Day10 3月9日

软件51 庞建业 2151601012

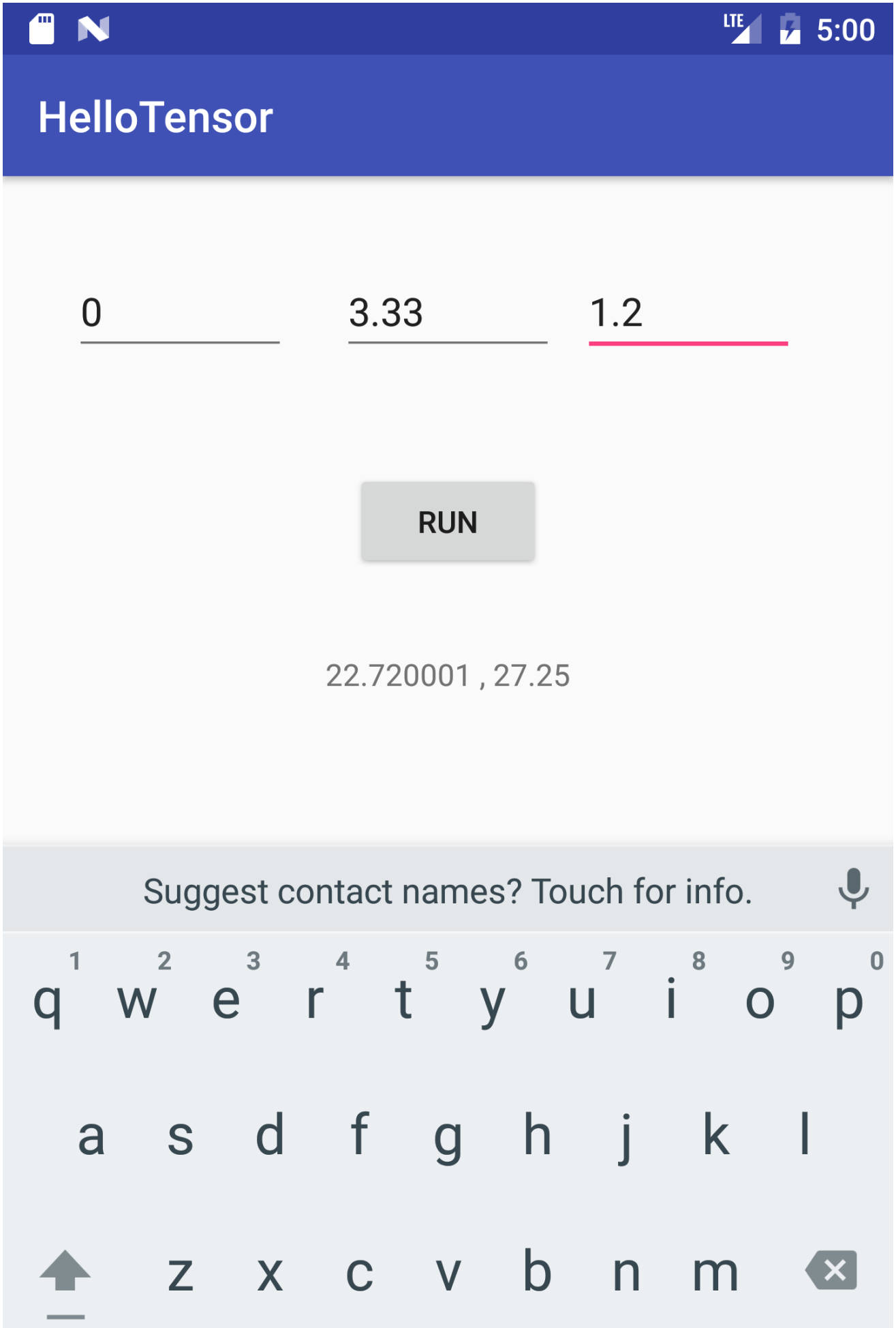
学习Build tensorflow on Andriod

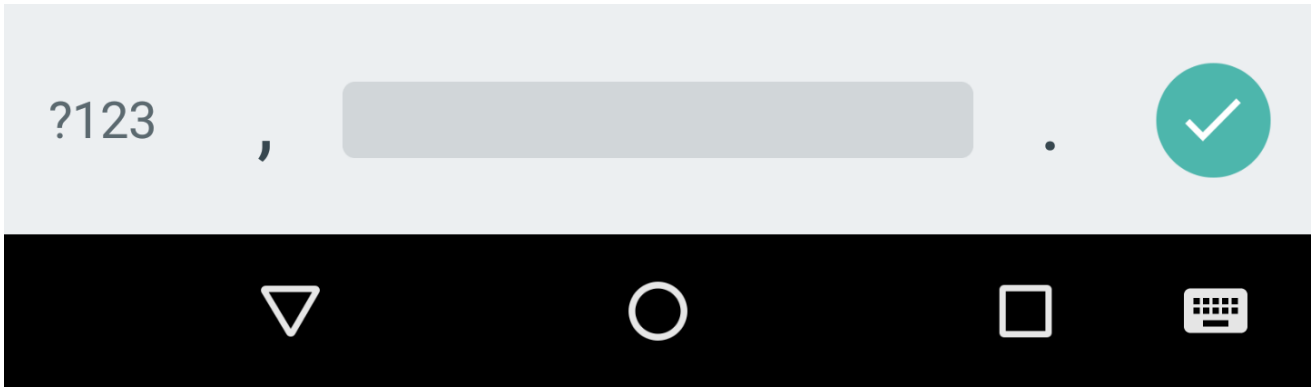
Add the TF Libraries to Your Project

For this example, these include the Java and `libtensorflow_inference` native library.

Copy the `libandroid_tensorflow_inference_java.jar` and the architecture folders inside of the `libtensorflow_inference.so` folder to `app/libs/`. The `libs/` folder should look like:

```
libs
|____arm64-v8a
| |____libtensorflow_inference.so
|____armeabi-v7a
| |____libtensorflow_inference.so
|____libandroid_tensorflow_inference_java.jar
|____x86
| |____libtensorflow_inference.so
|____x86_64
| |____libtensorflow_inference.so
```


5:00
HelloTensor
0
3.33
1.2
RUN
22.720001 , 27.25
Suggest contact names? Touch for info.



Accessing the TF Inference Interface

```
Inside of the MainActivity.java, we first import the TensorFlowInferenceInterface package:

And load the tensorflow_inference native library:

We use some constants to specify the path to the model file, the names of the input and output
nodes in the computation graph, and the size of the input data as follows:

And create a TensorFlowInferenceInterface instance that we use to make inferences on the graph
throughout the app:

We then initialize the inferenceInterface it and load the model file inside of the onCreate
event of the MainActivity:
```

Android TensorFlow support

- The TensorFlow Java API
