Booster 开发指南

变更历史

本工程所有的变更都记录在此。

文档格式基于 Keep a Changelog

[1.4.0] - 2019-04-22

Added

- 增加前端路由、前端布局等内容 3.5、3.6、3.11

[1.3.0] - 2019-03-06

Changed

- 修改vue-cli版本为3.4.0
- 更新Booster脚手架版本为1.3.0

[1.2.0] - 2019-01-17

Changed

- 更新Booster脚手架版本为1.2.0

Added

- 集成前端表格组件 3.6

[1.1.0] - 2018-12-02

Changed

- 更新war包部署方式 5.1.2

Added

- 增加NDI连接配置 4.9

1. 背景

前海人寿目前有超过200个IT系统，其中大部分是基于Java平台，使用JavaEE相关技术构建。由于没有一套统一的软件开发架构框架和相应的技术规范，导致在系统建设中存在基础功能重复、运维监控困难、技术标准不统一、人员技能难以共享等问题。为了确保技术的迭代更新、规范化系统建设、提升开发效率、促进人力资源共享，从而保证保险业务系统的稳定运营，我们构建了一套统一的技术架构和技术规范。

2. Booster框架概述

2.1 什么是Booster

Booster[ˈbu:stə(r)]原意为加速器或者助推器，选择这个名字的含义有：

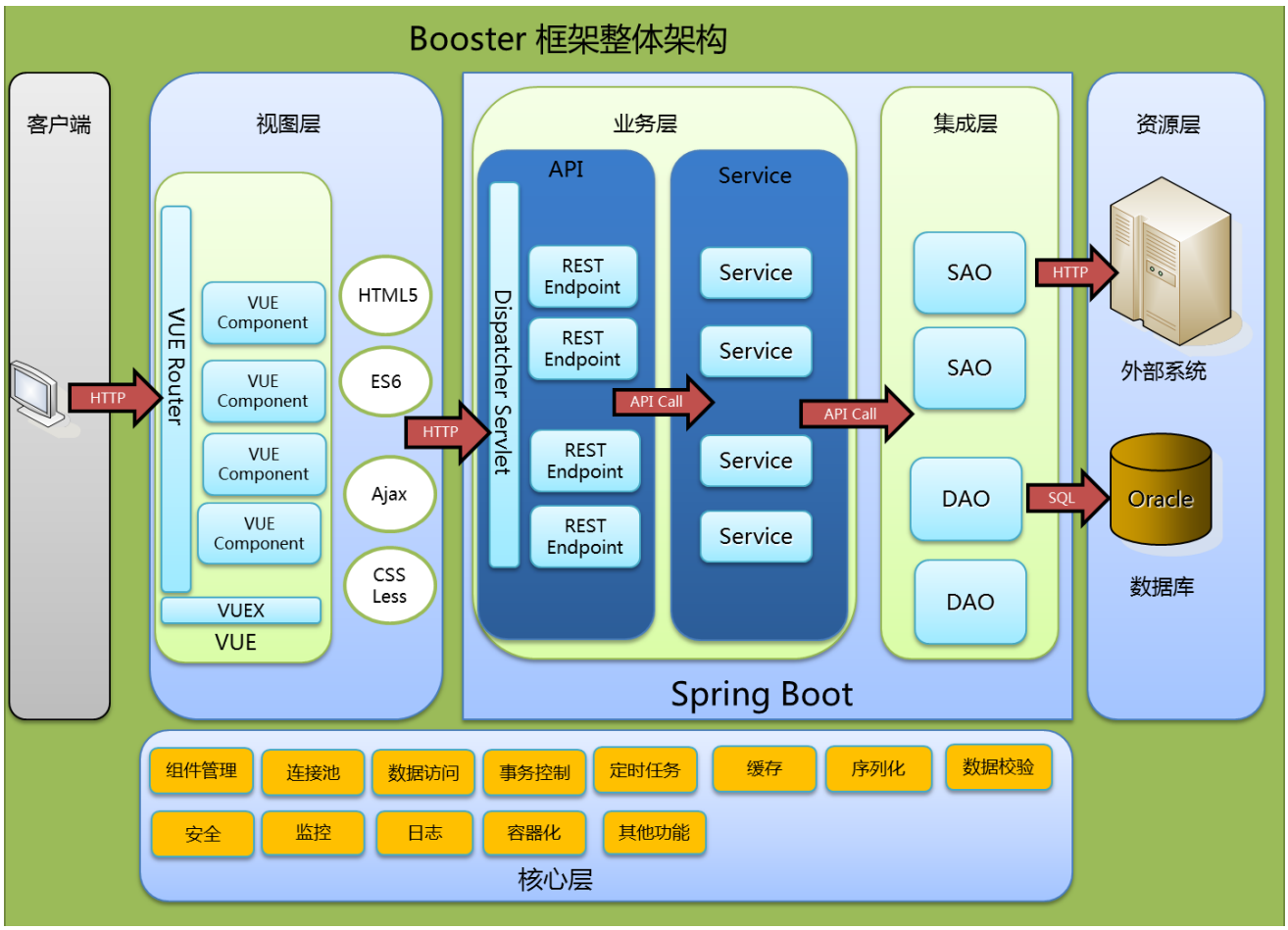
1. 这个开发框架能够提升开发效率；
2. 这个开发框架并非原创性，而是整合现有业界流行技术，以统一规范的方式来使用这些技术和框架；

2.2 Booster的设计目标

Booster的设计目标包括，但不局限于：

1. 采用业界通用和流行的技术和理念，以降低开发人员的学习成本；
2. 在保证安全、可靠的前提下，采用较新的技术以适应长期演进趋势；
3. 提供丰富的文档、指引和示例，使得开发人员容易上手；
4. 提供项目模板、工具，方便快速建立系统原型；
5. 提供统一的运维、监控及监管功能和接口，便于系统的运维；
6. 提供配置打包工具，支持在云虚拟机或容器中部署;
7. 提供依赖包管理、静态代码扫描、持续集成平台和工具集成;
8. 提供常见通用功能的集成，使开发人员更加关注业务；

2.3 Booster框架的整体架构



Booster整合了业界流行的技术和框架，整体采用前后端分离的策略，前端为基于VUE的单页应用，后端为基于Spring Boot的微服务应用。

Booster框架分为前端和后端两大模块，同时还提供一系列基于spring-boot-starter的第三发服务集成组件。

2.4 源码结构

项目采取前后端分离的模式，前后端独立开发，下面是整个项目的目录结构。

```
├─ backend                    // 后端项目
│   ├─ src                    // 后端源代码
├─ frontend                   // 前端项目
│   ├─ public                 // 前端静态资源，html等
│   ├─ src                    // 前端源代码
│   ├─ ...                    // 前端打包时用到的配置文件
├─ installer                  // 打包工具
├─ pom.xml                    // 项目的Maven POM
```

3.前端介绍

3.1 技术选型

前端使用的主要技术包括：

1. Html 5
2. ES 6
3. VUE 2
4. IView 2
5. Axios
6. Less

3.2 前端源码结构

下面是`frontend/src`中前端源代码的结构

```
├── api                         // 所有请求
├── assets                      // 主题 字体等静态资源
│   ├── icons                   // 图标
│   ├── images                  // 图片
│   ├── style                   // 样式
├── components                  // 全局公用组件
├── config                      // 全局配置
├── directive                   // 全局指令
├── libs                        // 封装一些公用方法，相当于utils
├── locale                      // 国际化支持
├── router                      // 路由
├── store                       // 全局store，状态管理
├── view                        // 所有页面
│   ├── components              // 封装页面中用到的公共组件
├── App.vue                     // 入口页面
├── main.js                     // 程序执行的入口，相当于java的Main方法， 加载组件 初始化等
├── index.less                  // 框架的主要样式文件，样式变量声明
```

3.3 登录界面

默认已经集成IAM，需要申请IAM的客户端配置文件iam-client.properties替换后端对应配置后，输入IAM用户名和密码登录：

> 对应代码：
>
> `@/view/login/login.vue`

@ 是 webpack 的 alias

3.4 前端安全

3.4.1 权限认证

该项目中权限的实现方式是：通过获取当前用户的权限去比对路由表，生成当前用户具的权限可访问的路由表，通过 `router.addRoutes` 方法动态挂载到 router 上。

