浙江树人大學

毕业设计过程材料

专业班级 电子商务 141 班

姓　　名 宋益晨

指导教师 杨晓波

所在学院 信息科技学院

2018 年 5 月

总　目　录

浙江树人大学

本科毕业设计

任 务 书

题　　目 “晨博依恋”服饰网站设计与实现

学　　院 信息科技学院

专　　业 电子商务

班　　级 141 班

学　　号 201405017127

学生姓名 宋益晨

指导教师 杨晓波

发放日期 2017 年 10 月 27 日

一、主要任务与目标

主要任务：通过对“晨博依恋”服饰在库存管理，品牌展示，销售拓展等进行需求分析和可行性分析，搭建后台管理系统以及前端展示页面，完成开题报告。

目标：开发出便于管理员管理网站内容以及向客户在官网上浏览购买商品的可视化页面。

二、主要内容与基本要求

主要内容：

1.首先进行需求分析

2.进行可行性分析，结合实际情况，借鉴其他相关网站，使网站功能名副其实。

3.开发系统，本次研究主要使用的 editor 为 vs-code，辅之以 cmder 命令行工具，开发环境是 WampServer。前端页面使用 html 和 javascript，后台系统为 php，数据库以 mysql 实现。以及图片处理的 Photoshop。

基本要求：

熟悉生产环境，明确需要的内容，了解网站制作流程，能够使用编程语言达到输出想法的目的。

三、计划进度

2017.09.01～2017.10.31	确定论文题目，收集资料，准备开题报告
2017.11.07～2017.11.20	完成文献综述
2017.11.21～2017.12.04	完成开题报告阶段
2017.12.05～2017.12.12	完成外文翻译
2017.12.13～2017.12.23	开题答辩
2017.12.24～2018.02.28	调查数据分析阶段
2018.03.01～2018.03.30	完成论文初稿阶段
2018.04.04～2018.04.30	毕业论文定稿
2018.05.01～2018.05.15	毕业论文答辩

四、主要参考文献

[1] 余明阳.品牌传播学[M].上海交通大学出版社，2016 (06)：4-5.

[2] Kyle banker, mongoDB in action second edition[M].2017(3)：10-20

[3] Ioannis K,ChaniotisKyriakos-Ioannis D,KyriakouNikolaos D,Tselikas.Is Node.js a viable option for building modern web applications? A performance evaluation study [J] .USA.Coputing.2015

[4] 王金龙，宋斌，丁锐.Node.js:一种新的 Web 应用构建技术[D].现代电子技术 2015.06

[5]朴灵.深入浅出 node.js[M].2013(12):240-256

[6] 李慧云,何震苇,李丽,陆钢.HTML5 技术与应用模式研究[M].中国电信股份有限公司广东研究院,2012-5

[7] Adam, The Definitive Guide to HTML5[M], 2011(4):4-5

[8] 谢华成,马学文. MongoDB 数据库下文件型数据存储研究[D]. 信阳师范学院网络信息与计算中心.2015-11

[9] Jon Duckett, HTML&CSS design and build website[M],2011(8):20-30

[10] 李慧云,何震苇,李丽,陆钢.基于 CSS 与 JavaScript 技术的 Tab 面板的设计与实现[J].计算机工程.2012-5

[11] 王艳.探析 HTML5 与 CSS3 在网页设计中的新特性和优势[J].电脑编程技巧与维护,2016

[12] Nicholas C,Zakas.Understanding ECMAScript 6:The Definitive Guide for JavaScript Developers[M].USA.Wrox.2012

[13] 刘旭光.基于 AJAX 与 JAVASCRIPT 技术在网页中传递数据的实现[M].淮北煤炭师范学院学报(自然科学版).2010-01

[14]闫岩.互联网+整合与跨界.2016(4):113

[15] 沈昕.基于 Node.js 及 Mongodb 的在线学习测试系统设计[J].北京:中国人民大学出版社,2010

[16] 刘星华.HTML5——下一代 Web 开发标准研究[J].计算机技术与发展, 2011

[17] Jesse James Garrett.用户体验要素[M].2007(10):27-50

[18] Steve McConnell, Code Complete (Second Edition)[M], 2004:13-83

[19] Dustin Luis Pedroia.Professional Node.js: Building Javascript Based Scalable Software[M].USA.2012-10

[20]郎为民.互联网+那些事儿[M].化学工业出版社.2017(2):14-15

指导教师　　年　月　日

教学院长　　年　月　日

浙江树人大学

本 科 生 毕 业 设 计

文 献 综 述

题　　目　晨博依恋服饰购物网站设计与实现

专　　业　电子商务

班　　级　141 班

姓　　名　宋益晨

指导教师　杨晓波

所在学院　信息科技学院

2017 年 11 月

前 言

传统的线下的销售模式服装企业在产品设计、生产、销售、品牌、营运等方面经过多年的发展，整体流程、人才、资源都已经相当成熟，因此品牌复制相当容易，同类品牌的企业同质化产品挤占市场， 竞争由蓝海快速进入红海，促使服装企业的获利能力严重下滑。

随着互联网的不断普及，中国服装企逐渐由“实体战场”向“虚拟战 场”延伸，企业对应用电子商务的意识也得到了质的飞跃。移动社交电商平台近年来异军突起，以口袋微店、微盟萌店及微卖等为代表。移动社交 电商平台较受资本青睐，对互联网理解深刻，产品系统完备且互联网运营/营销经验丰富， 分销渠道保证零库存的同时，有效集聚流量并促进转化购买，引领着行业的发展方向。目前服装类电子商务的主流模式有 B2B，B2C，C2C，O2O 自媒体与新媒体等。

晨博依恋服饰还是处于传统的线下战场，品牌营销和销售利润成为发展的突破口，目前公司独立官方网站的销售模式已经成为一种趋势，可以减少产品在经销商环节产生的费用，生产厂商能更加注重品牌意识和文化塑造。因此本次研究将围绕晨博依恋网站的销售模块进行整体设计开发。

正 文

（一）国内外服装电子商务发展现状

2015 年中国纺织服装电子商务交易总额为 37100 亿元 2016 年增长至 47006 亿元。纺织服装 B2B 电子商务交易规模：2016 年中国纺织服装 B2B 电子商务交易规模达 37114 亿元，较 2015 年的 22415 亿元，同比增长 27.2%。服装网购市场交易规模：2015 年，我国服装网购市场交易规模达 7457 亿元，2016 年达 9343 亿元，同比增长 21.2%。服装网购渗透率：2015 年，我国服装网购渗透率为 34.7%，2016 年服装网购渗透率达 36.9%，同比增长 10.2%。

电子商务的兴起于美国，可以说是世界上最早发展电子商务的国家，也是目前发展的最为成熟的国家之一，1995 年亚马逊开始一直保持着世界领先的地位。美国的服装产业从 2011 年至 2016 年美国服装市场从 2099.8 亿增长到 3588 亿美元。在全球化的进程中，国外服装品牌越来越重视中国的市场，在国内有效地结合线下与线上的资源，构造品牌形象，抢占市场

（二）服装类电子商务模式

纺织服装 B2B 电子商务发展迅猛，在行业电商中占主体地位，家纺服装网络零售线上线下融合发展，传统品牌企业整合线下门店协同发展，网络品牌企业 开辟线下门店速度加快。

1. B2B 服装电子商务

B2B 发展势头强，占据服装电子商务占据主要地位，发展 PC 端向移动端转移。通过便捷化、碎片化、个性化、智能化等特点扩大应用。纺织服装中有多个 B2B 交易结合点，供应链金融在中起到作用。衣联网依托开展直批(服装厂直接向服装店供货)业务的实体商家规模堪称全国之最，其数量已远远超过白马等实体市场。衣联网上的实体批发商主要来自十三行、沙河、白马、虎门等服装批发基地。由于衣联网上下游两端的用户都是商业用户，下游用户追求的更是低成本，与个人用户追求体验有所不同，这样一来，衣联网的用户就不会像淘宝上的个人用户那样在乎卖家态度、发货速度等细枝末节，这本身就解决了衣联网在提高用户体验上的成本问题。，衣联网根据开发服装行业独有的网络防抄版系统和区域保护系统，解决服装企业 对于服装款式遭遇抄袭模仿的后顾之忧。

2. B2C 服装电子商务

2001 年，韩都衣舍只是一个在淘宝上做韩国服装的代购网店。从名字便知，韩都衣舍更像是一个渠道名，于 2008 年开始自建品牌。从 2008 年到 2014 年间，韩都衣舍的销售额成长了 500 多倍，已经从年入 300 万元的小企业转身成为年入 16 亿元的电商黑马。传统品牌花大量的精力做渠道建设和扩张，而韩都衣舍把精力都集中在产品上，包括产品本身的设计、网站的视觉传达以及服务。围绕着小组制，韩都衣舍的整个管理架构分为三层，一是与品牌相关的企划、视觉、市场部门；二是 IT、供应链、物流、客服等互联网支持部门；三是人力、行政、财务等行政支持部门。整个公司的核心是产品小组，而市场、企划、设计、客服、行政、财务等部门全是小组的支持部门。在每个产品小组里，责、权、利完全统一，也高度自主。每个小组对于产品的款式、定价、生产量全由自己决定，但同时小组的 KPI 与销售额、毛利率、库存周转率相关。

3. 自媒体

“网红”们对粉丝强大的号召力，也让不少中国服装企业希望借助与“网红”之间的合作，迎来品牌销售的“第二春”。通过借助“网红”推广品牌单品，的确对一些服装企业的销售额起到一定的带动作用。服装品牌如何能实现与“网红”之间稳定有序的合作，对品牌聚焦客户、实现精准推广、赢得持续关注度起到巨大帮助。 茵曼：现在是社群和粉丝经济的时代，随着 90、95 后抢占市场购买力开始，消费更趋于小众化，社群化，个性化。想要抓住 90、95 后的消费者，以网红为首的社群电商绝对是时下最热趋势。在微博电商，网红就代表了一个个十万级，百万级，千万级的精准粉丝流量，流量转化最高的网红。有数据分析，每 20 个阅读，能够转化为一个点击。由此可见，网红其实代表了微博电商背后最核心的基础群，没有这些精准流量，那么微博的优势也无法施展。近两年，红人、达人、自媒体、专家等一批网红的快速爆发，已经可以预示着，社交化电商就是互联网行业的下一个“风口”。

