怎么理解 promise？关于 promise 常会有三种题型。第一种就是问概念；第二种就是 给一段代码让说出它的输出；第三种就是实现一个 promise ，来考察对 promise 的理解和编程能力（一般针对中高级程序员）。

promise 定义：A promise is an object that may produce a single value some time in the future: either a resolved value, or a reason that it’s not resolved (e.g., a network error occurred)。

```
console.time()
const openDoor=(cb)=>{
   setTimeout(cb,1000)
}
const putIn=(cb)=>{
   setTimeout(cb,3*1000)
}
const closeDoor=(cb)=>{
   setTimeout(cb,1000)
}
const done=()=>{
    console.timeEnd();
   console.log('done');
}
openDoor(()=> putIn(()=> closeDoor(done)));
```

改为用 promise 实现：

```
console.time()
const openDoor=()=>new Promise(res=>{
  setTimeout(res,1000)
})
const putIn=()=>new Promise(res=>{
  setTimeout(res,1000*3)
})
const closeDoor=()=>new Promise(res=>{
  setTimeout(res,1000)
})
const done=()=>new Promise(res=>{
      console.timeEnd();
  console.log('done2');
  res();
})
openDoor()
  .then(putIn)
  .then(closeDoor)
  .then(done)
```

类似 b.then().then().then() 的链式调用是如何实现的？

因为要不断的调用，所以一定是返回自己，或返回一个和自己类似的结构。后续实现 Promise 的时候，会用得上。

```
class Test1{
  then(){
    console.log(6666);
    return this;
  }
}
var a= new Test1();
a.then().then().then()
```

或

```
class Test2{
  then(){
    console.log(77777);
    return new Test2();
  }
}
var b= new Test2();
b.then().then().then()
```

promise 单元测试：

1.准备测试框架（参考 jest 官网）

```
# 1.初始化项目
npm init
# 2.安装 jest
yarn add --dev jest
#npm install --save-dev jest
# 3. 支持 es2015+ 和 ts
```

```
yarn add --dev babel-jest @babel/core @babel/preset-env @babel/preset-typescript @types/jest
# 4. 添加 文件 babel.config.js
// babel.config.js
module.exports = {
   presets: [
     ['@babel/preset-env', {targets: {node: 'current'}}],
      '@babel/preset-typescript',
   ],
};
# 5.配置命令
    "test": "jest",
    "test:watch": "jest --watchAll"
```

