

华中科技大学硕士学位论文

华中科技大学水电与数字化工程学院 硕士学位论文个人主要工作与学术成果总结

论文题目：基于敏捷方法的水轮机仿真软件设计与实现

研究方向：水电机组控制与故障诊断

论文中个人主要工作（不够可附页）

结合编程实践提出了一系列程序开发工作中有价值的开发思想和指导原则；总结了软件开发过程之中涉及的各个阶段及可选用的技术及开源辅助工具；较为深入探索了控制系统仿真软件的 OO 设计方案及实现方法。创造性的使用了基于敏捷的快速迭代开发方法进行了仿真程序的开发；进行了仿真程序 GUI 的开发；应用开发的程序对水轮机调速系统进行了可视化建模和暂态仿真；给出了 GSA 算法的一种并行的、面向对象的实现。

论文中的主要创新点（不够可附页）

开发了基于 java web 技术的仿真 demo 软件；

结合敏捷方法开发了面向对象的通用仿真程序，并探讨了敏捷方法在工程中的应用；

使用仿真程序对水轮机调速系统进行了暂态仿真；

开发了松散耦合的多线程并行水轮机调速系统参数辨识程序。

硕士学位论文主要研究成果的发表或获奖情况

序号	发表论文题目/获奖成果名称	刊物名称/ 获奖部门	刊物/奖励级别	发表论文/ 获奖时间	作者署名 名次
1					
2					
3					
4					

华 中 科技大学硕士学位论文

摘要

随着计算机技术的发展逐步深入到各行各业，经典的控制系统仿真领域也从中收益。以基于 matlab 的 Simulink 为代表的仿真软件，提供了一种可视化的建模方式。类似的软件操作环境开始大行其道，并逐渐深入人心。揭示这些软件背后的原理，有助于深入理解软件开发过程和仿真软件的实现方法

在“互联网+”呼声正高、主张“万物互联”的时代，传统的各行各业都不可避免地要经历各种尝试，并最终接受互联网的改造。在这个进程之中，工程和研究领域方面的软件开发也将受到一定的影响。为这些即将到来的改变做好准备，颇有裨益。

本世纪以来，笼罩在传统的软件开发方法上的困境迷雾逐渐被一股敏捷的清风吹散。软件开发过程不再死板地遵循瀑布方法，而是以客户为本，强调迅速得到可以运行的软件版本。敏捷不仅仅是一个方法论，还是一组价值观。探讨如何在特定的工程和研究领域如何遵循敏捷的方法进行软件的开发，具有积极的意义。

传统的软件架构方式中，出现过以技术为导向的架构方法，spring 等优秀框架的出现呼吁我们更多地放眼于业务。技术、方法只是完成业务的手段，而不应该干预基本的面向对象开发原则。综观软件行业的优秀实践，实事求是，灵活地运用最为合适的方法和技术，才能获得满意的效果。

本文以经典的控制系统可视化建模与仿真软件的设计、开发为例，基于 java 平台及其 web 技术，灵活运用 groovy、python 等动态语言，探讨了如何将敏捷的思想运用到相关领域软件的开发之中。

最终，本文探讨了面向对象的仿真程序开发方法，完成了仿真软件的开发，探讨了仿真程序的拓展方法，并使用软件对线性化的水轮机调节系统进行了仿真和参数辨识。

关键词：控制系统；仿真；敏捷开发；水轮机；调节系统

Abstract

with the computer science and technology developing all the time, affecting many other fields as well as the traditional simulation theory. The Simulink toolkit, which bases on the matlab platform, provides a way to modeling visually. Similar software operating environment become popular from then on. Reveal the principle of those works can help take a deep insight of the software development process and the implementation of simulation software.

Nowadays the "Internet Plus" action plan is making its way to help connecting almost everything in the world, thus creating a world joined together by the internet. The traditional fields will inevitably be affected and changed in the process, as well as the software development among the engineering and research field. Preparing for the incoming new age is necessary.

The century has brought us the cool breeze of agile methodology, blowing away the fog of the dilemma of traditional software development methodology. Agile method weighs more on customer demand and working software, other than following the rigid guide of the waterfall development approach. Agile is not only a methodology, but also a group of values. It is meaningful to study how agile method can help to develop software in the engineering field.

In the traditional software architecture, there has been a technology oriented architecture approach. The springframework stands out to tell us concentrate on the object-oriented principles, not on technologies and methods, which is just a way to achieve goals. Excellent practice in the software industry shows that only by seeing the truth from facts and adopting appropriate methods and technologies can we obtain satisfactory results.

This thesis takes the software design and development of simulation of hydraulic turbine regulating system as an example, using flexibly the java web technologies as well as two dynamic language -- groovy and python, to discuss how to apply agile method to software development in related areas.

As the last part, by using object-oriented method in developing the simulation program,

华 中 科技大学硕士学位论文

this thesis achieve its goal to develop a simulation software demo. It also discusses the method of extending the simulation program, uses the software to simulate the linearized model of hydraulic turbine regulating system, and perform parameter identification algorithm on it.

Keywords: control system, simulation, agile development, hydraulic turbine, regulating system

华 中 科技大学硕士学位论文

目录

摘要.....	2
Abstract.....	3
目录.....	5
1. 绪论.....	8
1.1 课题背景及研究意义	8
1.2 水轮机仿真软件系统概述	9
1.3 本文的基本内容	9
2. 开发约定及思想	11
2.1 开发约定	11
2.2 编程思想	15
3. 开发环境搭建	18
3.1 版本控制工具——git/github	20
3.2 项目构建工具——maven	21
3.3 后端架构	25
3.4 前端架构——jQuery 与 lodash	27
3.5 前后端通讯格式——json	28
3.6 derby 数据库和 jetty 服务器	29
3.7 脚本与动态语言支持——groovy 和 python	30
4. 仿真原理及调速系统模型	33
4.1 控制系统仿真的原理	33
4.2 控制系统的仿真算法	37
4.3 控制系统仿真程序的实现	38
4.4 仿真程序的约束	41
4.5 水轮机调速系统模型	42

华 中 科技大学硕士学位论文

4.6 本章小结	44
5. 程序原型及快速迭代开发	46
5.1 第 0 次迭代——groovy 实现	46
5.2 第 0 次迭代——java 实现	50
5.3 已经完成了?	53
5.4 第一次迭代	54
5.5 第二次迭代	56
5.6 是时候模块化了	57
5.7 类层次结构	60
5.8 仿真实例: ExciterTest	63
5.9 程序的进一步分离	66
5.10 迭代的计算顺序	70
5.11 本章小结	71
6. 仿真程序 GUI 开发简介	72
6.1 学习测试不可或缺	73
6.2 前端程序的基本结构	74
6.3 程序的用户界面	75
6.4 一次性脚本代码的使用	77
6.5 对程序开发的思考	79
6.6 本章小结	80
7. 仿真程序的应用	81
7.1 仿真软件的拓展	81
7.2 调速系统暂态仿真	83
7.3 调速系统参数辨识	86
8. 全文总结及工作展望	92
8.1 全文总结	92
8.2 工作展望	93

华 中 科技大学硕士学位论文

致谢..... 错误！未定义书签。

参考文献.....94

1. 绪论

1.1 课题背景及研究意义

近十年来,我国社会基础建设和公共设施建设已经高度发展和成熟,与此对应的,应用工程领域的理论和实践也得到了长足的发展,并在水利、航天、铁路交通等领域有较多的对外科技输出。在一些企业的带领下,软件开发领域也积累了十分宝贵的经验。特别是在“世界第一人口大国”的背景之下,应对诸如春运火车票网购、节日网络促销等等商用业务,对软件工程和软件架构的提出了极高的要求。

在水电站运行、电力调度等工程领域,自动化的软硬件监控系统也得到了广泛运用。在下一步的“5G”和物联网发展已成为热点之际,所有的传统行业都不免要或多或少地逐步与“互联网”相结合。认识到这一结合的重要性,研究针对工程应用领域的软件开发思想及方法、过程及手段,变得尤为重要。软件开发周期长,对组织管理、技术方法、商业运作等方面都有较高的要求;软件程序使用的封装、复用方法,存在于程序员和最终客户之间的“需求鸿沟”,都使得最终产品的失败率较高,客户的实际需求难以得到满足。解决“鸿沟”的最佳方法就是管理、开发、用户等各方面的通力合作。必须认识到,软件架构及开发并不仅仅是商业开发团队的禁脔,工程领域尤其如此,软件开发活动需要得到技术人员、操作人员的全力支持,部分最终客户、研究人员也需要熟悉其中的基本思想及原理,以看透隐藏在用户界面及软件包背后的“代码迷雾”。

从上世纪 90 年代“微软”进入中国以来,个人电脑得到了极大的普及,很多功能强大、界面精美的软件随之而来,极大地提高了人们的工作效率和生活品质;本世纪开始的第十年,“苹果”、“谷歌”携安卓和 IOS 手机终端平台进入大众的视野,同样的,一大批 app 丰富了人们的视野、方便了人们的生活。在工程领域,matlab 软件深受工程和研究人员的喜爱,尤其是 simulink,其“所见即所得”的界面风格和方便快捷的可视化建模,极大地简化了控制领域的设计及仿真。但是由于商业原因,simulink 及很多的其他软件都不是开源的,这些优秀的软件作品得不到

华 中 科技大学硕士学位论文

复用。即使现阶段大多数平台采用了面向服务的架构并开放了调用接口，也让一些初级开发人员和技術、研究人员不知所以。为此，同时也为普及和研究的目的，揭示这些优秀软件的内部原理，构建方式，具有重要的意义。对于工程软件的研究和开发，也具有一定的借鉴意义。

1.2 水轮机仿真软件系统概述

本系统采用 java 平台及其相关技术搭建而成。主要包含基于 html 的前端交互界面，后台仿真计算模块及仿真模型存储和管理单元。算法方面，采用了经典控制理论中的建模方法，并使用离散的方法进行计算求解。

在软件开发过程中，为了尽量模拟小组的开发环境，使用了 git 进行代码的版本管理，分别在两台机器上进行开发和同步。为了管理程序的依赖，以及对代码进行构建，使用了 maven 技术。前端开发采用了 html5，作为网页上 UI 界面的图形绘制技术，采用了 svg，并使用了 snap.svg 的 js 插件对 svg 进行编程操作。前台采用了 json 作为模型的存储格式和前后台数据交换格式。在后台采用 groovy 对模型进行解析并调用仿真算法。对于性能要求较高的仿真计算模块，使用 java 语言进行开发。数据持久化采用了 derby 数据库。完成后的系统运行在 jetty 服务器之上。在计算机上开启 jetty 服务之后，可以在浏览器中使用程序提供的控制系统仿真功能，对水轮机的典型过程进行仿真。同时，也可以基于系统已有的架构，从后台对程序进行拓展，以增强程序支持的仿真功能。

在实际使用过程中，用户可以调取数据库中存储好的模型进行设置并仿真，也可以根据需要设计新的模型进行仿真。此外，为了达到揭示软件开发和构建原理的目的，前端包含一些说明性的页面，对软件开发过程中的一些重要步骤进行说明。

1.3 本文的基本内容

本文使用 java 平台及其相关技术，综合运用各种开源流行的框架及技术，对以水轮机仿真程序为代表的工程领域软件的设计、开发方法进行了研究和探讨，并最终得到了一个基于 java web 技术的、实现了类似 simulink 基础仿真功能的软件系统。本文旨在揭示工程领域的软件架构方式及实现方法，同时对面向对象的仿真程序开

华 中 科技大学硕士学位论文

发方法进行了较深入的探讨。

第一章简述了开发背景及意义，并对本文内容进行了简要的介绍，说明了探讨工业软件开发方法的积极意义。

第二章结合编程的实践，针对 java 开发中的常用约定以及开发思想进行了较深入的说明。本文所开发的软件系统正是基于这些约定和说明才得以顺利的进行。各个正规的开发组织也是依赖于其内部的约定及规范来开发软件，并节约沟通成本。在阅读本文后续内容，尤其是第五章关于代码的部分时，也可以发现这些约定及思想的运用。在软件开发中遵循这些约定及思想，至关重要。

第三章结合业界比较流行的架构方式，论述了本文采取的架构模式。包括设计、开发采用的技术及环境。这些技术并非哪个平台独有的，在相关的软件开发平台上都具有类似的实现。因此，本章具有较为广泛的适用范围。在实际开发过程中，可以根据需要采用其中的一些技术，或者适当修改以运用于其他项目之中。

第四章集中论述了和本文相关的理论问题，包括控制系统仿真的原理及算法，并对业界现有的仿真程序实现进行了对比，总结了各种程序的实现方法和特点。揭示了背后的技术细节。最后，描述了将敏捷开发思想运用到工程领域软件之中的指导原则和方法。

第五章以实例的形式，详细论述了如何使用快速迭代的开发方法迅速搭建可以运行的骨架。以及如何在此基础上逐步通过重构的方式使程序符合面向对象的原则。在开发过程中，本文探讨了如何使用 groovy 开发程序原型，然后使用 java 进行精确建模的方式和方法。

第六章简述了开发用户界面的注意事项，揭示了隐藏在可视化建模技术背后的技术细节，并最终在前文关于仿真原理及算法、软件架构模式、java 平台相关技术等讨论的基础上完成了软件的开发工作。

第七章采用搭建好的软件对水轮机运行过程中的典型过程进行了仿真，并演示了软件的使用方法。然后拓展程序对调节系统进行了参数辨识。

第八章对全文进行了总结，回顾了整个软件开发过程中的关键点，并对未来的工作进行了展望。

2. 开发约定及思想

开发约定及思想是一个开发团队经过长期工作总结出来的最佳实践，它不是由特定平台或语言所施加的限制，而是从管理层面上对软件开发工作作出的一种补足。遵循好的约定及思想^[1]，可以增加项目各个模块的一致性，减少对各部分进行衔接时的沟通成本，并减少团队的学习、培养成本，进而在很大的程度上增加开发效率。团队工作应该确立科学、合理、简洁、高效的约定，当个人的习惯不符合团队的要求时，应该以团队的要求为准。

2.1 开发约定

开发约定包括开发范式带来的客观要求，开发语言及技术的客观限制，优秀实践积累的良好经验，以及其他任何被证明可以提高开发效率的指导原则。

2.1.1 bean 命名规范及命名的一致性

java bean 并不是一种编程模型，也不是一种语言机制，而是一种简单的命名规范。bean 规范对类作出了以下假设：

bean 必须具有一个无参构造方法。如果类没有定义构造方法，java 语言默认规定会由编译器生成一个无参构造方法。如果类定义了带参数的构造方法，则默认不会生成无参构造方法。所以，建议如果将一个类作为 bean，最好显式提供一个无参构造方法。

bean 的 getter 和 setter 方法被当做是 bean 类的属性。如 `getBeanProperty()`、`setBeanProperty()` 方法给定义他们的类添加了一个叫做 `beanProperty` 的属性。

可选的与属性同名的类字段。bean 规范并没有对存储属性的字段提出要求。也就是说，存储属性 `beanProperty` 的类字段可以有任意的命名和类型。但是，为了减少命名不一致，增加代码的可读性，建议字段名和属性名一致。

bean 给 java 类带来了属性，并提供了一些附加功能。首先，bean 可以作为值对象（value object）用来在不同的类之间传递数据；其次，bean 能对内部实现进行屏蔽，对属性的访问提供一个间接层，在间接层中加入额外的限制；最后，遵循 bean

华 中 科技大学硕士学位论文

命名规范的类是很多框架的宠儿，这些框架多使用 java 反射机制对 bean 进行操作，包括 bean 的创建，属性的注入等，这些操作符合约定优于继承的思想。

bean 最基本和最重要的功能还是作为值对象。只要符合了上述关于 bean 的三个约定，就可以作为值对象，但并不要求只作为值对象，而不具备任何的行为。在使用场景中，值对象不仅是数据容器，还可以作为一种接口，相当于对外部用户使用的数据结构进行的一个限定。如方法“void operate(SomeBean someBean);”指明了以类 SomeBean 作为给方法发送信息的一种数据结构。这样做的好处是，如果 bean 使用是 pojo (Plain Old Java Object, 简单 java 对象) 或者属性里包含 jdk 提供的基础数据结构（如集合框架），那么将不会不损害接口的简单性；除此以外，这样的接口更加容易修改，在 bean 中增减属性并不会引发接口的变动，只需要对方法的实现加以修改即可。如果软件项目的各个模块在需求分析、软件设计的阶段，或者极限编程活动当中搭建“可以行走的骨架”阶段，就确立好各部分要使用的 bean，就可以帮助各个层的开发人员更好地理解业务需求。经过实践，遵循上述约定开发的业务模块更容易修改。

