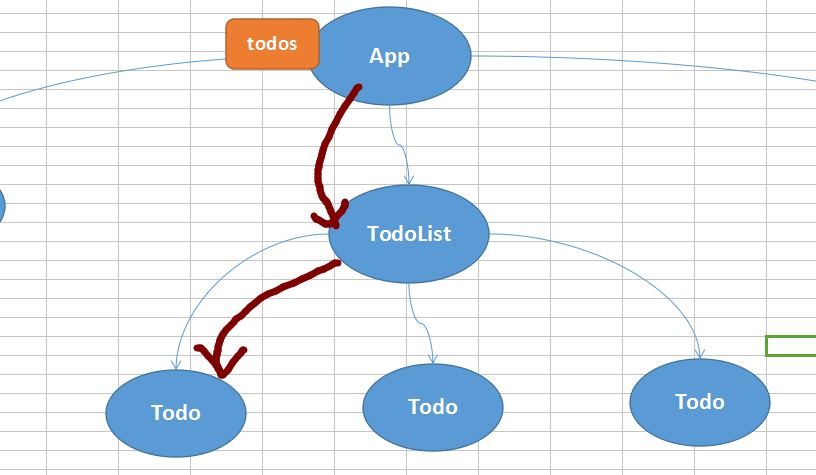
在修改的时候，有一个特殊情况，如果将todo内容清空了，实际上进行的是一个删除操作。

### 分析

第一点，非常关键，由于每个todo项的编辑状态是独立的，需要给todo单独的定义一个状态---**editing**，来表示是否处于编辑。

第二点，针对enter和esc键的按下，需要注册keyUp事件，针对失去焦点，需要注册blur事件。

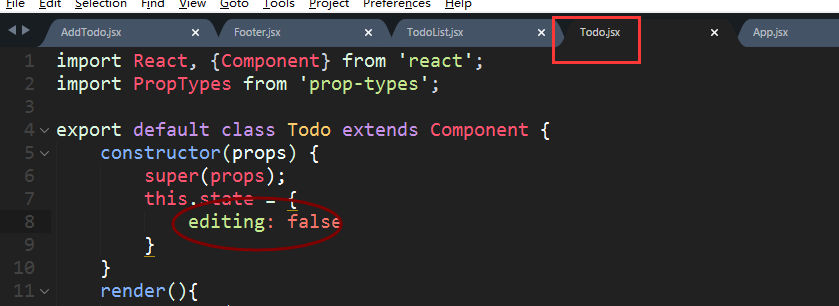
第三点，需要在App组件中定义一个更新指定的todo项，然后传递到Todo组件，需要两次传递，如图：



步骤：

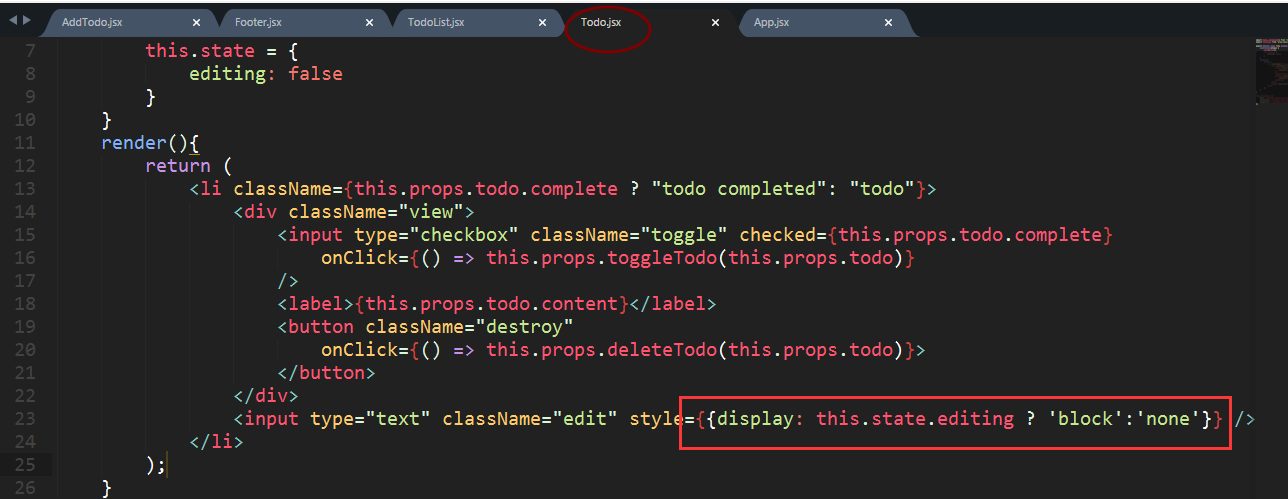
* 给Todo定义状态
* 需要根据状态来决定是否处于编辑状态
* 给label注册dlbClick事件，修改状态
* 在app组件中定义更新todo的方法
* 传递方法
* 在Todo组件中对input注册keyUp事件，完成操作
* 在Todo组件中对input注册blur事件，完成操作