2.为了检查 Promise 实现的正确性，要提前准备好单元测试（用于测试的文件）

promise 源码（如何实现一个 Promise）？

promise 是一个对象，一般是通过 new Promise（）来实例化的；所以这里要实现 Promise 类。

promise 的 then 是可以链式调用的，会用到链式调用的实现。根据逐个单元测试的要求来实现 Promise。

主要实现 Promise 的构造方法和 then 方法。

项目中的网络/api 请求模块是如何封装的（为什么要封装请求模块）？

1.接口请求一般是异步的，可以返回 promise，这样更加清晰。

2.网络请求 url 的公共部分可以单独配置到 网络请求内部。

3.针对所有的接口可以进行统一的处理，这也是面向切面编程的一个实践。

4.可以借助第三方库 axios 快速的封装 网络请求模块。

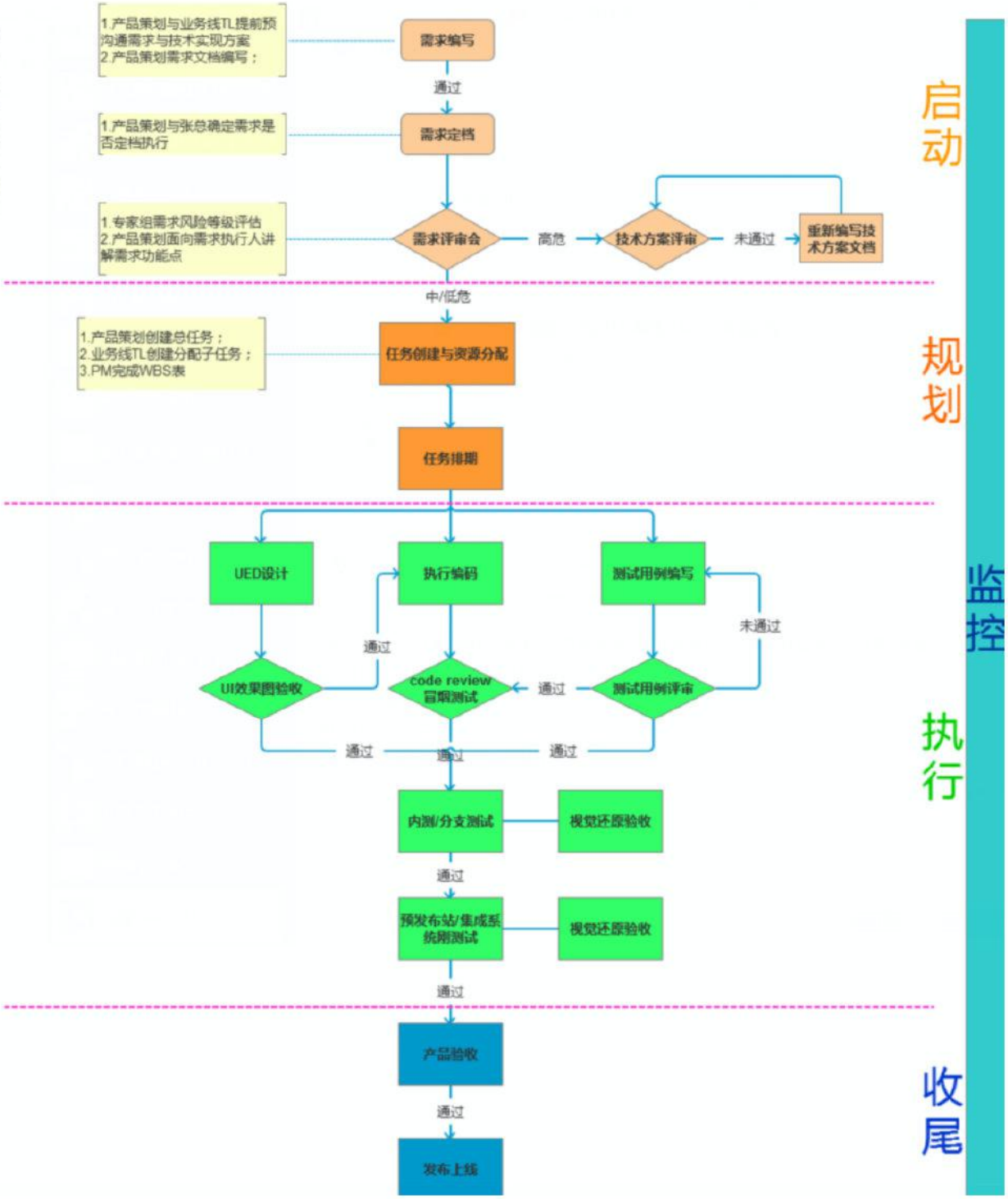
```
import axios from "axios";
import constant from "../constant";
import reactNavigationHelper from "./reactNavigationHelper";
import commonToast from "./commonToast";
//配置请求 url 的公共部分及超时时间
const commonHttp = axios.create({
  baseURL: constant.baseUri,
  timeout: 10 * 1000
});
commonHttp.interceptors.response.use(    //中间件拦截器？
  function(response) {
    return response;
  },
  function(error) {
    //针对所有接口统一处理登录过期的问题
    if (error.response.status === 401) {
      commonToast.show("登录过期");
      reactNavigationHelper.navigate("Login");
    }
    return Promise.reject(error);
  }
);

export default commonHttp;
```