你可以在利用每个路由的access属性来配置用户需要的权限：

```
meta: {
   access: ['role_admin'],  //配置需要的权限
   icon: 'arrow-graph-up-right',
   title: '二级-2'
}
```

在很多系统中每个页面的权限是动态配置的，不像本项目中是写死预设的。但其实原理是相同的。

如：你可以在后台通过一个 tree 控件或者其它展现形式给每一个页面动态配置权限，之后将这份路由表存储到后端。当用户登录后得到 roles，前端根据roles 去向后端请求可访问的路由表，从而动态生成可访问页面，之后就是 `router.addRoutes` 动态挂载到 `router` 上，你会发现原来是相同的，万变不离其宗。

3.4.2 跨站脚本攻击(XSS攻击)

服务器端输出到浏览器的数据，必须使用系统的安全函数来进行编码或转义来防范XSS攻击。 例如`<script>`在进行HTML编码后变成了`&lt;script&gt;`，而这段数据就会被浏览器认为只是一段普通的字符串，而不会被当做脚本执行了。

其他的防御措施，例如设置CSP HTTP Header、输入验证、开启浏览器XSS防御等等都是可选项，原因在于这些措施都存在被绕过的可能，并不能完全保证能防御XSS攻击。不过它们和输出编码却可以共同协作实施纵深防御策略。

你可以查阅OWASP XSS Prevention Cheat，里面有关于XSS及其防御措施的详细说明。

在网站上动态渲染任意 HTML 是非常危险的，因为容易导致 XSS 攻击。因此永远不要将用户提交的内容直接输出到页面上（必须经过html转义）

3.4.3 跨站请求伪造(CSRF攻击)

在关键业务点设置验证码验证。 在HTTP 请求中以参数的形式加入一个随机产生的token，并在服务器建立一个拦截器来验证这个token。

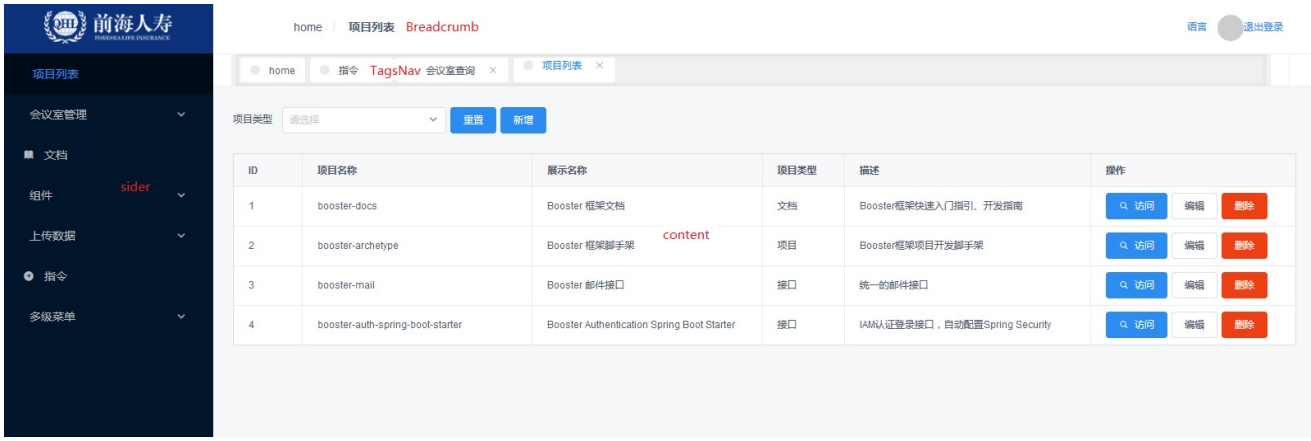
在本系统中，登录认证通过后，后续的前端http请求都会带上认证头`Authorization : Bearer <认证返回的token>`，服务器读取这个token值，会进行校验该请求当中的token是否合法，否则认为这次请求是违法的，拒绝该次服务。

3.4.4 文件上传漏洞（WebShell）

1. 检查服务器是否判断了上传文件类型及后缀。
2. 定义上传文件类型白名单，即只允许白名单里面类型的文件上传。
3. 文件上传目录禁止执行脚本解析，避免攻击者进行二次攻击。

3.5 界面布局

页面整体布局是一个产品最外层的框架结构，往往会包含导航、侧边栏、面包屑以及内容等。想要了解一个后台项目，先要了解它的基础布局。如图所示，整个页面就是一个`Main`组件：



3.5.1 Main

> 对应代码：
>
> `@/view/main/main.vue`

Main的整体结构大致如下：

```
<Layout>
    <Sider></Sider>
    <tags-nav></tags-nav>
    <Content></Content>
</Layout>
```

`Layout` 是 Main的布局，`Sider` 是左边侧边栏，`tags-nav` 是上方快捷导航条，`Content` 是右边具体页面内容

一般情况下，你增加或者修改页面只会影响 content 这个主体区域。其它配置在 layout 中的内容如：侧边栏或者导航栏都是不会随着你主体页面变化而变化的。

Layout

请参考iview官方文档：https://www.iviewui.com/components/layout

侧边栏

侧边栏基于 `iview` 的 `Sider` 布局以及 `Menu` 标签，对其样式进行了一些调整，大致调用方式如下：

```
<Sider>
    <side-menu></side-menu>
</Sider>
```

`side-menu` ：`src/view/main/components/side-menu/sider-menu.vue`

所有的样式都可以在 `src/view/main/components/side-menu/side-menu.less` 中找到，你可以自由修改之

快捷导航

快捷导航（tags-nav）即上方标签栏导航。用户可以自由切换打开过的页面，也可以点击"×"关闭已经打开的界面。

> 对应代码：
>
> `src/view/main/components/tags-nav/tags-nav.vue`

快捷导航基于 `router` 的 `name` 属性工作，如果你定义了两个router，它们的 name 属性一样，则快捷导航可能出现各种问题。

Content

Content区域的代码结构大致如下：

```
<keep-alive :include="cacheList">
    <router-view/>
</keep-alive>
```

这里 keep-alive 主要是为了缓存的，例如有一个含一大段表单输入的页面，如果不使用缓存，使用快捷导航切换页面后，用户输入将会丢失。

当然，这样对于某些业务是不适合的，比如文章详情页这种 `/article/1` `/article/2`，他们的路由不同但对应的组件却是一样的，所以他们的组件 name 就是一样的，这样必须设置禁止缓存，转而使用 localStorage 等浏览器缓存方案，自己进行缓存处理

```
meta: {
    notCache: true  //禁止 keep-alive 的缓存
}
```

router-view

booster框架中的 `router` 与 `vue-router` 功能一致。下面3.6章节会详细介绍

如果不同的路由，引用了同一个组件，默认情况下当这两个页面切换时并不会触发 vue 的 created 或者 mounted 钩子。

```
{ path: 'create', component: PostCreate, name: '发表文章' },
{ path: 'edit/:id(\\d+)', component: PostCreate, name: '编辑文章' },
```

其实可以简单的在 router-view 上加上一个唯一的 key，来保证路由切换时都会重新渲染触发钩子了。

```
<router-view :key="key"></router-view>

computed: {
  key() {
    // 只要保证 key 唯一性就可以了，保证不同页面的 key 不相同
    return this.$route.fullPath
  }
 }
```

`booster` 框架中大部分页面都是基于这个基础组件 `Main` 的，除了个别页面如： `login`，`404`，`401` 。

如果你想在一个项目中有多种不同的基础组件也是很方便的，只要在一级路由那里选择不同的组件就行。

```
{
    path: '/lab-project',
    name: 'lab-project',
    meta: {
      hide: true
    },
    // 你可以选择不同的基础组件
    component: Main,
}
```

3.5.2 响应式布局

booster框架针对桌面屏幕做了一些响应式配置，在开发时需要注意不同屏幕下的效果。由于booster主要针对的是后台管理系统，因此不建议作为移动端的框架。

1.通过媒体查询（media query）来区分不同屏幕上的样式

```
/* 小屏幕（平板） */
@media (min-width: 768px) { ... }

/* 中等屏幕（桌面显示器） */
@media (min-width: 992px) { ... }

/* 大屏幕（大桌面显示器1200px） */
@media (min-width: 1200px) { ... }
```