### 给Todo定义状态



### 需要editing来决定是否处于编辑状态

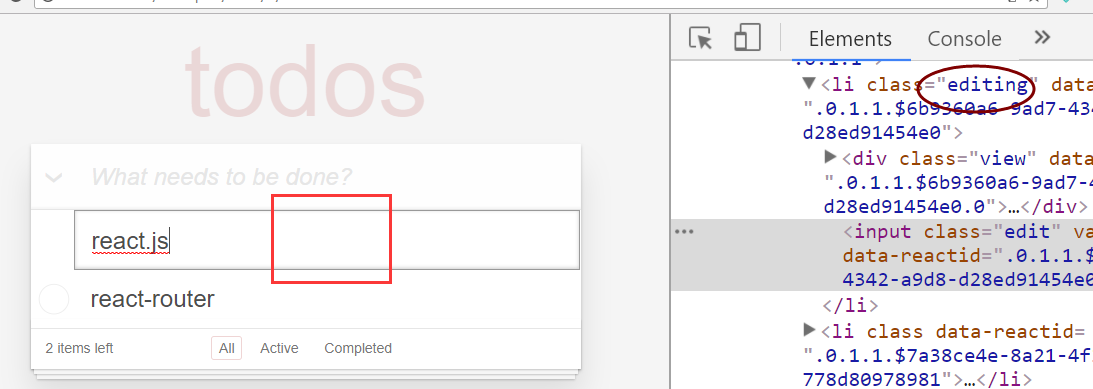
编写代码如下：



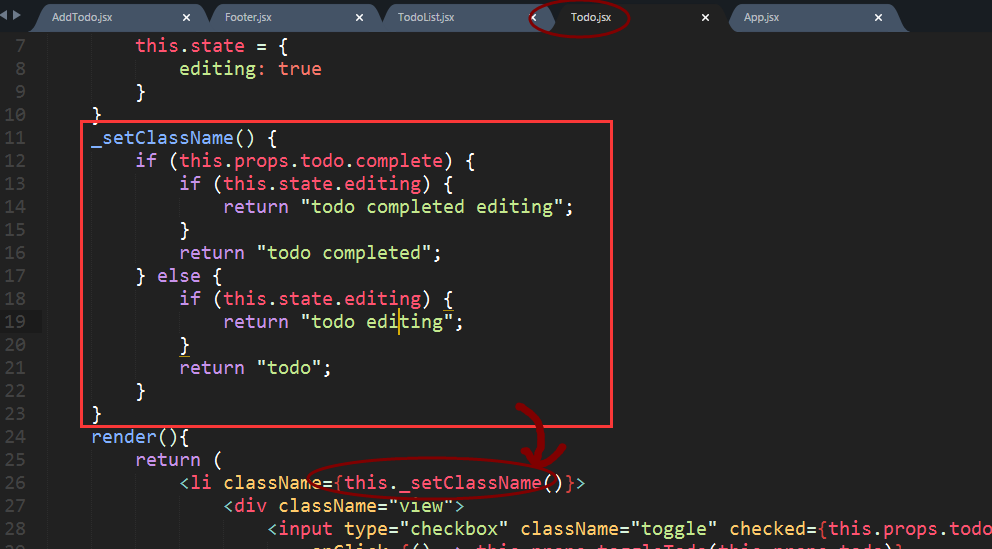
测试，如下：



发现，文本框没有盖住label。需要增加一个类

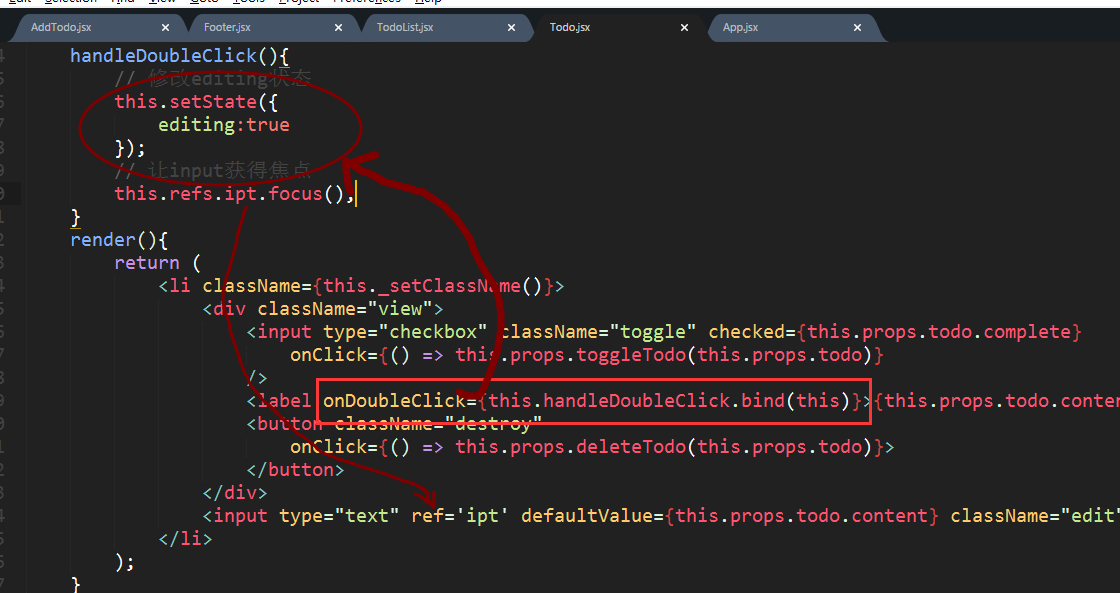


需要给li动态的设置editing 类，如下：