一般公司与项目流程有关的角色和职责：

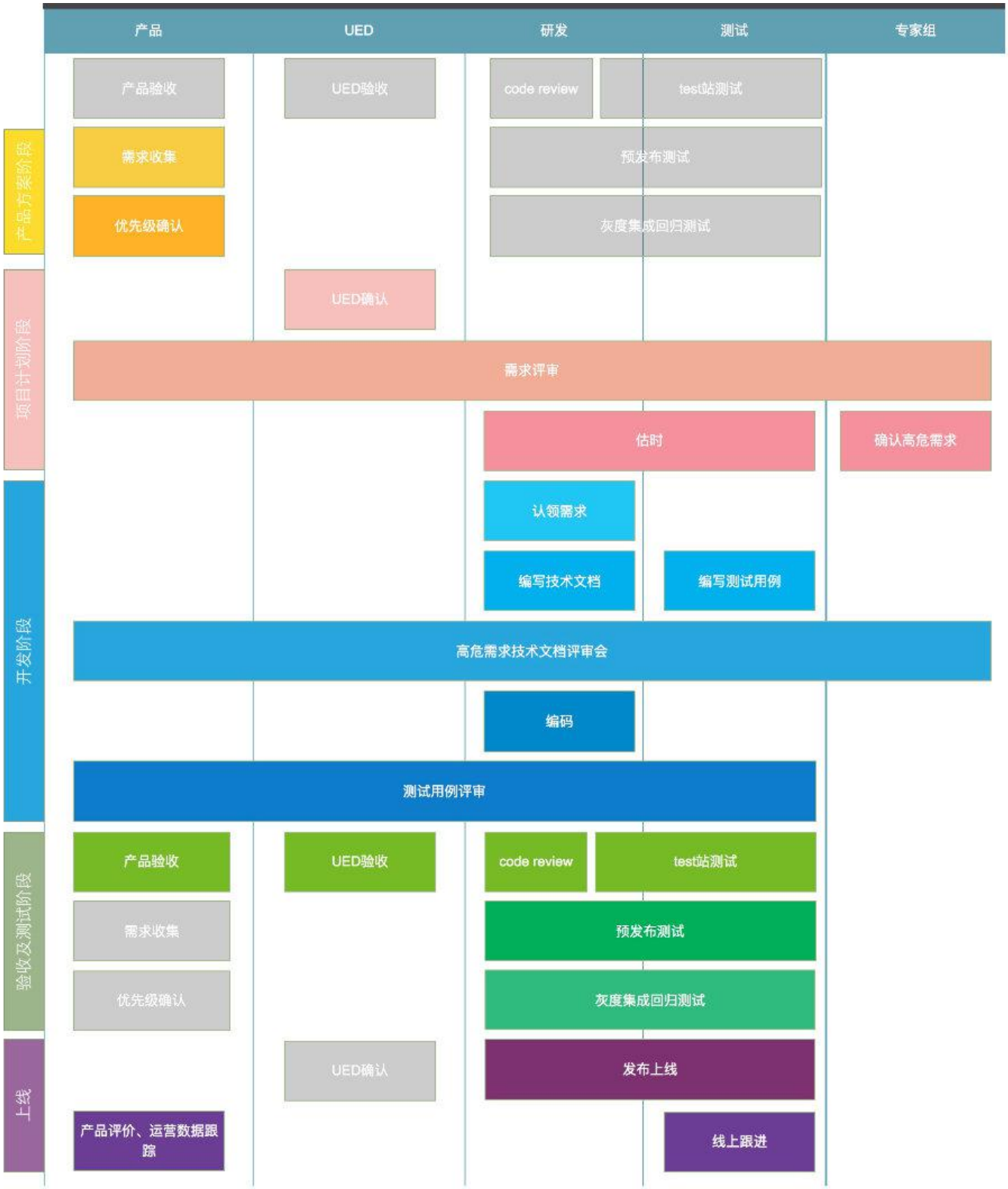
序号	部门/角色	职责
1	产品策划	1) 负责项目需求产品文档的编写的审核； 2) 负责组织项目需求分析会议； 3) 负责创建项目需求总任务。
2	项目管理组	1) 负责组织项目需求风险等级评估； 2) 负责组织项目需求技术方案评审会； 3) 负责对项目需求整体研发流程进行管理监控指导。
3	PM	1) 负责组织项目团队完成技术方案设计的讨论和编制； 2) 负责完成技术方案评审会议中的讲解和意见收集工作； 3) 负责协助产品策划完成项目团队 WBS 任务分解和进度计划的制定； 4) 负责整个项目需求开发过程中的技术风险管控和项目推进。
4	研发人员	1) 负责按照任务分配完成编码及联调工作，以及针对测试提出的 BUG 进行及时修复，保障所负责功能模块的正常运行； 2) 负责按照冒烟测试用例和提测标准完成代码自查。
5	测试人员	1) 负责提供各类质量保证标准文档，如：冒烟测试用例、提测标准； 2) 负责按照任务分配完成各阶段测试工作。
6	UED 部门	负责完成产品策划提出的各类项目需求相关的 UI 和 UE 设计工作。
7	大数据部门	负责完成产品策划提出的各类项目需求相关的数据分析及统计需求。
8	职能部门	负责配合产品策划提供研发人力资源和技术方面的项目需求等相关支持。
9	业务线负责人	1) 负责规划其业务线发展方向及相关需求业务； 2) 负责协助跨业务线之间的需求沟通对接工作。
10	业务线 TL	1) 负责管理业务线研发人员的工作任务分配； 2) 负责监督、管理和指导业务线内研发人员的工作任务； 3) 负责协助产品策划分析和把控需求分析会议中的技术风险。
11	业务线专家组	1) 负责评审项目需求风险等级； 2）负责评审项目需求技术方案设计。