- A `TensorFlowInferenceInterface` class that provides a smaller API surface suitable for inference and summarizing performance of model execution.
- build-gradle

```
allprojects {
    repositories {
        jcenter()
    }
}

dependencies {
    compile 'org.tensorflow:tensorflow-android:+'
}
```

Bazel

TF library

```
bazel build -c opt //tensorflow/contrib/android:libtensorflow_inference.so \
   --crosstool_top=//external:android/crosstool \
   --host_crosstool_top=@bazel_tools//tools/cpp:toolchain \
   --cpu=armeabi-v7a
```

vue-router2学习

可以使用npm直接安装插件

```
npm install vue-router --save
```

执行命令完成vue-router的安装，并在package.json中添加了vue-router的依赖。当我们在其他电脑上安装项目时只需要执行 `npm install` 即可完成安装。

package.json

```
  "dependencies": {
    ...
    "vue-router": "^2.1.1"
    ...
  },
```

如果是要安装在开发环境下，则使用以下命令行：

```
npm install vue-router --save-dev
```

package.json

```
  "devDependencies": {
    ...
    "vue-router": "^2.1.1",
    ...
  },
```

SPA中路由的简单实现

完成了SPA路由的简单实现demo（基于vue-cli的webpack模板）。 **main.js**

```
import Vue from 'vue'
import App from './App'
import VueRouter from 'vue-router'
import Page01 from './components/page01'
import Page02 from './components/page02'

Vue.use(VueRouter)//全局安装路由功能
//定义路径
const routes = [
  { path: '/', component: Page01 },
  { path: '/02', component: Page02 },

]
```

```
//创建路由对象
const router = new VueRouter({
  routes
})

new Vue({
  el: '#app',
  template: '<App/>',
  components: { App },
  router
})
```