4. O2O

优衣库以门店自助式购物体验和贴心服务著称，传统的、全部自营的线下门店一直是优衣库的核心渠道。在优衣库看来，O2O 的主要作用是为线下门店服务，帮助线下门店提高销量，并做到推广效果可查、每笔交易可追踪。门店是整个 O2O 布局的核心，布局 O2O 的目的主要是为线下门店导流、提高线下门店销量。这种模式主要适用于品牌号召力较强，同时以门店体验和服务拉动销售为主的服装品牌，这类品牌非常看重线下门店，自己的官网、网上商城和自有 app 等线上渠道也一般都会与线下门店互相打通，通过门店查找、优惠券发放、品牌宣传等方式将消费者吸引到线下门店去消费。自有 app 也会有下单购物功能，方便将用户数据沉淀下来做精准营销。

（三）服装电商网站存在的问题及解决方案

分析对比国内外的服装产品的电商平台，目前存在以下几个问题：

1. 注重 B 店销售，影响品牌塑造

如恒源祥,其官网的大致内容为企业在社会中的活动,2b 的销售通道只是很粗略的堆叠商品,2c 的销售通道则是几个天猫店铺的链接。这体现了恒源祥对第三方平台的重视程度——完全将线上的销售任务交给了 B 店。在多数服装企业进驻电商平台后，由于缺乏有效的管理，店铺之间都忙于竞争流量，转化率，店铺经常需要进行各种平台的促销活动，自身店铺的促销活动，促销有利有弊，在促销的过程中达到让消费者注意到产品目的，很容易就造成了难以塑造独特的品牌文化的问题。消费者经过一轮轮地促销后会习惯性地去寻找物美价廉的商品，弱化对品牌的敏感度，而品牌是附加值中最重要的部分。

2. 视觉展示趋同，辨识度低

很多知名品牌产品展示趋同，国内服装市场份额最大的品牌海澜之家的品类丰富，虽然页面整洁规划，但商品的布局方式却与淘宝等平台无异，都是配图加简短的文字说明，并且由于分类不明确，出现许多近似商品无法分辨的情况。在这种情况下，消费者难以感受到产品的特点，导致转换率低下。

解决方案：

如今电子商务模式已逐渐成熟，建立自己的直营网站既能更好的把控消费者认知，又能减少成本。如果要引起消费者的兴趣，加深消费者的印象，就要赋予产品内涵，建立品牌形象。如阿玛尼在产品的呈现上通过对某一主题的产品单独展示，同时适时地加入了产品的故事文化、所要表达的理念，来提升产品的魅力。整体主题直接影响了消费者的品牌认同感，应结合产品受众的审美与自身品牌的形象。如果能引起消费者的共鸣使其成为潜在消费者便已经成功了一半。UI 设

计既要有丰富的互动，也要兼顾人性化体验，如必要的文字说明可以采用固定页面位置的方式显示。

（四）服装电商网站发展趋势

受益于电商业务快速发展的服装企业在电商方面的动作，销售线下同季同款期货、推线上独有产品、打造电商销售平台、转为自营、新建电商物流基地，这些动作均显示传统服装品牌正在把线上渠道变“重”，加强对线上渠道的控制权，使其成为和实体渠道并驱齐驾的新的“传统”渠道。

原来企业很难与消费者沟通，个性需求不能被满足。现在互联网让沟通更容易实现，生产需求也可以高效化个性化。随着互联网第一批用户的年龄增长，网上零售也开启了高品质高客单的机会窗口。服装消费而言，高品质高度个性化是新中产的需求。而传统服装定制一直存在价格昂贵、定制周期长、消费者认知度低、消费频次低等消费痛点。基于互联网思维，从客户需求出发，采用按需定制模式的互联网服装定制应运而生。对于一心想要去产能、去库存的服装公司来说，电商成为了一条必经之路。包括森马服饰、步森股份、金发拉比在内的超过 80%的上市服装企业都同时进行线上线下布局。服装产业要想长期发展，必须转变新思路，与其被动适应新变化不如主动去迎合新变化， 当务之急是要短期内寻求到发展的新路径，以较短时间和较小的代价去实现突围。除此之外， 不光要解决短期的问题，还要解决长期制约发展的核心问题。对服装业而言，就是提升产品 价值和质量，用专业态度做产品，有追求极致的工匠精神。

总 结

服装品类一贯是网购市场的主力军。电商对服装行业及其零售终端渠道的冲击，不只是优化了产业链的中间环节，电商更改变了人们对服装行业的消费习惯，触动了传统服装行业进行转型升级开关。官网的建立有利于避开同质化竞争，在利用好客户数据的情况下能更好的品类拓展，增加销售渠道，减轻库存压力，建立品牌与消费者的良性互动。

设计和开发晨博依恋购物网站主要使用的语言为 JavaScript。开发完成后网站分为后台管理页面和商城页面。主要功能有会员注册、登录、购物车、产品详情、地址管理、前台商品显示、购买等功能模块。后台管理页面功能：用户评论管理，材料去向记录，商品增删改查，管理员分类。

参考文献:

[1] 余明阳.品牌传播学[M].上海交通大学出版社，2016（06）：4-5.
[2] Kyle banker.mongoDB in action second edition[M].2017(3)：10-20
[3] Ioannis K,ChaniotisKyriakos-Ioannis D,KyriakouNikolaos D,Tselikas.Is Node.js a viable

option for building modern web applications? A performance evaluation study [J] .USA.Coputing.2015

[4] 王金龙,宋斌,丁锐.Node.js:一种新的 Web 应用构建技术[D].现代电子技术 2015.06

[5] 朴灵.深入浅出 node.js[M].2013(12):240-256

[6] 李慧云,何震苇,李丽,陆钢.HTML5 技术与应用模式研究[M].中国电信股份有限公司广东研究院,2012-5

[7] Adam，The Definitive Guide to HTML5[M]，2011(4)：4-5

[8] 谢华成,马学文. MongoDB 数据库下文件型数据存储研究[D]. 信阳师范学院网络信息与计算中心.2015-11

[9] Jon Duckett，HTML&CSS design and build website[M],2011(8):20-30

[10] 李慧云,何震苇,李丽,陆钢.基于 CSS 与 JavaScript 技术的 Tab 面板的设计与实现[J] .计算机工程.2012-5

[11] 王艳.探析 HTML5 与 CSS3 在网页设计中的新特性和优势[J].电脑编程技巧与维护，2016

[12] Nicholas C,Zakas.Understanding ECMAScript 6:The Definitive Guide for JavaScript Developers[M].USA.Wrox.2012

[13] 刘旭光.基于 AJAX 与 JAVASCRIPT 技术在网页中传递数据的实现[M].淮北煤炭师范学院学报(自然科学版).2010-01

[14] 闫岩.互联网+整合与跨界.2016(4):113

[15] 沈昕.基于 Node.js 及 Mongodb 的在线学习测试系统设计[J].北京:中国人民大学出版社,2010

[16] 刘星华.HTML5——下一代 Web 开发标准研究[J].计算机技术与发展, 2011

[17] Jesse James Garrett.用户体验要素[M].2007(10):27-50

[18] Steve McConnell，Code Complete (Second Edition)[M].2004：13-83

[19] Dustin Luis Pedroia.Professional Node.js: Building Javascript Based Scalable Software[M].USA.2012-10

[20]郎为民.互联网+那些事儿[M].化学工业出版社.2017(2)：14-15

指导教师审核意见：

指导老师（签字）_____

2017 年 11 月 24 日

浙江树人大学

本 科 生 毕 业 设 计

开 题 报 告

题　　目 晨博依恋服饰购物网站设计与实现

专　　业 电子商务

班　　级 141 班

姓　　名 宋益晨

指导教师 杨晓波

所在学院 信息科技学院

开题时间 2017 年 12 月

一、 选题的背景与意义

2015 年中国纺织服装电子商务交易总额为 37100 亿元 2016 年增长至 47006 亿元。纺织服装

B2B 电子商务交易规模：2016 年中国纺织服装 B2B 电子商务交易规模达 37114 亿元，较 2015 年的 22415 亿元，同比增长 27.2%。服装网购市场交易规模：2015 年，我国服装网购市场交易规模达 7457 亿元，2016 年达 9343 亿元，同比增长 21.2%。服装网购渗透率：2015 年，我国服装网购渗透率为 34.7%，2016 年服装网购渗透率达 36.9%，同比增长 10.2%。

受益于电商业务快速发展的服装企业在电商方面的动作，销售线下同季同款期货、推线上独有产品、打造电商销售平台、转为自营、新建电商物流基地，这些动作均显示传统服装品牌正在把线上渠道变“重”，加强对线上渠道的控制权，使其成为和实体渠道并驱齐驾的新的“传统”渠道。

开发晨博依恋服饰官方网站的建立有利于建立品牌与消费者的良性互动，避开同质化竞争，在利用好客户数据的情况下能更好的品类拓展，提高转化率，减轻库存压力。

二、 研究的基本内容与拟解决的主要问题

（一） 研究的基本内容

1. 需求分析

晨博依恋官方网站定位是时尚中青年女士毛衫的购物网站，主要面向对象是有意向购买品牌产品的 C 端消费者，和有意向了解整体水平的批发客户，因此需要的前台展示板块有“企业信息”、“个人信息”、“热门商品”、“全部商品”、“购物车”、“产品详情”、“地址选择”和“订单确认”等。期望通过晨博依恋官方网站的整体设计、产品展示、UI 设计以达到塑造品牌文化形象，产生与消费者的良性互动，避开同质化竞争，在利用好客户数据的情况下能更好的品类拓展，提高转化率，减轻库存压力的目的。

2. 页面设计

结合产品受众的审美与自身品牌的形象设计相应主题的页面，要求页面有合理的模块分级，既能方便维护，又能使页面结构合理化。根据需求分析得出晨博依恋服饰官方网站 UI 应展示的版块有“企业信息”、“个人信息”、“热门商品”、“全部商品”、“购物车”、“产品详情”、“地址选择”和“订单确认”等。

3. 功能设计

后台功能设计，商品管理包含商品上架，商品下架，商品调价，调库存，商品简介修改。订单管理包含查看订单的详细信息及订单受理，订单配货，订单发货。站点管理可设置修改企业的各类信息及介绍。会员管理包括消息群发，删除会员，会员查询。

