学习方法

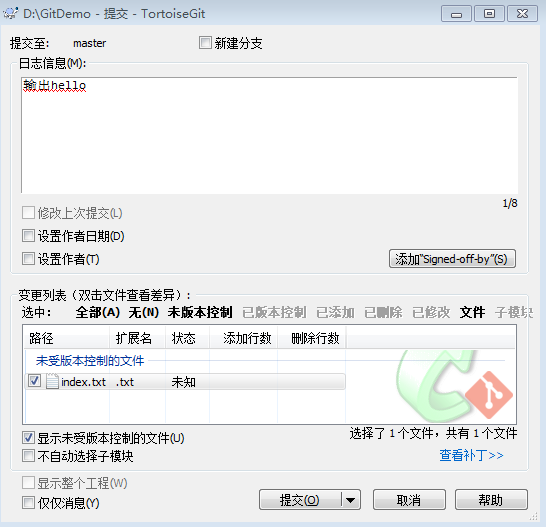
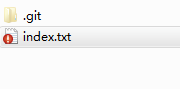
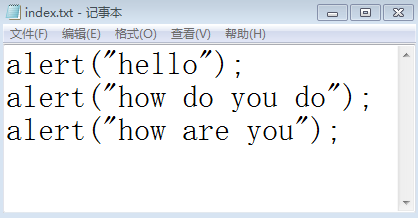
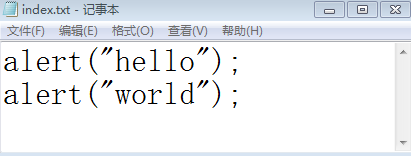
1. 建立逻辑思维
   1. 多分析
   2. 理解逻辑算法
   3. 多练、多敲、多读
2. 编程习惯
3. 大胆动手
4. 保持好奇好
5. 乐观面对错误