2.1.2 类层次分明且接口清晰

类层次的结构设计应该考量好业务和功能。业务始终是依赖于多个功能进行实现的，功能是实现业务的手段，但是功能是不依赖于业务而独立存在的。把功能从业务中独立出来，可以改善程序的结构，增加程序的可读性。接口清晰，是指模块的输入输出要明确，特别的，在 java 平台中可以使用 bean 和方法清晰地刻画功能模块和业务模块的接口。

2.1.3 实现的简单性

尽可能保持实现的简单性，到再也无法保持为止。简单的实现包括设计的简单性、实现的简单性、模块的简单性、关系的简单性、接口的简单性等。保持设计简单，得益于清晰的业务需求，需求越清晰，划分的模块规模就越小、功能也就越明确；保持实现的简单，要求不对业务做多余的假设，采用合乎要求、对已有环境影响较小的技术，消除代码中的重复等；保持模块的简单性，包括功能的单一，接口的清晰等；关系的简单性，要求处理好各部分的关系，保持单向的依赖，避免循环

华 中 科技大学硕士学位论文

依赖等；接口的简单性，要求从外部看去，各模块封装良好，没有模板代码，对用户调用而言是友好的。

2.1.4 不作多余的假设

瀑布开发模型适用于需求明确的场景，但是实际开发过程中很少有需求真正明确的时候，大多数情况下客户都不太清楚自己真正想要的是什么。大多数情况下，作为项目中下层模块的使用者，开发人员一般就是自己的客户，当然不希望面对一个复杂的、难以修改的基础框架，我们需要的是可以简单地使用并且能够应付一定变化的结构。

当某个功能显得“很炫酷”，但又不确定自己真的需要时，尽量做到不作多余的假设，来决定这些功能的去留。

不作多余的假设，就是避免“夸夸其谈的未来可能性”。YAGNI 原则（You Ain't Gonna Need It，你其实根本不需要它）指出，“不可能总是预测未来的需要”，过多的考虑会使系统提前变得复杂、难以维护。一旦假设自己在未来的某个时刻需要某种功能，或者假设自己应该给某个功能留下一个可以插入的接口，那么带来的困境将是实现上的复杂性和软件结构的复杂性。牢记 KISS 原则（Keep It Simple Stupid，使其简单而愚蠢），只在需要的时候才加入相应的功能，十分重要。

例如，当你预料到某个功能可能在将来有新的实现时，会留下一个接口方便未来的扩展，但当前这个接口只有一个实现。这个时候可以果断地抛弃留下接口的念头，因为可能这个接口再也不会会有新的实现了。如果保持了简单性，可以通过重构来加入新的接口。这样既保持了简单，也能拥抱变化。

2.1.5 可读性很重要

命名习惯（Naming Conventions）很重要。在软件设计领域，有一种开发方法叫做测试驱动的面向对象开发^[2]。该方法是测试驱动的开发（TDD，Test-Driven Development）在面向对象领域的延伸。除了要求测试先行以外，该方法还有一系列技术（如 junit，mock 技术）用于探测对象的行为，以确保对象之间的交互符合预期（Expectations）。当采用 TDD 时，写出来的测试就如同自然语言一般，清晰易懂。例如，对于 `testIllegalLoginWithWrongPasswordAndFailed`（测试使用错误的

密码登录而失败)方法,说明了测试的内容及结果,并且,这个方法名就是最好的注释。编程时应借鉴这种命名模式。

由于 c 语言根深蒂固的影响以及相关资料命名的随意性,很多 c 风格的代码都难以读懂。著名的循环变量 i, j 就是从 c 发扬光大的。现在依旧被广泛使用。但是 python 语言对这类命名不感冒,直接从语言层面上消除了使用这种 for 循环的必要^[3]。如代码 2-1 所示,判断列表中每个元素都大于 0, python 推荐的遍历列表方式与 c 拥有不同的风格。

代码 2-1 python 推荐的 for 循环

```
for n in [1, 2, 3]:  
    assert(n > 0)
```

汇编语言是机器的最爱,然而高级语言是面向开发人员的,不好的命名和代码组织方式让作者错失了与他人交流的机会。在 java 这类完全面向对象的语言之中,谨慎地选取包名、类名、字段名、方法名可以有效地表达作者的意图。谨慎地使用注释(包括类注释、字段注释、方法注释),一些需要行内注释的地方,往往都是隐错存在的地方。除了不必要的注释,如同作者自言自语的冗余注释,也应当避免。好的命名基本上可以做到零注释。

“高内聚,低耦合”的设计,是可读性的前提,如果类的设计不好,各个模块之间耦合严重,层次不清晰,就意味着可读性会受到影响。

提高可读性的最佳实践是,不放过任何一个可以解释程序做了什么的机会。如果对于局部变量的命名,也不轻易的使用 temp、ai(a1, a2 ...)、i、j 之类没有含义的命名风格,那么代码的维护者(很可能就是你自己)在六个月后将从中受益。关于注释,一个好的实践是,书写良好的类注释,说明类使用的上下文,依赖,采用的技术等具体事项;对公开接口方法编写简单的注释;显而易见的方法不要注释;尽量避免行内注释。

2.1.6 不断重构以改善软件的结构

重构,是对软件内部结构的一种调整,目的是在不改变软件可观察行为的前提下,调整其结构。文献^[4]提出了一系列重构的手法。重构是极限编程下的一员“猛

将”，也是所有开发过程都离不开的一环。

在重构的世界里，设计模式不再是一种手段，而是重构的结果^[5]。在编程的过程当中，由于需求变化、设计失误、第三方包变化等原因，代码面临着结构失控的风险。新功能的加入，接口的改变也会影响程序的可读性和可拓展性。重构可以在关键的时刻挽救逐渐腐烂的代码结构，并让代码重新满足最佳实践——也就是设计模式——的要求。

失控的代码往往都存在着“坏味道”。重构就是用来消除代码中的“坏味道”。如重复代码、过长函数、过大的类、发散式变化、霰弹式修改、冗赘类、夸夸其谈的未来性、令人迷惑的暂时字段、过渡耦合的消息链等等，都是代码的“坏味道”。“坏味道”是程序存在设计失误、编码失误的信号。

重构不是高端的理论，而是程序员每天都在做的事情。重构的手法大多由一系列小的操作完成，如：改变方法的签名，移动方法到其他类，抽取出新的类，封装字段等等，看似平凡的操作，都是强而有力的重构手法。

2.2 编程思想

2.2.1 模块化的思想

模块是组织代码的良好方式。这里的模块不仅仅是指代码块，在 java 中，项目、包、类、方法都可以作为模块。如，spring 项目由一个个子项目模块组成。模块化是面向对象的基础。接受模块化思想，意味着要屏蔽自然人根深蒂固的过程化的思想。组织代码功能模块的时候，要更多地考虑“这个做什么，那个做什么”，多过考虑“要先这么做，再那么做”。当然，在缺乏经验的时候不可能仅凭观察就将一个模块细化，分解为若干子模块，但是拥有重构的手法，使我们可以事后再进行“补救”，最终符合模块化的要求。模块化的代码可读、好维护、易重用。

2.2.2 面向对象的思想

从形式上看，面向对象要求数据和操作数据的方法一起进行组织，这符合客观世界的特点和人类的思考方式。从客观实际中有针对性地抽象出我们关心的部分，进行组织和刻画，就是面向对象。遵循面向对象的开发原则，很容易构建出高内聚、

低耦合的程序^[6]。

2.2.3 函数式编程的思想

函数式编程^[7]要求我们把函数当做一个加工厂，而数据经过函数的映射不断地变换形式，最终得到我们需要的结果。例如，在 java 语言中，有 bean 类 A、B、C 及其对象 a、b、c，那么链式调用：

```
a.toB().toC();
```

最终将 a 转换成了 c，就是一种函数式编程。函数式编程要满足以下两个要求：

- (1) 函数是无副作用的。也就是说，函数不会改变所处理的数据的内部状态。
- (2) 数据通常设计为不可变的。摒弃数据的可写性，虽然付出了复制数据的工作量和消耗更多内存的代价，但得到的好处是：不可变的数据是无状态的，数据高度一致。一经创立就不可改变的数据，比起在一处创立被多处改变的数据更好维护，它免除了跟踪对象状态的需要。

和函数式编程相对应的是命令式编程。和上面进行相同的假设，代码：

```
a.f1(b);  
a.f2(c);
```

就是一种命令式的风格。该风格符合面向对象的处理方式，a 分别依赖于 b 和 c 来完成功能。实际使用要根据需要采用上面两种风格。在 java 8 以前 java 语言并不支持函数式编程风格，但是，java 8 通过引入 lambda 表达式的语法糖和函数接口的特性，完美支持了强类型、静态语言环境下的函数式编程。此外，jdk 8 中还提供了相应的函数式 api，减少了书写模板代码的工作量。

2.2.4 约定优于配置的思想

约定是指按照既定的习惯做法来做事，配置是提供配置的接口，在使用前设值进行配置。约定优于配置（convention-over-configuration）是指，更多地提前采用经受了实践考验的默认配置，从而减少甚至省略每次使用时的配置工作量。

在软件开发中，可以提前约定的地方包括项目架构、编程接口、数据结构等。在基于 maven 构建的项目中，默认的配置约定，源代码存在于目录 src/java/main 中，测试代码位于 src/test/main 中，并且当这两个结构保持一致时，编译后的目录结构

也保持一致，这样路径一致的类属于同一个包命名空间，使得测试代码能以包可见性访问源代码，同时可以将源代码和测试代码分开组织，方便代码的管理。

对于 javascript、python 等动态、弱类型的语言，约定的使用更加方便。当 javascript 对象具有相同的方法签名或者同名的变量时，他们就遵循了相同的约定，相当于实现了同一个接口，由于弱类型的原因，可以使用相同的方式调用。这相当于 java 中的多态。

当然，约定隐藏了一些设计时的预定义项，在程序中引入“约定优于配置”的代码时，要充分写好文档注释，并提供测试代码，防止维护时摸不着头脑。

2.2.5 封装与暴露接口

封装是指隐藏具体的实现细节，向用户暴露一个清晰简洁的外部接口。spring jdbc template 技术就是一种对模板代码的封装。封装简化了调用，增强了程序的可读性，改善了代码结构。

在设计类的时候，应当考虑尽可能将大多数信息封装起来，只暴露必要的接口，供用户调用。这样，被封装的部分就保留了修改的自由。

2.2.6 数据视图

同一组数据在不同的层次有不同的表现形式，这一组表现形式就是数据的视图。比如，用于存储的视图、业务计算的视图和界面的视图。在工程领域，用于描述客观世界的模型数据是核心的视图，其他诸如存储方式和表现形式的视图都要依据这个视图来确立。数据视图同时是程序代码内部进行通信的协议所在。

2.2.7 敏捷开发

敏捷首先是一组思想，它强调关注个人与交互，要求迅速得到可以运行的软件，重视与客户的交流，提倡拥抱变化。

敏捷还是一组方法。极限编程就是敏捷方法的一种。它要求快速确立程序的可行走的骨架，不断地得到程序给予的反馈。极限编程适用于工程和研究领域。当需求不明确或者经常变动时，使用极限编程，为客户提供原型，从而验证和调整需求，使软件开发工作符合实际情况。

3. 开发环境搭建

从软件项目的开发周期来看，需要处理好诸如项目构建、版本控制、软件设计与开发、软件测试、最终产品的部署与发布等工作。过去的十五年间，java 平台下涌现了大量优秀的开源框架^[8-10]，在项目中合理使用这些框架能大大提高开发效率，使我们可以专注于业务开发，而将其他细节工作托管给框架来完成。

结合工程运用和软件开发实践，本文基于传统的 web 程序三层架构模式，充分整合当前流行的开源框架和技术，大大简化了软件开发的过程，加强了开发的可控性。相关框架与技术如表 3-1 所示：

表 3-1 demo 项目采用的相关框架及技术

适用场景	框架/技术	简述	说明
版本控制	git、github	分布式的代码版本控制软件，适用于同步存在于不同地区的计算机上的软件各个版本。	git 优于 svn 等集成管理工具
	git shell	git 命令行工具	
构建及依赖管理	maven	遵循约定优于配置的思想，为软件约定了一套实用的结构，便于从不同的“maven 仓库”中导出合适的依赖，	可以使用 gradle 代替
前端	jQuery	dom 操作框架	简化 dom 操作及事件处理
	lodash	给 javascript 提供函数式风格的 api	javascript 语言框架
	jQuery easyui	前端 ui 框架	易用性较强
后端	spring	java 语言框架	
	spring jdbc	基于 jdbc 技术提供数据库访问的模板 api	
	derby	使用 java 开发的数据库，支持以内嵌或服务器方式运行	derby 足够小巧并且 api 齐全
前后端数据通讯方式	ajax	异步 javascript	
	json	javascript 对象记法	高效的数据传输格式
	groovy json	groovy json 处理 api	

华 中 科技大学硕士学位论文

	spring+jackson	spring 内置 json 处理 api	
测试	junit	单元测试框架	
	matlab/simulink	用于验证仿真计算的正确性	
开发平台	jdk8	java se 8 sdk	提供了静态语言下的函数式风格编程
	groovy 2.7	基于 jvm 的动态类型语言	快速原型开发语言及脚本语言
	windows 7 32bit 操作系统+intel i3 处理器	计算机 1 配置	用于 demo 项目开发的计算机 1
	windows 10 64bit 操作系统+intel i5 处理器	计算机 2 配置	用于 demo 项目开发的计算机 2
	eclipse 4.6(Neon.1)	eclipse 集成开发环境	需要 java 8 及以上版本
	eclipse plugin groovy	groovy 的 eclipse 插件	负责 groovy 代码的编译及配置
	eclipse plugin maven	maven 的 eclipse 插件	集成 maven 功能
	eclipse git plugin	git 的 eclipse 插件	集成 git 功能
	eclipse plugin spring	spring 的 eclipse 插件	为 spring 开发提供 gui 支持
服务器	jetty	jetty servlet 服务器	使用 java 开发的小巧服务器

本文基于上述架构，对 java 8 平台下的水轮机仿真程序所涉及的技术进行了较深入的研究；基于 java web 技术，完成了的“水轮机调速系统仿真”样例（demo）项目的研究、设计及开发，以下简称“**demo 项目**”。

前文提到过，上述架构基于经典的三层 web 应用架构提出：展示层（UI Layer）、业务层（Business Logical Layer）、持久层（Persistence Layer）。没有考虑要求分布式、大吞吐量的场合。但是，遵循面向对象基本原则所开发出来的程序，更加容易利用面向服务的架构、分布式、缓存等技术，拓展至更广泛的适用范围之中。限于知识及实践水平，对上述非功能需求，本文不作过多的讨论。文献^[11]对大

型程序的架构进行了深入的探讨

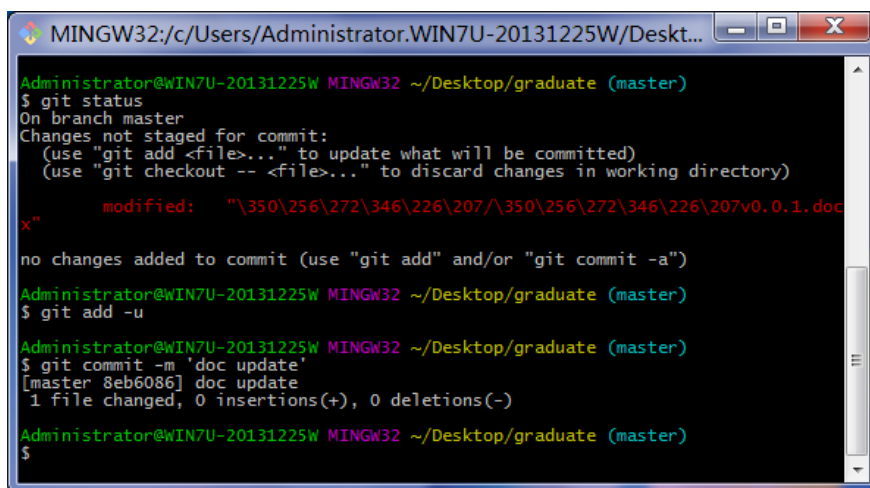
此外，文中讨论的背景虽然是 java 平台和工程领域，但是同样的问题存在于各种开发背景之中，因此本文对软件开发的讨论具有一般性。同时，即使技术在不断更新换代，也是以既有技术为基础逐渐演进而来的。研究这些技术的特点和使用方式，具有积极意义。接下来，有必要对表 3-1 中的选择作出考量。

3.1 版本控制工具——git/github

在软件项目中，如果不进行版本控制，时刻保持项目的各个版本处于一致的状态之中，就会同时存在数个版本，引发“版本灾”。git 是一个分布式的版本控制工具，它从文件的二进制编码上对同一文件的两个不同版本进行比对，并对不一致的地方进行合并或者冲突提醒，从而促使不同的版本重新回到统一的状态之中来。

除了版本同步，git 还提供了分支（branch）的概念用以从项目的一个版本出发开发适用于不同环境的软件分支；提供了里程碑（tag）的概念以标记项目开发过程中的重要版本。