开发流程/项目流程大概是什么样的？

业务线开发流程图


1.产品策划与业务线TL提前预沟通需求与技术实现方案
2.产品策划需求文档编写；
需求编写
通过
1.产品策划与张总确定需求是否定档执行
需求定档
1.专家组需求风险等级评估
2.产品策划面向需求执行人讲解需求功能点
需求评审会
高危
技术方案评审
未通过
重新编写技术方案文档
中/低危
启动
1.产品策划创建总任务；
2.业务线TL创建分配子任务；
3.PM完成WBS表
任务创建与资源分配
任务排期
规划
UED设计
执行编码
测试用例编写
通过
未通过
UI效果图验收
code review
冒烟测试
测试用例评审
通过
通过
通过
通过
内测/分支测试
视觉还原验收
通过
预发布站/集成系统测试
视觉还原验收
通过
执行
产品验收
通过
发布上线
收尾
监控

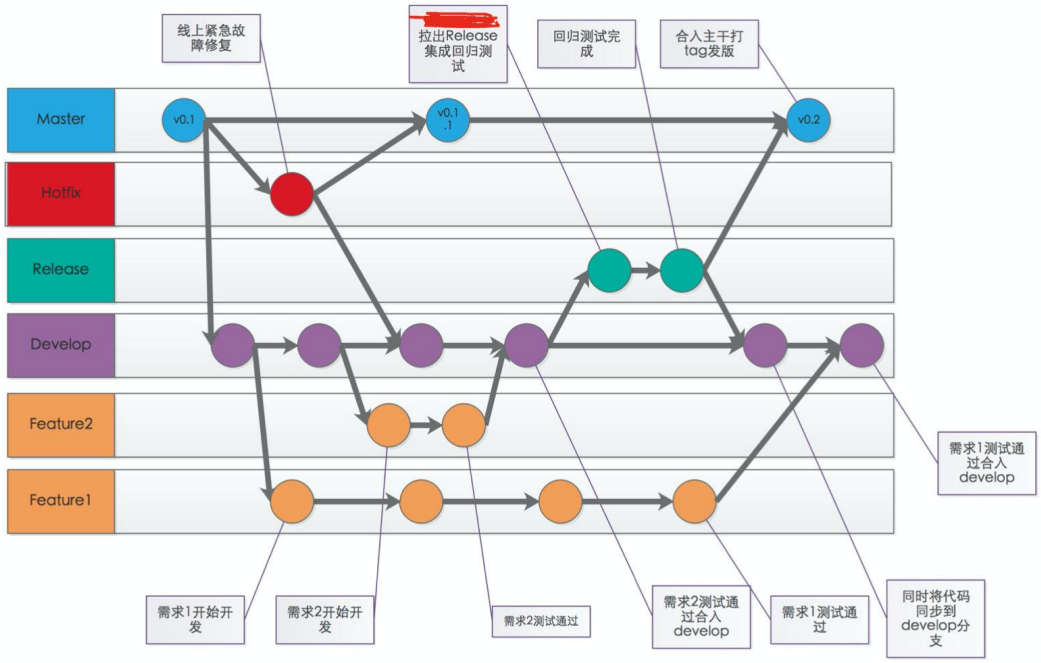
研发流程管理规范描述					
阶段	关键项	输入	输出	责任人	说明
启动	需求评审会 （需求风险评估会）	需求文档	需求排期（产品策划完成） 需求风险等级（由项目助理完成）	产品策划 项目管理人员	1.产品策划提前发送会议邀请； 2.需求讲解沟通； 3.会后确定项目PM；
	技术方案评审会议	技术方案文档	通过的技术方案报告	需求PM 业务线专家组 项目管理人员	1.需求通过后，产品策划提前发送会议邀请（见相关模板）； 2.需求讲解沟通； 3.记录需求疑问及风险
规划	建立总任务	通过技术方案评审的需求	Redmine总任务	产品策划	1.产品策划在解牛项目管理平台完成总任务创建
	任务分配	总任务	Redmine子任务	业务线TL	1.TL完成子任务创建
	UED设计	产品策划申请邮件	UED部门任务	产品策划 UED部门负责人	1.产品策划与UED部门沟通
	大数据打点	产品策划申请邮件	大数据部门任务	产品策划 大数据部门负责人	1.产品策划与大数据部门沟通
	职能部门需求	产品策划申请邮件	职能部门任务	产品策划 职能部门负责人	1.产品策划与职能部门沟通