就业能力

1. 分析能力
2. 独立思考能力
3. 程序阅读能力
4. 程序排错能力
5. 独立编写程序能力
6. 团队协作能力
7. 学习能力
8. 善于总结的能力

专业技能

git/模块化开发

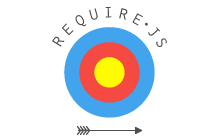
1. git和SVN的区别
   1. git是什么？
      1. git是目前世界上最先进的分布式版本控制系统（没有之一）。
      2. 什么是版本控制系统？
         1. 如果你用Microsoft Word写过长篇大论，那你一定有这样的经历：想删除一个段落，又怕将来想恢复找不回来怎么办？有办法，先把当前文件“另存为……”一个新的Word文件，再接着改，改到一定程度，再“另存为……”一个新文件，这样一直改下去，最后你的Word文档变成了这样：过了一周，你想找回被删除的文字，但是已经记不清删除前保存在哪个文件里了，只好一个一个文件去找，真麻烦。看着一堆乱七八糟的文件，想保留最新的一个，然后把其他的删掉，又怕哪天会用上，还不敢删，真郁闷。更要命的是，有些部分需要你的财务同事帮助填写，于是你把文件Copy到U盘里给她（也可能通过Email发送一份给她），然后，你继续修改Word文件。一天后，同事再把Word文件传给你，此时，你必须想想，发给她之后到你收到她的文件期间，你作了哪些改动，得把你的改动和她的部分合并，真困难。于是你想，如果有一个软件，不但能自动帮我记录每次文件的改动，还可以让同事协作编辑，这样就不用自己管理一堆类似的文件了，也不需要把文件传来传去。如果想查看某次改动，只需要在软件里瞄一眼就可以，岂不是很方便？  
             这个软件用起来就应该像这个样子，能记录每次文件的改动：  
            版本 用户 说明 日期  
            1 张三 删除了软件服务条款5 7/12 10:38  
            2 张三 增加了License人数限制 7/12 18:09  
            3 李四 财务部门调整了合同金额 7/13 9:51  
            4 张三 延长了免费升级周期 7/14 15:17
   2. git的诞生
      * 1. 很多人都知道，Linus在1991年创建了开源的Linux，从此，Linux系统不断发展，已经成为最大的服务器系统软件了。Linus虽然创建了Linux，但Linux的壮大是靠全世界热心的志愿者参与的，这么多人在世界各地为Linux编写代码，那Linux的代码是如何管理的呢？事实是，在2002年以前，世界各地的志愿者把源代码文件通过diff的方式发给Linus，然后由Linus本人通过手工方式合并代码！你也许会想，为什么Linus不把Linux代码放到版本控制系统里呢？不是有CVS、SVN这些免费的版本控制系统吗？因为Linus坚定地反对CVS和SVN，这些集中式的版本控制系统不但速度慢，而且必须联网才能使用。有一些商用的版本控制系统，虽然比CVS、SVN好用，但那是付费的，和Linux的开源精神不符。不过，到了2002年，Linux系统已经发展了十年了，代码库之大让Linus很难继续通过手工方式管理了，社区的弟兄们也对这种方式表达了强烈不满，于是Linus选择了一个商业的版本控制系统BitKeeper，BitKeeper的东家BitMover公司出于人道主义精神，授权Linux社区免费使用这个版本控制系统。安定团结的大好局面在2005年就被打破了，原因是Linux社区牛人聚集，不免沾染了一些梁山好汉的江湖习气。开发Samba的Andrew试图破解BitKeeper的协议（这么干的其实也不只他一个），被BitMover公司发现了（监控工作做得不错！），于是BitMover公司怒了，要收回Linux社区的免费使用权。Linus可以向BitMover公司道个歉，保证以后严格管教弟兄们，嗯，这是不可能的。实际情况是这样的：Linus花了两周时间自己用C写了一个分布式版本控制系统，这就是Git！一个月之内，Linux系统的源码已经由Git管理了！牛是怎么定义的呢？大家可以体会一下。Git迅速成为最流行的分布式版本控制系统，尤其是2008年，GitHub网站上线了，它为开源项目免费提供Git存储，无数开源项目开始迁移至GitHub，包括jQuery，PHP，Ruby等等。历史就是这么偶然，如果不是当年BitMover公司威胁Linux社区，可能现在我们就没有免费而超级好用的Git了。
   3. 集中式VS分布式（SVN VS GIT）
      1. Linus一直痛恨的CVS及SVN都是集中式的版本控制系统，而Git是分布式版本控制系统，集中式和分布式版本控制系统有什么区别呢？
         1. 先说集中式版本控制系统，版本库是集中存放在中央服务器的，而干活的时候，用的都是自己的电脑，所以要先从中央服务器取得最新的版本，然后开始干活，干完活了，再把自己的活推送给中央服务器。中央服务器就好比是一个图书馆，你要改一本书，必须先从图书馆借出来，然后回到家自己改，改完了，再放回图书馆。  
              
            集中式版本控制系统最大的毛病就是必须联网才能工作，如果在局域网内还好，带宽够大，速度够快，可如果在互联网上，遇到网速慢的话，可能提交一个10M的文件就需要5分钟，这还不得把人给憋死啊。那分布式版本控制系统与集中式版本控制系统有何不同呢？首先，分布式版本控制系统根本没有“中央服务器”，每个人的电脑上都是一个完整的版本库，这样，你工作的时候，就不需要联网了，因为版本库就在你自己的电脑上。既然每个人电脑上都有一个完整的版本库，那多个人如何协作呢？比方说你在自己电脑上改了文件A，你的同事也在他的电脑上改了文件A，这时，你们俩之间只需把各自的修改推送给对方，就可以互相看到对方的修改了。和集中式版本控制系统相比，分布式版本控制系统的安全性要高很多，因为每个人电脑里都有完整的版本库，某一个人的电脑坏掉了不要紧，随便从其他人那里复制一个就可以了。而集中式版本控制系统的中央服务器要是出了问题，所有人都没法干活了。在实际使用分布式版本控制系统的时候，其实很少在两人之间的电脑上推送版本库的修改，因为可能你们俩不在一个局域网内，两台电脑互相访问不了，也可能今天你的同事病了，他的电脑压根没有开机。因此，分布式版本控制系统通常也有一台充当“中央服务器”的电脑，但这个服务器的作用仅仅是用来方便“交换”大家的修改，没有它大家也一样干活，只是交换修改不方便而已。  
            