使用 git 的原因除了其易用性以外，还包括 github (<https://github.com>) 网站提供的免费服务，通过这些服务，可以将项目提交到远端进行存储及同步。图 3-1 演示了如何通过 git 命令行工具对项目更改进行追踪与提交。



```
Administrator@WIN7U-20131225W MINGW32 ~/Desktop/graduate (master)
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   "\350\256\272\346\226\207\350\256\272\346\226\207v0.0.1.doc
x"

no changes added to commit (use "git add" and/or "git commit -a")
Administrator@WIN7U-20131225W MINGW32 ~/Desktop/graduate (master)
$ git add -u
Administrator@WIN7U-20131225W MINGW32 ~/Desktop/graduate (master)
$ git commit -m 'doc update'
[master 8eb6086] doc update
1 file changed, 0 insertions(+), 0 deletions(-)
Administrator@WIN7U-20131225W MINGW32 ~/Desktop/graduate (master)
$
```

图 3-1 通过 git bash 命令行提交更改

需要强调的是，git 不仅能对文本数据提供同步，二进制文件（如 pdf，doc）的同步也能从中获益，比如：软件开发的文档、参考文献等非纯文本文件，都可以使

用 git 进行管理，这类似于时下流行的云存储技术。此外，git 项目每次提交，都要求给出当次提交的有意义的描述信息，便于项目开发成员进行交流。git 的这些特性使得 github 被称为“程序员的微信”。eclipse 4.6 版本内置集成了对 git 的支持，导入项目时如果检测到 git 版本库（.git 文件夹下的所有内容），可以提供可视化工具，简化 git 的使用。

文献^[12]详细介绍了 git 的原理以及如何使用 git shell 命令行工具及 git gui 对个人及团队项目进行版本控制。

demo 项目使用 git 进行版本控制，并将相关资料及代码提交至 github 网站，访问 demo 项目的 github url 为：

<https://github.com/yy550956983/graduate.git>

可以在该网址检出项目、查看更新历史等。项目开发过程中主要使用 github gui、git gui 以及 git 的 eclipse 插件对位于两个不同机器中的项目版本进行同步。如图 3-2 所示，在“计算机 1”中开发并提交至本地版本库，然后使用 push 操作将更改推送至远端服务器进行同步，再到“计算机 2”中执行 pull 操作拉出服务器中的版本并与本地版本进行合并。使用这种方式，既对项目的各个版本进行了备份，也方便在多个终端推进项目进度。

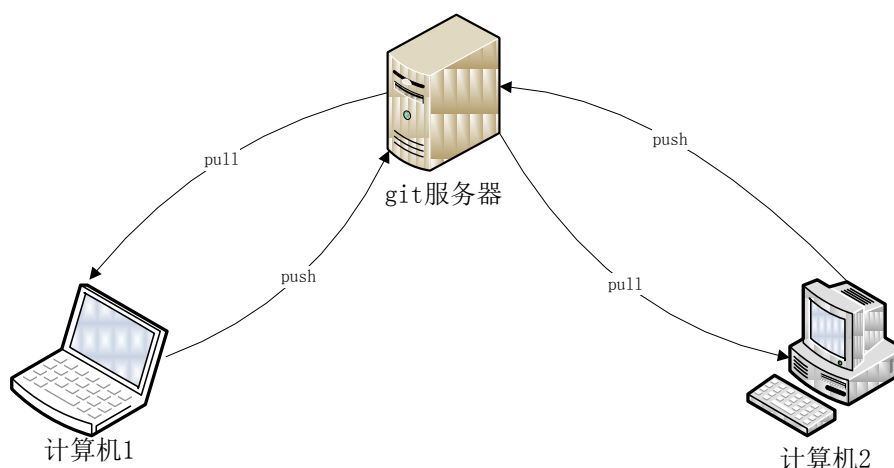


图 3-2 使用 git 进行项目远程同步操作

3.2 项目构建工具——maven

项目构建工具一般约定了项目的结构（见 2.2.4 节），提供了对第三方依赖的

华 中 科技大学硕士学位论文

管理，并对代码的编译、测试，项目的发布等特定生命周期的任务提供支持和管理，大大简化了项目的构建工作。常用的构建工具包括 maven 和 gradle。

典型的 maven web 项目约定结构如图 3-3 所示。通过 src/main/java 和 src/main/test 文件夹，可以轻易地将源码和测试代码分置在不同的文件夹中，分开管理。如果其下的包结构相同，那么包下的类的命名空间也相同，这样源类和测试类同属一个包，测试类就能访问源类的包可见域，从而既保护了封装，又避免了源代码和测试放在同一个文件夹下导致的管理不便。此外，在该目录结构中，webapp 文件夹用于存放前端程序，target 用于存放程序的生成内容，如编译好的.class 文件、jar 包、war 包、文档等等。

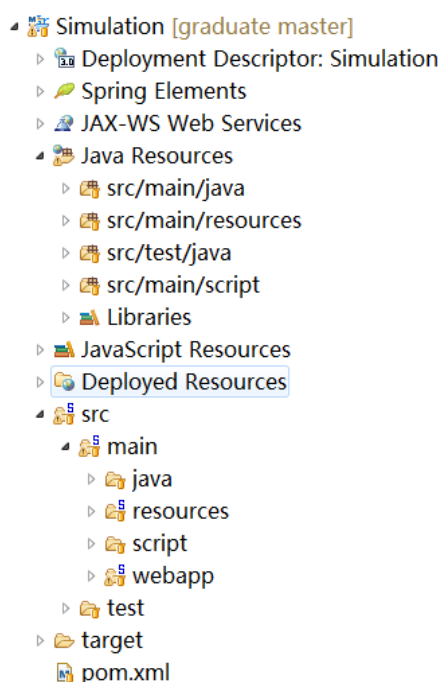


图 3-3 典型的 maven 项目结构

maven 通过 pom.xml 文件进行配置。项目坐标唯一定位了一个项目，通过坐标添加对某项目的依赖。maven 将项目开发分为了很多个详细的阶段（如构建、开发、测试、发布等），maven 插件以一种可插拔的方式，通过作用于不同的阶段来完成对 maven 功能的拓展。与第三方库相同，插件也通过坐标来进行定位。如代码 3-1 所示，通过 groupId、artifactId 和 version 三个元素组成的坐标，在 pom.xml 的 dependencies 标记下添加对 apache math3 项目（提供高等数学的算法包）的依赖：

华 中 科技大学硕士学位论文

代码 3-1 maven 添加对 math 的依赖

```
<dependency>
  <groupId>org.apache.commons</groupId>
  <artifactId>commons-math3</artifactId>
  <version>3.5</version>
</dependency>
```

通过使用 maven 的 eclipse 插件，在 pom.xml 发生变更时，插件可以自动检测到依赖的变化，并在底层调用 maven 调整程序的依赖项。流程如图 3-4 所示：

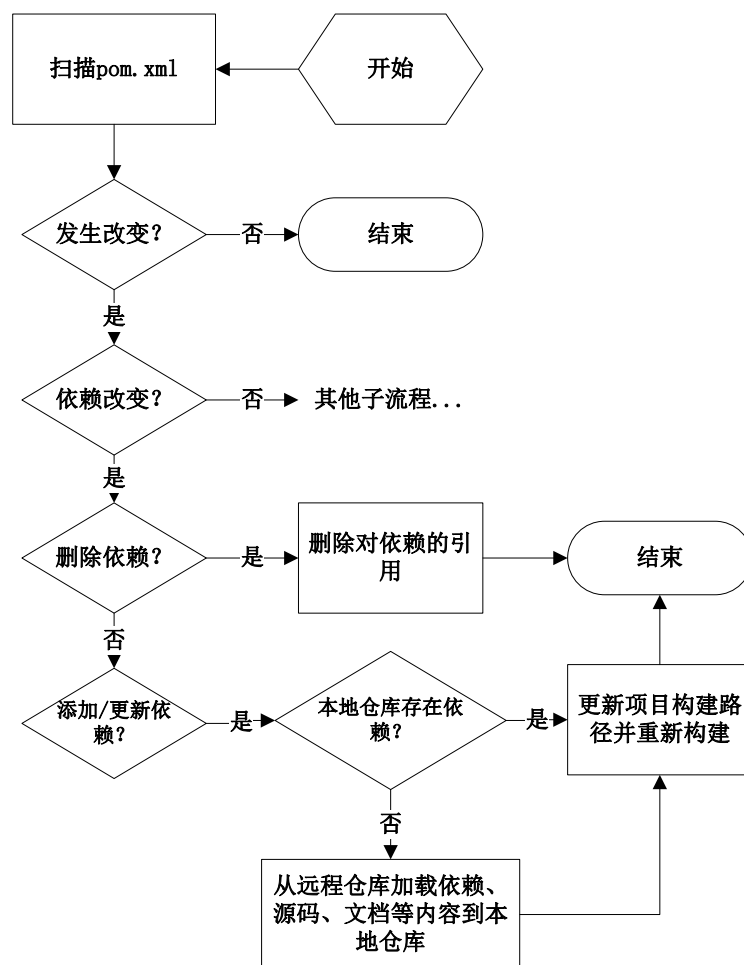


图 3-4 maven 依赖管理流程图

demo 项目还依赖于两个 maven 插件：compiler 插件和 jetty 插件。compiler 插件指定了 java 代码的编译器，jetty 插件在开发过程中内嵌对 jetty 容器的支持。详细配置参数表可以在可以在插件相关网站上查询。使用搜索引擎搜索 maven xxx 或 maven xxx plugin 可以查询到第三方依赖项目的坐标以及配置方式。

华 中 科技大学硕士学位论文

代码 3-2 为 maven 设置了 compiler 插件，覆盖了 maven 默认使用的 jdk 1.6 编译器。代码 3-3 为 maven 添加了 jetty 插件支持，并添加了对 jetty 进行配置的描述符文件 webdefault.xml、前端资源路径 src/main/webapp 以及项目的类路径 target/classes。有了上述的配置，当使用插件命令 jetty:run 开启服务器的时候，可以确保类得到正确的编译、项目得到正确的构建，前端的页面也能够正确地被服务器程序寻找到。

代码 3-2 compiler 插件配置：

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-compiler-plugin</artifactId>
  <version>3.6.0</version>
  <configuration>
    <source>1.8</source>
    <target>1.8</target>
  </configuration>
</plugin>
```

代码 3-3 jetty 插件配置：

```
<plugin>
  <groupId>org.eclipse.jetty</groupId>
  <artifactId>jetty-maven-plugin</artifactId>
  <version>9.2.10.v20150310</version>
  <configuration>
    <webApp>
      <contextPath>/Simulation</contextPath>
      <defaultsDescriptor>
        src/main/resources/webdefault.xml
      </defaultsDescriptor>
    </webApp>
    <webAppSourceDirectory>src/main/webapp
  </webAppSourceDirectory>

  <classesDirectory>target/classes</classesDirectory>
</configuration>
</plugin>
```

gradle 构建工具集成了 maven、ant 等项目的功能，并提供了脚本化的配置方式。gradle 构建脚本使用了 DSL（领域特化语言）的风格，简化了配置文件的编写方式。例如，常见的 gradle 配置文件 build.gradle 如代码 3-4 所示。

代码 3-4 使用 gradle 作为构建工具

```
apply plugin: 'java' // 插件配置
apply plugin: 'jetty'
```



```
apply plugin: 'eclipse-wtp'
sourceCompatibility = 1.8 // jdk 配置
targetCompatibility = 1.8

repositories{ // 仓库配置
    mavenLocal() // 指定 maven 本地仓库
    mavenCentral() // 指定 maven 中央仓库
}
dependencies{ // 依赖配置
    compile 'org.apache.commons: commons-math3: 3.5'
}
jettyRun{ contextPath = 'Simulation'}// 服务器配置
eclipse{
    wtp{
        component{contextPath = 'Simulation'}
    }
}
```

可以看出，以上的配置比 maven 基于 xml 的配置方式更加的友好。由于 gradle 是 groovy 平台下的项目，上述配置文件符合 groovy 的语法，简洁明了。gradle 是 maven 的可选替代构建工具，并且有相关的 gradle eclipse 插件可以使用。

demo 项目使用 maven 作为构建工具，由于 eclipse 4.6 版本内置了对 maven 的支持，使用 maven 插件不需要额外的配置。文献^[13]对 maven 的原理及使用作了深入细致的剖析。

3.3 后端架构

3.3.1 新一代平台——java 8

java 8 特指 Java SE 8 版本。是 Oracle 公司收购 sun 以后推出的首个 java 重要版本。其引入的最重要的语言特性为 lambda 表达式，通过函数式接口和 lambda 表达式语法糖，创造性地为静态类型语言添加了对函数式编程的支持。此外，jdk 8 还基于这些新特性提供了一组新的支持函数式编程的流式 api。

如代码 3-5 所示，判断列表中所有的元素都大于 0。其中，lambda 表达式 “n -> n>0” 实现了内置的 Predicate<T>接口（这里是 Predicate<Integer>），由编译器在编译时将这一语法糖转化为该接口的匿名内部类的实现。语法和之前版本的 for 循环和迭代器相比，有了进一步的简化。在本文的 7.3.1 小节中，实现了的 GSA 算法，流式 api 和 lambda 表达式的使用让代码变得更加紧凑和易读。

代码 3-5 java8 的流式 api 及 lambda 表达式支持

```
List<Integer> numbers=Arrays.asList(1,2,3);  
numbers.stream().allMatch(n->n>0); // true
```

java 程序最终是由（java virtual machine, java 虚拟机）解释执行的。可能熟悉 c/c++ 开发环境的人会质疑 java 语言的性能问题，但是，随着 jvm^[14] 中 JIT（即时编译）等最新编译技术的使用，java 的执行速度已经可以和本地代码相媲美。本文 5.2 节和 7.3.2 节的试验足以说明，java 能满足绝大多数性能上的要求。

3.3.2 spring mvc 和 spring jdbc

关于 spring 框架的原理与使用业界已经有较为深入的了解。文献^[9]介绍了 spring 框架的原理与使用。spring mvc 是基于 spring 的 web 框架，采用了命令模式对前端的请求进行拦截、解析、处理及结果渲染。其基本原理如图 3-5 所示。

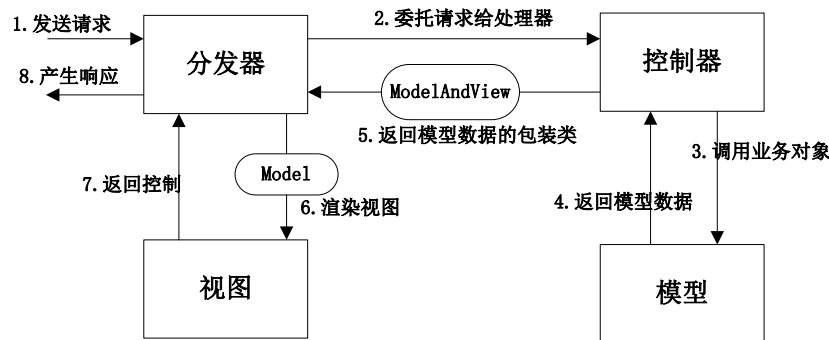


图 3-5 spring mvc 的原理

spring 框架内置了对 json 和 java bean 互相转换的支持。为了利用这一特性，需要在 pom.xml 中添加对 jackson 包的依赖，并在 spring web 的配置文件 dispatcher-servlet.xml 中添加如下标记：

```
<context:component-scan base-package="hust.hx.action" />  
<mvc:annotation-driven />
```

其中，第一句指明了控制器 bean 的搜索包路径，第二句表明了通过注解进行配置。

假设有如图 3-6 所示的类及如代码 3-6 所示的控制器实现。

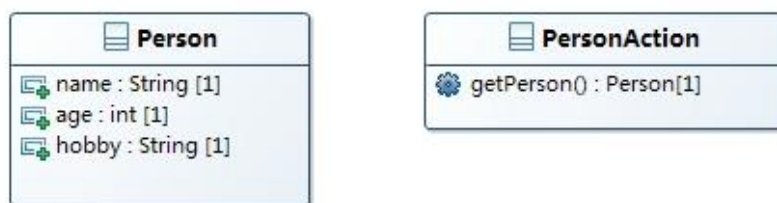


图 3-6 简单的 spring mvc 控制器

代码 3-6 spring mvc 的控制器

```
@Controller
public class PersonAction {
    @ResponseBody // json 是默认返回类型
    @GetMapping(path = "/get-person")
    public Person getPerson() {
        return new Person("hx", 25, "java");
    }
}
```

当完成上述配置，开启服务器并采用 url:

<http://localhost:8080/Simulation/get-person.action>

访问该控制器时，spring 框架会自动对结果进行转换并得到与 Person 类对应的 json，如下所示：