	跨组需求	产品策划申请邮件	跨业务线任务	产品策划 业务线负责人	1.产品策划与各业务线负责人沟通
	WBS完成	需求文档	技术WBS文档	PM	PM组织完成工作任务拆分
执行	进度推进	执行过程问题	解决方案	产品策划	需求问恩解答
				PM	技术方案问题推进
				业务线TL	技术问题解答
	交付测试 （live站）	完成的执行内容	发起live站部署邮件	产品策划	发部署邮件
		live站部署邮件	回复live站部署邮件	需求执行人	回复部署邮件
		live站部署邮件	回复live站测试报告	测试人员	回复测试结果邮件
	交付验收 （预发布站）	live站测试报告	发起提交验收部署邮件	产品策划	发部署邮件
		提交验收部署邮件	回复预发布站部署邮件	需求执行人	回复部署邮件
		提交验收部署邮件	回复预发布验收报告	测试人员	回复测试结果邮件
监控	进度	流程缺陷	流程指导和纠正	产品策划 项目管理人员	提供流程指导和进度推进
	需求变更	需求变更	需求变更邮件 需求文档更新	产品策划	发起需求变更邮件通知
		技术方案变更	技术方案评审申请	需求PM 项目管理人员	重新进行技术方案评审
	代码审核	完成的代码	部门内部流程	业务线TL	完成代码审核
	质量	提交的功能	测试验收报告	测试人员	完成质量验收
收尾	验收	回复预发布验收报告	发布验收报告	产品策划	完成最终项目验收报告
	线上跟踪	发布上线的功能	结案报告	产品策划 开发人员 测试人员	上线功能跟踪和质量保障
	项目总结报告	成功发布上线	项目总结文档	项目管理人员 产品经理	收尾项目



如何做项目缺陷（bug）管理？

序号	角色	职责
1	测试人员	主要指发现和报告缺陷的测试人员。 1. 缺陷上报 2. 缺陷回归测试 3. 缺陷关闭
2	开发人员	主要指对缺陷进行调查和修复的开发人员。 1. 缺陷修复
3	策划人员	主要指软件的产品策划人员 1. 缺陷修复（需求文档类缺陷） 2. 延期修复状态的缺陷确认
4	项目管理	主要指关注项目过程质量的管理人员。 1. 项目缺陷报表数据查看 2. 缺陷进度跟进