App.vue

```
<template>
  <div id="app">
    <router-link to="/">01</router-link>
    <router-link to="/02">02</router-link>
    <br/>
    <router-view></router-view>
  </div>
</template>
```

page01.vue

```
<template>
  <div>
    <h1>page02</h1>
  </div>
</template>
```

page02.vue

```
<template>
  <div>
    <h1>page02</h1>
  </div>
</template>
```

实现步骤：

npm安装vue-router
Vue.use(VueRouter)全局安装路由功能
定义路径数组routes并创建路由对象router
将router对象传到Vue对象中
在根组件中使用<router-link>定义跳转路径
在根组件中使用<router-view>来渲染组件
创建子组件

路由的跳转

router-link

`router-link`标签用于页面的跳转

```
<router-link to="/page01">page01</router-link>
```

点击这个`router-link`标签`router-view`就会渲染路径为`/page01`的页面。 **注意：** `router-link`默认是一个`<a>`标签的形式，如果需要显示不同的样子，可以在`router-link`标签中写入不同标签元素，如下显示为`button`按钮。

```
<router-link to="/04">
    <button>to04</button>
</router-link>
```

router.push

通过JS代码控制路由的界面渲染，官方文档写法如下：

```
// 字符串
router.push('home')
// 对象
router.push({ path: 'home' })
// 命名的路由
router.push({ name: 'user', params: { userId: 123 }})
// 带查询参数，变成 /register?plan=private
router.push({ path: 'register', query: { plan: 'private' }})
```

如果是全局注册的路由`Vue.use(VueRouter)`

```
// 字符串
this.$router.push('home')
// 对象
this.$router.push({ path: 'home' })
// 命名的路由
this.$router.push({ name: 'user', params: { userId: 123 }})
// 带查询参数，变成 /register?plan=private
this.$router.push({ path: 'register', query: { plan: 'private' }})
```

> 能这么写猜测是将router对象传递给Vue对象后，复制router对象为Vue.$router上了

push方法其实和`<router-link :to="...">`的写法是等同的。 **注意：**push方法的跳转会向 history 栈添加一个新的记录，当我们点击浏览器的返回按钮时可以看到之前的页面。

router.replace

push方法会向 history 栈添加一个新的记录，而replace方法是替换当前的页面，不会向 history 栈添加一个新的记录。用法如下 **template**

```
<router-link to="/05" replace>05</router-link>
```

script

```
this.$router.replace({ path: '/05' })
```

router.go

go方法用于控制history记录的前进和后退

```
// 在浏览器记录中前进一步，等同于 history.forward()
this.$router.go(1)
// 后退一步记录，等同于 history.back()
this.$router.go(-1)
// 前进 3 步记录router.go(3)
// 如果 history 记录不够用，那就默默地失败呗
this.$router.go(-100)
this.$router.go(100)
```

go方法就是浏览器上的前进后退按钮，方法中传递的数字参数就是前进和后退的次数

路由的传参方式

在路由跳转的过程中会传递一个object，可以通过`watch`方法查看路由信息对象。

```
watch: {
  '$route' (to, from) {
      console.log(to);
      console.log(from);
  },
},
```

console中看到的路由信息对象

```
{
    ...
    params: { id: '123' },
    query: { name: 'jack' },
    ...
}
```

这两个参数会在页面跳转后写在路径中，路径相当于`/page/123?name=jack`

传递数据

在路由配置文件中定义参数

```
 export default [
    ...
    { name: 'Page05', path: '/05/:txt', component: Page05 },
]
```

下面有两种传递params的方式

1. 通过path传递

路径后面的`/:txt`就是我们要传递的参数。

```
this.$router.push({ path: '/05/441'})
```

此时路由跳转的路径

```
http://localhost:8080/#/05/441
```

此时我们看到查看路由信息对象：

```
{
    ...
    params: {
        txt: '441'
    }
    ...
}
```

2. 通过params传递

```
this.$router.replace({
    name: 'Page05',
    params: { abc: 'hello', txt: 'world' },
    query: { name: 'query', type: 'object' }
})
```

通过name获取页面，传递params和query。 得到的URL为

```
http://localhost:8080/#/05/world?name=query&type=object
```

而获取到的参数为

```
{
    ...
    params: {
        abc: "hello",
        txt: "world"
    }
    ...
}
```

获取数据

template

```
<h2> {{ $route.params.txt }} </h2>
```

script

```
console.log(this.$route.params.txt)
```

2.query

query传递数据的方式就是URL常见的查询参数，如`/foo?user=1&name=2&age=3`。很好理解，下面就简单写一下用法以及结果

传递数据

template

```
<router-link :to="{ path: '/05', query: { name: 'query', type: 'object' }}" replace>05</router-link>
```

script

```
this.$router.replace({ path: '/05', query: { name: 'query', type: 'object' }})
```