### 给label注册dlbClick事件，修改状态

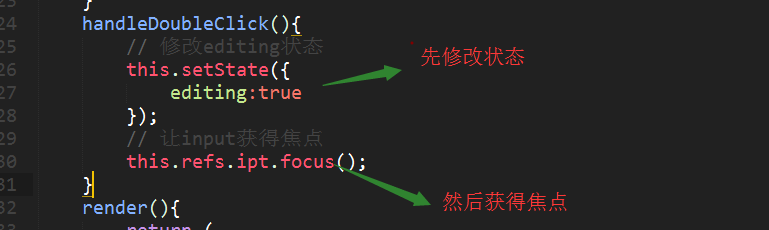
编写代码如下：



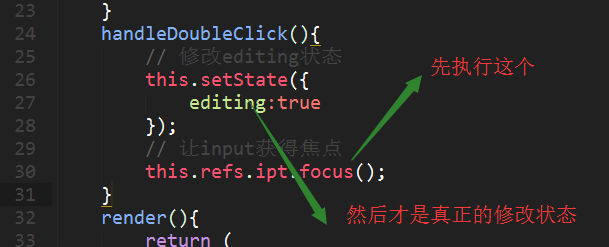
测试，发现可以变为编辑状态，但是并没有获得焦点。

原因是由于setState的异步特性。

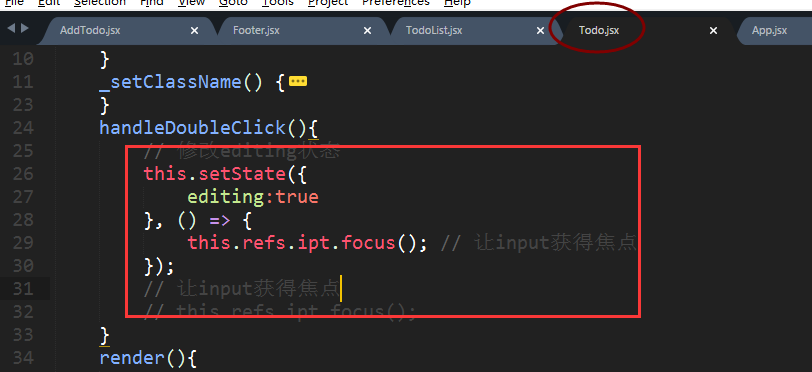
我们是这么想的：



代码却是这样执行的