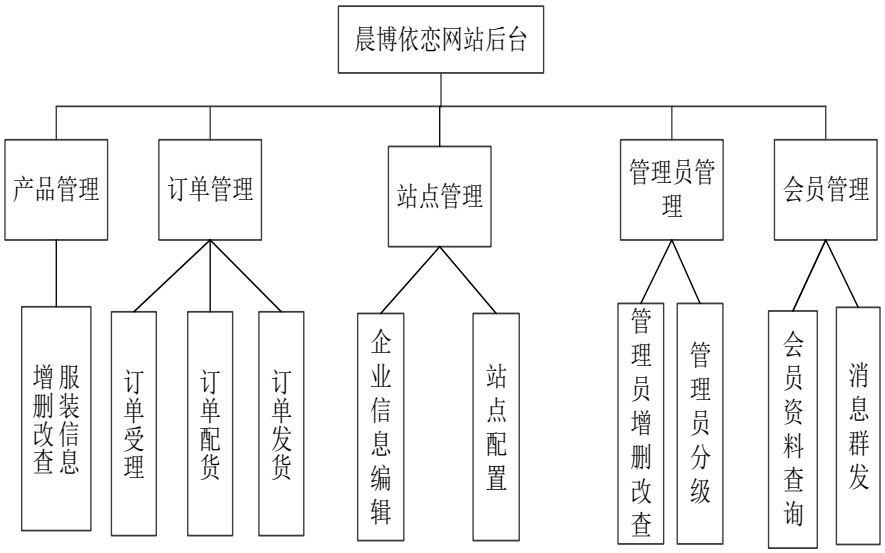

图 1 后台版块

前台功能设计为会员注册登录包含，个人信息消息查询，购物流程。

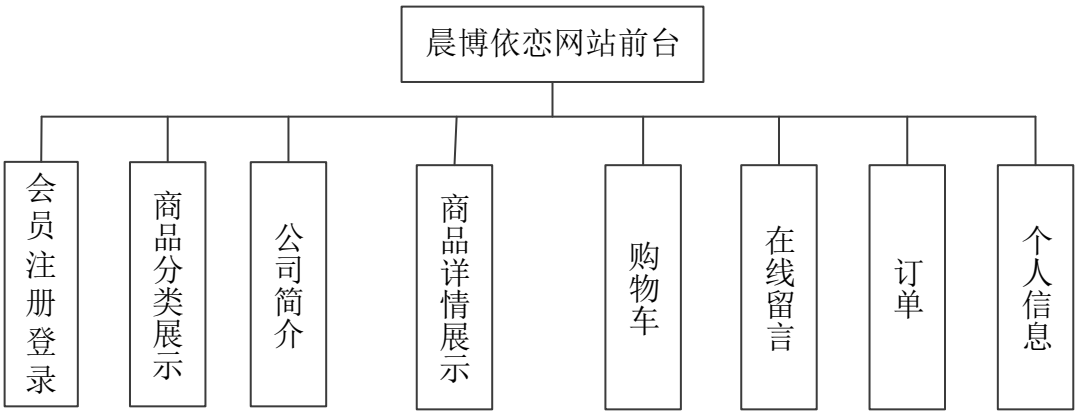

图 2 前台展示版块

数据库设计，晨博依恋官方网站是垂直小型电子商务平台可以使用构建数据库，MySQL 的 SQL“结构化查询语言”。由于其体积小、速度快、总体拥有成本低，尤其是开放源码这一特点，许多中小型网站为了降低网站总体拥有成本而选择了 MySQL 作为网站数据库。

4. 网站开发

使用到的语言有 html，css，JavaScript，php。以及对图片处理的 ps 软件。商城页面内容适合数据驱动。使用 Photoshop 工具美化,修改图片,使图片更适合网站风格。后端开发使用 php,运行在 wamp 环境。

5. 测试

配合官网管理员，用真实材料进行各个功能的测试，在阿里云服务器上测试性能，用不同浏览器打开测试展示情况，根据测试反馈进行修改。

（二） 拟解决的主要问题

1. 前端设计

在对晨博依恋官网的网页外观设计过程中，应注意从美观、简约等方面着手。在网页整体设计中注意利用合理的布局。另外网站主题色彩的搭配要与晨博依恋服饰所要传达的形象相符，达到让浏览用户过目不忘的效果。运用 CSS 语言与 JavaScript 语言，达到流畅的人机交互。

2.. 功能开发

在设计电子商务网站的功能时，首先要确定网站的目标、规模和具体内容，也可参考其他同类网站的功能设计情况，再决定网站应该具有哪些功能,才能保证达到网站实际需求。商城页面功能包括等。会员注册、登录、购物车、产品详情、地址管理、前台商品显示、购买等功能模块。后台管理页面功能：用户评论管理，商品编辑，管理员分类，会员管理，站点设置。实现具体模块所涉及到的主要算法、数据结构、类的层次结构及调用关系，需要说明软件系统各个层次中的每一个程序(每个模块或子程序)的设计考虑，以便进行编码和测试。应当保证软件的需求完全分配给整个软件。详细设计应当足够详细，能够根据详细设计报告进行编码

3. 连接数据库

在设计数据库时需要把将要用到的主体与属性提前设计，对属性的字段名严谨的分析设计,字段的属性也需要符合情理，不过多浪费数据库空间或过少导致信息缺失。通过使用数据库可视化软件 Navicat for MySQl 管理，在配置文件 config/config.php 中配置以链接数据库。

4. 网站内容展示优化

传输的内容体积要小图片要压缩图片根据支持情况选择体积更小的格式(如 webp)css、js 内容压缩服务端开启 Gzip，在传输数据之前再次压缩。传输的内容数量要少图片图标合并 (css sprite)、svg 图标合并 (svg sprite)。css、js 文件打包合并网速要足够快服务器出口带宽要够考虑到南北差异、运营商差异，在不同地区部署服务器静态资源放 CDN 服务器响应要及时接口响应速度要快(数据库优化、查询优化、算法优化)cpu、内存、硬盘读写不要成为瓶颈；多加几台机器重要页面(首页)静态化。服务端提前渲染后首页生成静态页面，用户访问首页直接返回静态

页面，不需要像其他页面那样还需加载 css、js 再获取数据渲染展示能重复利用的资源要利用好服务器设置合适的静态资源缓存时间前端文件打包时做合理的分块，让公共的资源缓存后能被多个页面复用

三、 研究的方法与技术路线

（1）研究的方法

首先使用调查法对实际情况分析，包括市场环境，生产环境与销售环境，评估需求的实现难度，自身水平限制。确定实际设计方案。再对系统分析后确定系统的模块，对每个模块进行研究。整体项目开发设计的技术涵盖对 HTML、CSS、JavaScript、php、sql 的语法和 jquery、Echarts 类库的了解。所以需要掌握大量的知识，在项目准备阶段，使用文献调研法学习了解事物。开发阶段，利用开源的 github 平台中的介绍，工具的官方文档学习。

（2）研究的技术路线

在晨博依恋购物网站管理系统的开发过程中主要使用的 editor 为 vs-code，辅之以 cmder 命令行工具，开发环境是 WampServer。前端页面使用 html 和 javascript，后台系统为 php，数据库以 mysql 实现。以及图片处理的 Photoshop。

1. js 图表

数据可视化是数据得以应用的最关键的步骤。解决方案为依靠一个纯 Javascript 的图表库，它可以流畅的运行在 PC 和移动设备上，兼容当前绝大部分浏览器（IE8/9/10/11，Chrome，Firefox，Safari 等），底层依赖轻量级的 Canvas 类库 ZRender，提供直观，生动，可交互，可高度个性化定制的数据可视化图表。ECharts 由数据驱动，数据的改变驱动图表展现的改变。因此动态数据的实现也变得异常简单，只需要获取数据，填入数据，ECharts 会找到两组数据之间的差异然后通过合适的动画去表现数据的变化。配合 timeline 组件能够在更高的时间维度上去表现数据的信息。ECharts 提供了常规的折线图，柱状图，散点图，饼图，K 线图，用于统计的盒形图，用于地理数据可视化的地图，热力图，线图，用于关系数据可视化的关系图，treemap，多维数据可视化的平行坐标，还有用于 BI 的漏斗图，仪表盘，并且支持图与图之间的混搭。

2. Mysql 数据库

MySQL 是一种关联数据库管理系统，关联数据库将数据保存在不同的表中，而不是将所有数据放在一个大仓库内。这样就增加了速度并提高了灵活性。MySQL 的 SQL“结构化查询语言”。由于其体积小、速度快、总体拥有成本低，尤其是开放源码这一特点，许多中小型网站为了降低

网站总体拥有成本而选择了 MySQL 作为网站数据库。

3. php

PHP，超文本预处理器（Hypertext Preprocessor ），是一种通用的开源脚本语言。编程范型为面向对象和命令式编程语言，同时 PHP 可以在 windows/Mac/linux 跨平台中操作，它几乎支持所有流行的数据库。 PHP 语言容纳了 Java、C 语言和 Perl 的特点，是众多开发语言中较为简单的一种开发语言，在 Web 开发领域中使用广泛，利于学习。

PHP 拥有很多其他语言的语法特点，当然也有自己自创新的语法。PHP 将程序嵌入到 HTML 文档中，可以将 HTML 写的静态网页与数据库联系在一起，形成动态网页。

4. HTML 和 CSS

HTML5 规范定义了我们用来标记内容的元素，并指明了它的内容意义。CSS 允许我们控制标记内容呈现给用户的外观。JavaScript 允许我们操纵 HTML 文档的内容，响应用户交互，并且利用新的 HTML5 元素的一些以编程为中心的特性。HTML 语义化是编写 HTML 的方式，HTML 语义化既方便开发者阅读，又能让机器去理解。语义化要让数据和表述的本体的映射成为可能,那么结构首先要可以表达出来，并且通过一些结构的约定俗成（或者直接声明）让计算机可以找到这些结构的本体，然后计算机就可以通过本体的关系来进行逻辑演绎。合理使用 html5 的语义化才能有利于机器 seo 优化，同时让爬虫和机器可以很好的解析页面。

5. Photoshop

Photoshop 技术吸取多种技术的强大功能不断提高自身实力，成为现如今被广大民众使用最为广泛的技术，该软件主要用来处理图形和图像，在处理图像时更加完善，拥有强大的功能。在设计网站时，用 Photoshop 对网站的设计素材进行加工和处理，使素材图片更完美更容易的用于网站建设中。

四、 研究的总体安排与进度

2017.09.01 ~ 2017.10.31	确定论文题目，收集资料，准备开题报告
2017.11.07 ~ 2017.11.20	完成文献综述
2017.11.21 ~ 2017.12.04	完成开题报告阶段
2017.12.05 ~ 2017.12.12	完成外文翻译
2017.12.13 ~ 2017.12.23	开题答辩
2017.12.24 ~ 2018.02.28	调查数据分析阶段
2018.03.01 ~ 2018.03.30	完成论文初稿阶段
2018.04.04 ~ 2018.04.30	毕业论文定稿
2018.05.01 ~ 2018.05.15	毕业论文答辩

五、 参考文献

[1] 余明阳.品牌传播学[M].上海交通大学出版社，2016（06）：4-5.