```
{"name": "hx", "age": 25, "hobby": "java"}
```

通过使用上述特性，demo 项目在前端页面和控制器之间传输数据，并且避免了使用语法复杂和对调试不友好的页面脚本技术（如 jsp），使代码具有简洁性和一致性。

eclipse 以插件的形式对 spring 提供了支持，在 eclipse ide 中选择 help -> install new software-> add，键入以下 url：

<http://dist.springsource.com/release/TOOLS/update/e4.6/>

可以安装适用于 eclipse 4.6 版本的 spring 插件。

3.4 前端架构——jQuery 与 lodash

jQuery 是目前最为流行的前端 javascript 框架。个人认为，可以把 jQuery 看作是处理 DOM（文档对象模型）的 DSL，它封装并屏蔽了底层浏览器的差异，提供了对于 DOM 的 CRUD（Create Retrieve Update Delete）操作接口、事件处理机制、

ajax 调用、css 操作、以及一些简化语言复杂性的 api，结构清晰，使用方便。文献^[15]对 jQuery 作了详细的阐述。

lodash 是纯粹的 javascript 语言框架，提供了一组易用的 api，用于完成诸如语言类型检测，数组、集合操作，字符串操作等常见任务，并赋予 javascript 函数式编程的风格。可以在 <https://lodash.com> 上查看有关 lodash 的信息。使用 lodash 完成“集合中的所有数都大于 0”的代码 3-7：

代码 3-7 lodash 提供的函数式编程支持

```
var numbers = [1, 2, 3];
_.every(numbers, function(n) {return n > 0;}); // true
```

上述代码同样比较简洁，当然，如果 javascript 引入了 lambda 表达式，还能进一步得到简化。简洁的代码通常就意味着更高的可读性。

3.5 前后端通讯格式——json

json（JavaScript Object Notation，javascript 对象注记）是 javascript 语言的对象字面量表示法，可以表达具有一定层次结构的数据类型。由于其动态弱类型的特性，javascript 语言有更加简洁的字面量表示方法，表 3-2 的对比表明，json 在字面量的表示上有更大的灵活性，且受到语言类型系统的影响最小。

表 3-2 javascript 和 java、groovy 字面量的对应关系

javascript 类型	javascript 字面量	java 字面量		groovy 字面量	
		字面量	类型	字面量	类型
Number	1.0	1	int	1	Integer
		1.0	double	1.0	BigDecimal
String	"str"	"str"	String	'str'	String
Boolean	true, false				
Array	[1,2]	{1,2}	int[]	[1,2]	ArrayList
Object	{a:1,b:2}	无		[a:1,b:2]	LinkedHashMap

在 3.3.2 小节中已经看到，后端生成的遵循 java bean 规范的 pojo 对象，可以通过 spring 的支持和 json 数据进行互相转换。文献^[16, 17]说明了 json 数据在网络传输时

华 中 科技大学硕士学位论文

的性能表现，json 在大多数情况下可以满足前后端的数据交换需求。

groovy 语言从 api 的层面对 json 提供了支持。通过使用工具类 JsonSlurper 和 JsonOutput，可以很容易地在 json 和 groovy 对象字面量之间进行互相转换。代码 3-8 完成了 json 字符串和 groovy 的 LinkedHashMap 之间的转换。

代码 3-8 JsonTest.groovy

```
// 导入相应的工具类
import groovy.json.JsonOutput
import groovy.json.JsonSlurper

// json 转换为 groovy 对象
def slurper = new JsonSlurper()
def obj = slurper.parseText """
{"name":"hx","age":24,"hobby":"java"}"""

assert obj.name == 'hx'
assert obj.age == 24
assert obj.hobby == 'java'

// groovy 对象转换为 json
def person = [name:'hx',age:24,hobby:'java']
def out = new JsonOutput()
def personJson = out.toJson(person)
assert personJson ==
'{"name":"hx","age":24,"hobby":"java"}'
```

3.6 derby 数据库和 jetty 服务器

derby 数据库^[18]是由 java 语言实现的基于文件系统进行组织的数据库，其特点是非常小巧，既可以作为数据库单独运行，也可以内嵌于程序中运行。derby 数据库符合 sql-92 和 jdbc 3.0 标准。最为重要的是，derby 和绝大多数托管于 apache 基金会的 java 项目一样，具有十分完善且详细的文档支持，且文档会随同 derby 的发布版本一同发布，derby 的下载地址为：<http://db.apache.org/derby/>。

demo 项目使用 derby 数据库存储程序相关数据。在项目开发过程中，计算机 1 属于便携式笔记本，由于配置方面的原因无法在上面部署重量级的商业数据库，然而得益于 derby 的小巧，可以克服上述不足，满足研究和软件开发中对于数据持久化的需求。

此外，eclipse 的 Database Development 透视图（perspective）提供了一个便捷的

华 中 科技大学硕士学位论文

数据库连接与交互 gui，可以书写标准 sql 语句对数据库进行 crud 操作。

选择 jetty 作为服务器是基于和 derby 一样的考量。jetty 是一个 servlet 容器，同样是基于纯 java 语言实现的，小巧易用，可以嵌入到程序中或单独作为服务器运行。与生产环境下使用的服务器（比如 tomcat）相比，jetty 显得更加轻量，灵活。同时，提供的拓展功能更少，不过，对于 demo 项目的研究性质与大多数非生产环境而言，功能已经足够使用了。

在大型项目开发当中，开发和部署选择不同的配置是常见的策略。像 derby 和 jetty 这样小巧的程序具有更加平缓的学习曲线，适合程序员在开发过程中单独使用，更加复杂的持久化需求可以交由专业人士进行，在项目后期再迁移到适用于生产环境的重量级工具进行测试、集成和部署。

maven 使用插件 jetty-maven-plugin 对 jetty 提供了支持，使用 maven 命令:mvn jetty-run 可以开启 jetty 内嵌服务器，默认监听 8080 端口上的请求。另外，可以配置 jetty 扫描资源文件的时间间隔，以迅速地将文件的变更反映到程序当中，和同样条件下运行的 tomcat 相比，jetty 对改变更加灵敏。tomcat 在变化频繁的时候有可能停止继续响应，需要重启服务器才能正确运行。

3.7 脚本与动态语言支持——groovy 和 python

groovy 是 jvm 上的一种强大的、可选类型的、动态语言。groovy 旨在像 python 和 c/c++之间的关系那样成为 java 平台下的“粘合剂”。同时，groovy 是一门独立的、完整的、成熟的语言。groovy 代码可以作为脚本运行，也可以编译运行。groovy 代码和 java 代码是二进制兼容的，也就是说，二者编译而来的.class 文件格式是一致的。正因为如此，二者相互调用无需额外的配置，只要能在类路径下寻找到彼此即可——groovy 代码可以和 java 代码互相调用，无缝集成。

从语言层面来讲，groovy 提供了很多很强大的特性。早在 java 8 以前，groovy 已经通过闭包（closure）特性来支持函数式风格的编程；groovy 特殊的函数调用语法和闭包的代理特性使其天然可以支持 DSL（Domain-Specific Language，领域特定语言）的编写；元编程（meta-programming）的特性支持 groovy 动态地对类进行拓

华 中 科技大学硕士学位论文

展；特质（trait）则提供了“胖接口”和混入（mixin）的特性。gradle（构建工具，<https://gradle.org/>）和 grails（一站式的 web 应用解决方案 <http://www.grails.org/>）是应用广泛的两个 groovy 平台下的项目。

eclipse 下通过插件对 groovy 提供了支持。在 eclipse ide 中选择 help > install new software > add，键入以下 url：

<http://dist.springsource.org/snapshot/GRECLIPSE/e4.6/>

可以安装适用于 eclipse 4.6 版本的 groovy 插件。通过新建 groovy class 可以创建 groovy 类，通过快捷键 shift+alt+x, g 可以以脚本形式运行 groovy 程序。

在实际使用中，建议下载并安装 groovy，在满足 groovy 要求的 java 版本正确安装的前提下，只需要解压 zip 压缩文件到特定文件夹（即安装目录），然后指定环境变量 GROOVY_HOME 并指向 groovy 安装目录，在 path 变量中添加 %GROOVY_HOME%\bin，就能够在 cmd 命令行中通过 groovyConsole 命令进入 groovy 自带的控制台环境，实现 repl（read-evaluate-print loop，输入-执行-打印循环）风格的编程，以得到迅速的反馈、加深对 groovy 语言的理解。groovy 控制台如图 3-7 所示，其中使用了 groovy 的函数式编程支持判断列表中的每一项均大于 0。

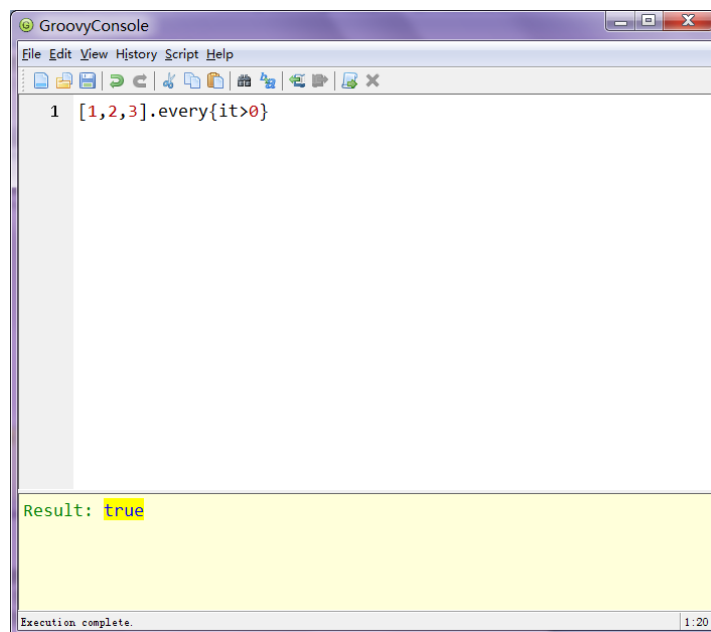


图 3-7 groovy 控制台实现 repl

对于 demo 项目的“仿真”主题而言，对程序的执行速度有较高的要求，然而，

groovy 由于其内部实现机理，执行速度远远不及 java（同样的问题也存在于 python 之中），但是，groovy 语言的动态特性以及对 json 的支持十分适合开发接口及上层驱动程序。在这种情况下，可以将对性能敏感的业务程序使用 java 编写，将对变化敏感的接口程序使用 groovy 编写。从而同时发挥静态类型严谨和动态语言灵活的优势。在本文的 7.3.2 小节有更多关于 java 和 groovy 性能的讨论。

4. 仿真原理及调速系统模型

前两个章节明确了开发工程应用软件所遵循的思想，并结合工程实际运用确立了一种实用性较高的 java web 软件开发架构，接下来需要研究计算机控制系统仿真的原理、算法及其程序实现方式。同时，水轮机调速系统的仿真模型也是题中应有之义。

4.1 控制系统仿真的原理

仿真活动的本质在于通过模型去模拟实际系统的表现。从历史发展来看^[19]，仿真包括物理仿真和数字仿真。物理仿真借助物理模型对实物进行近似，至今仍应用于水轮机设计和制造之中，使用水轮机原型和相似理论去近似模拟实际水轮机的特性。数字仿真特指借助数学模型和数字计算机所进行的仿真。和物理仿真相比，数学模型的建立更加容易，开发周期较短，适应能力强。今天所说的仿真，大多数是指数字仿真。

仿真的步骤一般如图 4-1 所示。从仿真对象来看，仿真主要分为连续系统的仿真和离散系统的仿真。调速系统的仿真属于连续系统的仿真。

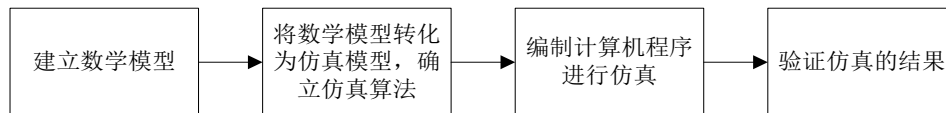


图 4-1 仿真的一般步骤

从仿真的时间标尺 τ 和实际时间标尺 t 的比例关系来看，仿真分为实时仿真（ $t/\tau = 1$ ）和非实时仿真（ $t/\tau \neq 1$ ）。对调速系统的仿真当中，当应用于分析极值、验证调节规律、调节保证计算等对实时性要求不高的场合，可以采用非实时的仿真。实时仿真一般指有实物参与的仿真，这时候，仿真模型必须具有接收实际数据和发出计算数据的接口，还要等待实物的响应才能进行进一步的仿真，这种仿真适用于输入容易获取、仿真对象操作代价高昂的情况。

水轮机调节系统属于动力系统，这种系统的数学模型一般可以由一组微分方程来表征。根据经典的控制理论，描述系统的微分方程组、控制方框图和状态方程之

华中科技大学硕士学位论文

间可以相互转化，但其物理本质是等价的。另外，由于迟滞、饱和等实际因素的存在，水轮机还存在一定的非线性结构。这些都是编制仿真程序时应该考虑到的因素。

对于系统的三种等效模型，由于转态方程在数学描述上的简单性，适合用来进行理论推导；方框图的形式适合用来进行模块化的开发；微分方程的形式适合用来构造算法。

4.1.1 离散相似法

设一线性定常系统的状态方程如下所示：

$$\dot{X} = AX + BU \quad (4-1)$$

$$Y = CX + DU \quad (4-2)$$

其方框图如图 4-2 所示：

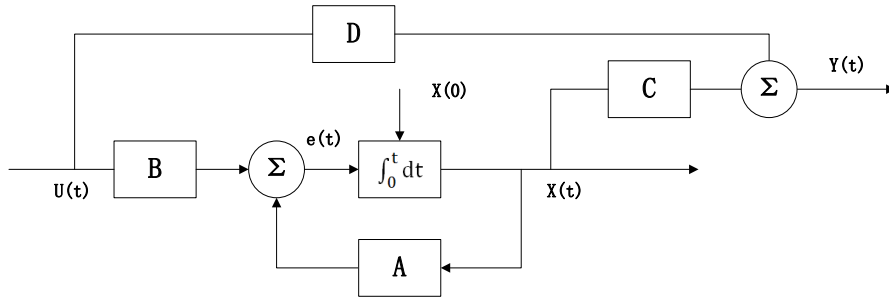


图 4-2 系统方框图

在连续时间下，式(4-1)可表述为：

$$\dot{X}(t) = AX(t) + BU(t) \quad (4-3)$$

式中 A、B 均为常系数矩阵。对上式进行拉普拉斯变换，得到：

$$(sI - A)X(s) = X(0) + BU(s) \quad (4-4)$$

以 $(sI - A)^{-1}$ 左乘上式两端，并作拉普拉斯反变换，可得：

$$X(t) = \Phi(t)X(0) + \int_0^t \Phi(t - \tau)BU(\tau)d\tau \quad (4-5)$$

其中 $\Phi(t)$ 为转移矩阵，且有：

$$\Phi(t) = L^{-1}(sI - A)^{-1} \quad (4-6)$$

同时，我们也可以变换一种思路，使用 e^{-At} 左乘(4-3)式两端，整理后得到：

$$e^{-At}[\dot{X}(t) - AX(t)] = e^{-At}BU(t) \quad (4-7)$$

利用微分计算规则，可以将上式改写为：

华 中 科技大学硕士学位论文

$$\frac{d}{dt}[e^{-At}X(t)] = e^{-At}BU(t) \quad (4-8)$$

上式两端同时积分并整理, 可得:

$$X(t) = e^{At}X(0) + \int_0^t e^{A(t-\tau)}BU(\tau)d\tau \quad (4-9)$$

比较(4-5)式及(4-9)式, 可以得到:

$$\Phi(t) = e^{At} = L^{-1}(sI - A)^{-1} \quad (4-10)$$

由拉普拉斯反变换的公式可知, 上式中矩阵分式可以展开成级数的形式并定义为:

$$e^{At} = I + At + \frac{A^2}{2!}t^2 + \dots \quad (4-11)$$

由上述(4-9)式可求得该线性定常系统的准确解, 当 e^{At} 不易求解时, 可以通过式(4-11)求过渡矩阵的近似解。上式是在连续的条件下得到的, 计算机无法处理连续的情况, 因此要得到适合编制程序进行计算的算法, 需要将原来的模型进行离散化处理, 从而对实际的连续系统加以近似。

对于上述系统, 当 $t=kT$ 时, 带入(4-9)可得:

$$X(kT) = e^{AkT}X(0) + \int_0^{kT} e^{A(kT-\tau)}BU(\tau)d\tau \quad (4-12)$$

当 $t=(k+1)T$ 时, 有:

$$X[(k+1)T] = e^{A[(k+1)T]}X(0) + \int_0^{[(k+1)T]} e^{A[(k+1)T-\tau]}BU(\tau)d\tau \quad (4-13)$$

由上述两式得到系统离散解的递推格式为:

$$X[(k+1)T] = e^{AT}X(kT) + \int_{kT}^{[(k+1)T]} e^{A[(k+1)T-\tau]}BU(\tau)d\tau \quad (4-14)$$

当采用零阶保持器并且不进行补偿时, 有:

$$\sim U(t) = U(kT), \text{ 当 } kT \leq t \leq (k+1)T \quad (4-15)$$

上式带入(4-14)式可得:

$$X[(k+1)T] = e^{AT}X(kT) + \left(\int_0^T e^{A(T-\tau)}d\tau\right)U(kT) \quad (4-16)$$

令上式中 $\Phi(T) = e^{AT}$, $\Phi_m(T) = \int_0^T e^{A(T-\tau)}d\tau$, 可见, 实际计算过程就是要根据系统的状态方程求解这两个系数、并进行迭代的过程。

华 中 科技大学硕士学位论文

4.1.2 数值积分法

离散相似法本质上是对微分进行离散化处理从而近似求解微分方程。从数值积分的思路出发，可以建立起一套新的求解方法。数值积分法本质上属于离散相似法。

重写离散相似法中的系统如下：

$$\dot{X} = AX + BU \quad (4-1)$$

$$Y = CX + DU \quad (4-2)$$

由系统的结构可知：

$$X(t) = X(0) + \int_0^t e(t)dt \quad (4-17)$$

对上述系统进行离散化处理，分别令 $t=kT$ 和 $t=(k+1)T$ ：

$$X(kT) = X(0) + \int_0^{kT} e(t)dt \quad (4-18)$$

$$X[(k+1)T] = X(0) + \int_0^{(k+1)T} e(t)dt \quad (4-19)$$

用(4-19)减去(4-18)式可得：

$$X[(k+1)T] = X(kT) + \int_{kT}^{(k+1)T} e(t)dt \quad (4-20)$$

于是，对积分 $\int_{kT}^{(k+1)T} e(t)dt$ 的不同近似算法就构成了系统不同的近似求解方法。从数值分析的知识可知，主要的近似方法有欧拉法、梯形法、龙格-库塔法、Adams法等。

当采用欧拉法时，使用零阶保持器，对偏差的离散处理为

$$e_h(t) = e(kT), \quad kT \leq t \leq (k+1)T \quad (4-21)$$

上式带入(4-20)可得：

$$X[(k+1)T] = X(kT) + Te(kT) \quad (4-22)$$

简记为：

$$X(k+1) = X(k) + Te(k) \quad (4-23)$$

由 $e(k)$ 近似 \dot{X} 将上式带入(4-1)和(4-2)，可得：

$$X(k+1) = (I + AT)X(k) + BU(k) \quad (4-24)$$

$$Y(k+1) = CX(k+1) + DU(k+1) \quad (4-25)$$

上文从理论上考察了控制系统仿真的方法，然而，使用计算机编制程序进行计

算还需要将上述理论转化为对机器友好的仿真算法。

4.2 控制系统的仿真算法

在计算过程中，可以选取几个基本的模块作为标准，然后将其他的模块转化为这种基本模块，简化问题的求解。例如，对具有传递函数

$$G_o(s) = \frac{1}{s(1+s)}$$

的系统，可以拆分为一个积分环节和一个惯性环节相连的等效系统：

$$G_o(s) = G_1(s)G_2(s) = \frac{1}{s} \times \frac{1}{1+s}$$

再分别对上述两个环节进行离散化处理。

在 demo 项目中，为了实现上的简便，使用离散相似法^[19]进行仿真计算。在这里，我们选取如下的两个模块作为基本模块：

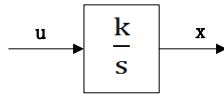


图 4-3 积分环节

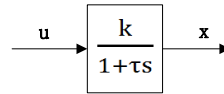


图 4-4 惯性环节

对于积分环节，其微分方程表述如下：

$$\dot{x} = ku$$

由公式(4-9)和(4-16)可以得到上述微分方程离散化后的系数：

$$\begin{cases} \Phi(t) = 1 \\ \Phi(T) = 1 \\ \Phi_m(T) = kT \end{cases}$$

可知，使用零阶保持器进行离散化，积分环节的差分方程为：

$$x(k+1) = x(k) + kTu(k) \quad (4-26)$$

同理可得，对于惯性环节，其微分方程和离散化的差分方程分别为：

$$\dot{x} = -\frac{1}{\tau}x + \frac{k}{\tau}u$$

$$x(k+1) = e^{-\frac{T}{\tau}}x(k) + k(1 - e^{-\frac{T}{\tau}})u(k) \quad (4-27)$$

应当注意到，当采取零阶保持器进行离散化时，所得到的差分方程是“无后效性”的，也就是说， $k+1$ 时刻的输出仅和 k 时刻的状态和输入有关，这为编程带来

了便利，但是必须指出，系统模块越多，需要加入的保持器就越多，计算的精度也就越低。

利用上述模块化、系统等效转换的思路，可以得到基本的仿真流程如图 4-5 所示：

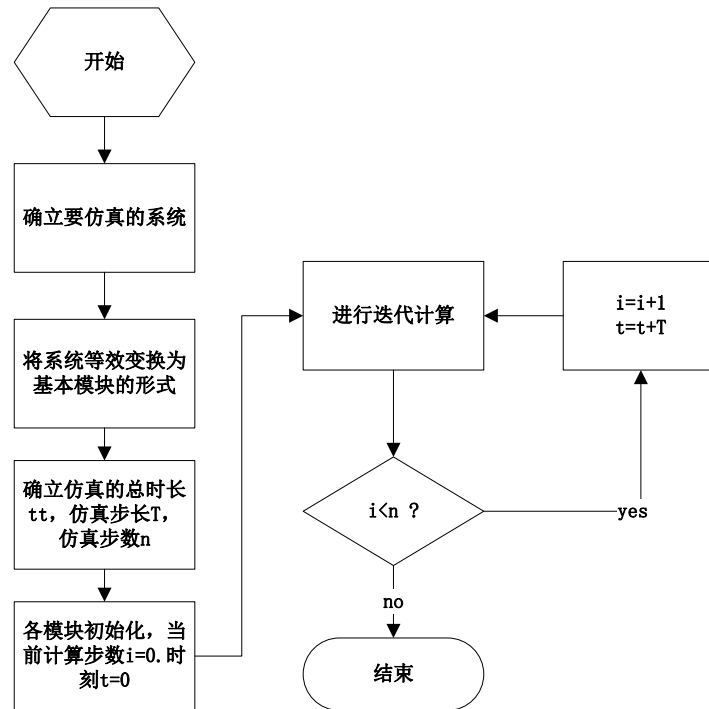


图 4-5 基本仿真流程

4.3 控制系统仿真程序的实现

依照 4.2 节的流程和算法，可以利用任何一种高级语言编制程序进行仿真计算。但是，针对每个系统都编制特定的程序，从实现上来讲会造成大量的重复代码，既不利于阅读也容易出错，进而导致难以进行测试、调试和改进。2.1.6 小节关于重构的讨论表明，重复的代码是程序中可以优化的地方。

采用离散相似法进行仿真计算时，复杂的系统最终可以转化为由一个个较小模块组成的简单系统。显然，针对单一复杂系统的过程化程序，也可以通过模块化的思路，“分治”地改造成多个简单系统组装而成的对象化程序。这正是 `simulink` 仿真程序成功的原因之一。`java` 对于面向对象的开发范式提供了良好的支持，在这类编程语言中，如何较好地封装仿真程序，已经有较多的探讨。

文献^[19]采用了一种基于 c 语言的模块化程序的开发方法。但是，c 语言不支持类等高级编程模型，虽然使用结构体和操作结构体的一组函数，可以有效地模拟类对数据和行为的封装，但是其可读性、api 易用性和对内存管理的要求等问题的都会给编程带来较大的难度。该文献虽然提出了面向对象的模块结构，但是将底层的运算逻辑和仿真模块的视图界面信息封装在一处，不符合面向对象编程的“单一职责原则”，更不会产生“易读易懂”的代码。另外，可能是出于商业的考虑，作者对面向对象下的仿真算法如何实现，并没有做过多的讨论。

simulink 采用后缀为.mdl 的文件统一记录模块的参数和渲染的信息，在仿真时采取解析该模型文件的方法，对文件中数据包含的信息进行提取，并将和绘制有关的信息渲染到用户视图上，将和仿真有关信息载入到模型之中进行仿真计算，从而将可视化的模型界面和计算逻辑分开处理，满足了“单一职责原则”“职责分离”的要求。采用 simulink 建立如图 4-6 所示的仿真模型，在 matlab 工作空间中会生成后缀为.mdl 文件，采用文本编辑器打开可以看到如代码 4-1 所示的片段。从 5.9 节可以看出，采用这种数据结构及文件结合的方式记录模型，有利于模型的持久化存储和传输。综观第五章的迭代开发过程，采用这种.mdl 文件或本文采用的方式，处理仿真模型，不是实现上的“偶然抉择”，而是重构的“必然结果”。文献^[20]使用 simulink 实现了对水轮机组的模块化建模与仿真。

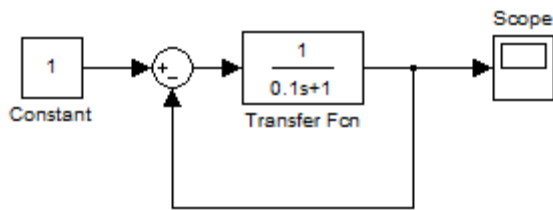


图 4-6 简单的 simulink 惯性环节

代码 4-1 图所示模型生成的.mdl 文件片段