代码大概怎么管理的（怎么管理 git 分支的）？



项目为什么需要打包/编译/构建？

1.es2015+ 转 es5

2.Js/css/图片等文件合并

3.ts 转 js

4.模板编译 jsx 转 js

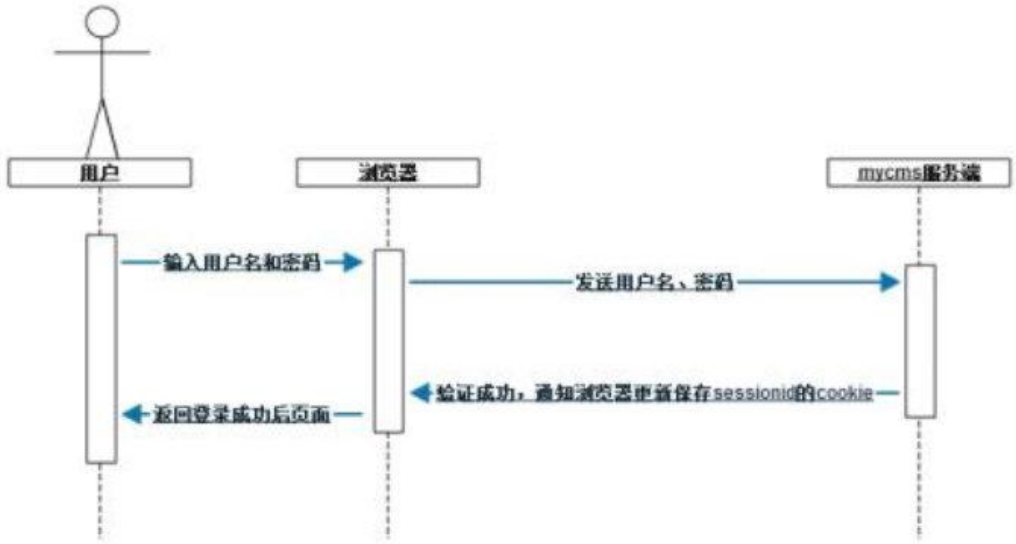
5.less/sass 转 css

6.代码压缩

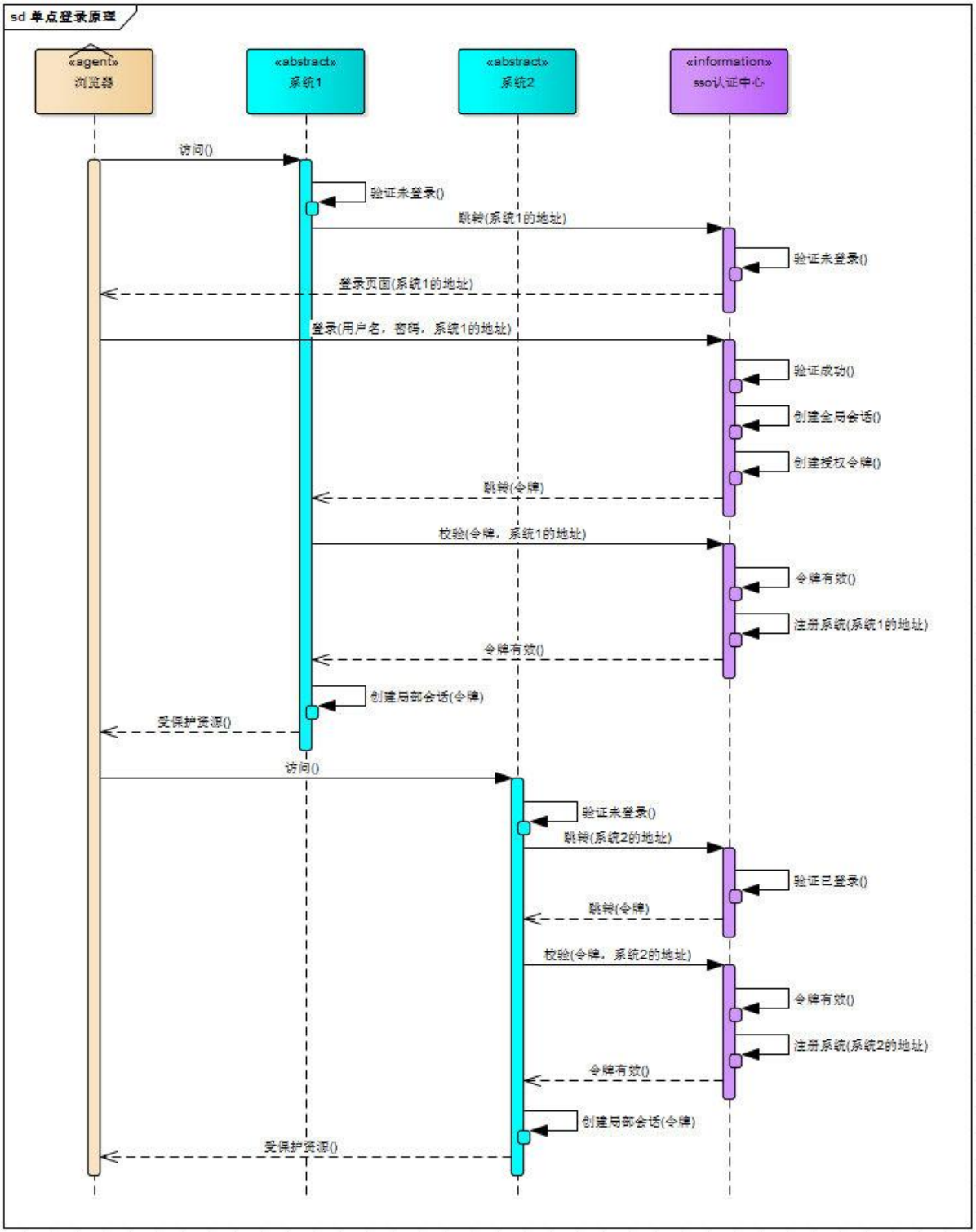
7.针对不同环境生成不同的代码

项目中登录大概是怎么实现？

1.最简单的登录流程：



2.端点登录-sso


sd 单点登录原理
«agent»
浏览器
«abstract»
系统1
«abstract»
系统2
«information»
sso认证中心
访问()
验证未登录()
跳转(系统1的地址)
验证未登录()
登录页面(系统1的地址)
登录(用户名，密码，系统1的地址)
验证成功()
创建全局会话()
创建授权令牌()
跳转(令牌)
校验(令牌，系统1的地址)
令牌有效()
注册系统(系统1的地址)
令牌有效()
创建局部会话(令牌)
受保护资源()
访问()
验证未登录()
跳转(系统2的地址)
验证已登录()
跳转(令牌)
校验(令牌，系统2的地址)
令牌有效()
注册系统(系统2的地址)
令牌有效()
创建局部会话(令牌)
受保护资源()

项目中权限管理是怎么实现？

1.常规权限模型：RBAC 基于角色的权限访问控制（Role-Based Access Control）。

2.常规管理系统中的权限：根据登录用户返回菜单列表（可能是一颗树）。

3.根据登录用户控制接口权限：前台系统中的权限管理。

如何根据数组构建树？

单元测试：

```
import arrayToTree from "./arrayToTree";
import { originData, treeData } from "./data";

test("arrayToTee", () => {
  expect(arrayToTree(originData)).toEqual(treeData);
});
```

实现步骤：

```
import { Item, Node } from "./data";
const arrayToTree = (arr) => {
  if (!Array.isArray(arr) || arr.length < 1) return null;
  const [root] = arr.filter(item => item.parentId === null);
  const addChildren = (node, dataList) => {
    const children = dataList
      .filter(item => item.parentId === node.id)
      .map(item => addChildren(item, dataList));
    return { ...node, children };
  };
  return addChildren(root, arr);
};
export default arrayToTree;
```

如何把树的数据转成数组？

单元测试：

```
import treeToArray from "./treeToArray";
import { originData, treeData } from "./data";
test("arrayToTee", () => {
  expect(
    treeToArray(treeData).sort((a, b) => (a.name > b.name ? 1 : -1))
  ).toEqual(originData.sort((a, b) => (a.name > b.name ? 1 : -1)));
});
```