路径结果

```
http://localhost:8080/#/05?name=query&type=object
```

路由信息对象

```
{
    ...
    query: {
        name: "query",
        type: "object"
    }
    ...
}
```

获取数据

获取数据和params是一样的。 **template**

```
<h2> {{ $route.query.name }} </h2>
```

script

```
console.log(this.$route.query.type)
```

Webpack学习

> webpack是一个模块打包工具。
>
> **用vue项目来举例**：浏览器它是只认识js，不认识vue的。而我们写的代码后缀大多是.vue的，在每个.vue文件中都可能html、js、css甚至是图片资源；并且由于组件化，这些.vue文件之间还有错综复杂的关系。所以项目要被浏览器识别，我们就要使用webpack将它们打包成js文件以及相应的资源文件。
>
> 以vue项目的形式编写项目逻辑，浏览器以他理解的方式来运行项目。webpack把我们的vue项目想表达的所有意图传递给浏览器让浏览器去运行。

hello.js

```
console.log("hello~~")
```

app.js

```
console.log("hello app");
require("./hello.js")
```

`app.js`中导入了`hello.js`，它们之间有导入关系。我们假如直接将`app.js`放到html中是会报错的。

```
hello app
Uncaught ReferenceError: require is not defined at app.js:2
```

如果我们要维持这种关系我们就必须使用打包工具进行打包。在命令行中输入：

```
// 安装webpack
$ npm install webpack -g
// 打包app.js
$ webpack app.js bundle.js
```

然后我们会发现项目中多了一个bundle.js文件，我们在html中导入这个js文件。 **index.html**

```
<!DOCTYPE html>
<html>
  <head>
    <title>demo01</title>
  </head>
  <body>
    <h1>demo01</h1>
    <script src="bundle.js"></script>
  </body>
</html>
```

最后输出正确结果

```
hello app
hello~~
```

webpack.config.js

webpack.config.js文件是webpack的默认配置文件。之前我们使用命令行`$ webpack entry.js output.js`来实现打包，其实webpack可以有更多的打包配置，这些配置都是在webpack.config.js中完成的

```
const webpack = require("webpack")

module.exports = {

  entry: {
```

```
    entry: "./app/entry.js",
  },
  output:
  {
    path: __dirname + "/dist",
    filename: 'bundle.js',
  },
  module: {
    loaders: [
      {
        test: /\.js$/,
        loader: 'babel',
        exclude: /node_modules/
      },
    ]
  }
}
```

entry&output

entry是配置webpack的入口文件，上面的代码中我们将app目录下的entry.js作为入口文件。webpack会将与entry.js有关的资源都进行打包。 output是出口文件，即打包好的文件的存放地址和文件名。

单文件，单输出

```
const webpack = require("webpack");
module.exports = {
  context: __dirname + "/src",
  entry: {
    app: "./app.js",
  },
  output: {
    path: __dirname + "/dist",
    filename: "[name].bundle.js",
  },
};
```

多文件，单输出

```
const webpack = require("webpack");
module.exports = {
  context: __dirname + "/src",
  entry: {
    app: ["./home.js", "./events.js", "./vendor.js"],
  },
  output: {
    path: __dirname + "/dist",
    filename: "[name].bundle.js",
  },
};
```

多文件，多输出

```
const webpack = require("webpack");
module.exports = {
  context: __dirname + "/src",
  entry: {
    home: "./home.js",
    events: "./events.js",
    contact: "./contact.js",
  },
  output: {
    path: __dirname + "/dist",
    filename: "[name].bundle.js",
  },
};
```

打包出来的单个或者多个文件直接可以在html中使用

```
<script src="./dist/entry.js"></script>
```

loaders

> loader是webpack的加载器，可以帮我们处理各种非js文件。如css样式，vue、jsx、weex等后缀的代码，JPG、PNG图片等。所以我们一般会在package.json中看到各种***-loader。这些就是各类资源的loader加载器。 在module的loaders数组中可以有多个对象，每个对象就是一个加载器。下面是babel-loader的最简单配置方式

```
  module: {
    loaders: [
      {
        test: /\.js$/,
        loader: 'babel',
      },
    ]
  }
```

对象中的test是正则表达式，用于搜索后缀为.js的文件。loader是所用加载器名称。