2.宽度、字体大小不固定，应该使用百分比

```
#head{
   width:100%;
}
#content{
   width:50%;
}

body {
   font-size: 100%
}

h1 {
    font-size: 1.5em;
}
```

3.内容流动

随着屏幕尺寸变小和种类的增多，内容将会占据更多的空间，而下方的内容就有可能被挤到下一行，因此，应该尽量使用流动布局（或者使用flex布局），例如

```
.main {
    float: right;
    width: 70%;
}
.leftBar {
    float: left;
    width: 25%;
}
```

3.6 前端路由

3.6.1 路由配置项

在Booster前端开发中，路由配置承担着重要的作用，它影响着如下内容：

- 左侧菜单内容

- 是否缓存该页
- 该页面图标（显示在菜单、面包屑和Tab标签）

本框架侧边栏和路由是绑定在一起的，你只需要在 `src/router/routers.js` 中配置对应的路由 ，侧边栏就能动态的生成了。大大减轻了手动重复编辑侧边栏的工作量。当然这样就需要在配置路由的时候遵循一些约定的规则。

接下来来看如何配置路由

路由的配置是在 `/src/router` 文件夹下， `./src/router/index.js` 文件中定义路由拦截的逻辑， `./src/router/routers.js` 文件中定义页面路由信息。

```
path: '/join',      //(二级)路由的访问路径
name: 'join',       //设定路由的名字，必填，且必须唯一，否则使用时会出现各种问题
meta: {
     // 路由的属性
},
component: Main      //路由的组件
```

meta除了原生参数外可配置的参数:

```
meta: {
  notClose: (default: false) 新增属性，设为true后，在快捷导航条上不可被关闭
  hideInMenu: (default: false) 设为true后在左侧菜单不会显示该页面选项
  showAlways: (default: false) 设为true后如果该路由只有一个子路由，在菜单中也会显示该父级菜单
  notCache: (default: false) 设为true后页面不会缓存
  access: (default: null) 可访问该页面的权限数组，当前路由设置的权限会影响子路由
  icon: (default: -) 该页面在左侧菜单、面包屑和标签导航处显示的图标，如果是自定义图标，需要在图标名称前加下划线'_'
  href: 'http://www.baidu.com' (default: null) 用于跳转到外部连接
  title: '' 菜单显示在侧边栏中的标题，这里的title可以自动被国际化机制翻译
}
```

完整的示例：

```
 {
    path: '/join',
    name: 'join',
    meta: {
      title: 'QQ群',
      icon: '_qq',
      access: ['role_admin'],
    },
    component: Main,
    children: [
      {
        path: 'join_page',
        name: 'join_page',
        meta: {
          icon: '_qq',
          title: 'QQ群'

        },
```

```
      component: () => import('@/view/join-page.vue')
    }
  ]
}
```

3.6.2 在独立页面展示的页面

如登录页、错误页这种**独立**的页面，无需使用Main组件，则直接定义在 `routers.js` 中 `export default []` 的数组中 定义，例如登录界面的定义如下：

```
{
  path: '/login',
  name: 'login',
  meta: {
    title: 'Login - 登录',
    hideInMenu: true
  },
  component: () => import('@/view/login/login.vue')
}
```

在 `router.js` 的结尾也定义了 `404、500` 等错误页面：

```
{
  path: '/401',
  name: 'error_401',
  component: () => import('@/view/error-page/401.vue')
},
{
  path: '/500',
  name: 'error_500',
  component: () => import('@/view/error-page/500.vue')
},
{
  path: '*',
  name: 'error_404',
  component: () => import('@/view/error-page/404.vue')
}
```

3.6.3 在Main组件展示区域中展示的页面

页面作为多Tab页展示，在Main组件的视图区域显示，**也就是我们平时开发用到的模式**，具体路由需要在 `children` 中定义为二级路由。定义如下：

```
{
  path: '/lab-project',
  name: 'lab-project',
  meta: {
    hide: true
  },

  component: Main,   //在Main组件中展示
```

```
    children: [
      {
        //定义二级路由对象
        path: '/lab-project_page',
        name: '项目列表',
        meta: {
          icon: 'md-list',
          title: '项目列表',
        },
        component: () => import('@/view/lab-project/lab-project.vue')
      }
    ]
  }
```

这种模式即vue router嵌套路由，外层组件是 Main ，内层是具体的业务组件。

3.6.4 路由懒加载

这一部分请查看vue router的官方文档

当打包构建应用时，Javascript 包会变得非常大，影响页面加载速度。如果我们能把不同路由对应的组件分割成不同的代码块，然后当路由被访问的时候才加载对应组件，这样就更加高效了。

结合 Vue 的异步组件和 Webpack 的代码分割功能 ，轻松实现路由组件的懒加载。如：

```
component: () => import('@/view/lab-project/lab-project.vue')
```

3.7 UI组件库

Booster框架使用IView UI 3.x提供的控件库作为标准控件库，使用标准控件时请参考
官方文档中使用步骤。

如果使用自动化项目生成工具生成的项目，默认已经引入IView和相关的Library，直接根据官方文档使用各种控件即可。
否则，需要按照IView官方文档步骤先安装，然后才可以使用。

常见的控件列表包括：

- 基础组件：色彩、字体、按钮、图标
- 布局组件：栅格、布局、卡片、折叠面板、面板分割、分割线、单元格
- 导航组件：菜单、标签、下拉菜单、分页、面包屑、徽标数、步骤条、进度条
- 表单组件：输入框、单选框、多选框、开关、表格、选择器、自动完成、滑块、日期选择器、时间选择器、级联选择、穿梭框、数字输入框、评分、上传、颜色选择器、表单
- 视图组件：警告提示、全局提示、通知提醒、对话框、树形组件、文字提示、气泡提示、进度条、头像、标签、走马灯、时间轴、相对时间

具体使用方法请参考IViews 官方文档

当然，如果上述组件仍然不能满足你的要求，那么你仍然可以使用其他第三方组件。但是，我们要求这个组件必须是一个标准的Vue组件。

前端表格组件

在iview框架中默认的Table组件，需要写一个复杂的 `column` 对象，如果表格列数较多，column对象比较难维护。 因此booster集成了 `iv-table` 组件，支持以标签的形式，写iview的表格

具体可以查看文档：https://www.npmjs.com/package/iv-tables

3.8 后台数据访问

Booster框架采用axios与后端进行接口调用。

由于我们采用Json Web Token与后端进行认证，因此我们创建了src/lib/httpRequest.js公共库来保证每次请求都包含了JWT的认证头。

为了方便管理维护，统一的请求处理都放在 `@/src/api` 文件夹中，并且一般按照 `model` 纬度进行拆分文件，例如 `@/api/lab-project.js`

通常情况下，所有的与后端交互的http请求都应该是 `Rest API` 的形式，所有从前端向后端发起的http请求都应该采用httpRequest.js库。前后端的交互有两种方式， `json` 格式提交和 `form` 表单格式提交，两种提交方式，后端返回给前端的数据都是 `json` ，只是前端给后端提供的参数不同。

3.8.1 JSON格式提交

这种方式下，Content-Type为 `application/json` ，数据以 `json` 的形式在请求主体中，后端用 `@RequestBody` 注解接收，例如：

前端：

```
import httpRequest from '@/libs/httpRequest'

export const query = (id,name) => {
    return httpRequest.request({    //注意这里是httpRequest.request，表示是json格式提交
        url: '/api/query',
        method: 'post',
        data : {
            id : id,
            name : name
        }
    })
}
```

后端：

```
class TestEntity{
    private int id;
    private String name;

    //省略 get 、set 方法
}

@RequestMapping(value = "/api/query")
public void query(@RequestBody TestEntity entity) throws IOException{

}
```

这时，在前端执行 `query(1,'test')` 方法，HTTP请求如下：

```
POST /api/query HTTP/1.1
Content-Type: application/json; charset=utf-8
Content-Length: 22
Authorization: Bearer <token>

{"id":1,"name":"test"}
```

Authorization是登录后的token验证信息，参考安全章节。

3.8.2 Form表单格式提交

这种方式下，Content-Type为 `application/x-www-form-urlencoded` ，提交方式跟正常表单post提交一致，后端可以直接接收，也可以用 `@RequestParam` 注解接收，例如：