   4. git的使用
      1. 下载git：<https://git-scm.com/download/win>
      2. 下载tortoisegit：<https://tortoisegit.org/download/>
      3. 先安装git，安装好后，右键，会发现菜单多了几项关于git的选项
      4. 安装tortoisegit,安装好以后，右键，会发现菜单多了几项关于tortoisegit的选项
   5. 使用tortoisegit
      1. git属于分布式版本控制器，其实每个人的电脑都是一个完整的版本库；比如两个人在局域网写一个项目，那么这两个人的电脑里面都是一个完整的项目，当有新的代码出现的时候，直接push给对方即可。当然人多的时候，又不在局域网里面，不可能push给每一个人，git的解决方案是用一台电脑当做集中的版本库，大家把新代码都push到上面去，要得时候在pull到自己电脑上。国外比较著名的就是github了,国内的oschina也广为人知。
         1. <https://git.oschina.net/> 码云
      2. 创建一个版本库
         1. 创建一个文件夹GitDemo，然后  右键-在这里创建版本库
         2. 成功后，在GitDemo文件夹下会多一个.git文件夹（隐藏状态），此文件夹，尽量不要乱修改或删除。
         3. 创建一个文件，然后提交(commit)到版本库【这个操作可以在离线状态操作】
         4. 右键，选择Git提交->”master”
         5. 填写日志信息：  
            
         6. 然后在index中添加一个新的语句。当文件代码改变，那么文件会显示为一个红色感叹号  
            
         7. 将新增的代码，提交到版本库中。参考4和5步。
         8. 查看日志：右键-TortoiseGit-显示日志  
            在日志中，文件更改的信息，都很清楚。  
            
         9. git还有个特别好用的功能，分支。  
            发现问题：你代码写了很多，运行OK；但是突然想加个新功能进去，这个功能你也不知道能否正常运行，而且修改过程中，除了新加代码和文件进去，还会修改以前的代码。要是万一失败，修改回来也是一种很麻烦的事情。这种时候很多人就用备份方式。来看看git是怎么优雅的处理这个问题的。  
            git的处理方式：当你想加一个新功能进去的时候，你可以新建一个分支，例如名字叫newfunction，然后在分支中把新功能加上去，如果OK，将代码合并到master分支上，如果新功能失败，切换回master分支上来，在newfunction写的代码，又全看不到了。
            1. 新建一个分支，右键-tortoiseGit-创建分支  
                 
               右键，你会发现当前的分支为你新建的newfunction分支了（Git commit-“newfunction”）
            2. 接下来，就开心新增你的功能；我们再index.txt中不仅增加了输出how are you方法，还修改了第二条输出how do you do。  
               
            3. 提交我们的代码，此时我们发现这个新功能不行，存在很多漏洞等等问题。我们想回到之前的代码。怎么办呢？
            4. 为了回到之前的项目，我们直接切回master分支。  
               右键-tortoiseGit-切换/检出  
               选择master即可，然后确定  
                 
               再来看看index中的代码是否回到了从前呢？  
                 
               无论是修改，还是添加，都没影响到复原。这就是分支的作用。这个功能是不是让你省去了很多工作量呢。
2. Hbuilder的插件使用
   1. 工具-》插件安装
   2. EGIT-》安装
   3. 共享项目
      1. 项目上右键点击
      2. 选择Team-》共享项目
      3. 在Repository中选择一个本地的GIT工作区
         1. 工作区可以先选择一个目录文件夹
         2. 点击Greate按钮，会将选中的文件夹创建成git工作区
      4. 点击完成，就会将要共享的项目剪贴到工作区中
         1. 以后修改和提交的项目，都是git管理的项目，可以进行修改、提交、回退等等各种操作。

第二节：应用

1. 掌握模块化的思想
2. 掌握AMD规范

第三节

1. requireJS模块化开发
   1. 使用模块化开发的好处：  
      **通过 exports 暴露接口**。这意味着不需要命名空间了，更不需要全局变量。这是一种彻底的命名冲突解决方案。  
      **通过 require 引入依赖。**这可以让依赖内置，开发者只需关心当前模块的依赖，其他事情 Sea.js/ Require.js 都会自动处理好。对模块开发者来说，这是一种很好的 关注度分离，能让程序员更多地享受编码的乐趣。
   2. 实现JS模块化开发，目前有两个规范，一个是AMD规范，一个是CMD规范。  
      RequireJS遵循的是AMD规范。AMD推崇依赖前置。  
      SeaJS遵循的CMD规范。CMD推崇依赖就近(按需加载)  
      AMD:API根据使用范围有区别，但使用同一个api接口  
      CMD:每个API的职责单一
   3. 为什么要用require.js?