[2] Kyle banker.mongoDB in action second edition[M].2017(3)：10-20

[3] Ioannis K,ChaniotisKyriakos-Ioannis D,KyriakouNikolaos D,Tselikas.Is Node.js a viable option for building modern web applications? A performance evaluation study [J] .USA.Coputing.2015

[4] 王金龙,宋斌,丁锐.Node.js:一种新的 Web 应用构建技术[D].现代电子技术 2015.06

[5]朴灵.深入浅出 node.js[M].2013(12):240-256

[6] 李慧云,何震苇,李丽,陆钢.HTML5 技术与应用模式研究[M].中国电信股份有限公司广东研究院,2012-5

[7] Adam，The Definitive Guide to HTML5[M]，2011(4)：4-5

[8] 谢华成,马学文. MongoDB 数据库下文件型数据存储研究[D]. 信阳师范学院网络信息与计算中心.2015-11

[9] Jon Duckett，HTML&CSS design and build website[M],2011(8):20-30

[10] 李慧云,何震苇,李丽,陆钢.基于 CSS 与 JavaScript 技术的 Tab 面板的设计与实现[J] .计算机工程.2012-5

[11] 王艳.探析 HTML5 与 CSS3 在网页设计中的新特性和优势[J].电脑编程技巧与维护，2016

[12] Nicholas C,Zakas.Understanding ECMAScript 6:The Definitive Guide for JavaScript Developers[M].USA.Wrox.2012

[13] 刘旭光.基于 AJAX 与 JAVASCRIPT 技术在网页中传递数据的实现[M].淮北煤

炭师范学院学报(自然科学版).2010-01

[14]闫岩.互联网+整合与跨界.2016(4):113

[15] 沈昕.基于 Node.js 及 Mongodb 的在线学习测试系统设计[J].北京:中国人民大学出版社,2010

[16] 刘星华.HTML5——下一代 Web 开发标准研究[J].计算机技术与发展, 2011

[17] Jesse James Garrett.用户体验要素[M].2007(10):27-50

[18] Steve McConnell.Code Complete (Second Edition)[M].2004：13-83

[19] Dustin Luis Pedroia.Professional Node.js: Building Javascript Based Scalable Software[M].USA.2012-10

[20]郎为民.互联网+那些事儿[M].化学工业出版社.2017(2)：14-15

指导教师审核意见：

指导老师（签字）_____

2017 年 12 月 15 日

浙江树人大学

本 科 生 毕 业 设 计

外 文 资 料 翻 译

题　　目 晨博依恋服饰购物网站规划与设计

专　　业 电子商务

班　　级 141 班

姓　　名 宋益晨

指导教师 杨晓波

所在学院 信息科技学院

附　　件 1.外文资料翻译译文;2.外文原文

外文翻译一：

了解 ECMAScript 6

var 声明与变量提升 (Var Declarations and Hoisting)

使用 var 关键字声明的变量，不论在何处都会被视作在函数级作用域内顶部的位置发生（如果不包含在函数内则为全局作用域内）。为了说明变量提升到底是什么，查看如下函数定义：

```
function getValue(condition) {
    if (condition) {
        var value = "blue";
        // 其它代码
        return value;
    } else {
        // value 可以被访问到，其值为 undefined
        return null;
    }
    // 这里也可以访问到 value，值仍为 undefined
}
```

如果你不太熟悉 JavaScript，或许会认为只有 condition 为 true 时变量 value 才会被创建。实际上，value 总是会被创建。JavaScript 引擎在幕后对 getValue 函数做了调整，可以视为：

```
function getValue(condition) {
    var value;
    if (condition) {
        value = "blue";
        // 其它代码
        return value;
    } else {
        return null;
```

```
    }
}
```

变量的声明被提升至顶部，但是初始化的位置并没有改变，这意味着在 else 从句内部也能访问 value 变量，但如果真的这么做的话，value 的值会是 undefined，因为它并没有被初始化或赋值。

刚开始接触 JavaScript 的开发者总是要花一段时间来习惯变量提升，对该独特概念的陌生也会造成 bug。因此 ECMAScript 6 引入了块级作用域的概念使得变量的生命周期变得更加可控。

块级声明（Block-Level Declarations）

块级声明指的是该声明的变量无法被代码块外部访问。块作用域，又被称为词法作用域（lexical scopes），可以在如下的条件下创建：

函数内部

在代码块（即 { 和 }）内部

块级作用域是很多类 C 语言的工作机制，ECMAScript 6 引入块级声明的目的是增强 JavaScript 的灵活性，同时又能与其它编程语言保持一致。

let 声明

let 声明的语法和 var 完全一致。你可以简单的将所有 var 关键字替换成 let，但是变量的作用域会限制在当前的代码块中（稍后讨论其它细微的差别）。既然 let 声明不会将变量提升至当前作用域的顶部，你或许要把它们手动放到代码块的开头，因为只有这样它们才能被代码块的其它部分访问。举个例子：

```
function getValue(condition) {
    if (condition) {
        let value = "blue";
        // 其它代码
        return value;
    } else {
        // value 并不存在（无法访问）
        return null;
    }
    // 这里 value 也不存在
```

```
}
```

本次 getValue 函数的写法的默认行为更贴近你脑海中 C 和其它类 C 语言的印像。既然变量的声明由 var 替换成了 let，那么它们就不会自动提升到当前函数作用域的顶部，除非执行流程到了 if 从句内部，其它情况时是没有办法对该变量进行访问的。如果 if 从句中 condition 的值为 false，那么 value 变量就不会被声明或者初始化。

禁止重复声明

如果一个标识符在当前作用域里已经存在，那么再用 let 声明相同的标识符或抛出错误

```
var count = 30;
// 语法错误
let count = 40;
```

在本例中，count 被声明了两次： 一次是被 var 另一次被 let 。因为 let 不会重新定义已经存在的标识符，所以会抛出一个错误。反过来讲，如果在当前包含的作用域内 let 声明了一个全新的变量，那么就不会有错误抛出。正如以下的代码演示：

```
var count = 30;
// 不会抛出错误
if (condition) {
    let count = 40;
    // 其它代码
}
```

本次 let 声明没有抛出错误的原因是 let 声明的变量 count 是在 if 从句的代码块中，并非和 var 声明的 count 处于同一级。在 if 代码块中，这个新声明的 count 会屏蔽掉全局变量 count ，避免在执行流程未跳出 if 之前访问到它。

const 声明（Constant Declarations）

在 ECMAScript 6 中也可使用常量（const）语法来声明变量。该种方式声明的变量会被视为常量，这意味着它们不能再次被赋值。由于这个原因，所有的 const 声明的变量都必须在声明处初始化。示例如下：

```
// 合法的声明
const maxItems = 30;
// 语法错误：未初始化
```

```
const name;
```

变量 maxItems 已经被初始化，所以这里不会出现任何问题。至于 name 变量，由于你未对其进行初始化赋值所以在运行时会报错。

const 声明 vs let 声明（Constants vs Let Declarations）

const 和 let 都是块级声明，意味着执行流跳出声明所在的代码块后就没有办法在访问它们，同样 const 变量也不会被提升，示例如下：

```
if (condition) {
    const maxItems = 5;
    // 其它代码
}
// maxItems 在这里无法访问
```

在这段代码中，maxItems 在 if 从句的代码块中作为常量被声明。一旦执行流跳出 if 代码块，外部就无法访问这个变量。

另一处和 let 相同的地方是，const 也不能对已存在的标识符重复定义，不论该标识符由 var（全局或函数级作用域）还是 let （块级作用域）定义。例如以下的代码：

```
var message = "Hello!";
let age = 25;
// 下面每条语句都会抛出错误
const message = "Goodbye!";
const age = 30;
```

以上两条 const 声明如果单独存在即是合法的，很遗憾的是在本例中前面出现了 var 和 let 声明的相同标识符（变量）

尽管 const 和 let 使用起来很相似，但是必须要记住它们的根本性的差异：不管是在严格（strict）模式还是非严格模式（non-strict）模式下，const 变量都不允许被重复赋值。

```
const maxItems = 5;
maxItems = 6;          // 抛出错误
```

和其它编程语言类似，maxItems 不能被赋予新的值，然而和其它语言不同的是，const 变量的值如果是个对象，那么这个对象本身可以被修改。

将对象赋值给 const 变量（Declaring Objects with Const）

const 声明只是阻止变量和值的再次绑定而不是值本身的修改。这意味着 const 不能

限制对于值的类型为对象的变量的修改，示例如下：

```
const person = {
    name: "Nicholas"
};
// 正常
person.name = "Greg";
// 抛出错误
person = {
    name: "Greg"
};
```

在这里，person 变量一开始已经和包含一个属性的对象绑定。修改 person.name 是被允许的因为 person 的值（地址）未发生改变，但是尝试给 person 赋一个新值（代表重新绑定变量和值）的时候会报错。这个微妙之处会导致很多误解。只需记住：const 阻止的是绑定的修改，而不是绑定值的修改。

暂存性死区（The Temporal Dead Zone）

let 或 const 声明的变量在声明之前不能被访问。如果执意这么做会出现错误，甚至是 typeof 这种安全调用（safe operations）也不被允许的：

```
if (condition) {
    console.log(typeof value);  // ReferenceError!
    let value = "blue";
}
```

在这里，变量 value 由 let 声明并被初始化，但是由于该语句之前抛出了错误导致其从未被执行。这种现象的原因是该语句存在于被 JavaScript 社区泛称为暂存性死区（Temproal Dead Zone, TDZ）之内。ECMAScript 规范中未曾对 TDZ 有过显式命名，不过在描述 let 和 const 声明的变量为何在声明前不可访问时，该术语经常被使用。本章会涵盖在 TDZ 中关于声明位置的一些微妙部分。此外示例虽然全部用的是 let ，但是实际用法和 const 别无二致。

当 JavaScript 引擎在作用域中寻找变量声明时，会将变量提升到函数/全局作用域的顶部（var）或是放入 TDZ（let 和 const）内部。任何试图访问 TDZ 内部变量的行为都会以抛出运行时 (runtime) 错误而告终。当执行流达到变量声明的位置时，变量才会被移出 TDZ ，

代表它们可以被安全使用。

由 let 或 const 声明的变量，如果你想在定义它们之前就使用，那么以上所述的准则都是铁打不变的。正如之前的示例所演示的那样，typeof 操作符都不能幸免。不过，你可以在代码块之外的地方对变量使用 typeof 操作符，但结果可能并不是你想要的。考虑如下的代码：

```
console.log(typeof value);        // "undefined"
if (condition) {
    let value = "blue";
}
```

当对 value 变量使用 typeof 操作符时它并没有处在 TDZ 内部，因为它的位置在变量声明位置所在的代码块之外。这意味着没有发生任何绑定，所以 typeof 仅返回 “undefined”