前端：

```
import httpRequest from '@/libs/httpRequest'

export const query = (id,name) => {
    return httpRequest.formRequest({   //注意这里是httpRequest.formRequest，表示是form表单格式提交
        url: '/api/query',
        method: 'post',
        data : {
            id : id,
            name : name
        }
    })
}
```

后端：

```
class TestEntity{
    private int id;
    private String name;

    //省略 get 、set 方法
```

```
}

@RequestMapping(value = "/api/query")
public void query(TestEntity entity) throws IOException{

}

//或者使用@RequestParam接收
@RequestMapping(value = "/api/query")
public void query(@RequestParam("id") int id,
                  @RequestParam("name") String name) throws IOException{

}
```

这时，在前端执行 `query(1,'test')` 方法，HTTP请求如下：

```
POST /api/query HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Content-Length: 14
Authorization: Bearer <token>

id=1&name=test
```

3.8.3 非Rest API

对于非 `Rest API` ，需要直接访问URL的，可以在请求参数中带上token，这里以文件下载为例，后端的代码为：

```
@RequestMapping(value = "/api/download")
public void downLoadFile(HttpServletResponse response) throws IOException {
     response.setContentType("application/octet-stream");       //设置ContentType为文件下载
     response.addHeader("Content-Disposition","attachment;filename=" + fileName);   //设置文件名
     response.getOutputStream().write(...)                      //向流中写入具体的文件内容
}
```

在前端可以以下面的方式访问：

```
import {getUser} from '@/libs/util'


const user = getUser()
window.open("/api/download?token=" + user.token)
```

3.8.4 在组件中使用

在组件使用时，引入在 `@/api` 中定义的函数，例如

```
import {save} from '@/api/lab-project'
```

对于返回的函数对象，可以采取如下操作：

```
save(formData).then((response) => {
   // handle success
   console.log(response);
 })
 .catch((error) => {
   // handle error
   console.log(error);
 })
 .finally(() => {
   // always executed
 });
```

对于请求的处理请参考axios官方文档。

3.9 开发流程

前端开发通常只需要以下步骤：

1. 创建一个标准的Vue组件；
2. 添加路由和菜单；
3. 调用后端REST接口进行数据交互；

这里以创建一个简单的Hello World页面为例：

1.创建src/view/hello-world.vue文件,其内容如下:

```
<template>
  <div>
  <p>{{ message }}</p>
 </div>
 </template>
<script>
  export default {
    name: "hello-world",
    data() {
       return {message: 'Hello '}
     }
 </script>
```

2.修改src/router/routers.js添加路由和菜单:

```
export default [
...
 {
     path: '/hello-world',
     name: 'hello-world',
     meta: {

       hide: true
```

```
        },
        component: Main,
        children: [
          {
            path: '/hello-world-page',
            name: 'hello world',
            meta: {
              icon: 'social-buffer',
              title: 'Hello World'
            },
            component: () => import('@/view/hello-world.vue')
          }
        ]
      }
    ...
    ]
```

3.修改src/view/hello-world.vue增加后台数据访问

```
<template>
  <div>
    <p>{{ message }}</p>
    <button v-on:click="getMessage">Say Hello</button>
  </div>
</template>

<script>
  import httpRequest from '@/libs/httpRequest'

  export default {
    name: "hello-world",
    data() {
      return {message: 'Hello '}
    },
    methods: {
      getMessage() {
        httpRequest.request({
          url: '/api/hello',
          method: 'get'
        }).then(({data}) => {
          this.message = data.message
        })
      }
    }
  }
</script>
```

3.10 引入第三方模块

除了 booster 框架提供的内置的业务组件，有时我们还需要引入其他外部组件，我们可以通过 `npm install` 的方式引入。

这里以 `loadsh` （一个 JavaScript 实用工具库）为例：

执行 `npm i --save lodash` ，执行完毕后，在 `package.json` 中可以看到 `loadsh` 的声明。

使用：

```
// Load the full build.
import _ from 'loadsh'

// Load the core build.
import _  from 'lodash/core'

// Load method categories.
import array from 'lodash/array'
import object from 'lodash/fp/object'
```

第三方库，推荐代理到 `Vue.prototype` 上，作为 `Vue` 原型对象的一个属性，便于管理，例如：

```
import _ from 'loadsh'
Object.defineProperty(Vue.prototype, '$loadsh', {
    value: _
})
```

由于所有的组件都会从 `Vue` 的原型对象上继承它们的方法, 因此在所有组件/实例中都可以通过 this.$loadsh: 的方式访问 `loadsh` 而不需要定义全局变量或者手动的引入.

MyNewComponent.vue

```
const _ = this.$loadsh
```

同时以这种方式引入后，可以直接在渲染页面时作为一个普通变量来使用，例如，`loadsh` 有一个工具方法 `capitalize` （字符串首字母大写），渲染时可以直接使用：

```
<span>{{$loadsh.capitalize('foo')}}</span>
```

最后渲染效果为

```
<span>Foo</span>
```

如果是不兼容webpack的库，可以放到public目录中，在 `index.html` 中以静态资源的方式引入，

任何放置在 `public` 文件夹的静态资源都会被简单的复制，而不经过 webpack。

注意我们推荐将资源作为你的模块依赖图的一部分导入，这样它们会通过 webpack 的处理并获得如下好处：

- 脚本和样式表会被压缩且打包在一起，从而避免额外的网络请求。
- 文件丢失会直接在编译时报错，而不是到了用户端才产生 404 错误。
- 最终生成的文件名包含了内容哈希，因此你不必担心浏览器会缓存它们的老版本。

路径之前，需要加上 `<%= BASE_URL %>` （自动处理相对路径），以 `ckplayer` 为例：

```
<script type="text/javascript" src="<%= BASE_URL %>ckplayer/ckplayer.js" charset="utf-8">
</script>
```

3.11 国际化

本项目默认集合了国际化 i18n 方案，通过 `vue-i18n` 实现，参考 `src/locale/index.js`

默认提供了简体中文、繁体中文、英语三种语言支持（点击右上角语言可以切换，并且自动识别系统设置语言），所有语言文件都在 `src/locale/lang` 下。

示例

在`src/locale/lang/zh-CN.js` 新增属性：

```
{
    hello: '你好，世界'
}
```

在`src/locale/lang/en-US.js` 新增属性：

```
{
    hello: 'Hello World'
}
```

菜单国际化

修改菜单meta的title属性，例如：

```
meta: {
    title: 'hello'
}
```

组件国际化

```
<h1>{{ $t('hello') }}</h1>
```

```
console.log(this.$t('hello'))
```

属性嵌套

国际化属性也支持嵌套，例如定义：

```
{
    hello: {
        world: '你好，世界'
    }

}
```

就可以通过 `hello.world` 的形式访问

4. 后端介绍

4.1 技术选型

后端使用的技术包括:

1. Java 1.8+
2. Spring Boot 2.0.2 release
3. Spring Framework 5.0+
4. Mybatis 3.4.6
5. Quartz
6. Ehcache

4.2 后端源码结构

下面是后端源代码的结构

```
├── src/main/java/com.foresealife.xxx/  // 后端项目源代码
│   ├── Application.java                 //Spring Boot应用的执行入口,包含了main方法
│   ├── RootConfig.java                 // 全局性的资源，如数据源、事务控制等配置
│   ├── config/                         // 分模块和类别的配置，如安全，任务调度等
│   ├── entity/                         // 数据类、实体类、DTO、或者bean的定义
│   ├── exception/                      // 自定义的异常
│   ├── mapper/                         // Mybatis Mapper接口类
│   ├── remoting/                       // 定义远程接口调用实现
│   ├── service/                        // 对应Spring的@Service
│   ├── task/                           // 异步、定时任务
│   ├── web/api/                        // REST类型的API接口
│   ├── web/controller/                 // 定义访问的页面，对应Spring的@Controller
├── src/main/resources/                 // 配置文件
│   ├── mapper/*.xml                   // Mybatis的SQL配置文件，与mapper接口一一对应
│   ├── application.properties         // Spring Framework 主配置文件
│   ├── ehcache.xml                    // ehcache 的缓存配置文件
│   ├── iam-client.properties          // IAM客户端配置文件
│   ├── log4j2.xml                     // 日志配置文件
│   ├── mybatis-config.xml             // Mybatis的全局配置文件
├── src/test/java/com.foresealife.xxx/  // 单元测试
├── target                             // 编译出的class、jar包
├── pom.xml                            // 后端项目的Maven POM
├── readme.MD                          // 源代码使用的说明，以Markdown格式编写
```