需要使用setState方法的第二个参数，是一个回调函数，意思是指状态修改之后，需要执行的代码，修改如下：



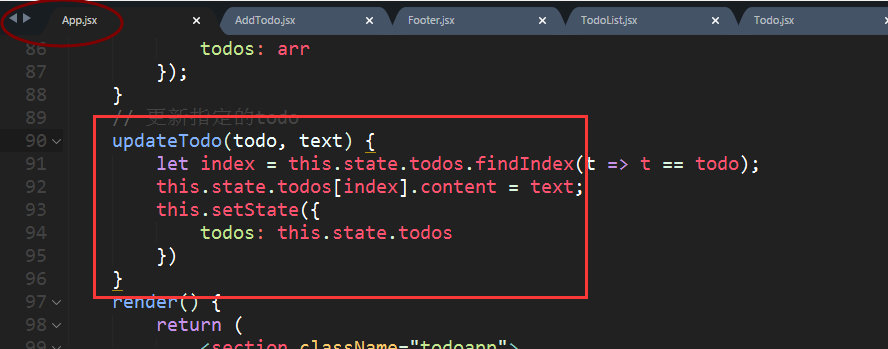
测试，ok。

### 在app组件中定义更新todo的方法

针对更新，需要两个参数：

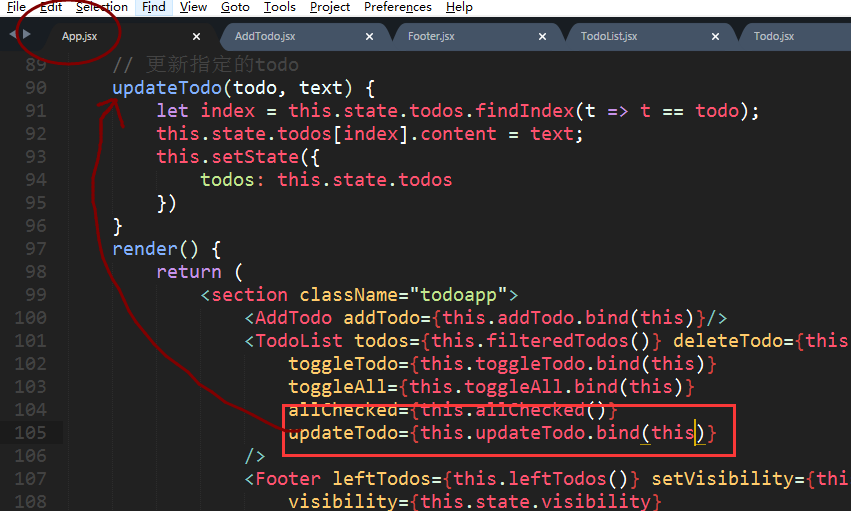
* 当前todo对象
* text文本内容

在App组件中定义方法如下：