TDZ 只是发生在块级绑定中独特的特设定之一，另一个特殊设定发生在循环中。

循环中的块级绑定（Block Binding in Loops）

或许开发者对块级作用域有强烈需求的场景之一就是循环，因为它们不想让循环外部访问到内部的索引计数器。举个例子，以下的代码在 JavaScript 编程中并不罕见：

```
for (var i = 0; i < 10; i++) {
    process(items[i]);
}
// 这里仍然可以访问到 i
console.log(i);                          // 10
```

块级作用域在其它语言内部是默认的，以上的代码的执行过程也并无差异，但区别在于变量 i 只能在循环代码块内部使用。然而在 JavaScript 中，变量的提升导致块外的部分在循环结束后依然可以访问 i 。若将 var 替换为 let 则更符合预期：

```
for (let i = 0; i < 10; i++) {
    process(items[i]);
}
// 在这里访问 i 会抛出错误
console.log(i);
```

在本例中变量 i 只存在于 for 循环代码块中，一旦循环完毕变量 i 将不复存在。

循环中的函数（Functions in Loops）

长久以来 var 声明的特性使得在循环中创建函数问题多多，因为循环中声明的变量在块外也可以被访问，考虑如下的代码：

```
var funcs = [];
for (var i = 0; i < 10; i++) {
    funcs.push(function() { console.log(i); });
}
funcs.forEach(function(func) {
    func();        // 输出 "10" 共 10 次
});
```

你可能认为这段代码只是普通的输出 0 - 9 这十个数字，但事实上它会连续十次输出“10”。这是因为每次迭代的过程中 i 是被共享的，意味着循环中创建的函数都保持着对相同变量的引用。当循环结束后 i 的值为 10，于是当 console.log(i)被调用后，该值会被输出。

为了修正这个问题，开发者们在循环内部使用即时调用函数表达式（immediately-invoked function expressions, IIFEs）来迫使每次迭代时创建一份当前索引值的拷贝，示例如下：

```
var funcs = [];
for (var i = 0; i < 10; i++) {
    funcs.push((function(value) {
        return function() {
            console.log(value);
        }
    }(i)));
}
funcs.forEach(function(func) {
    func();        // 输出 0，1，2 ... 9
});
```

这种写法在循环内部使用了 IIFE，并将变量 i 的值传入 IIFE 以便拷贝索引值并存储起来，这里传入的索引值为同样被当前的迭代所使用，所以循环完毕后每次调用的输出值正如所期待的那样是 0 - 9 。幸运的是，ECMAScript 6 中 let 和 const 的块级绑定对循环代码进行简化。

循环中的 let 声明（Let Declarations in Loops）

let 声明通过有效地模仿 上例中 IIFE 的使用方式来简化循环代码。在每次迭代中，一个新的同名变量会被创建并初始化。这意味着你可以抛弃 IIFE 的同时也能获得相同的结果。

```
var funcs = [];
for (let i = 0; i < 10; i++) {
    funcs.push(function() {
        console.log(i);
    });
}
funcs.forEach(function(func) {
    func();        // 输出 0，1，2 … 9
})
```

这段循环代码的执行效果完全等同于使用 var 声明和 IIFE，但显然更加简洁。let 声明使得每次迭代都会创建一个变量 i，所以循环内部创建的函数会获得各自的变量 i 的拷贝。每份拷贝都会在每次迭代的开始被创建并被赋值。这同样适用于 for-in 和 for-of 循环，如下所示：

```
var funcs = [],
    object = {
        a: true,
        b: true,
        c: true
    };
for (let key in object) {
    funcs.push(function() {
        console.log(key);
    });
}
funcs.forEach(function(func) {
    func();        // 输出 "a"，"b" 和 "c"
```

});

在本例中，for-in 循环的表现和 for 循环相同。每次循环的开始都会创建一个新的变量 key 的绑定，所以每个函数都会有各自 key 变量值的备份。结果就是每个函数都会输出不同的值。如果循环中 key 由 var 声明，那么所有的函数会输出 "c" 。

外文原文一：

Understanding ECMAScript 6

Block Bindings

Traditionally, the way variable declarations work has been one tricky part of programming in JavaScript. In most C-based languages, variables (or bindings) are created at the spot where the declaration occurs. In JavaScript, however, this is not the case. Where your variables are actually created depends on how you declare them, and ECMAScript 6 offers options to make controlling scope easier. This chapter demonstrates why classic var declarations can be confusing, introduces block-level bindings in ECMAScript 6, and then offers some best practices for using them.

Var Declarations and Hoisting

Variable declarations using var are treated as if they are at the top of the function (or global scope, if declared outside of a function) regardless of where the actual declaration occurs; this is called hoisting. For a demonstration of what hoisting does, consider the following function definition:

```
function getValue(condition) {
    if (condition) {
        var value = "blue";
        // other code
        return value;
    } else {
        // value exists here with a value of undefined
        return null;
    }
    // value exists here with a value of undefined
}
```

If you are unfamiliar with JavaScript, then you might expect the variable value to only be created if condition evaluates to true. In fact, the variable value is created regardless. Behind

the scenes, the JavaScript engine changes the getValue function to look like this:

```
function getValue(condition) {
    var value;
    if (condition) {
        value = "blue";
        // other code
        return value;
    } else {
        return null;
    }
}
```

The declaration of value is hoisted to the top, while the initialization remains in the same spot. That means the variable value is actually still accessible from within the else clause. If accessed from there, the variable would just have a value of undefined because it hasn't been initialized.

It often takes new JavaScript developers some time to get used to declaration hoisting, and misunderstanding this unique behavior can end up causing bugs. For this reason, ECMAScript 6 introduces block level scoping options to make the controlling a variable's lifecycle a little more powerful.

Block-Level Declarations

Block-level declarations are those that declare variables that are inaccessible outside of a given block scope. Block scopes, also called lexical scopes, are created:

Inside of a function

Inside of a block (indicated by the { and } characters)

Block scoping is how many C-based languages work, and the introduction of block-level declarations in ECMAScript 6 is intended to bring that same flexibility (and uniformity) to JavaScript.

Let Declarations

The let declaration syntax is the same as the syntax for var. You can basically replace var with let to declare a variable, but limit the variable's scope to only the current

code block (there are a few other subtle differences discussed a bit later, as well). Since letdeclarations are not hoisted to the top of the enclosing block, you may want to always place let declarations first in the block, so that they are available to the entire block. Here's an example:

```
function getValue(condition) {
    if (condition) {
        let value = "blue";
        // other code
        return value;
    } else {
        // value doesn't exist here
        return null;
    }
    // value doesn't exist here
}
```

This version of the getValue function behaves much closer to how you'd expect it to in other C-based languages. Since the variable value is declared using let instead of var, the declaration isn't hoisted to the top of the function definition, and the variable value is no longer accessible once execution flows out of the if block. If condition evaluates to false, then value is never declared or initialized.

No Redeclaration

If an identifier has already been defined in a scope, then using the identifier in a let declaration inside that scope causes an error to be thrown. For example:

```
var count = 30;
// Syntax error
let count = 40;
```

In this example, count is declared twice: once with var and once with let. Because let will not redefine an identifier that already exists in the same scope, the let declaration will throw an error. On the other hand, no error is thrown if a let declaration creates a new variable with the same name as a variable in its containing scope, as demonstrated in the following code:

```
var count = 30;
// Does not throw an error
if (condition) {
    let count = 40;
    // more code
}
```

This let declaration doesn't throw an error because it creates a new variable called count within the if statement, instead of creating count in the surrounding block. Inside the if block, this new variable shadows the global count, preventing access to it until execution leaves the block.

Constant Declarations

You can also define variables in ECMAScript 6 with the const declaration syntax. Variables declared using const are considered constants, meaning their values cannot be changed once set. For this reason, every const variable must be initialized on declaration, as shown in this example:

```
// Valid constant
const maxItems = 30;
// Syntax error: missing initialization
const name;
```

The maxItems variable is initialized, so its const declaration should work without a problem. The name variable, however, would cause a syntax error if you tried to run the program containing this code, because name is not initialized.

Constants vs Let Declarations

Constants, like let declarations, are block-level declarations. That means constants are no longer accessible once execution flows out of the block in which they were declared, and declarations are not hoisted, as demonstrated in this example:

```
if (condition) {
    const maxItems = 5;
    // more code
}
```

```
// maxItems isn't accessible here
```

In this code, the constant maxItems is declared within an if statement. Once the statement finishes executing, maxItems is not accessible outside of that block.

In another similarity to let, a const declaration throws an error when made with an identifier for an already-defined variable in the same scope. It doesn't matter if that variable was declared using var (for global or function scope) or let (for block scope). For example, consider this code:

```
var message = "Hello!";
let age = 25;
// Each of these would throw an error.
const message = "Goodbye!";
const age = 30;
```

The two const declarations would be valid alone, but given the previous var and let declarations in this case, neither will work as intended.

Despite those similarities, there is one big difference between let and const to remember. Attempting to assign a const to a previously defined constant will throw an error, in both strict and non-strict modes:

```
const maxItems = 5;
maxItems = 6;          // throws error
```

Much like constants in other languages, the maxItems variable can't be assigned a new value later on. However, unlike constants in other languages, the value a constant holds may be modified if it is an object.

Declaring Objects with Const

A const declaration prevents modification of the binding and not of the value itself. That means const declarations for objects do not prevent modification of those objects. For example:

```
const person = {
    name: "Nicholas"
};
// works
```

```
person.name = "Greg";
// throws an error
person = {
    name: "Greg"
};
```

Here, the binding person is created with an initial value of an object with one property. It's possible to change person.name without causing an error because this changes what person contains and doesn't change the value that person is bound to. When this code attempts to assign a value to person (thus attempting to change the binding), an error will be thrown. This subtlety in how const works with objects is easy to misunderstand. Just remember: const prevents modification of the binding, not modification of the bound value.

The Temporal Dead Zone

A variable declared with either let or const cannot be accessed until after the declaration. Attempting to do so results in a reference error, even when using normally safe operations such as the typeof operation in this example:

```
if (condition) {
    console.log(typeof value);    // ReferenceError!
    let value = "blue";
}
```

Here, the variable value is defined and initialized using let, but that statement is never executed because the previous line throws an error. The issue is that value exists in what the JavaScript community has dubbed the temporal dead zone (TDZ). The TDZ is never named explicitly in the ECMAScript specification, but the term is often used to describe why let and const declarations are not accessible before their declaration. This section covers some subtleties of declaration placement that the TDZ causes, and although the examples shown all use let, note that the same information applies to const.