最早的时候，所有Javascript代码都写在一个文件里面，只要加载这一个文件就够了。后来，代码越来越多，一个文件不够了，必须分成多个文件，依次加载。下面的网页代码，相信很多人都见过。  
 <script src="1.js"></script>  
　　<script src="2.js"></script>  
　　<script src="3.js"></script>  
　　<script src="4.js"></script>  
　　<script src="5.js"></script>  
　　<script src="6.js"></script>  
这段代码依次加载多个js文件。  
这样的写法有很大的缺点。首先，加载的时候，浏览器会停止网页渲染，加载文件越多，网页失去响应的时间就会越长；其次，由于js文件之间存在依赖关系，因此必须严格保证加载顺序（比如上例的1.js要在2.js的前面），依赖性最大的模块一定要放到最后加载，当依赖关系很复杂的时候，代码的编写和维护都会变得困难。  
require.js的诞生，就是为了解决这两个问题：  


实现js文件的异步加载，避免网页失去响应；

管理模块之间的依赖性，便于代码的编写和维护。

* 1. require.js的加载
     1. 使用require.js的第一步，是先去官方网站[下载](http://requirejs.org/docs/download.html)最新版本。  
        下载后，假定把它放在js子目录下面，就可以加载了。  
        　<script src="js/require.js"></script>  
        有人可能会想到，加载这个文件，也可能造成网页失去响应。解决办法有两个，一个是把它放在网页底部加载，另一个是写成下面这样：  
        <script src="js/require.js" **defer async="true"** ></script>  
        async属性表明这个文件需要异步加载，避免网页失去响应。IE不支持这个属性，只支持defer，所以把defer也写上。  
        加载require.js以后，下一步就要加载我们自己的代码了。假定我们自己的代码文件是main.js，也放在js目录下面。那么，只需要写成下面这样就行了：  
        <script src="js/require.js" **data-main="js/main"**></script>  
        data-main属性的作用是，指定网页程序的主模块。在上例中，就是js目录下面的main.js，这个文件会第一个被require.js加载。由于require.js默认的文件后缀名是js，所以可以把main.js简写成main。
  2. 主模块的写法
     1. 上面的main.js，我把它称为"主模块"，意思是整个网页的入口代码。它有点像C语言的main()函数，所有代码都从这儿开始运行。  
        下面就来看，怎么写main.js。  
        如果我们的代码不依赖任何其他模块，那么可以直接写入javascript代码。  
         // main.js  
         alert("加载成功！");  
        但这样的话，就没必要使用require.js了。真正常见的情况是，主模块依赖于其他模块，这时就要使用AMD规范定义的的require()函数。  
         // main.js  
         require(['moduleA', 'moduleB', 'moduleC'], function (moduleA, moduleB, moduleC){  
         // some code here（其它代码  
         });  
        require()函数接受两个参数。第一个参数是一个数组，表示所依赖的模块，上例就是['moduleA', 'moduleB', 'moduleC']，即主模块依赖这三个模块；第二个参数是一个回调函数，当前面指定的模块都加载成功后，它将被调用。加载的模块会以参数形式传入该函数，从而在回调函数内部就可以使用这些模块。  
        require()异步加载moduleA，moduleB和moduleC，浏览器不会失去响应；它指定的回调函数，只有前面的模块都加载成功后，才会运行，解决了依赖性的问题。
     2. 下面，我们看一个实际的例子。  
        假定主模块依赖jquery、underscore和backbone这三个模块，main.js就可以这样写：  
         require(['jquery', 'underscore', 'backbone'], function ($, \_, Backbone){  
         // some code here  
         　});  
        require.js会先加载jQuery、underscore和backbone，然后再运行回调函数。主模块的代码就写在回调函数中。
  3. 模块的加载
     1. 上面最后的示例中，主模块的依赖模块是['jquery', 'underscore', 'backbone']。默认情况下，require.js假定这三个模块与main.js在同一个目录，文件名分别为jquery.js，underscore.js和backbone.js，然后自动加载。  
        使用require.config()方法，我们可以对模块的加载行为进行自定义。require.config()就写在主模块（main.js）的头部。参数就是一个对象，这个对象的paths属性指定各个模块的加载路径。  
         require.config({  