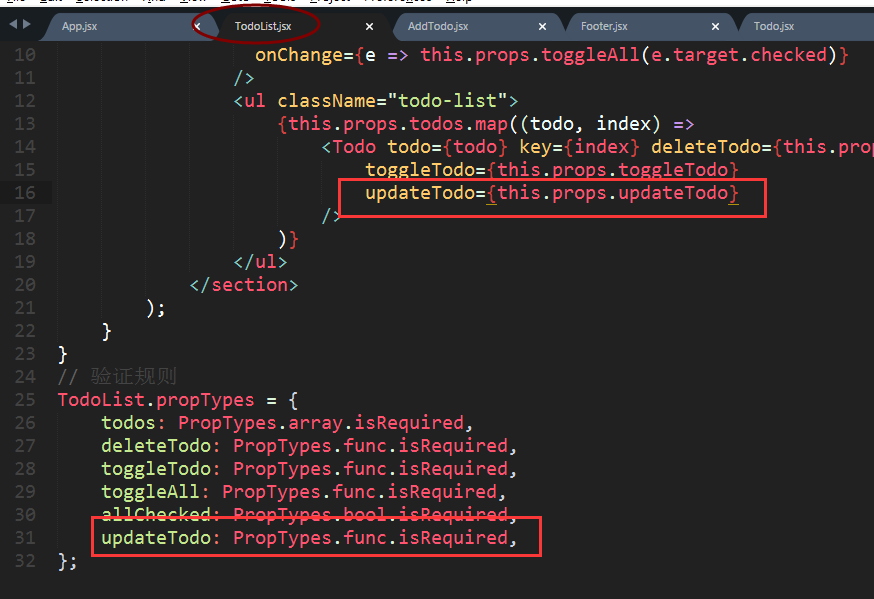


### 传递方法

第一次传递，如下：

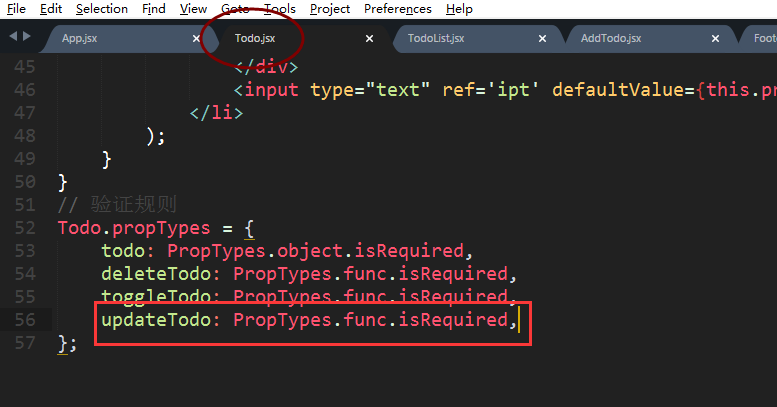


第二次传递，如下：

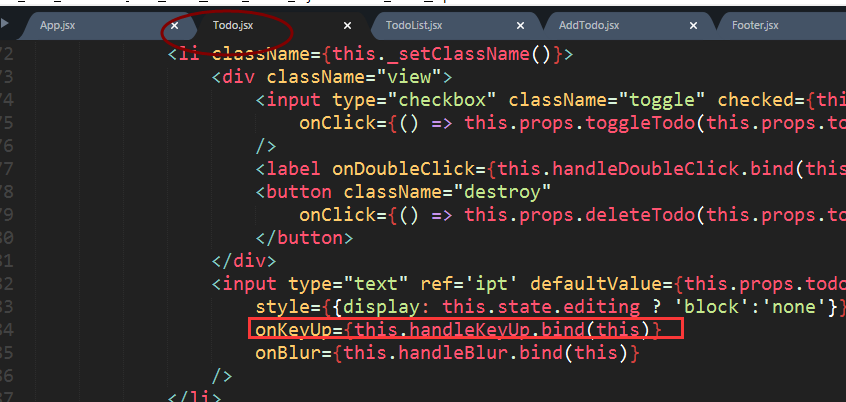


### 在Todo组件中对input注册keyUp事件，完成操作

先编写验证规则，如下：



注册keyup事件，如下：