When a JavaScript engine looks through an upcoming block and finds a variable declaration, it either hoists the declaration to the top of the function or global scope (for var) or places the declaration in the TDZ (for let and const). Any attempt to access a variable in the TDZ results in a runtime error. That variable is only removed from the TDZ, and therefore

safe to use, once execution flows to the variable declaration.

This is true anytime you attempt to use a variable declared with let or const before it's been defined. As the previous example demonstrated, this even applies to the normally safe typeof operator. You can, however, use typeof on a variable outside of the block where that variable is declared, though it may not give the results you're after. Consider this code:

```
console.log(typeof value);        // "undefined"
if (condition) {
    let value = "blue";
}
```

The variable value isn't in the TDZ when the typeof operation executes because it occurs outside of the block in which value is declared. That means there is no value binding, and typeof simply returns "undefined".

The TDZ is just one unique aspect of block bindings. Another unique aspect has to do with their use inside of loops.

Block Binding in Loops

Perhaps one area where developers most want block level scoping of variables is within for loops, where the throwaway counter variable is meant to be used only inside the loop. For instance, it's not uncommon to see code like this in JavaScript:

```
for (var i = 0; i < 10; i++) {
    process(items[i]);
}
// i is still accessible here
console.log(i);                    // 10
```

In other languages, where block level scoping is the default, this example should work as intended, and only the for loop should have access to the i variable. In JavaScript, however, the variable i is still accessible after the loop is completed because the vardeclaration gets hoisted. Using let instead, as in the following code, should give the intended behavior:

```
for (let i = 0; i < 10; i++) {
    process(items[i]);
}
```

```
// i is not accessible here - throws an error
console.log(i);
```

In this example, the variable i only exists within the for loop. Once the loop is complete, the variable is no longer accessible elsewhere.

Functions in Loops

The characteristics of var have long made creating functions inside of loops problematic, because the loop variables are accessible from outside the scope of the loop. Consider the following code:

```
var funcs = [];

for (var i = 0; i < 10; i++) {
    funcs.push(function() { console.log(i); });
}
funcs.forEach(function(func) {
    func();         // outputs the number "10" ten times
});
```

You might ordinarily expect this code to print the numbers 0 to 9, but it outputs the number 10 ten times in a row. That's because i is shared across each iteration of the loop, meaning the functions created inside the loop all hold a reference to the same variable. The variable i has a value of 10 once the loop completes, and so when console.log(i) is called, that value prints each time.

To fix this problem, developers use immediately-invoked function expressions (IIFEs) inside of loops to force a new copy of the variable they want to iterate over to be created, as in this example:

```
var funcs = [];

for (var i = 0; i < 10; i++) {
    funcs.push((function(value) {
        return function() {
            console.log(value);
```

```
            }
        }(i)));
    }
    funcs.forEach(function(func) {
        func();         // outputs 0, then 1, then 2, up to 9
    });
```

This version uses an IIFE inside of the loop. The i variable is passed to the IIFE, which creates its own copy and stores it as value. This is the value used by the function for that iteration, so calling each function returns the expected value as the loop counts up from 0 to 9. Fortunately, block-level binding with let and const in ECMAScript 6 can simplify this loop for you.

Let Declarations in Loops

A let declaration simplifies loops by effectively mimicking what the IIFE does in the previous example. On each iteration, the loop creates a new variable and initializes it to the value of the variable with the same name from the previous iteration. That means you can omit the IIFE altogether and get the results you expect, like this:

```
var funcs = [];

for (let i = 0; i < 10; i++) {
    funcs.push(function() {
        console.log(i);
    });
}
funcs.forEach(function(func) {
    func();         // outputs 0, then 1, then 2, up to 9
})
```

This loop works exactly like the loop that used var and an IIFE but is, arguably, cleaner. The let declaration creates a new variable ieach time through the loop, so each function created inside the loop gets its own copy of i. Each copy of i has the value it was assigned at the beginning of the loop iteration in which it was created. The same is true for for-in and for-

of loops, as shown here:

```
var funcs = [],
    object = {
        a: true,
        b: true,
        c: true
    };
for (let key in object) {
    funcs.push(function() {
        console.log(key);
    });
}
funcs.forEach(function(func) {
    func();         // outputs "a", then "b", then "c"
});
```

In this example, the for-in loop shows the same behavior as the for loop. Each time through the loop, a new key binding is created, and so each function has its own copy of the key variable. The result is that each function outputs a different value. If var were used to declare key, all functions would output "c".

外文翻译二：

Javascript 高级程序设计

JavaScript 与 HTML 的交互是通过事件来处理的，这些事件指示在文档或浏览器窗口中何时发生特定时刻。事件可以使用侦听器（也称为处理程序）来订阅，只有在事件发生时才能执行。这个模型在传统软件工程中被称为观察者模式，它允许页面行为（用 JavaScript 定义）和页面外观（用 HTML 和 CSS 定义）之间的松散耦合。事件最初出现在 Internet Explorer 3 和 Netscape Navigator 2 中，作为从服务器到浏览器的一些表单处理的一种方式。到 Internet Explorer 4 和 Netscape 4 发布时，每个浏览器都提供了几代相似但不同的 API。 DOM Level 2 是以合乎逻辑的方式标准化 DOM 事件 API 的第一次尝试。 Internet Explorer 9，Firefox，Opera，Safari 和 Chrome 都已经实现了 DOM Level 2 事件的核心部分。 Internet Explorer 8 是使用纯专有事件系统的最后一个主要浏览器。浏览器事件系统是一个复杂的系统。尽管所有主流浏览器都实现了 DOM Level 2 事件，但这个规范并没有涵盖所有的事件类型。 BOM 也支持事件，并且这些事件与 DOM 事件之间的关系经常令人困惑，因为长期缺乏文档（HTML5 试图澄清的东西）。更为复杂的是由 DOM 级别 3 增加 DOM 事件 API。根据您的要求，使用事件可能相对简单或非常复杂。不过，还有一些重要的核心概念要理解。事件流

当第四代网络浏览器（Internet Explorer 4 和 Netscape Communicator 4）开始开发时，浏览器开发团队遇到了一个有趣的问题 :页面的哪一部分拥有特定的事件？为了理解这个问题，在一张纸上考虑一系列同心圆。当你把手指放在中间时，它不仅仅是一个圆圈，而是纸张上的所有圆圈。两个开发团队都以同样的方式浏览浏览器事件。当你点击一个按钮时，他们得出结论，你不仅点击按钮，而且还点击它的容器和整个页面。事件流描述事件在页面上的接收顺序，有趣的是，Internet Explorer 和 Netscape 开发团队提出了一个几乎完全相反的事件流概念。 Internet Explorer 将支持事件冒泡流，而 Netscape Communicator 将支持事件捕获流。

事件冒泡 Internet Explorer 事件流称为事件冒泡，因为事件被称为从最特定的元素（文档树中最深的可能点）开始，然后向上流向最不特定的节点（文档）。考虑下面的 HTML 页面：

<!DOCTYPE html> <html> <head> <title>Event Bubbling Example</title> </head> <body> Click Me </body> </html>

点击页面中的 元素，点击事件按以下顺序发生：1. 2. <body> 3. <html> 4. document click 事件首先在< div>，这是被点击的元素。然后点击事件沿着 DOM 树行进，在每个节点上沿着它的方向进行，直到到达文档对象。图 13-1 显示了这种效果。所有现代浏览器都支持事件冒泡，尽管在实现方式上有一些变化。 Internet Explorer 5.5 和更早版本将<冒泡>跳到<html>元素（从<body>直接转到文档）。Internet Explorer 9, Firefox, Chrome 和 Safari 继续事件冒泡到窗口对象。事件捕获 Netscape Communicator 团队提出了一种称为事件捕获的替代事件流程。事件捕获的理论是最少的特定节点应该首先接收事件，最特定的节点应该最后接收事件。事件捕获的目的是在事件到达预定目标之前拦截事件。如果前面的例子与事件捕获一起使用，单击 元素按以下顺序对 click 事件进行处理 :1. document 2. <html> 3. <body> 4. 通过事件捕获，click 事件首先被文档接收，然后继续沿着 DOM 树到达事件的实际目标 元素。这个流程如图 13-2 所示。尽管这是 Netscape Communicator 唯一的事件流模型，但 Internet Explorer 9，Safari，Chrome，Opera 和 Firefox 目前支持捕获事件。所有这些实际上都是在窗口级事件开始事件捕获的，尽管 DOM 级别 2 事件规范指出事件应该从文档开始。通常不会使用事件捕获，因为旧版浏览器缺少支持。一般的建议是使用事件冒泡自由，同时保留特殊情况下的事件捕获。

DOM 事件流 DOM 级别 2 事件指定的事件流具有三个阶段：事件捕获阶段，目标阶段和事件冒泡阶段。事件捕获首先发生，如果有必要的话提供机会来拦截事件。接下来，实际目标接收事件。最后一个阶段是冒泡，最终可以对事件做出反应。考虑到以前使用的简单 HTML 示例，单击 按照图 13-3 中指示的顺序对事件进行解析。在 DOM 事件流中，实际目标（ 元素）在捕获阶段不会收到事件。这意味着捕获阶段从文档移动到<html>到<body>并停止。下一个阶段是“目标”，这个阶段在 上进行，在事件处理方面被认为是冒泡阶段的一部分（稍后讨论）。然后，冒泡阶段发生并且事件返回到文档。大多数支持 DOM 事件流的浏览器都实现了一个怪癖。即使 DOM 级别 2 事件规范指出捕获阶段没有达到事件目标，Internet Explorer 9，Safari，Chrome，Firefox 和 Opera 9.5 以及更高版本都会在事件目标的捕获阶段找到事件。最终的结果是有两个机会来处理目标上的事件。事件处理程序

事件是用户或浏览器本身执行的某些操作。这些事件具有点击，加载和鼠标悬停等名称。为响应事件而调用的函数称为事件处理函数（或事件侦听器）。事件处理程序名称以“on”开头，所以 click 事件的事件处理程序被称为 onclick，而 load 事件的事件处理程序被称为 onload。分配事件处理程序可以通过许多不同的方式完成。

HTML 事件处理程序可以使用具有事件处理程序名称的 HTML 属性来分配特定元素支

持的每个事件。属性的值应该是一些 JavaScript 代码来执行。例如，要点击一个按钮来执行一些 JavaScript，可以使用下面的代码：

<input type="button" value="Click Me" onclick="alert('Clicked')" />

点击此按钮时，会显示警报。 这种交互是通过指定 onclick 属性并将一些 JavaScript 代码指定为值来定义的。 请注意，由于 JavaScript 代码是一个属性值，因此不能使用＆符号，双引号，小于或大于等 HTML 语法字符，而不能将其转义。 在这种情况下，使用单引号而不是双引号来避免使用 HTML 实体。 要使用双引号，请将代码更改为以下内容：