         　　　paths: {  
         　　　　　 "jquery": "jquery.min",  
         　　　　　 "underscore": "underscore.min",  
        　　　　　　 "backbone": "backbone.min"  
         }  
         });  
        上面的代码给出了三个模块的文件名，路径默认与main.js在同一个目录（js子目录）。如果这些模块在其他目录，比如js/lib目录，则有两种写法。一种是逐一指定路径。  
         require.config({  
         　　　　paths: {  
         　　　　　"jquery": "**lib/**jquery.min",  
        　　　　　　 "underscore": "**lib/**underscore.min",  
        　　　　　　 "backbone": "**lib/**backbone.min"  
         　　　}  
         　　});  
        另一种则是直接改变基目录（baseUrl）。  
         require.config({  
         　　　　**baseUrl: "js/lib",** 　　　　paths: {  
         　　　　　"jquery": "jquery.min",  
        　　　　　　 "underscore": "underscore.min",  
        　　　　　　 "backbone": "backbone.min"  
         　　　}  
         　　});
  4. AMD模块的写法
     1. require.js加载的模块，采用AMD规范。也就是说，模块必须按照AMD的规定来写。  
        具体来说，就是模块必须采用特定的define()函数来定义。如果一个模块不依赖其他模块，那么可以直接定义在define()函数之中。  
        假定现在有一个math.js文件，它定义了一个math模块。那么，math.js就要这样写：  
         // math.js  
         　　define(function (){  
         　　　　var add = function (x,y){  
         　　　　　　return x+y;  
         　　　　};  
         　　　　return {  
         　　　　　　add: add  
        　　　　 };  
         });  
        加载方法如下：  
         // main.js  
        　　require(['math'], function (math){  
        　　　　alert(math.add(1,1));  
        　　});  
        如果这个模块还依赖其他模块，那么define()函数的第一个参数，必须是一个数组，指明该模块的依赖性。  
         define(['myLib'], function(myLib){  
        　　　　function foo(){  
        　　　　　　myLib.doSomething();  
        　　　　}  
        　　　　return {  
        　　　　　　foo : foo  
        　　　　};  
        　　});  
        当require()函数加载上面这个模块的时候，就会先加载myLib.js文件。

1. seaJS介绍

* 1. <http://seajs.org/docs/>
  2. 如何使用seaJs
     1. 引入sea.js的库
     2. 如何变成模块

//module1.js

define(function(require,exports,module){ //sea下的参数：不允许修改

//exports : 对外提供接口的对象

function show(){

alert(1);

}

exports.show = show;

});

* + 1. 如何调用模块  
       //两个参数：第一个参数：模块的地址

//sea的默认根目录：sea.js这个文件

//ex==exports seajs.use("./js/module1.js",function(ex){

ex.show();

});

1. AMD和CMD规范
   1. [AMD](https://github.com/amdjs/amdjs-api/wiki/AMD)是"Asynchronous Module Definition"的缩写，意思就是"异步模块定义"。它采用异步方式加载模块，模块的加载不影响它后面语句的运行。所有依赖这个模块的语句，都定义在一个回调函数中，等到加载完成之后，这个回调函数才会运行。
   2. 在 Sea.js 中，所有 JavaScript 模块都遵循 CMD（[Common Module Definition](https://github.com/cmdjs/specification/blob/master/draft/module.md)） 模块定义规范。该规范明确了模块的基本书写格式和基本交互规则。  
      在 CMD 规范中，一个模块就是一个文件。代码的书写格式如下：  
      define(factory);

第四节：应用

1. 掌握requireJS的使用

第五节：综合应用

1. 掌握模块化开发
2. 掌握git