说明:

- com.foresealife.xxx 系统的顶级包名，通常xxx应该对应系统名或者模块名，如果使用booster-archetype

  > 生成源码，则取-Dpackage或者-DgroupId；

4.3 应用配置方式

系统的配置文件源码位于src/main/resources路径。

在开发阶段，默认情况下源码编译后会将此路径下所有文件复制到target/classes中，因此这些配置文件处于java classpath的根路径，如果生成jar包则jar包中也会包含这些文件。 在发布阶段，应考虑将需要放置在jar外的配置文件排除，部署时单独提供，以classpath成员的方式加入。

application.properties中包含了Spring Framework中各种各样的配置，详细内容请参考Common Applicaiton Properties。

mapper 路径下存放了Mybatis的Mapper XML 文件 ，mybatis-config.xml存放了Mybatis的XML 映射配置文件

log4j2.xml 为Log4j2日志框架需要的各种logger定义。

iam-client.properties 为IAM客户端配置文件。

其他的配置文件根据系统的实际情况，但是都要求放置在src/main/resources路径下。

相应地，对于单元测试是需要的各种配置，都应该放置在src/test/resources路径下。

4.4 日志

Booster框架中的日志采用slf4j+log4j2的模式, 即在源码中应该使用slf4j的API，而实际的日志输入是由Log4j2负责的。

当需要自定logger时，可以在log4j2.xml中增加：

```
<Configuration status="ERROR">
  <Appenders>
    ...
    <Console name="console" target="SYSTEM_OUT">
      <PatternLayout>
        <pattern>%d{ISO8601}|%level|%thread|%c{1.}|%L|%msg%n</pattern>
      </PatternLayout>
    </Console>
    ...
  </Appenders>
 <Loggers>
     ...
    <Logger name="com.foresealife" level="INFO" additivity="false">
      <AppenderRef ref="console"/>
    </Logger>
    ...
  </Loggers>
</Configuration>
```

详细说明请参考log4j2官方文档，这里只做简要说明：

- Appenders 定义了Log输出的目的地和格式，多个Logger可以共用1个Appender；
- Logger 定义了Log的来源，其中"name"对应了Java的package名称，可以对不同的package定义不同的logger来实现精确的日志控制；

在源代码中使用Log的方式如下：

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
...
private static final Logger LOG = LoggerFactory.getLogger(HeartbeatJob.class);
...
LOG.info("type_code: {} , type_desc: {} ", rs.getString("type_code"), rs.getString("type_desc"));
...
```

注意以下几点:

- 记录日志是不要使用字符串拼接，采用{}让日志框架自动将参数填到指定位置；
- 日志记录时需要区分不同的日志等级；
  具体请参考《日志开发规范》；

4.5 安全性

后端使用Spring Security来实现安全机制。

认证(Authentication)由Spring Security代理给IAM完成，Spring Security和IAM对接的代码已经在工程模板中自动生成，一般情况下不需要改动。各个业务系统只需要申请自己系统的iam-client.properties文件替换默认的即可。在调用IAM认证的过程中，不仅会验证用户名密码，而且会获取用户配置在本系统中的角色信息，此角色信息将用于授权。

授权(Authorization)由SecurityConfig.java类负责，采用Ant风格的URL路径表达式和角色之间建立映射关系的方式来实现权限控制。

例如：

```
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
        ...
                .antMatchers("/js/**","/css/**").permitAll()
                .antMatchers("/api/**").hasRole("USER")
                .anyRequest().authenticated();
        ...

    }
```

通过这段代码，我们实现了如下控制:

- 所有匹配/js/**和/css/**的URL允许任何人访问，不需要登录也可以；
- 所有匹配/api/**的URL只允许拥有USER角色的人访问，即必须登录且登录用户拥有USER角色；
- 除上述以外，所有其他URL必须要登录以后才能访问；

注意: 由于Spring Secrutiy的默认约定，代码中角色名为"USER",则认证源数据中角色名应该为"ROLE_USER",即IAM中要配置"ROLE_USER"角色。

另外，在IAM中提供了ACL的功能，我们可以通过自定义的AccessDecisionManager来集成，使得上述Spring Security的内置的授权和IAM的ACL共同起作用。

通过在SecurityConfig中配置自动以定义的AccessDecisionManager，并注册到Spring Security来启用：

```
    @Autowired
    private IAMAccessDecisionVoter accessDecisionVoter;

    private AccessDecisionManager accessDecisionManager() {
        List<AccessDecisionVoter<? extends Object>> decisionVoters = new ArrayList<>();
        decisionVoters.add(new WebExpressionVoter());
        decisionVoters.add(new RoleVoter());
        decisionVoters.add(new AuthenticatedVoter());
        decisionVoters.add(accessDecisionVoter);
        return new UnanimousBased(decisionVoters);
    }
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .cors().and().csrf().disable()
            .authorizeRequests().accessDecisionManager(accessDecisionManager())
            .antMatchers("/", "/js/**").permitAll()
            .antMatchers("/api/login").permitAll()
            .anyRequest().authenticated();
    }
```

我们还采用jasypt来对配置文件application.properties中存在的明文密码进行加密, 例如：

```
app.datasource.secondary.password=ENC(P7xVJnbrn/MCzyVEOejTRwrrerer)
...
jasypt.encryptor.algorithm=PBEWithMD5AndDES
jasypt.encryptor.password=mykey
```

说明:

- 凡是以ENC()包裹起来的配置项都是加密后密文，应用程序读取配置项时会自动根据jasypt.encryptor.algorithm和jasypt.encryptor.password进行解密。

4.6 事务控制

我们使用Spring Framework的事务控制机制,默认情况下使用基于数据源的事务控制。 事务控制器的申明位于RootConfig.java:

```
    @Bean
    public PlatformTransactionManager transactionManager(DataSource dataSource) {
        return new DataSourceTransactionManager(dataSource);
    }
```

这是一种单数据源的事务控制，使用大多数业务系统的情况。如果应用系统需要访问多个数据源，应该考虑：

1. 首先应该考虑架构设计上将访问不同数据库的逻辑划分在不同的微服务中，而不应该构建一个大而全的服务；

2. 如果的确需要在一个服务中访问两个数据源，如果需要将两个数据库的操作包含在一个事务中，那么需要考虑使用JTA事务控制器；
3. 如果的确需要在一个服务中访问两个数据源，如果不存在跨两个数据源的事务，则可以为每一个数据源申明一个事务控制器；

关于JTA事务控制器的申请，请参考Spring Framework的文档。

在事务控制器后，在源代码中通过@Transactional的注解来申请事务：

```
    @Transactional(readOnly = false, propagation = Propagation.REQUIRES_NEW)
    public void updateFoo(Foo foo) {
        // do something
    }
```

注意以下几点:

1. 如果交易中只包含单次数据操做，则无需申明事务;
2. 应该将事务的范围控制在最小，避免浪费数据库资源;
3. 应该设置事务超时值，避免长时间阻塞;
4. 如需回滚事务，但未指定异常类型，则需要将checked异常捕获并转换为运行时异常;
5. 凡是在包含事务的代码，不允许再直接操作底层数据库连接;
6. 应该尽量将事务控制放在Service层，而不是DAO层;
7. 应该尽量在实现类，而不是接口申明上申明事务;
8. 不能在私有的方法上申明事务;
9. 不能使用未申明事务的方法调用申明了事务的方法;
10. 应该尽量减少分布式事务，考虑将操作不同数据源的代码放到不同的服务中;

4.7 缓存

我们采用Spring Framework Cache Abstraction来实现缓存。

使用缓存时，我们只需要在public的方法增加@Cacheable注解，指定缓存名称和key，则方法的返回值自动加入缓存。在有效期内，后续的方法调用 将直接返回被缓存值，而不会调用实际的方法。 例如：