<input type="button" value="Click Me" onclick="alert("Clicked")" />

在 HTML 中定义的事件处理程序可能包含要执行的确切操作，或者可以调用页面上其他位置定义的脚本，如下例所示：

<script type="text/javascript"> function showMessage(){ alert("Hello world!"); }</script> <input type="button" value="Click Me" onclick="showMessage()" />

事件是用户或浏览器本身执行的某些操作。这些事件具有点击, 加载和鼠标悬停等名称。为响应事件而调用的函数称为事件处理函数（或事件侦听器）。事件处理程序名称以"on"开头, 所以 click 事件的事件处理程序被称为 onclick, 而 load 事件的事件处理程序被称为 onload。分配事件处理程序可以通过许多不同的方式完成。

HTML 事件处理程序可以使用具有事件处理程序名称的 HTML 属性来分配特定元素支持的每个事件。属性的值应该是一些 JavaScript 代码来执行。例如，要点击一个按钮来执行一些 JavaScript，可以使用下面的代码：

<input type ="button"value ="Click Me"onclick ="alert（'Clicked'）"/>

点击此按钮时，会显示警报。这种交互是通过指定 onclick 属性并将一些 JavaScript 代码指定为值来定义的。请注意，由于 JavaScript 代码是一个属性值，因此不能使用＆符号，双引号，小于或大于等 HTML 语法字符，而不能将其转义。在这种情况下，使用单引号而不是双引号来避免使用 HTML 实体。要使用双引号，请将代码更改为以下内容：

在 HTML 中定义的事件处理程序可能包含要执行的确切操作，或者可以调用页面上其他位置定义的脚本，如下例所示：

函数 showMessage（）{alert（"Hello world！"）；在这个代码中，当按钮被点击时，按钮调用 showMessage（）方法。 showMessage（）函数定义在一个单独的<script>元素中，也可以包含在外部文件中。作为事件处理程序执行的代码可以访问全局范围内的所有内容。

以这种方式分配的事件处理程序具有一些独特的方面。 首先，创建一个包装属性值的

函数。 该函数有一个特殊的局部变量叫做 event，它是事件对象（在本章稍后讨论）：

这使您可以访问事件对象，而无需自己定义它，也无需将其从封闭函数的参数列表中拉出。 函数内部的这个值等同于事件的目标元素，例如：

<!-- outputs “Click Me”--> <input type=”button” value=”Click Me” onclick=”alert(this.value)”>

这个动态创建的函数的另一个有趣的方面是它如何扩大范围链。 在函数中，文档和元素本身的成员可以像访问局部变量一样被访问。 该函数通过使用以下内容的范围链增强来完成此操作：

function(){with(document){ with(this){//attribute value }} }

这意味着事件处理程序可以轻松访问自己的属性。 以下功能与上例相同：

<!-- outputs “Click Me” --> <input type=”button” value=”Click Me” onclick=”alert(value)”>

如果元素是表单输入元素，则作用域链也包含父表单元素的条目，使该功能等同于以下内容：

function(){with(document){with(this.form){with(this)//attribute value } } } }

基本上，这种扩充允许事件处理程序代码访问同一表单的其他成员，而不用引用表单元素本身。 例如

<form method=”post”> <input type=”text” name=”username” value=””> <input type=”button” value=”Echo Username” onclick=”alert(username.value)”> </form>

点击这个例子中的按钮会显示文本框中的文本。 请注意，它只是直接引用用户名。 在 HTML 中分配事件处理程序有一些缺点。 第一个是计时问题：HTML 元素可能出现在页面上，并且在事件处理程序代码准备就绪之前由用户交互。 在前面的例子中，假设 showMessage（）函数没有被定义，直到页面稍后的按钮代码之后。 如果用户在定义 showMessage（）之前单击该按钮，则会发生错误。 由于这个原因，大多数 HTML 事件处理程序被包含在 try-catch 块中，以便它们安静地失败，如下例所示：

<input type=”button” value=”Click Me” onclick=”try{showMessage();}catch(ex){}”>

var btn = document.getElementById(“myBtn”); btn.onclick = function(){ alert(“Clicked”); };

这里，从文档中检索一个按钮，并分配一个 onclick 事件处理程序。 请注意，事件处理

程序在运行代码之前不会被分配，所以如果代码出现在页面中按钮的代码之后，可能会有一段时间，在点击按钮时该按钮将不会执行任何操作。 当使用 DOM Level 0 方法分配事件处理程序时，事件处理程序被认为是元素的一个方法。 因此事件处理程序在元素的范围内运行，这意味着这与元素是等价的。 这里是一个例子：

```
var btn = document.getElementById(“myBtn”); btn.onclick = function(){       alert(this.id); //”myBtn” };
```

这个代码显示单击按钮时元素的 ID。 该 ID 使用 this.id 检索。 可以使用它来访问事件处理程序中的任何元素的属性或方法。 以这种方式添加的事件处理程序旨在用于事件流的冒泡阶段。 您可以通过将事件处理程序属性的值设置为 null 来删除通过 DOM Level 0 方法分配的事件处理程序，如下例所示：

```
btn.onclick = null;       //remove event handler
```

一旦事件处理程序设置为空，该按钮不再需要单击时采取任何操作。

外文原文二：

Professional JavaScript for web developers

JavaScript's interaction with HTML is handled through events, which indicate when particular moments of interest occur in the document or browser window. Events can be subscribed to using listeners (also called handlers) that execute only when an event occurs. This model, called the observer pattern in traditional software engineering, allows a loose coupling between the behavior of a page (defined in JavaScript) and the appearance of the page (defined in HTML and CSS). Events first appeared in Internet Explorer 3 and Netscape Navigator 2 as a way to offload some form processing from the server onto the browser. By the time Internet Explorer 4 and Netscape 4 were released, each browser delivered similar but different APIs that continued for several generations. DOM Level 2 was the first attempt to standardize the DOM events API in a logical way. Internet Explorer 9, Firefox, Opera, Safari, and Chrome all have implemented the core parts of DOM Level 2 Events. Internet Explorer 8 was the last major browser to use a purely proprietary event system. The browser event system is a complex one. Even though all major browsers have implemented DOM Level 2 Events, the specification doesn't cover all event types. The BOM also supports events, and the relationship between these and the DOM events is often confusing because of a longtime lack of documentation (something that HTML5 has tried to clarify). Further complicating matters is the augmentation of the DOM events API by DOM Level 3. Working with events can be relatively simple or very complex, depending on your requirements. Still, there are some core concepts that are important to understand. EVENT FLOW

When development for the fourth generation of web browsers began (Internet Explorer 4 and Netscape Communicator 4), the browser development teams were met with an interesting question: what part of a page owns a specific event? To understand the issue, consider a series of concentric circles on a piece of paper. When you place your finger at the center, it is inside of not just one circle but all of the circles on the paper. Both development teams looked at browser events in the same way. When you click on a button, they concluded, you're clicking not just on the button but also on its container and on the page as a whole.

Event flow describes the order in which events are received on the page, and interestingly, the Internet Explorer and Netscape development teams came up with an almost exactly opposite concept of event flow. Internet Explorer would support an event bubbling flow, whereas Netscape Communicator would support an event capturing flow.

Event Bubbling The Internet Explorer event flow is called event bubbling, because an event is said to start at the most specific element (the deepest possible point in the document tree) and then flow upward toward the least specific node (the document). Consider the following HTML page:

<!DOCTYPE html> <html> <head> <title>Event Bubbling Example</title> </head> <body> Click Me </body> </html>

When you click the element in the page, the click event occurs in the following order: 1. 2. <body> 3. <html> 4. document The click event is first fired on the , which is the element that was clicked. Then the click event goes up the DOM tree, firing on each node along its way until it reaches the document object. Figure 13-1 illustrates this effect. All modern browsers support event bubbling, although there are some variations on how it is implemented. Internet Explorer 5.5 and earlier skip bubbling to the <html> element (going from <body> directly to document). Internet Explorer 9, Firefox, Chrome, and Safari continue event bubbling up to the window object. Event Capturing The Netscape Communicator team came up with an alternate event flow called event capturing. The theory of event capturing is that the least specific node should receive the event first and the most specific node should receive the event last. Event capturing was really designed to intercept the event before it reached the intended target. If the previous example is used with event capturing, clicking the element fires the click event in the following order: 1. document 2. <html> 3. <body> 4. With event capturing, the click event is first received by the document and then continues down the DOM tree to the actual target of the event, the element. This flow is illustrated in Figure 13-2. Although this was Netscape Communicator's only event flow model, event capturing is currently supported in Internet Explorer 9, Safari, Chrome, Opera, and Firefox. All of them actually begin event capturing at the window-level event despite the fact that the DOM Level 2 Events specification indicates that the events should begin at document. Event capturing is generally not used because of a lack of support in older

browsers. The general advice is to use event bubbling freely while retaining event capturing for special circumstances.

DOM Event Flow The event flow specified by DOM Level 2 Events has three phases: the event capturing phase, at the target, and the event bubbling phase. Event capturing occurs first, providing the opportunity to intercept events if necessary. Next, the actual target receives the event. The final phase is bubbling, which allows a final response to the event. Considering the simple HTML example used previously, clicking the fires the event in the order indicated in Figure 13-3. In the DOM event flow, the actual target (the element) does not receive the event during the capturing phase. This means that the capturing phase moves from document to <html> to <body> and stops. The next phase is "at target," which fires on the and is considered to be part of the bubbling phase in terms of event handling (discussed later). Then, the bubbling phase occurs and the event travels back up to the document. Most of the browsers that support DOM event flow have implemented a quirk. Even though the DOM Level 2 Events specification indicates that the capturing phase doesn't hit the event target, Internet Explorer 9, Safari, Chrome, Firefox, and Opera 9.5 and later all fire an event during the capturing phase on the event target. The end result is that there are two opportunities to work with the event on the target. EVENT HANDLERS

Events are certain actions performed either by the user or by the browser itself. These events have names like click, load, and mouseover. A function that is called in response to an event is called an event handler (or an event listener). Event handlers have names beginning with "on", so an event handler for the click event is called onclick and an event handler for the load event is called onload. Assigning event handlers can be accomplished in a number of different ways.