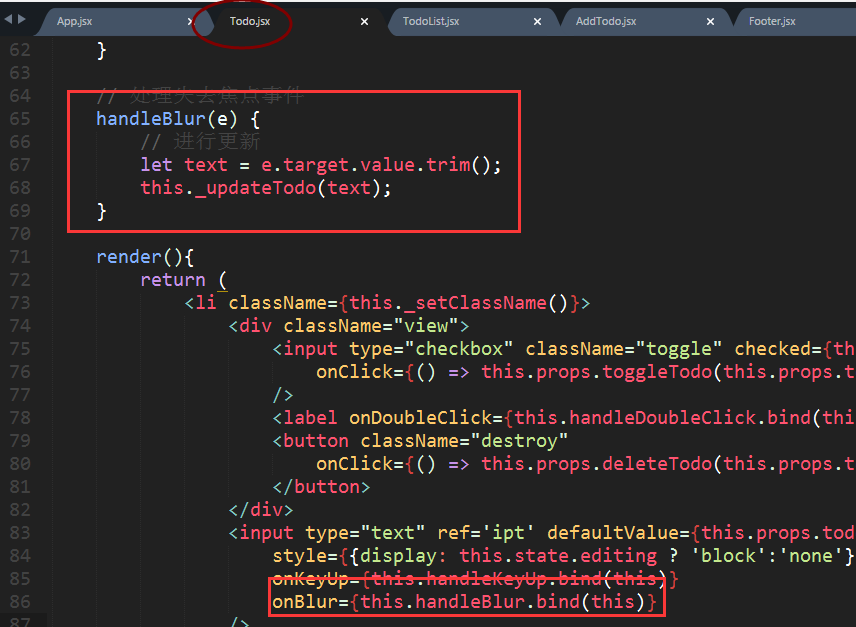
对应的代码如下：



测试，ok。

### 在Todo组件中对input注册blur事件，完成操作

编写代码如下：



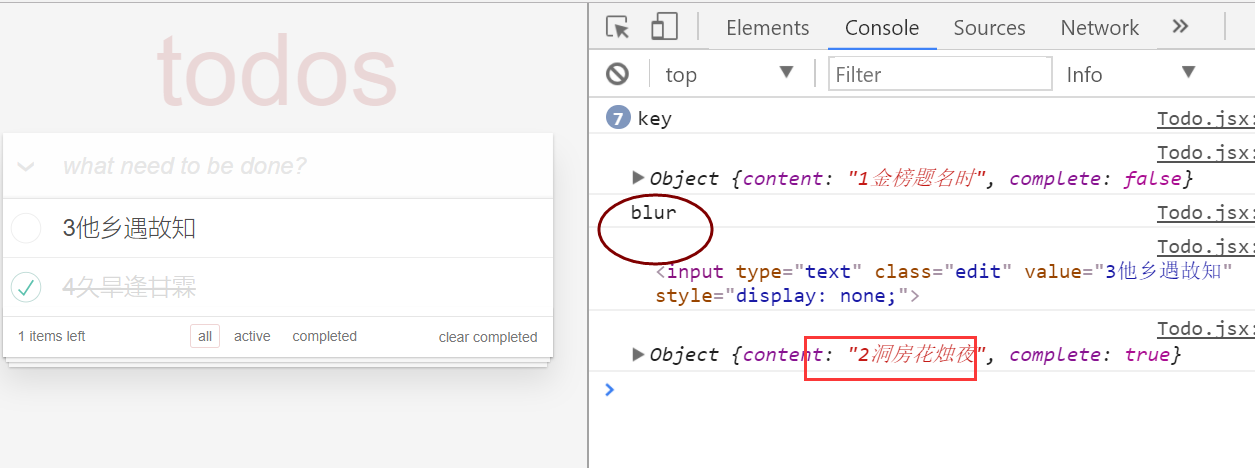
测试，ok。

出现了一个小bug？退格删除时，删除的不对。

原因在于在enter的时候，会触发blur事件。

**【补充】：**

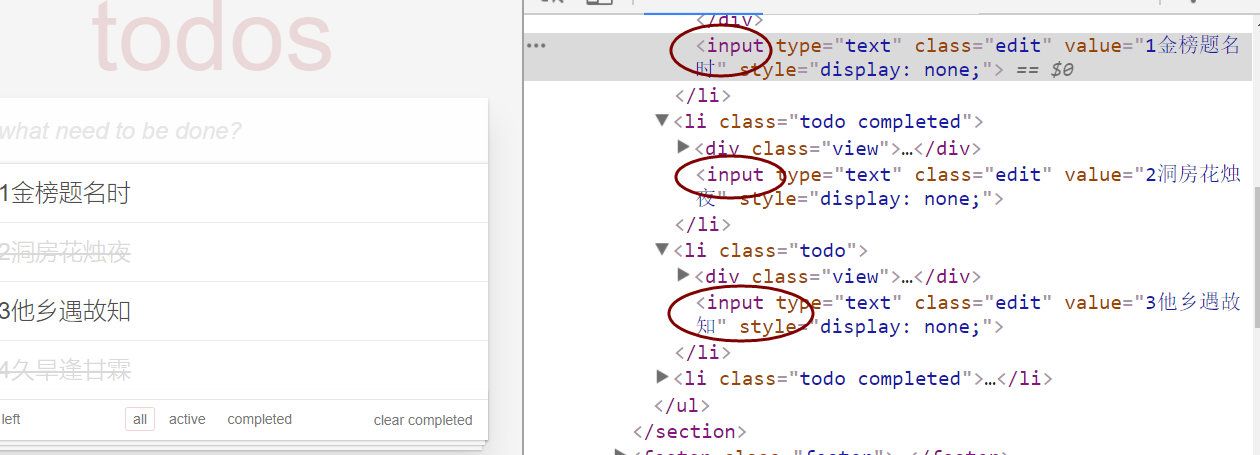