```
@Cacheable(cacheNames = {"userToken"})
public UsernamePasswordAuthenticationToken getAuthenticationToken(String username) {
         ...
        return new UsernamePasswordAuthenticationToken(userInfo, null, roles);
    }
```

缓存的过程可以理解为:

1. 当方法getAuthenticationToken首次调用时，Spring首先检查缓存userToken中的key为username的是否在有效期内；
2. 如果有则直接返回值，否则执行实际的方法获取值；
3. 获取到值后，可以username为key，将值放入缓存userToken中，然后返回值；

一般情况下，被缓存的值会根据缓存设定的清理策略，如超时、空闲或者容量限制FIFO等，从缓存中自动清理。但是，如果需要手工清理，则需要利用@CacheEvict注解。 例如：

```
@CacheEvict(cacheNames = {"userToken"})
public void logout(String username){
        ...
}
```

当方法logout被调用时，缓存userToken中可以为username的值将被清理出缓存。

我们采用ehcache来实现底层的缓存，配置文件位于src/resources/ehcache.xml,详细配置请参考官方文档。

关于缓存更多的用法请参考Spring Framework官方文档。

4.8 定时任务

我们采用Spring Framework和Quartz集成的方式来实现定时任务，在API调用使用Spring 提供的接口类，而不直接使用Quartz的API。

Quartz的配置文件位于src/resources/application.properties中，以spring.quartz.*开头的配置项为quartz的配置项。具体配置项的含义请参考文档, 重要的配置项如下：

- spring.quartz.job-store-type=memory
  持久化Job状态的方式，取值为memory或者jdbc,即将Job的状态放在内存中或者存入数据库
- spring.quartz.jdbc.initialize-schema=embedded
  如果选择job-store-type=jdbc, 那么相关数据表的初始化方式，取值为ALWAYS/EMBEDDED/NEVER, 分别对应：每次启动时都重新初始化数据库 ，使用内存数据库，从不初始化数据库。
- spring.quartz.jdbc.schema=classpath:org/quartz/impl/jdbcjobstore/tables_@@platform@@.sql
  数据库初始化脚本, 即用来建立quartz相关的数据库表的脚本，这里platform指数据库类型，支持的有:
  oracle,mysql,postgres,h2,hsqldb等具体脚本可以展开quartz-.jar 在路径org/quartz/impl/jdbcjobstore/下查看 。
- spring.quartz.properties.*=

  这里是quartz的自身的properties，需要根据quartz的文档来设置。

开发定时任务时，请按照如下步骤进行：

1.实现org.springframework.scheduling.quartz.QuartzJobBean接口 例如:

```
@PersistJobDataAfterExecution
public class HeartbeatJob extends QuartzJobBean {
    private static final Logger LOG = LoggerFactory.getLogger(HeartbeatJob.class);
    private int count;

    @Override
    protected void executeInternal(JobExecutionContext context)
            throws JobExecutionException {
        this.count++;
        LOG.info("{} Hi, I am still alive!", this.count);
        context.getJobDetail().getJobDataMap().put("count",this.count);
    }

    public void setCount(int count) {
        this.count = count;

    }
```

```
}
```

这里的注意:

1. 凡是private的成员变量，且提供了setter，quart会尝试从JobDataMap中找同名的key，并且将值注入;
2. 如果要抛出异常，请将异常封装为JobExecutionException后再抛出；
3. 如果需要持久化Job状态，如成员变量的值，则可以添加@PersistJobDataAfterExecution注解;

2.定义JobDetailFactoryBean

例如在SchedulerConfig.java中定义

```
    @Bean
    public JobDetailFactoryBean sampleJobDetail() {
        JobDetailFactoryBean jobDetailFactory = new JobDetailFactoryBean();
        jobDetailFactory.setJobClass(HeartbeatJob.class);
        jobDetailFactory.setName("HeartBeatJob");
        jobDetailFactory.setDescription("Say hello repeatably");
        jobDetailFactory.setDurability(true);
        return jobDetailFactory;
    }
```

3.定义SimpleTriggerFactoryBean

例如在SchedulerConfig.java中定义

```
    @Bean
    public SimpleTriggerFactoryBean sampleJobTrigger(JobDetail job) {
        SimpleTriggerFactoryBean trigger = new SimpleTriggerFactoryBean();
        trigger.setJobDetail(job);
        trigger.setRepeatInterval(10000);
        trigger.setRepeatCount(SimpleTrigger.REPEAT_INDEFINITELY);
        return trigger;
    }
```

注意: 这里的Trigger和JobDetail是一一对应的；

4.9 数据库访问

我们采用Mybatis来实现数据库访问。

Mybatis的配置文件位于:

- src/resources/mybatis-config.xml
  这是Mybatis的全局配置文件，可以根据XML映射配置文件进行配置，这里主要的作用是注册mapper
- src/resources/mapper/xxx_mapper.xml
  Mapper文件用于实现具体的SQL语句与接口中方法的对应，也用于定义ResultSet和JavaBean的映射。

  这里要求一个Mapper接口类对应一个XML配置文件。可以根据Mapper XML 配置文件进行配置。

开发步骤如下： 1.定义数据源 在application.properties中修改数据源配置:

```
#Oracle 数据库配置样例
spring.datasource.type=com.zaxxer.hikari.HikariDataSource
spring.datasource.driver=oracle.jdbc.driver.OracleDriver
spring.datasource.url=jdbc:oracle:thin:@10.50.64.1:1521:mydata
spring.datasource.username=*****
spring.datasource.password=*****

#Postgres 数据库配置样例
spring.datasource.type=com.zaxxer.hikari.HikariDataSource
spring.datasource.driver=org.postgresql.Driver
spring.datasource.url=jdbc:postgresql://xxx.db.foresealife.com:5503/mydata
spring.datasource.username=*****
spring.datasource.password=*****

#Mysql 数据库配置样例
spring.datasource.type=com.zaxxer.hikari.HikariDataSource
spring.datasource.driver=com.mysql.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/database
spring.datasource.username=*****
spring.datasource.password=*****
```

关于HikariCP的更多配置选项请参考文档,示例如下：

```
spring.datasource.hikari.maximumPoolSize=30
spring.datasource.hikari.connectionTimeout=30000
spring.datasource.hikari.idleTimeout=300000
spring.datasource.hikari.minimumIdle=10
spring.datasource.hikari.leakDetectionThreshold=10000
spring.datasource.hikari.maxLifetime=1800000
```

注意，以上都为单数据源的情况，如果一个应用中需要连接多个数据库，可以参考下面样例。
但是，为了避免分布式事务，应该考虑:

- 首先，应该从架构上是否应该将系统拆分为多个模块，每一个模块负责访问一个数据库；
- 其次，如果是处理ETL逻辑则应该将功能放到专门做ETL的系统中；

```
app.datasource.primary.type=com.zaxxer.hikari.HikariDataSource
app.datasource.primary.driver=oracle.jdbc.driver.OracleDriver
app.datasource.primary.url=jdbc:oracle:thin:@10.50.64.1:1521:mydata
app.datasource.primary.username=******
app.datasource.primary.password=******


app.datasource.secondary.type=com.zaxxer.hikari.HikariDataSource
app.datasource.secondary.driver=oracle.jdbc.driver.OracleDriver
app.datasource.secondary.url=jdbc:oracle:thin:@10.50.82.2:1521:core
app.datasource.secondary.username=******
app.datasource.secondary.password=******
```

在RootConfig.java中分别读取两个数据源的配置:

```
    @Bean
    @Primary
    @ConfigurationProperties("app.datasource.primary")
    public DataSourceProperties primaryDatasourceProperties() {
        return new DataSourceProperties();
    }

    @Bean("primary-datasource")
    @Primary
    @ConfigurationProperties("app.datasource.primary")
    public DataSource primaryDatasource() {
        return primaryDatasourceProperties().initializeDataSourceBuilder().build();
    }

    @Bean
    @ConfigurationProperties("app.datasource.secondary")
    public DataSourceProperties secondaryDatasourceProperties() {
        return new DataSourceProperties();
    }

    @Bean("secondary-datasource")
    @ConfigurationProperties("app.datasource.secondary")
    public DataSource secondaryDatasource() {
        return secondaryDatasourceProperties().initializeDataSourceBuilder().build();
    }
```

JNDI数据库配置

方法一，修改applicaiton.properties中关于spring.datasource的配置项如下：