解题步骤：

```
import { Node, Item } from "./data";
const treeToArray = (node: Node) => {
  const nodeToArray = (node: Node, arr: Item[]) => {
    const { children, ...item } = node;
    arr.push(item);
    children.forEach(child => nodeToArray(child, arr));
    return arr;
  };
  return nodeToArray(node, []);
};
export default treeToArray
```

如何限制接口访问频次？接口请求部分的逻辑可以封装成一个函数 fun。假设存在一个处理函数的函数 better，接受 fun，返回一个 betterFun。当 betterFun 被多次调用，fun 只在必要的时候执行。const betterFun=better(fun);
限制接口访问一般涉及到用户的输入。主要有两个经典场景：
debounce（防抖）（在多段连续的时间内只执行最后那次时间的一次）：强制函数在某段时间内只执行一次；
throttle （截流）（平均在每段时间里的每次时间固定的执行一次，可以在每次时间的刚开始或刚结束执行函数）：强制函数以固定的速率执行。

1.debounce 单元测试

```
import debounce from "./debounce";
jest.useFakeTimers();
test("debounce", () => {
  const func = jest.fn();
  const debounced = debounce(func, 1000);
  debounced();
  debounced();
  jest.runAllTimers();
  expect(func).toHaveBeenCalledTimes(1);
});
test("debounce twice", () => {
  const func = jest.fn();
  const debounced = debounce(func, 1000);
  debounced();
  debounced();
  setTimeout(debounced, 20);
  setTimeout(debounced, 2000);
  jest.runAllTimers();
  expect(func).toHaveBeenCalledTimes(2);
});
```

实现 debounce

```
const debounce = (fn, delay: number) => {
  let inDebounce;
  return function(...args) {
    const context = this;
    clearTimeout(inDebounce);
    inDebounce = setTimeout(() => {
      fn.apply(context, args);
    }, delay);
  };
};
export default debounce;
```

实现 throttle

```
/*
* @param fn {Function}   实际要执行的函数
* @param delay {Number}   执行间隔，单位是毫秒（ms）
* @return {Function}     返回一个“节流”函数
*/
function throttle(fn, threshold) {
  // 记录上次执行的时间
  var last
  // 定时器
  var timer
  // 默认间隔为 250ms
  threshold || (threshold = 250)
  // 返回的函数，每过 threshold 毫秒就执行一次 fn 函数
  return function () {
    // 保存函数调用时的上下文和参数，传递给 fn
    var context = this
    var args = arguments
    var now = +new Date()
```

```
    // 如果距离上次执行 fn 函数的时间小于 threshold，那么就放弃
    // 执行 fn，并重新计时
    if (last && now < last + threshold) {
      clearTimeout(timer)
      // 保证在当前时间区间结束后，再执行一次 fn
      timer = setTimeout(function () {
        last = now
        fn.apply(context, args)
      }, threshold)
    // 在时间区间的最开始和到达指定间隔的时候执行一次 fn
    } else {
      last = now
      fn.apply(context, args)
    }
  }
}
```

debounce.leading（截流的函数先执行） vs debounce.trailing（截流的函数后执行）

```
function debounce(func, wait, immediate) {
    var timeout;
    return function() {
        var context = this, args = arguments;
        var later = function() {
            timeout = null;
            if (!immediate) func.apply(context, args);
        };
        var callNow = immediate && !timeout;
        clearTimeout(timeout);
        timeout = setTimeout(later, wait);
        if (callNow) func.apply(context, args);
    };
};
```

一般是怎么进行前后端联调的？

1.项目是前后端分离的项目；后端负责提供数据接口，前端负责页面。

2.一般接口完成之后，后端会给一份接口文档，包含接口地址，请求类型，请求的参数，可能的数据返回。

3.部分后端可能不写文档，会进行口头沟通。口头沟通的时候需要记录必要的信息。

4.联调的时候最好是工作时间（免得晚上，别人下班了），因为可能涉及到沟通。

5.最好开发之前想好你需要哪些数据，然后和后端提前沟通一下。避免一下你需要的数据，后端没有开发，导致项目工期延误。