```
Block {
    BlockType      Constant
    Name           "Constant"
    Position       [150, 100, 180, 130]
}
Block {
```

华 中 科技大学硕士学位论文

```

    BlockType
    Name      "Scope"
    Ports     [1]
    Position   [385, 99, 415, 131]
    NamePlacement "alternate"
    Location   [270, 365, 594, 604]
    Open      off
    NumInputPorts "1"
    List {
    ListType    AxesTitles
    axes1      "%<SignalLabel>"
    }
    SaveToWorkspace on
    SaveName      "inertia_out"
    DataFormat    "Array"
    LimitDataPoints off
}
Block {
    BlockType
    Name      "Sum"
    Ports     [2, 1]
    Position   [215, 105, 235, 125]
    ShowName   off
    IconShape  "round"
    Inputs     "|+-"
    InputSameDT off
    OutDataTypeMode "Inherit via internal rule"
    SaturateOnIntegerOverflow off
}
Block {
    BlockType
    Name      "Transfer Fcn"
    Position   [260, 97, 320, 133]
    Denominator "[0.1 1]"
}

```

由于 simulink 出色的封装和简单的使用接口，出现了一些基于 simulink 的仿真程序^[21-23]。这些程序为了兼顾业务开发的简便和用户界面的美观，采用了其他平台（如 java、C#、visual basic 等）调用 simulink 的方式。本文并不打算基于这种方式来开发仿真程序。

首先，这种方式不可避免地会降低程序的性能。在本文所使用的计算机 1 上，由于硬件资源的限制，使用 java 调用 matlab 核心的方式编写的程序，其启动速度与调用、返回结果的速度几乎到了无法忍受的程度。

其次，这种方式开发的程序可移植性不好，开发出来的程序加入了对 matlab 核心程序的依赖，matlab 核心不是开源的，而且没有对应的其他语言的实现版本。因

而采用这种思路开发的程序难以部署和移植。仅仅为了用户界面的美观就作出上述妥协是不可取的，直接基于 matlab 开发 GUI 程序，或者开发新的 simulink 插件包，是更好的方法。

最后，很多人对仿真和控制系统的了解，还停留在方框图和微分方程的概念之上，对于 simulink、PSASP^[24]（电力系统分析综合程序）等仿真程序如同魔法般的“拖拽式建模”，无法透过现象观察到本质；或者虽然懂得仿真算法，但不明白如何运用到仿真之中。本文的其中一个目的就是揭示隐藏在仿真程序用户界面下的技术细节。

综合上述考量，demo 项目探索了如何使用面向对象的方法，从底层开始建立起一个支持可视化建模的仿真系统。

4.4 仿真程序的约束

从程序开发的客观要求上来看，除了遵循面向对象开发的基本原则以外，仿真程序应当满足以下约束：

（1）对功能的约束。为了支撑水轮机调速系统仿真的上层功能，需要提供足够的仿真基础模块，如表 4-1 所示。demo 项目最终实现了表中的大部分基本模块。

表 4-1 仿真程序的基础模块

模块类型	说明	举例
信号源（Source）	提供标准输入和随机输入的模块及接口	阶跃信号源，斜坡信号源
线性控制（Linear Controller）模块	典型的控制模块	积分环节，放大器，惯性环节，分式型传函
非线性控制模块	典型的非线性环节	迟滞环节，死区环节
联结（joint）模块	联结其他模块的模块	加法器，乘法器
记录模块	记录数据	示波器
自定义模块	用户自定义模块的行为，输入和输出	
开关模块	按时间对某模块的输入输出进行控制。	定时开启开关，定时关闭开关

（2）对易用性的约束。程序应当有一组易用的用户接口，包括面向仿真模块

拓展的程序员接口和面向最终用户的图形界面接口；

（3）对精度的约束。和 `simulink` 仿真结果相对比，计算结果应具备合理的精度，如最大误差不超过 5%；

（4）对性能的约束。程序的计算部分应当足够快速，这一约束可以具体描述为“对于具有水轮机调速系统规模的一类系统，应当在 20ms 内完成时长为 30s，步长为 0.01s 的仿真计算”。

其中，约束（1）为主要约束，是开发过程中需要优先满足的，它直接关系到软件的主要功能是否实现；约束（2）是十分重要的约束，接口对程序员友好意味着更好的可读性和可拓展性，不可读的代码基本是不可拓展的；约束（3）是评价功能实现正确性的约束，相当于软件的测试标准；约束（4）是对程序非功能特性的要求，在开发过程中可以先暂时不予考虑，而在代码性能调优的时候再加以满足，当然，结构良好的代码通常具有不错的性能。

在开发的过程中，`demo` 项目采用和 `simulink` 相类似的模式对程序的结构进行组织。在开发方法上，由于仿真程序的结构基本是清晰明确的，只是在实现细节上存在模糊的地方。面对这种场景，犹豫迟疑只会让程序员原地踏步。可以遵循敏捷的思想，使用敏捷的 `groovy` 语言，采取一种快速原型的开发方法，一开始就着重、就简地实现程序的核心仿真计算功能，然后通过反复迭代与重构，改善程序的结构，并使用 `java` 实现其中对性能要求较高的部分，最终将原型演变为功能较为完备的仿真程序。

4.5 水轮机调速系统模型

水轮机调速系统可以使用控制模型进行描述。其主要结构如图 4-7 所示。完整的调速系统包括引水系统，调速器，执行机构（接力器），水轮发电机组。遵循面向对象的思考方式，对这些组成部分，分别采用相应的模型，单独进行描述。

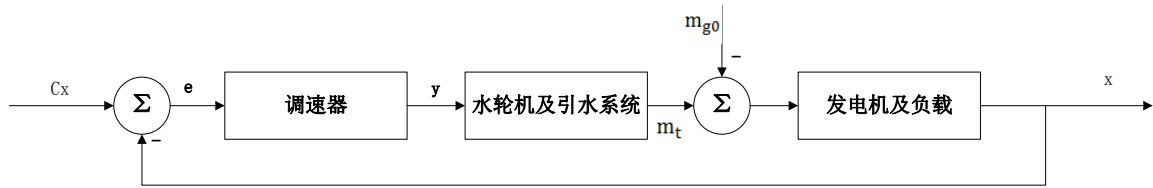


图 4-7 调速系统结构

文献^[25]给出了一种微机调速器的线性模型。其各部分的控制模型如下。

压力引水系统：

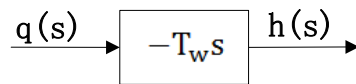


图 4-8 压力引水系统模型

执行机构经过简化处理，可以近似看成一个惯性系统：

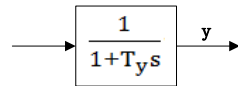


图 4-9 一阶惯性系统

引水系统及水轮机系统模型：

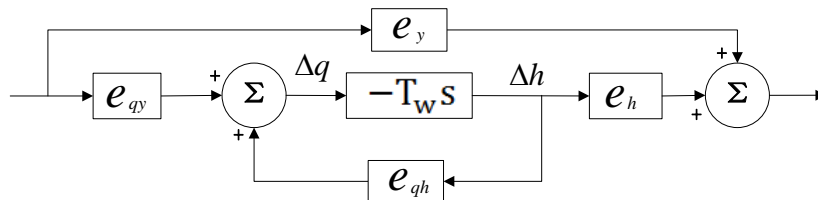


图 4-10 引水系统及水轮机模型

发电机及负载模型：

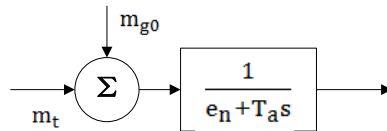


图 4-11 发电机及其负载

最终得出在稳定工况点附近，水轮机调节系统的线性化数学模型如图 4-12 所示。

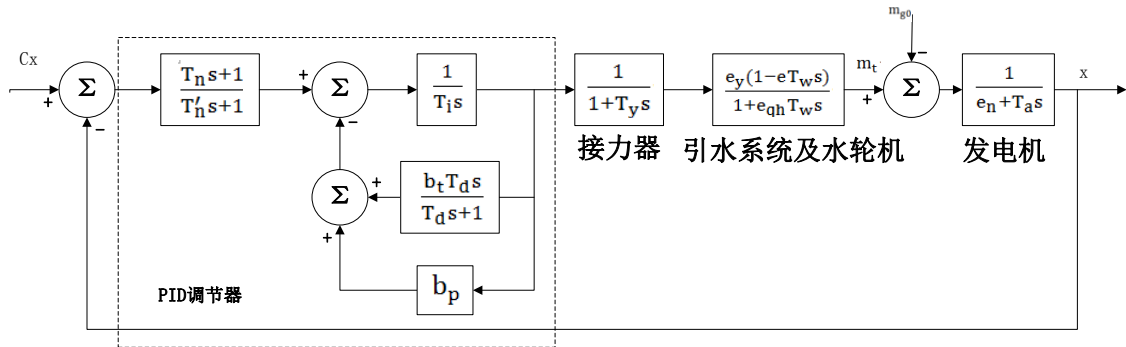


图 4-12 调速系统线性化模型

需要注意的是，上述线性化模型在水轮机运行与额定工况附近时能获得较高的精度，但不适用于大波动过程下的水轮机暂态运行仿真。要进行大波动过程下的暂态仿真，还需要在上述模型的基础上，引入更加精确的模型。考虑到工作量和时间的限制，以及实现上的简便，demo 项目仅考虑上述的线性化模型。对于更加复杂的业务需求，如考虑水轮机的非线性因素等，可以通过对仿真程序进行拓展来满足。

4.6 本章小结

本章讨论了构筑水轮机仿真程序所必须的基础设施：控制系统仿真算法和调速系统的数学模型。

在仿真算法中，着重讨论了使用零阶保持器的离散相似法。该算法的优点在于递推公式形式简单，容易使用编程语言实现。缺点在于保持器数量越多，算法的精度越低。然而，由于实现上的简便，demo 项目将在原型和实现中都采用离散相似法。在需要确保精度的场景中，可以考虑使用数值积分方法中更为精确的龙格-库塔法等方法，该方法是 simulink 默认采用的方法。

本章说明了将快速迭代的“敏捷”方法运用到工程领域软件的开发思路，探讨了这种方式的可能性和方法论。在带有研究性质的项目之中，对需求的理解不是一成不变的，也不是一蹴而就的，而是不断变化的，日趋成熟的。我们的编程方法也要能够敏捷地应对研究过程中的持续变化，这正是可以使用迭代方法的基础所在。

敏捷过程提倡的四个核心原则，也就是著名的敏捷宣言：

- (1) 个人和交互高于过程和工具；
- (2) 工作软件高于详细的文档；

(3) 与客户合作高于合同谈判;

(4) 对变更及时做出反应高于遵循计划。

遵循上述原则,有助于将开发的注意力集中在最终的产品上。然而,正如大多数讨论敏捷的文献指出的,敏捷过程是一个需要勇气的过程,其提倡的“交流”“合作”“拥抱变化”等价值观不仅需要个人具有创新的勇气,同时需要团队具有坚定不移贯彻“敏捷精神”的勇气。

本章还给出了用于水轮机调速系统仿真的线性化的数学模型。下一章将探讨如何运用前述各章节所讨论的思想、方法,使用基于 java 平台的技术实现仿真程序。

5. 程序原型及快速迭代开发

敏捷方法的关键在于迅速得到可以运行的软件，以通过不断地运行程序，得到第一手的反馈信息，进而不断地改进和完善程序。为此，可以从原型出发，通过对原型的迭代，在每次迭代周期中增量地引入新的特性。对于 demo 项目而言，可以先采用过程化的脚本实现简单的仿真模型，再重构为面向对象的实现，并不断加入新的仿真元素和特性，最终得到完整的程序。

5.1 第 0 次迭代——groovy 实现

首先，采用最简单的惯性环节和阶跃电源组成的开环系统，系统的方框图和程序的计算流程分别如图 5-1、图 5-2 所示。

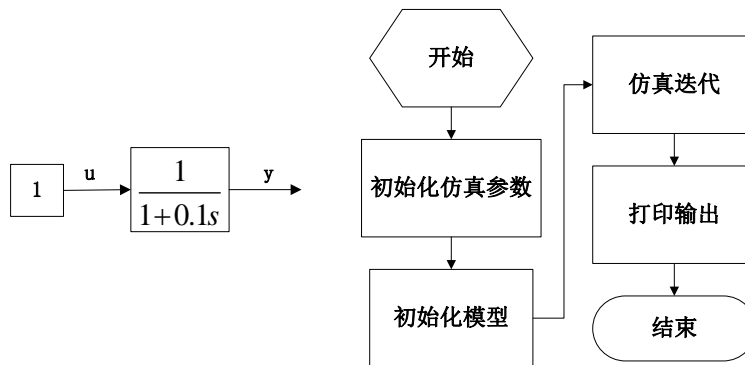


图 5-1 简单的惯性环节

图 5-2 程序计算流程

由公式（4-27），采用离散相似法离散化后的模型为：

$$y(k+1) = e^{-T}y(k) + (1 - e^{-T})u(k)$$

式中，T 为仿真步长。仿真程序如代码 5-1 所示

代码 5-1 惯性环节仿真计算 inertiaTest.groovy