```
spring.datasource.jndi-name=jdbc/jndidatasource
```

方法二，删除 `application.properties` 中，以 `spring.datasource.` 开头数据源相关的配置项，在 `RootConfig` 类中，添加方法：

```
private static final JNDI_NAME="jdbc/jndidatasource";
@Bean
public DataSource dataSource() {
    JndiObjectFactoryBean factory = new JndiObjectFactoryBean();
    factory.setJndiName(JNDI_NAME);
    try {
        factory.afterPropertiesSet();
    } catch (NamingException e) {
       LOGGER.error("Failed to get datasource from JNDI "+JNDI_NAME, e);
    }
    return (DataSource) factory.getObject();
}
```

这里的 `JNDI_NAME` 与容器中配置的JNDI名称对应，具体如何在容器中配置JNDI，属于中间件架构的内容，这里不再赘述。

2.定义Mapper接口类
例如，创建接口类com.foresealife.booster.dao.mapper.UserInfoMapper如下：

```
package com.foresealife.booster.dao.mapper;

import com.foresealife.booster.entity.UserInfo;
import org.apache.ibatis.annotations.Mapper;

@Mapper
public interface UserInfoMapper {
    UserInfo getUserInfo(int id);
}
```

注意: 需要添加@Mapper注解

3.创建Mapper XML配置文件 例如，创建src/main/resources/mapper/user_info_mapper.xml
内容如下:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN" "http://mybatis.org/dtd/mybatis-3-
mapper.dtd">
<mapper namespace="com.foresealife.booster.dao.mapper.UserInfoMapper">
  <select id="getUserInfo" resultType="UserInfo">
            select * from users where id=#{id}
    </select>
</mapper>
```

注意:

- "namespace"应该等于Mapper接口类的全名;
- select/delete/update/insert的id应该等于接口方法名；
- 接口参数与返回值需要与SQL查询语句的结果集类型兼容；

4.在需要调用Mapper的地方注入Mapper接口实例
例如：

```
package com.foresealife.booster.service.impl;

import com.foresealife.booster.dao.mapper.UserInfoMapper;
import com.foresealife.booster.entity.UserInfo;
import com.foresealife.booster.service.UserInfoService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class SimpleUserInfoService implements UserInfoService {
    @Autowired
    private UserInfoMapper userInfoMapper;
    @Override
    public UserInfo findUserInfo(int id) {
```

```
        return userInfoMapper.getUserInfo(id);
    }
}
```

这里需要理解为UserInfoMapper的实例是由Mybatis框架在运行时根据XML配置文件生成的。

注：如果需要控制SQL语句相关的日志，Logger应该对应接口类，例如为了显示执行的SQL语句，可以添加Logger：

```
    <Logger name="com.foresealife.booster.dao.mapper.UserInfoMapper" level="DEBUG"
additivity="false">
      <AppenderRef ref="console"/>
    </Logger>
```

4.10 IAM集成

我们采用了Spring Boot的自动配置机制来实现了booster-auth-spring-boot-starter，只需要在pom.xml中引入如下依赖既可以：

```
<dependency>
    <groupId>com.foresealife.booster</groupId>
    <artifactId>booster-auth-spring-boot-starter</artifactId>
    <version>1.1.0</version>
</dependency>
```

并且在classpath中提供IAM的客户端配置文件iam-client.properties，无需编码就可以实现与IAM的对接。

技术解释如下：

1. booster-auth-spring-boot-starter实现了一个基于IAM的AuthenticationProvider并且注册到了AuthenticationManager中；
2. booster-auth-spring-boot-starter注册了一个JWTAuthFilter来校验后续的请求中是否包含正确的Token；
3. booster-auth-spring-boot-starter提供了/api/login接口，用于校验IAM的用户名密码，如果校验通过则返回JWT的token；

接口的参数定义如下（实际调用时请采用对应的JSON或者XML格式）：

```
public class LoginRequest {
    private String username; //IAM用户名
    private String password;  //IAM密码
}
```

```
public class LoginResponse {
    private boolean success=false;  //是否登录成功
    private String message;         //登录异常消息
    private String token;           //如果登录成功，JWT的token
    private List<String> roles;     //用户在IAM中角色
    private String schema = "Bearer";  //后续返回头中需要提供的认证方式，目前为固定值
}
```

登录成功后，后续的请求头中都应该带上JWT的认证头，例如：

```
Authorization: Bearer <token>
```

4.11 监控接口

我们采用Spring Boot Actuator提供的监控接口。
配置方式如下：

- 在SecuriyConfig.java中增加监控路径授权,例如.antMatchers("/actuator/**").permitAll()
- 在配置文件application.properties中配置需要启用的监控接口,例如：
  management.endpoints.web.exposure.include=health,beans,metrics

注意，监控接口可能暴露应用的敏感信息，权限控制应该慎重。

通过HTTP客户端访问监控接口获取监控信息：

```
##获取可用监控接口列表
curl http://localhost:8080/actuator

##获取可以监控指标列表
curl http://localhost:8080/actuator/metrics

##获取连接池中空闲连接数
curl http://localhost:8080/actuator/metrics/hikaricp.connections.idle
```

4.12 调试

前后端分离开发时，经常需要独立调试。前后端分离是各自分工，协同敏捷开发，二者互不影响。

前端开发者主要会使用 `mock server` 进行模拟测试，后端人员采用 `junit` 进行API单元测试，或者使用 `Postman` 进行测试

5. 打包与部署

5.1 编译打包方式

在Booster框架中，我们使用Maven作为编译打包工具，spring-boot-maven-plugin插件提供了打包的功能。默认情况下，在pom.xml中已经包含如下配置：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <!-- ... -->
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <version>2.0.4.RELEASE</version>
                <executions>
                    <execution>
                        <goals>
                            <goal>repackage</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</project>
```

通过pom.xml中配置，spring-boot-maven-plugin可以自动将编译后的工程编译为可执行的jar包或者标准的war包。

5.1.1 编译为可执行jar包

在pom.xml中配置packaging为jar，如下:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <!-- ... -->
    <packaging>jar</packaging>
    <!-- ... -->
</project>
```

在执行如下命令后，编译后的源码会被打包为一个可执行的jar包。

```
$ mvn package
```

在target目录下回存在原始的jar包和可执行jar包，例如：

```
2018/08/21  17:04        28,411,897 booster-service-1.0.0.jar
2018/08/21  17:04            17,725 booster-service-1.0.0.jar.original
```

这里booster-service-1.0.0.jar.original是原始jar包，中包含了src目录下源代码编译后的class和资源。
booster-service-1.0.0.jar是整合jar包，包括所有依赖包和配置。
可以通过如下命里启动jar包：

```
java -jar booster-service-1.0.0.jar
```

可执行的jar可以以独立进程方式进行部署。

5.1.2 编译为war包

对于比较大型的Web应用，我们推荐war包部署的方式，这样，服务的启动/停止，JVM参数、连接池的配置可以由中间件管理，而不用开发者去管理了。这里以 `imeeting` 项目为例：

建议打包前去除 `installer` 包，删除 `imeeting-installer` 目录，并且修改根目录下的 `pom.xml` 文件，去除 `<module>imeeting-installer</module>` 节点。

5.1.2.1 前后端合并部署

1.修改 `imeeting-backend/pom.xml` ，`packaging` 改成 `war` ：

```
  <groupId>com.foresealife</groupId>
  <artifactId>imeeting-backend</artifactId>
  <version>1.0.0</version>
  <packaging>war</packaging>
<!--<packaging>jar</packaging>-->
```

2.找到`spring-boot-starter-web`依赖节点，在其中添加如下代码，

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <!-- 移除嵌入式tomcat插件 -->
    <exclusions>
        <exclusion>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-tomcat</artifactId>
        </exclusion>
        <exclusion>
            <groupId>org.apache.tomcat.embed</groupId>
            <artifactId>tomcat-embed-websocket</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

3.增加 `frontend` 的依赖，增加：

```
        <dependency>
            <groupId>com.foresealife</groupId>
            <artifactId>imeeting-frontend</artifactId>
            <version>1.0.0</version>
        </dependency>
```