HTML Event Handlers Each event supported by a particular element can be assigned using an HTML attribute with the name of the event handler. The value of the attribute should be some JavaScript code to execute. For example, to execute some JavaScript when a button is clicked, you can use the following:

<input type="button" value="Click Me" onclick="alert('Clicked')" />

When this button is clicked, an alert is displayed. This interaction is defined by specifying the onclick attribute and assigning some JavaScript code as the value. Note that since the

JavaScript code is an attribute value, you cannot use HTML syntax characters such as the ampersand, double quotes, less-than, or greater-than without escaping them. In this case, single quotes were used instead of double quotes to avoid the need to use HTML entities. To use double quotes, you will change the code to the following:

<input type="button" value="Click Me" onclick="alert("Clicked")" />

An event handler defined in HTML may contain the precise action to take or it can call a script defined elsewhere on the page, as in this example:

<script type="text/javascript"> function showMessage(){ alert("Hello world!"); }</script> <input type="button" value="Click Me" onclick="showMessage()" />In this code, the button calls showMessage() when it is clicked. The showMessage() function is defined in a separate <script> element and could also be included in an external file. Code executing as an event handler has access to everything in the global scope.

Event handlers assigned in this way have some unique aspects. First, a function is created that wraps the attribute value. That function has a special local variable called event, which is the event object (discussed later in this chapter):

<!-- outputs "click" --> <input type="button" value="Click Me" onclick="alert(event.type)">

This gives you access to the event object without needing to define it yourself and without needing to pull it from the enclosing function's argument list. The this value inside of the function is equivalent to the event's target element, for example:

<!-- outputs "Click Me"--> <input type="button" value="Click Me" onclick="alert(this.value)">

Another interesting aspect of this dynamically created function is how it augments the scope chain. Within the function, members of both document and the element itself can be accessed as if they were local variables. The function accomplishes this via scope chain augmentation using with:

function(){with(document){ with(this){//attribute value }} }

This means that an event handler can access its own properties easily. The following is functionally the same as the previous example:

<!-- outputs "Click Me" --> <input type="button" value="Click Me"

onclick="alert(value)">

If the element is a form input element, then the scope chain also contains an entry for the parent form element, making the function the equivalent to the following:

function(){with(document){with(this.form){with(this)//attribute value } } } }

Basically, this augmentation allows the event handler code to access other members of the same form without referencing the form element itself. For example:

<form method="post">

<input type="text" name="username" value="">

<input type="button" value="Echo Username" onclick="alert(username.value)">

</form>

Clicking on the button in this example results in the text from the text box being displayed. Note that it just references username directly. There are a few downsides to assigning event handlers in HTML. The first is a timing issue: it's possible that the HTML element appears on the page and is interacted with by the user before the event handler code is ready. In the previous example, imagine a scenario where the showMessage() function isn't defined until later on the page, after the code for the button. If the user were to click the button before showMessage() was defined, an error would occur. For this reason, most HTML event handlers are enclosed in try-catch blocks so that they quietly fail, as in the following example:

<input type="button" value="Click Me" onclick="try{showMessage();}catch(ex){}">

If this button is clicked before the showMessage() function is defined, no JavaScript error occurs because the error is caught before the browser can handle it. Another downside is that the scope chain augmentation in the event handler function can lead to different results in different browsers. The rules being followed for identifier resolution are slightly different amongst JavaScript engines, and so the result of accessing unqualified object members may cause errors. The last downside to assigning event handlers using HTML is that it tightly couples the HTML to the JavaScript. If the event handler needs to be changed, you may need to change code in two places: in the HTML and in the JavaScript. This is the primary reason that many developers avoid HTML event handlers in favor of using JavaScript to assign event handlers.

DOM Level 0 Event Handlers The traditional way of assigning event handlers in JavaScript

is to assign a function to an event handler property. This was the event handler assignment method introduced in the fourth generation of web browsers, and it still remains in all modern browsers because of its simplicity and cross-browser support. To assign an event handler using JavaScript, you must first retrieve a reference to the object to act on. Each element (as well as window and document) has event handler properties that are typically all lowercase, such as onclick. An event handler is assigned by setting the property equal to a function, as in this example:

```
Var btn = document.getElementById("myBtn"); btn.onclick = function(){ alert("Clicked"); };
```

Here, a button is retrieved from the document and an onclick event handler is assigned. Note that the event handler isn't assigned until this code is run, so if the code appears after the code for the button in the page, there may be an amount of time during which the button will do nothing when clicked. When assigning event handlers using the DOM Level 0 method, the event handler is considered to be a method of the element. The event handler, therefore, is run within the scope of element, meaning that this is equivalent to the element. Here is an example:

```
var btn = document.getElementById("myBtn");
btn.onclick = function(){
  alert(this.id);       //"myBtn"
};
```

This code displays the element's ID when the button is clicked. The ID is retrieved using this.id. It's possible to use this to access any of the element's properties or methods from within the event handlers. Event handlers added in this way are intended for the bubbling phase of the event flow. You can remove an event handler assigned via the DOM Level 0 approach by setting the value of the event handler property to null, as in the following example:

```
btn.onclick = null;       //remove event handler
```

Once the event handler is set to null, the button no longer has any action to take when it is clicked.

浙江树人大学

本 科 生 毕 业 设 计

工 作 指 导 记 录

题　　目 晨博依恋服饰购物网站设计与实现

专　　业 电子商务

班　　级 141 班

姓　　名 宋益晨

指导教师 杨晓波

所在学院 信息科技学院

工作指导记录

通过与宋益晨同学的交流，最终帮助他确立了研究的课题，并且初步确定了系统整体的规范方向。同时把毕业设计（论文）的格式规范及基本要求，对毕业设计的开发和论文的写作工作提了具体要求，解释了选题的含义及基本思路，回答了他的有关疑问。

指导老师（签字）：

2016 年 10 月 10 日

工作指导记录

发放《过程材料》，宋益晨同学以“晨博依恋服饰购物网站规划与设计”为题目。审读以后，发现以下问题：系统背景分析不全面，系统部分功能不明确。不过，总体还是把握住了此项目。对以上问题，我对他进行了一定的技术的交流，并提了一点建议。同时讨论了确定使用的开发语言。

指导老师（签字）：

2016 年 10 月 20 日

工作指导记录

分析了宋益晨同学的《文献综述》，发现其对文献综述主要要表达的内容还不明确。针对此问题，我给他提供了如何写文献综述的材料，并与其进行了沟通，分析了一下“文献综述”的具体要介绍的内容，比如：目前国内外服装网站等。

除此之外，还发现其借鉴的参考文献部分年份较老，内容都比较过时。所以介绍了相关的下载文献的网址和关键词搜索的优化技术。同时布置了开题报告的撰写，讨论了开题报告包含的内容。

指导老师（签字）：

2016 年 11 月 18 日

工作指导记录

对宋益晨同学的《开题报告》进行审阅分析后，发现一个很大的问题：他对开题报告的整体方向有点偏离。因为他的系统开发，而在其内容中大多数都是背景分析等内容，系统本身功能模块没有介绍。

为此，与他进行了沟通，详细的分析了开题报告的具体结构，帮助其明确了具体内容。此外，对功能模块方面提供了一些建议。同时对于外文翻译做出一定的意见指导。

指导老师（签字）：

2016 年 12 月 12 日

工作指导记录

经过一番努力，宋益晨同学顺利通过开题答辩。接下来进入修改老师的建议的环节，并检查他的过程材料。

对开题报告审阅，发现提出的问题没有给出解决方法，让他修改开题报告。

指导老师（签字）：

2016 年 12 月 24 日

工作指导记录

由于寒假的来临，要求宋益晨同学来办公室进行项目进度的报告。他在报告完后提出了几个功能系统开发方面的问题。通过沟通与交流，对其问题一一进行了解答。

指导老师（签字）：

2016 年 12 月 30 日

工作指导记录

宋益晨同学整体项目进展情况还可以，许多问题也得到了解决。今天要求其上报寒假计划。他的寒假计划主要为：开发部分功能模块。

同时讨论了数据的 E-R 图的内容和数据库的概念结构设计。

指导老师（签字）：

2016 年 1 月 16 日

工作指导记录

在这次指导中把一些事情进行布置，同时对于某些问题的存在进行一个提前的提醒。同时布置接下来的内容安排：

首先，对网站各个模块的界面进行显示的实现。然后初步开发各个模块的功能。

指导老师（签字）：

2016 年 1 月 23 日

工作指导记录

与宋益晨同学交流了前面的任务进度，交流了寒假的毕业设计情况，发现功能模块开发的较少，要求及时去做，并按照网站里的模块来填充内容。同时解答了一些他提出的问题。

指导老师（签字）：

2017 年 2 月 29 日

工作指导记录

询问了宋益晨同学该阶段的毕业设计进展情况。对他网站进行评阅，指出其存在的问题。在听取了其汇报后，针对其提出的问题一一进行解答

指导老师（签字）：

2017 年 3 月 24 日

工作指导记录

与宋益晨同学就其毕业设计论文初稿进行了讨论，发现他在网站模块设计阶段存在问题，而且摘要比较口语化，文字不怎么通顺，摘要翻译直接复制网上翻译内容。论文参考文献 上标没有，格式不对。建议其重新修饰一下摘要的内容和自己一句句翻译出摘要内容，对网站模块设计进行修改。

指导老师（签字）：

2017 年 3 月 31 日

工作指导记录

对于宋益晨同学论文的写作情况做一个了解和指导，同时对于系统做一个了解。对论文进行了初步的审阅，发现存在的问题：内容分段不合理，图标标注不合理等问题，要求及时改正所提出的问题。

指导老师（签字）：

2017 年 4 月 17 日

工作指导记录

询问了宋益晨同学毕业论文的修改情况。在审阅了他修改后的论文，发现经过上一次的指导后，发现改正了大多数问题。和他交流一些论文撰写过程中难点疑点问题，帮助其更好的完善论文。

另外建议他在毕业论文里加入前端设计，要求该生结合设计对毕业论文进行修改，并于尽快将论文上交以便修改。

指导老师（签字）：

2017 年 4 月 28 日

工作指导记录

对宋益晨同学的论文进行了最后的修改，对论文的定稿，发现论文的格式还是存在许多问题，详细设计的地方还不够清晰，要求该生在本周内完成上述内容的修改。

同时对该生答辩 ppt 的内容进行指导，要求该生做好答辩 ppt。

指导老师（签字）：

2017 年 5 月 3 日

工作指导记录

对宋益晨同学的论文进行再一次的审读，对论文中的图表部分进行查阅，发现一些绘制上的错误，与他进行沟通后，及时做了修改。

同时指导该生如何进行答辩，做好网站演示的准备。

指导老师（签字）：

2017 年 5 月 4 日

工作指导记录

对宋益晨同学的论文和过程材料进行最后一次的审读，对论文中的格式进行修改，与他进行沟通后，及时做了修改。指导他做好答辩的准备。

指导老师（签字）：

2017 年 5 月 5 日