```
def start=System.currentTimeMillis()

def dt=0.01 //仿真间隔
def totalTime=10 //seconds
def e=Math.E;

def n=totalTime/dt
```

华 中 科技大学硕士学位论文

```
def time=[] //仿真时间
def u=[] // 阶跃输入
(0..<n).each{k->
    time+=k*dt
    u+=1.0
}

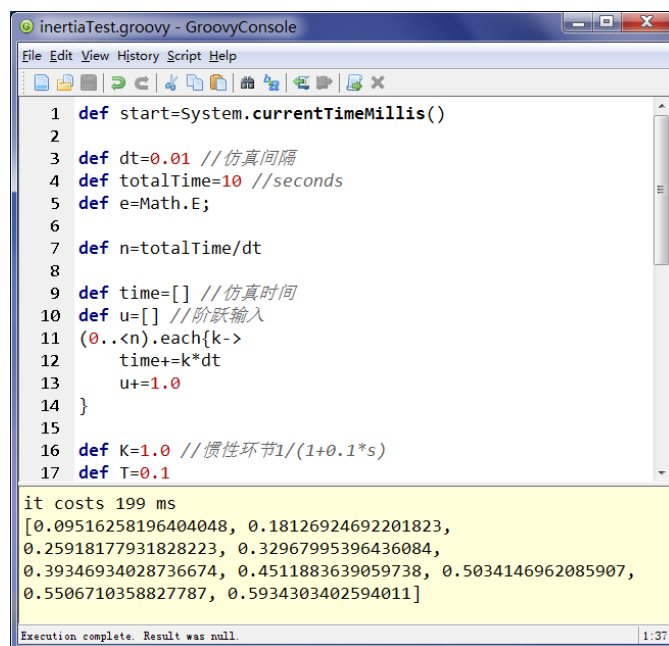
def K=1.0 //惯性环节 1/(1+0.1*s)
def T=0.1
def out=[]
out[0]=0

def c1=e**(-dt/T)
def c2=K*(1-e**(-dt/T))
(1..<n).each{k->
    //迭代
    double d= c1*out[k-1]+c2*u[k]
    out+=d
}

def end=System.currentTimeMillis()
println "it costs ${end-start} ms"

println out[1..<10] // 输出结果打印, 后文中省略
def pw=new PrintWriter('D:\\out.txt')
(0..<n).each{k->
    pw.println("${time[k]} ${out[k]}")
}
pw.flush()
pw.close()
```

程序的输出如图 5-3 所示。值得注意的是, 上述程序首次运行耗时为 199ms, 在第二次及之后的运行中消耗的时间保持在 25ms 左右, 这是因为 groovy 控制台编译并缓存了该脚本产生的.class 文件, 使得性能得到了提升。另外, 上述程序运行在计算机 1 上, 在计算机 2 上进行相同的试验时, 耗时为 3ms 左右, 这是二者硬件配置的差异带来的性能差异。

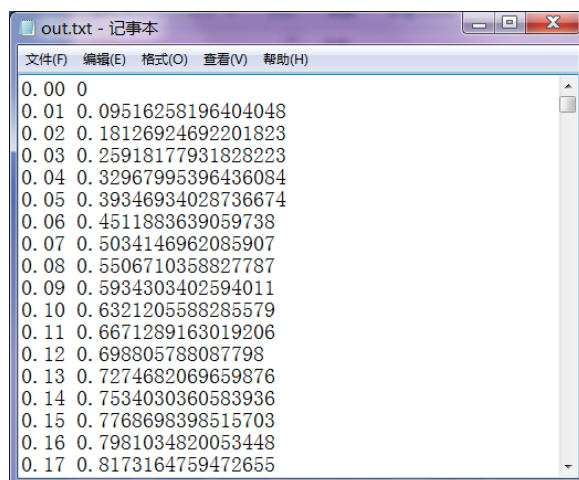


```
1 def start=System.currentTimeMillis()
2
3 def dt=0.01 // 仿真间隔
4 def totalTime=10 //seconds
5 def e=Math.E;
6
7 def n=totalTime/dt
8
9 def time=[] // 仿真时间
10 def u=[] // 阶跃输入
11 (0..<n).each{k->
12     time+=k*dt
13     u+=1.0
14 }
15
16 def K=1.0 // 惯性环节1/(1+0.1*s)
17 def T=0.1

it costs 199 ms
[0.09516258196404048, 0.18126924692201823,
0.25918177931828223, 0.32967995396436084,
0.39346934028736674, 0.4511883639059738, 0.5034146962085907,
0.5506710358827787, 0.5934303402594011]
```

图 5-3 程序输出

程序输出打印到文本文件中如图 5-4 所示。



```
0.00 0
0.01 0.09516258196404048
0.02 0.18126924692201823
0.03 0.25918177931828223
0.04 0.32967995396436084
0.05 0.39346934028736674
0.06 0.4511883639059738
0.07 0.5034146962085907
0.08 0.5506710358827787
0.09 0.5934303402594011
0.10 0.6321205588285579
0.11 0.6671289163019206
0.12 0.698805788087798
0.13 0.7274682069659876
0.14 0.7534030360583936
0.15 0.7768698398515703
0.16 0.7981034820053448
0.17 0.8173164759472655
```

图 5-4 文本输出

下面需要验证计算的正确性。如同 4.3 节所述，采用和 simulink 的仿真结果进行对比的方式来验证程序的正确性。建立 simulink 模型如图 5-5 所示。

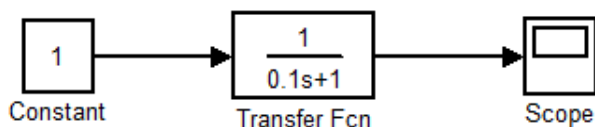


图 5-5 simulink 惯性环节模型

华 中 科技大学硕士学位论文

仿真过程中采用和 groovy 程序相同的配置，在 simulink 的 Simulation 菜单下配置即可，simulink 默认使用 ode3 算法，在试验过程中始终采用这一默认的配置。同时，设置示波器 Scope 将仿真结果以矩阵的形式保存在工作空间中，在图 5-6 所示的配置下，输出为 1001×2 double>类型的矩阵，将其命名为 inertiaOut。

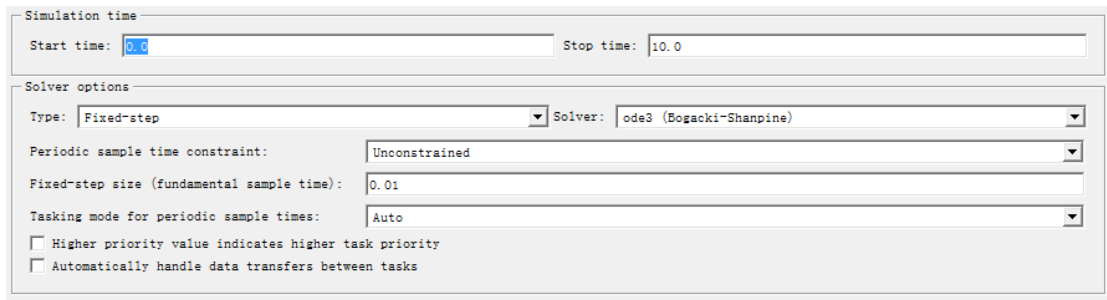


图 5-6 simulink 仿真配置

编写如代码 5-2 所示的 matlab 函数，读取保存在路径“D:\\out.txt”下的仿真计算结果，并绘制图形对仿真的结果进行对比。例如，在本例中使用 compare(inertiaOut)得到如图 5-7 所示的曲线图。

代码 5-2 绘制曲线仿真结果进行对比的函数

```
function compare(a)
    out=importdata('D:\\out.txt');
    plot(a(:,1),a(:,2),'red -',out(:,1),out(:,2),'blue --');
end
```

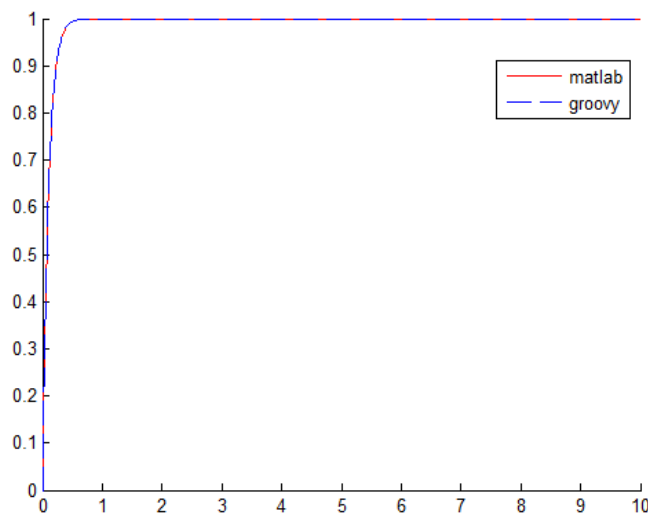


图 5-7 T=0.01s 仿真结果对比

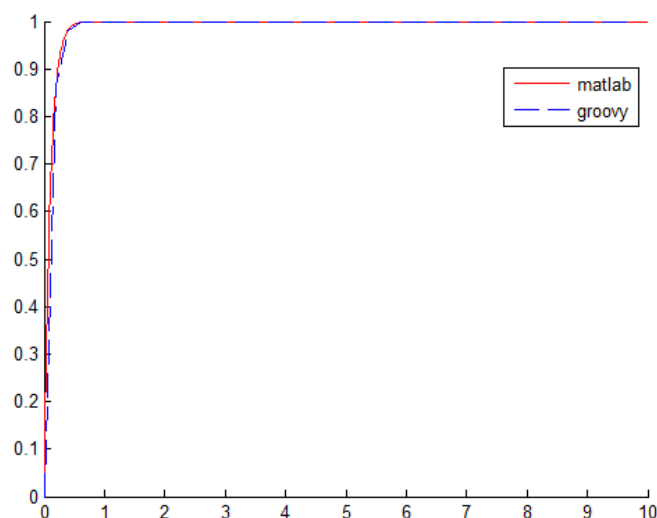


图 5-8 $T=0.2s$ 时的仿真结果对比

可以看出，当仿真步长增大 20 倍、并且为惯性环节时间常数（0.1s）的 2 倍时，仿真精度有所下降。

现在，我们已经有了程序的第一个版本。虽然采用的模型十分简单，仿真的过程代码也无法被复用，但至少，我们实实在在看到了程序运行的结果。这多少能给我们一些信心。接下来，在我们进一步丰富这个简单的程序之前，先来看一下它的 java 版本，观察一下两种语言的异同。

5.2 第 0 次迭代——java 实现

5.1 节中的仿真程序的 java 版本如代码 5-3 所示。这段代码完成了和代码 5-2 完全相同的功能，其主体为方法 `simulate`。

代码 5-3 仿真程序的 java 版本

```
public class InertiaTest {
    private static double[] time;
    private static double[] out;

    public static void main(String[] args) {
        TestUtil.timeIt(() -> simulate()); // 计时
        TestUtil.printFirst(out, 10);