4.增加 `war plugin` ：

```
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-war-plugin</artifactId>
                <version>3.0.0</version>
                <configuration>
                    <webResources>
                      <resource>
                         <directory>${project.parent.basedir}/imeeting-frontend/dist</directory>
                      </resource>
                    </webResources>
                    <warName>imeeting</warName> <!-- war包的名字 -->
                    <failOnMissingWebXml>false</failOnMissingWebXml>
                </configuration>
            </plugin>
```

5.修改启动类，将启动类 `Application` 修改为

```
@SpringBootApplication
public class Application extends SpringBootServletInitializer {

    public static void main(String[] args) {
        final SpringApplication application = new SpringApplication(Application.class);
        application.run(args);
    }

    @Override
    protected SpringApplicationBuilder configure(SpringApplicationBuilder application)   {
        return application.sources(Application.class);
    }

}
```

注意：该类继承了 `SpringBootServletInitializer` 并且重写了configure方法，这是关键所在。

最后，在 `imeeting` 根目录，执行命令 `mvn package` ，在 `imeeting-backend/target` 下，可以看到打好的war包（包括前后端代码）

5.1.2.2 前后端分离部署

该方式与上面方式大体上是一致的，去除第3步，去除第4步的 `<directory>...</directory>` 节点即可

```
    <!-- <dependency>
         <groupId>com.foresealife</groupId>
         <artifactId>imeeting-frontend</artifactId>
         <version>1.0.0</version>
     </dependency> -->
```

```
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-war-plugin</artifactId>
            <version>3.0.0</version>
            <configuration>
                <!--<webResources>
                  <resource>
                     <directory>${project.parent.basedir}/imeeting-frontend/dist</directory>
                    </resource>
                </webResources>-->
                <warName>imeeting</warName> <!-- war包的名字 -->
                <failOnMissingWebXml>false</failOnMissingWebXml>
            </configuration>
        </plugin>
```

在 `imeeting` 根目录，执行命令 `mvn package` ，在 `imeeting-backend/target` 下，可以看到打好的war包（仅后端代码），在 `imeeting-frontend/dist` 下，可以看到编译好的前端工程。

5.2 部署方式

5.2.1 作为独立进程启动

如上所述，我们可以将Booster框架编译为可执行jar包，因此部署时可以作为独立进程直接启动。在标准项目模板中，我们已经提供了适用于Unix/Linux平台的启动脚本。

启动脚本有配置文件和bash脚本组成：配置文件booster.conf用于定于启动相关的配置项，booster.sh用于启停Java进程。默认约定：

1. booster.conf必须位于booster.sh的同一路径或者booster.sh上一级路径下的config路径下；
2. booster.conf本身也是标准的bash脚本，只不过仅包含变量申明；

booster.conf的内容如下：

```
#!/bin/bash
#os user used to start the process
RUN_USER=jboss

#workding dir
WORKING_DIR="/qhapp/appsystems"

#configuartion/resources/properties
CONF_PATH="$WORKING_DIR/config"

#Additional jars and libs
LIB_PATH="$WORKING_DIR/lib"
```

```
#runnable jar file generated by spring boot maven plugin
APP_JAR="$WORKING_DIR/apps/imeeting.jar"

#application name
APP_NAME=imeeting

#Use to save the pid file
PID_FILE="$WORKING_DIR/${APP_NAME}.pid"

#log
LOG_DIR="/qhapp/applogs/$APP_NAME"
LOG_FILE="$LOG_DIR/${APP_NAME}_system_startup.log"

#max seconds waiting, when stop the process
STOP_WAIT_TIME=60

#if set to false, will run the command directly
USE_START_STOP_DAEMON=true

#JRE installation path, the default will be used if omitted
JAVA_HOME=

#JVM parameters
JAVA_OPTS="-Dsun.misc.URLClassPath.disableJarChecking=true -Xms1024m -Xmx1024m -
Dfile.encoding=utf-8 -XX:+PrintGCDetails -XX:+PrintGCDateStamps -XX:+UseGCLogFileRotation -
XX:NumberOfGCLogFiles=10 -XX:GCLogFileSize=10M -Xloggc:$LOG_DIR/${APP_NAME}_system_gc.log -
XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=$LOG_DIR/${APP_NAME}_<pid>.hprof"

#main class to start the application
MAIN_CLASS=org.springframework.boot.loader.JarLauncher

#program parameters,can be access from args[]
RUN_ARGS=
```

在默认的配置项中，我们假设应用安装后目录结构如下，如实际情况不同，则需要根据配置项含义调整booster.conf配置文件。

```
/qhapp/appsystems/apps
├── bin                          // 可执行脚本
│   ├── booster.sh               // booster启动脚本
├── config                       // 配置文件
│   ├── booster.conf             // booster启动配置文件
│   ├── application.properties   // 应用配置文件
│   ├── log4j2.xml               // 日志配置文件
│   ├── ...                      // 其他需要放到Java的classpath中的配置文件
├── lib                          // 依赖包库，一般不要，因为booster生成包中已经包含了所有依赖
│   ├── *.jar                    // 第三方可选依赖jar包
│   ├── ...

/qhapp/applogs/
├── <app_name>
```

```
|   ├── <app_name>_system_startup.log        // 进程启动日志
|   ├── <app_name>_system_gc.log             // JVM垃圾回收日志
|   ├── <app_name>_system_access.log         // 应用访问日志
|   ├── ...                            // 其他类型的日志文件
```

启动脚本支持的命令选项：

```
Usage: ./booster.sh {start|stop|force-stop|restart|force-reload|status|run}
start              启动Booster应用进程
stop               停止Booster应用进程
force-stop         强制停止进程，相当于kill -9
restart            重启进程
status             显示当前进程运行状态
run                直接执行启动命令，即不启动后台进程
```

5.2.3 以war包方式部署到应用服务器

由于Booster开发框架采用了Spring Boot 2, 其要求应用服务器兼容Servlet 3.1+, 常见的应用服务器要求：

- Tomcat 8.5+
- Jetty 9.4+
- JBoss 7+

具体的部署方式和普通的war包一样。

5.2.4 前后端跨域问题

有些系统前后端采取分离部署的方式，即前后端在不同的机器上（或者不同端口），

由于浏览器对于javascript的同源策略的限制。前端调用后端就会遇到跨域的问题。

开发环境

修改 `frontend/vue.config.js` 文件，`port` 是前端服务启动的端口，`targe` 是后端启动的地址 ，这样所有 `/api` 开头的请求都会被自动转发到后端上

```
  devServer: {
      port: 8081,
      open: true,
      proxy: {
        '/api': {
          target: 'http://localhost:8080/'
        }
      }
    }
```

参考 Vue Config

生产环境：

生产环境有两种方式，后端跨域或者nginx端口转发，选择其中一种即可

后端跨域配置简单，修改代码即可，但可能存在CSRF攻击漏洞，而nginx部署比较麻烦。

后端跨域设置

在 `SecurityConfig` 类中增加以下代码：

其中 `"http://localhost:8081"` 是前端的地址，如果前端地址不确定，可以以*代替

```
    @Bean
    public CorsConfigurationSource corsConfigurationSource() {
        CorsConfiguration configuration = new CorsConfiguration();
        configuration.setAllowedOrigins(Arrays.asList("http://localhost:8081"));
        configuration.setAllowedMethods(Arrays.asList("POST", "GET", "OPTIONS", "DELETE"));
        configuration.setAllowedHeaders(Arrays.asList("Content-Type", "Authorization"));
        UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();
        source.registerCorsConfiguration("/api/**", configuration);
        return source;
    }
```

nginx端口转发

建议配置为Nginx端口转发，参考[ngx_http_proxy_module]

这种方式可以实现前后端分离部署，在打包后，前端发布 `frontend\dist` 下的所有文件（可以放到nginx的html目录下），后端发布 `backend\target` 下的jar包即可

nginx的配置文件参考如下：

```
server {
    listen 80;
    server_name your.domain.name;
    location / {
        # 把跟路径下的请求转发给前端工具链（如gulp）打开的开发服务器
        # 如果是产品环境，则使用root等指令配置为静态文件服务器
        proxy_pass http://localhost:5000/;
    }

    location /api/ {
        # 把 /api 路径下的请求转发给真正的后端服务器
        proxy_pass http://localhost:8080/;
        # 把host头传过去，后端服务程序将收到your.domain.name，否则收到的是localhost:8080
        proxy_set_header Host $http_host;
    }
}
```