可以增加调试代码，输出如下：



由于在enter的时候，又触发了blur事件。

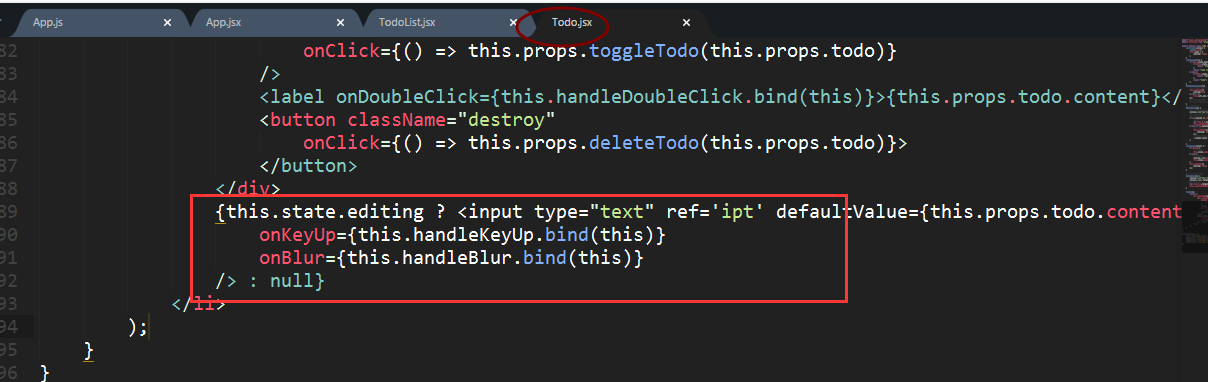
唯一的办法，就是不给你触发blur的机会。

查看html结构，如下：



发现，其实，这几个input每时每刻都是存在于DOM中的， 只不过由于设置了display:none，所以我们没有看见而已。

所以，我们不要控制样式，就控制这个元素是否应该存在于dom之上。



测试，ok。