        PrintUtil.print(w -> { // 打印结果到 "D:\\out.txt"
            for (int i = 0; i < time.length; ++i) {
                w.println(String.format("%f %f",
                    time[i], out[i]));
            }
        })
    }
}
```

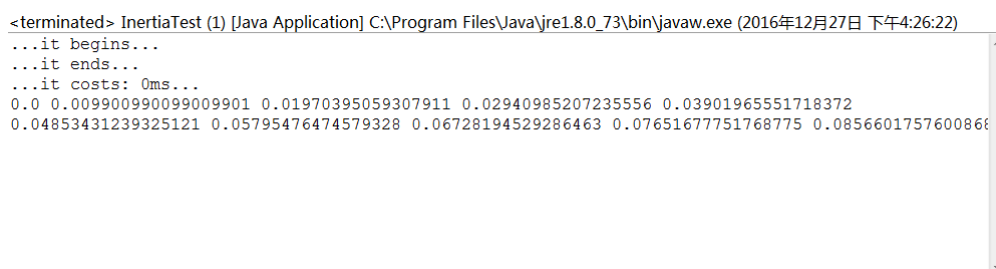
```
        });  
    }  
  
    private static void simulate() {  
        double dt = 0.01; // 仿真配置  
        double totalTime = 10; //seconds  
  
        int n = (int) (totalTime / dt);  
  
        time = new double[n]; // 输入配置  
        double[] in = new double[n];  
        for (int i = 0; i < n; ++i) {  
            time[i] = i * dt;  
            in[i] = 1.0;  
        }  
  
        double K = 1.0; // 参数配置  
        double T = 1.0;  
  
        out = new double[n];  
        out[0] = 0;  
  
        double c1 = Math.pow(Math.E , -dt/T); // 缓存系数  
        double c2 = K * (1 - c1);  
        for (int k = 1; k < n; ++k) { // 迭代仿真  
            out[k] = c1 * out[k - 1] + c2 * in[k];  
        }  
    }  
}
```

这里我们可以直观的感受得到，尽管有动、静态类型的差别，java 和 groovy 的语法十分接近。groovy 代码更加简洁，编写起来更加容易，java 代码占用了更多的行数，其中，还依赖了两个辅助方法（TestUtil、PrintUtil 中的静态方法）。但是，既然 groovy 更加简洁优雅，为什么 demo 项目的主体还是要使用 java 而不是 groovy 呢？

如 3.7 节中讨论的，java 的强类型更加严谨，能够有效的借助编译器的能力在编译而不是运行时就找出绝大多数和类型相关的错误。而动态语言（如 javascript、python、groovy）在多数情况下直到运行时才会抛出错误提醒你代码某处弄错了类型，甚至有时会抛出含义模糊、难以调试的错误。从开发的经验也可以看出，如果使用 groovy 来开发底层业务的模型，长时间的接触动态类型会使开发者迷失在“类型灾”之中。所以，使用类似的语言进行原型、脚本等规模较小的程序的开发，能更好地发挥其优势。

华 中 科技大学硕士学位论文

此外，从如图 5-9 所示的运行结果来看，在仿真时长为 10s，步长为 0.01s 时，java 代码的耗时小于 1ms，可见二者的性能差异十分明显。值得一提的是，把仿真时长从 10s 调整至 300s 到 1000s，程序运行时间最大不超过 10ms。对仿真程序而言，计算的耗时取决于迭代次数和模型的复杂程度，在性能优化的时候可以通过调整仿真时间、仿真步长，简化模型来获得更高的效率。但是目前而言，没有必要过多地考虑性能问题。程序的性能优化一般放在开发的最后阶段，甚至测试版发布以后。现阶段，使程序和设计保持良好更加重要。实践表明，遵循良好的编程原则有助于获得较好的性能。



```
<terminated> InertiaTest (1) [Java Application] C:\Program Files\Java\jre1.8.0_73\bin\javaw.exe (2016年12月27日 下午4:26:22)
...it begins...
...it ends...
...it costs: 0ms...
0.0 0.009900990099009901 0.01970395059307911 0.02940985207235556 0.03901965551718372
0.04853431239325121 0.05795476474579328 0.06728194529286463 0.07651677751768775 0.0856601757600866
```

图 5-9 java 版本的运行结果

对于功能明确的代码，有必要使用函数的手段进行“模块化”的封装。借助诸如 eclipse 这样的现代 ide，查看封装良好的代码变得更加容易。代码 5-3 中的 TestUtil.timeIt()方法和 PrintUtil.printTo()方法，运用了 java8 语言的“函数式编程”特性。代码 5-4 和 5-5 展示了使用封装的方式。

代码 5-4 辅助计时函数 timeIt，使用 java 实现

```
public static void timeIt(Container container) {
    print("...it begins...");
    long start = System.currentTimeMillis();
    long end;
    try { // 错误处理
        container.process();
        end = System.currentTimeMillis();
    } catch (Exception e) {
        print("error occured: " + e.getMessage());
        return;
    }
    print("...it ends...");
    long cost = (end - start);
    print("...it costs: " + cost + "ms...");
}
```

```
@FunctionalInterface
public static interface Container {
    void process();
}
```

代码 5-5 辅助打印函数 printTo，使用 groovy 实现

```
class PrintUtil {
    static def path='D:\\out.txt'

    static def printTo(path,Printer p){
        def pw=new PrintWriter(path)
        p.print(pw)
        pw.flush()
        pw.close()
    }

    static def print(Printer p){
        printTo(path,p)
    }
}

@FunctionalInterface
interface Printer{
    void print(PrintWriter pw)
}
```

从敏捷开发的思想出发，上述代码并不是事先计划的产物：文档和设计都不会事先决定有这么一段代码。这段代码应该是在实现业务的过程中，不断遇到类似的逻辑，于是重构得到的通用“模块”。事实上，代码 5-4 源于计量程序运行时间的一个简单但又不断重复的代码片段，代码 5-5 则是 demo 项目开发过程中可以预计的将会不断使用的一段代码。两段代码都用于辅助对程序进行测试。在编程过程中，如果碰到需要以“复制粘贴”的形式复用代码的地方，在“重复的代码”（见 2.1.6 重构）把事情弄得更糟糕之前，不妨考虑类似上面的“静态全局功能函数”。

5.3 已经完成了？

有的观点认为，我们可以止步于“第 0 次迭代”，因为所有功能都已经完成了，没有再往下走的必要。如果需求改变，大不了打上补丁。这种观点的思维止步于程式的代码，其适用范围同样止步于小型、一次性、用完就丢掉的程序。

那么问题在哪里呢？“第 0 次迭代”的代码无法被重用，模型的改变、仿真过程的改变等都要求修改源代码，因此对很多变化都不是封闭的；又由于不能添加新

的模型、算法等问题，所以对拓展而言也不是“开放”的。这样的代码规模越大，越难进行测试。更重要的是，这段代码不可读，代码中随处可见的“魔幻数”和随意命名的变量，经过很久以后，即使作者自己，也会忘记了创作的初衷，从而无法继续维护自己的作品。

诚然，还有注释可以帮助他。但是实践告诉我们，过多和过少的注释都将是问题所在^[26]，良好的程序应具备“自解释性”：通过良好的命名和模块划分使程序脉络清晰。抛开维护注释带来的工作量不谈，注释可能仅仅告诉读者：老兄，注意这，打了斜杠和星号的这部分，这里可能会出问题！

当然，像第 0 次迭代中这样的“一次性代码”也有它的使用场景，这就是“脚本代码”，这种代码在本文前台 GUI 程序开发过程中得到了运用。众所周知，由于历史的原因，不同的操作系统底层一般使用 c 语言实现，并提供 c 语言的 api 供程序员使用。但是，如果任何任务都必须接触底层，难免是一件枯燥乏味的事情。python 语言和 java 一样，提供了很多接口简化并统一了访问操作系统服务的过程。但是，python 更进一步，提供了更为简便的 api。这是巨大的进步。使用 python 编写可以跨平台运行的任务脚本，是一种愉悦的体验。6.4 节对脚本代码进行了更多的探讨。

“过早的止步”体现出不严谨的态度和对编程语言理解上的谬误。对于 java 而言，jvm 就如同一台精准的机器，使用它的人相当于这台机器的操作员，只有操作得当，机器才能运转良好。

下面，是时候迈出迭代开发的第一步了，为了说明开发的思路，最初的几步都会迈得足够小。

5.4 第一次迭代

考虑给图 5-1 中的模型加入一个新的环节，如图 5-10 所示。

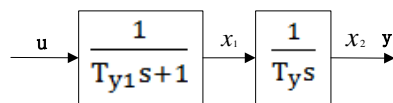


图 5-10 第一步迭代的模型

由公式（4-27），采用离散相似法离散化后的模型为：

$$x_1(k+1) = e^{-10T}x_1(k) + (1 - e^{-10T})u(k+1)$$

$$x_2(k+1) = x_2(k) + 2Tx_1(k+1)$$

其中， T 为仿真步长。这里，我们取 $T_{y1} = 0.1$ ， $T_y = 0.5$ 。

从模型可以知道，引入一个新的模块（积分环节），就需要增加一个新的变量来进行计算。可以看出，模型各部分的连接顺序为：电源——惯性环节——积分环节，计算过程中要保证先计算 $x_1(k+1)$ ，再计算 $x_2(k+1)$ 。

代码 5-6 的变化仅仅是添加了一个新的配置和计算过程。具体的仿真程序代码可以参考 demo 项目的 simple_serial.groovy。

代码 5-6 添加积分环节后新增的代码片段

```
def ty=0.5 // 配置
.....
def x2=[] // 初始化
x2[0]=0
def c2=2*dt
.....
(1..n).each{k-> // 迭代
    .....
    x2[k]=(x2[k-1]+c2*x1[k-1]).trunc(4)
}
.....
```

与 simulink 仿真对比的结果如图 5-11 所示：

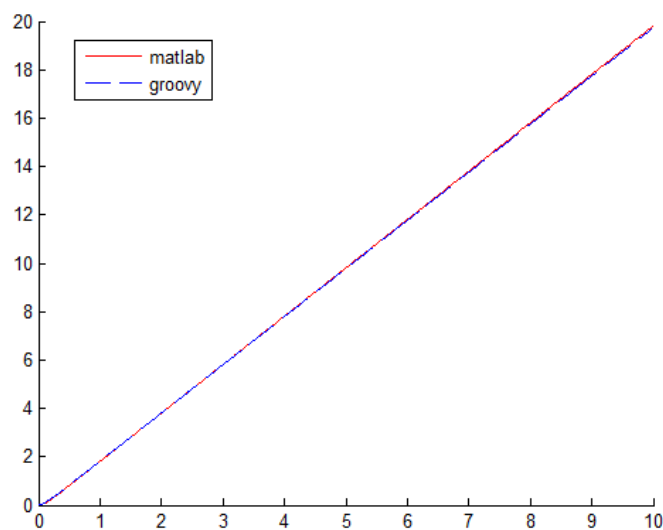


图 5-11 简单串连环节的仿真结果

5.5 第二次迭代

这个小节中我们再次前进一小步，将“第一次迭代”中的模型闭环，就得到如图 5-12 所示的模型。

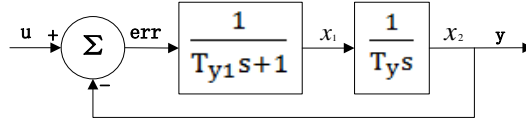


图 5-12 简单的闭环模型

依旧采用离散相似法，结合公式（4-27），得到离散化的模型为：

$$\text{err}(k+1) \approx u(k+1) - x_2(k)$$

$$x_1(k+1) = e^{-10T}x_1(k) + (1 - e^{-10T})\text{err}(k+1)$$

$$x_2(k+1) = x_2(k) + 2Tx_1(k+1)$$

其中，T 为仿真步长。这里，我们同样取 $T_{y1} = 0.1$ ， $T_y = 0.5$ 。

由模型可知，引入一个求和模块之后，对算法产生了更多的限制。在迭代的过程中，欲求 $x_2(k+1)$ ，须先求 $x_1(k+1)$ ，而欲求 $x_1(k+1)$ ，需要知道 $\text{err}(k+1)$ ，这相当于对求解顺序施加了更严格限制，计算时需要从电源出发，遍历所有的仿真块。在开发通用的仿真程序时，必须考虑到这一点。在开发用户界面的时候，还必须注意，不能假设用户的输入符合求解的计算顺序，必须使用 5.10 节中讨论的算法确定计算的顺序。尽管存在上述的影响，代码的变化足够小，因为我们的步伐也足够小。

代码 5-7 闭环后新增的代码片段

```
...
def err=[] // 误差
...
def c11=e**(-10*T) // buffered coefficient
def c12=1-e**(-10*T)
def c21=2*T
(1..n).each{k ->
    err[k]=u[k]-x2[k-1] // iteration
    x1[k]=c11*x1[k-1]+c12*err[k]
    x2[k]=x2[k-1]+c21*x1[k-1]
    y[k]=x2[k] // save output
}
...
```


与 simulink 仿真对比的结果如图 5-13 所示。可以看出，仿真结果有比较高的精度，通过对数据比较可知计算结果精确到小数点后第二位。

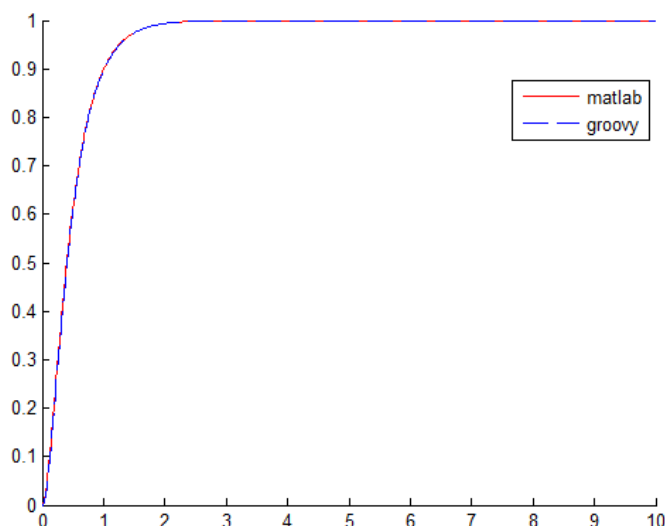


图 5-13 简单闭环系统的仿真结果

5.6 是时候模块化了

连续三个系统的仿真程序编制，已经可以让我们“嗅到”重复的味道了。如同反复被指出的，重复意味着可以被重构。前面几次迭代中重复的部分，都是可以封装模板代码；而不同的代码，也可以使用多态来统一。我们总可以通过不断地重构，尽可能地消除代码中的“重复”。

这一个小节依旧以图 5-14 所示的系统为例。从物理结构来看，这个系统包括三个部分：输出源（Source）、惯性块（Inertia Block）和这两者之间的连接线（Line）。从发生的物理过程来看，输出源驱动惯性块运动并发生状态变化。从数据的角度来看，输出源为阶跃源，其输出通过连接线传到惯性环节，惯性环节就像一个黑箱转换器，将输入进行转化后输出。

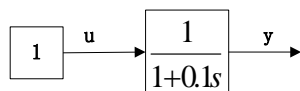


图 5-14 简单的惯性环节

从上面的描述中，我们提可以炼出 Source、Block、Line 三个基本模型，就像面向对象分析中那样。但是，我们显然做得更多，因为我们已经有了前面的三次尝

华 中 科技大学硕士学位论文

试，所以，得到上面的抽象更加的自然。使用类进行建模，类图如图 5-15 所示，Config 用以描述一次仿真的参数设置，块儿 StepSource 和 Inertia 分别代表阶跃电源和惯性环节，Line 用以连接两个块，Simulation 作为上层驱动协调各个类的对象进行仿真计算。

仿真程序如代码 5-8 所示。这段程序依旧很简洁，但是和过程化的程序相比，已经有了一定的复杂性，而且还存在两个明显的缺陷：

(1) 程序依旧不能复用。除了惯性环节，控制系统还有很多其他的基础环节没有考虑到，这说明程序的抽象程度还不够；Simulation 类没有普适性，到目前为止，它还只能处理图 5-14 中的简单模型；

(2) 程序的可拓展性不好。当两个类具有相同的方法签名时，类似 groovy 这样的动态语言认定它们具有相同的接口，但是这种约定而来的接口降低了可读性；此外，不同类型的模块接口并不是完全一致的，需要考虑到模块的类型。

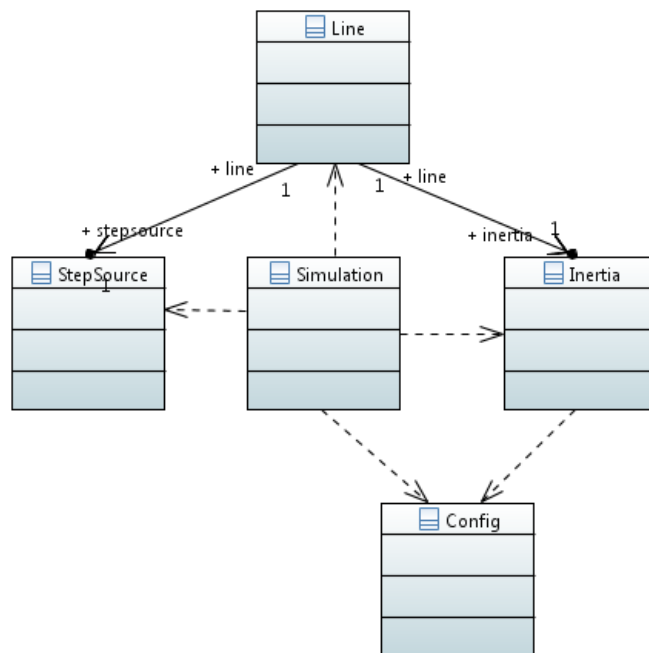


图 5-15 程序类图

代码 5-8 模块化的仿真程序

```
class Config{ // 仿真配置
    double T=0.01
    double totalTime=10
```

华 中 科技大学硕士学位论文

```
int n=totalTime/T
double currentTime=0.0
int i=0

void next(){
    ++i
    currentTime+=T
}

}

class Inertia{ // 惯性环节
    Config config
    def e=Math.E

    double k=1
    double t=0.1

    double out=0.0 //initial value

    double c1
    double c2

    def init(){
        c1=e**(-config.T/t)
        c2=k*(1-e**(-config.T/t))
    }

    double next(double input){
        out=c1*out+c2*input
        return out
    }
}

class Line{ // 连接线
    def start
    def end

    void next(){
        end.next(start.next())
    }
}

class StepSource{ // 阶跃源
    double next(){
        return 1.0
    }
}

class Simulation{
    def simulate(){
        def config=new Config() //仿真配置
```

```
def source=new StepSource() //元件设置

def inertia=new Inertia()
inertia.config=config
inertia.init()

def line1=new Line() // 连接设置
line1.start=source
line1.end=inertia

def y=[]
y[0]=inertia.out

(1..config.n).each{ //仿真迭代
    config.next()
    line1.next()
    y[it]=inertia.out
}
return y
}

def simu=new Simulation()
def out=simu.simulate()
println out[0..10]
```

可以进行重构，通过使用继承体系（5.7 节）改善程序结构，使用强类型和接口（5.7 节）明确程序边界，通过分离模型数据（5.9 节）提升程序的通用性。有人指出，可以利用 groovy 的可选类型特性，交替使用强弱类型，同时达到增强类型限制和保持编程的灵活性的目的。但是，编程实践表明，强弱类型联合起来使用，容易让人迷惑，哪里使用强类型，哪里使用弱类型都会让人纠结不已。项目的规模愈大，类型的问题愈加明显。还是将这种特性作为对 java 兼容的手段，而把建模交给 java 来完成吧。从这里开始，我们将不再使用 groovy 建模，但是从代码 5.15 可以看到，groovy 适合用于开发上层驱动程序，发挥它“胶水”的作用。

本节开发的程序还存在不足，但是它实现了基本的模块，作为仿真程序的雏形已经初具规模，所谓“麻雀虽小五脏俱全”。下一节将引入类层次结构。

5.7 类层次结构

在前面的代码中，阶跃电源可以进一步抽象为电源模型，它为系统提供能量，

同时也是计算的出发点；惯性环节可以抽象为控制块，它接受输入环节（可能是电源、连接点或其他控制块）提供的能量，并进行转化，得到新的输出。上面的描述提醒我们在程序中引入继承体系。改善后的程序结构如图 5-16 所示。

可以看到，结构的主要部分包括了一组接口，其根接口为类 Block（块），其下继承有 ControlBlock（控制块）和 Source（源）。ControlBlock 下继承有两个标记接口：LinearBlock（线性控制块）和 NonlinearBlock（非线性控制块）。

程序中几个主要类的 CRC（Classes, Responsibilities, Collaborations, 类-职责-协作）卡如表 5-1 至 5-4 所示。CRC 卡描述了类完成的功能，以及为了完成某项功能，他的协作者类所提供的功能。

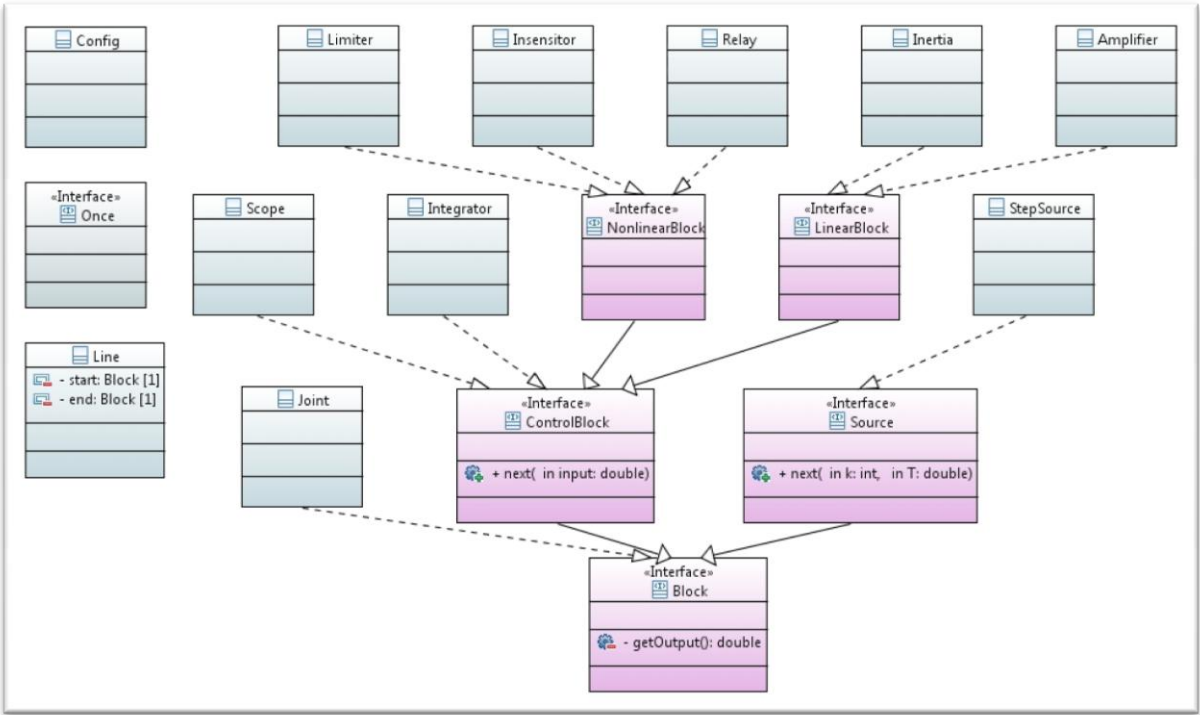


图 5-16 程序的继承结构

表 5-1 Block 的 CRC 卡

仿真块	
建模的基本模块	

表 5-2 Line 的 CRC 卡

华 中 科技大学硕士学位论文

连接线	
记录系统结构	仿真块
传递 start 块的输出，并促使 end 块的转态发生改变	仿真块

表 5-3 Joint 的 CRC 卡

连接点	
记录系统结构	
汇聚多个输入	连接线

表 5-4 Simulation 的 CRC 卡

仿真	
进行一次仿真计算	连接线、连接点、仿真块

通过引入继承结构，程序显得更加复杂，但我们得到了以下好处：多个基础模块可以被复用；消除了重复；程序结构更加清晰，使代码容易阅读；拓展程序变得更加容易，只需要向继承体系中添加新的实现类即可；Line 类分离了 Block 之间的依赖关系，专门用于处理系统的结构，Block 只需要描述客观对象的物理特征即可。

结合继承结构和 CRC 卡，可以迅速弄清程序的结构。因而，保留这些图表制品作为文档，可以使系统更好维护。仅仅保留这些制品就足以说明设计意图。敏捷开发提倡只留下“刚好够用”的文档，并勇敢地剔除过时的文档，减少文档的维护成本，避免过度开发文档所耗费的不必要的精力。

综观软件工程的发展过程，随着敏捷方法的提出，文档所扮演的角色越来越淡出软件设计的舞台。Martin Fowler 也在其著作中一再透露^[27]，文档制品更多的应该被视为一种沟通交流和解释说明的存在。在敏捷开发中，提倡和客户及用户进行面对面的交流，提倡发挥人的主观创造力，以尽快得到可以运行的软件。敏捷开发实践还指出^[28]，使用“用户故事卡片”和“燃尽图”等简单的工具，为畅通的交流提供辅助。即便如此，敏捷也并非抵制文档、提倡“无文档”，而是只保留适当的、最低限度的、有用的文档。

华 中 科技大学硕士学位论文

从实现角度看，由于引入了继承，在计算中，需要集中进行类型判断，并正确造型。这主要体现在 **Line** 类的改变上，其关方法如下：

代码 5-9 **Line** 类的 java 实现

```
public class Line {
    final Block start;
    final Block end;

    public Line(Block start, Block end) {
        super();
        this.start = start;
        this.end = end;
    }

    void push(int k, double T) {
        if (start instanceof Source) {
            ((Source) start).next(k, T);
        } else if (end instanceof ControlBlock) {
            ((ControlBlock) end).next(start.getOutput());
        }
    }
}
```

从代码中可以看出，**Line** 类记录了其连接的两个模块，并推动它们的状态发生改变（通过正确造型并委托给相关的仿真模块）。其关键方法 **push()** 使用了当前计算步数 **k** 和时间间隔 **T** 作为参数。值得注意的是，当 **start** 模块为电源时，**Line** 通知电源改变至下一状态；当 **end** 为控制块时，**Line** 通知控制块改变至下一状态。其他情况下，调用 **push** 方法都不会使模块状态发生改变。比如，**end** 为连接点时，仅仅读取 **start** 的输出，而不会调用其 **next** 方法。这保证计算正确地进行。

代码中的类型转换有些丑陋，但好在他们有了一个归宿，不至于分散在程序的各处，在它们造成问题（如测试失败）时，我们可以集中处理。下个小节将使用一个例子说明仿真程序的初步应用。

5.8 仿真实例：ExciterTest

采用文献[18]中的励磁系统模型，如图 5-17 所示。模型中的与各个模块对应的、代码 5-10 中的变量名也已经标出（如 **step** 模块在代码中的变量名 **stepSource**）。

可以发现，程序特点在于对象和对象之间的信息传递，通过各个对象的配合完成仿真。还可以注意到，依照从 11 到 110 的顺序调用 **push** 方法符合 5.4 节中关于计

算顺序的讨论。

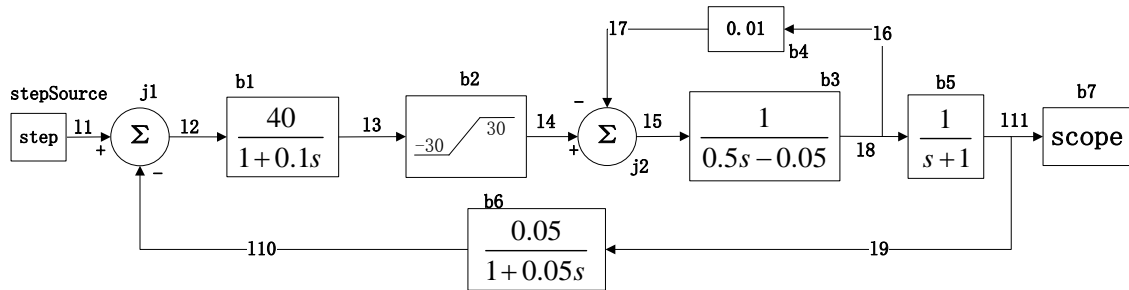


图 5-17 励磁系统模型

代码 5-10 励磁系统阶跃响应仿真，仿真时长 10s

```
public class ExciterTest {
    public static void main(String[] args) {
        Config config = new Config();

        Source stepSource = new StepSource();
        Joint j1 = new Joint();
        Block b1 = new Inertia().config(40, 0.1);
        Block b2 = new Limiter(30);
        Joint j2 = new Joint();
        Block b3 = new Inertia().config(-20, -10);
        Block b4 = new Amplifier(0.01);
        Block b5 = new Inertia().config(1, 1);
        Block b6 = new Inertia().config(0.05, 0.05);

        Line 11 = new Line(stepSource, j1);
        Line 12 = new Line(j1, b1);
        Line 13 = new Line(b1, b2);
        Line 14 = new Line(b2, j2);
        Line 15 = new Line(j2, b3);
        Line 16 = new Line(b3, b4);
        Line 17 = new Line(b4, j2);
        Line 18 = new Line(b3, b5);
        Line 19 = new Line(b5, b6);
        Line 110 = new Line(b6, j1);

        j1.addLine(11, Joint.ADD); // 配置连接点
        j1.addLine(110, Joint.SUB);

        j2.addLine(14, Joint.ADD);
        j2.addLine(17, Joint.SUB); // 负反馈

        List<Line> lines = Arrays.asList(11, 12, 13,
            14, 15, 16, 17, 18, 19, 110);
        TestUtil.timeIt(() -> {
            config.iterate((i, T) -> {
                lines.forEach(l -> l.push(i, T));
            });
        });
    }
}
```



```
    }  
    }  
};
```

按照惯例，通过和 matlab 结果进行对比以验证程序的正确性。如图 5-18 所示，当仿真间隔 $T=0.01s$ 时，结果和 matlab 相比基本吻合，但对于要求高精度的场合，仍存在比较大的误差；通过把 T 调整为更小的 $0.001s$ ，结果和 matlab 相比基本重合，仿真精度大大提高。从而我们验证了程序的正确性，并证明了误差是算法精度所致。

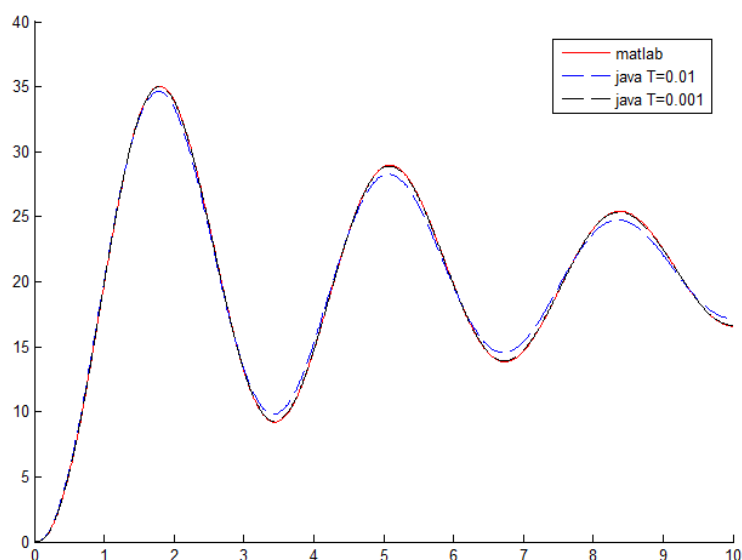


图 5-18 励磁系统仿真结果比较

和 4.4 节中讨论的一样，我们优先考虑功能的实现，延后考虑计算精度的问题。因为在合适的时候重构程序，采用更高精度的算法，总能够解决仿真精度方面的问题。现阶段，我们需要着重考虑的是，如何设计程序，使其对于仿真算法的变化是“开放”的，根据经验，遵循 DIP（依赖倒置原则）的原则，最终可以将程序重构为采用“策略模式”^[29]的解决方案，使程序可以灵活地切换为不同的算法。但是现在，使整个程序迅速运行起来才是当务之急，简单的算法虽然精度略为不如，但是胜在编码实现的简单。不得不再指出，在开发中保持实现的简单性，在需要时进行重构和优化，加入新的特性和技巧，远比过早考虑精度、性能等问题更加高效。

最后讨论上述程序的性能来结束这个小节。对象方法和接口方法的调用会略微降低性能。在“计算机 1”上，当仿真时长为 10s， $T=0.01s$ 时，上述仿真程序运行一次的时间平均为 11ms；当 T 减小到 $0.001s$ 时，这一时间增长到 25ms。7.3 节指

出，groovy 实现的 Simulator 类可能要为这多出的 14ms 买单。下个小节我们考虑如何通过分离模型数据，来统一程序的接口，并进一步提高仿真程序的通用性。

5.9 程序的进一步分离

上一小节中的仿真程序已经有了面向对象的调用风格，但不足之处是模型构建的过程，如代码 5-11 所示，大量的模型声明语句依然是重复代码的滋生处。不难预见，不断出现的命令风格的 Block、Line、Joint 声明语句也将会在程序复用时不断重现。“重复”又一次出现了，程序依旧存在“重构”的可能。在这个小节中，我们依旧采用 5.8 节中的励磁系统仿真模型。

代码 5-11 模型配置中产生的模板代码

```
Config config = new Config();

Source stepSource = new StepSource();
Joint j1 = new Joint();
Block b1 = new Inertia().config(40, 0.1);

..... // 省略重复的声明语句

j2.addLine(14, Joint.ADD);
j2.addLine(17, Joint.SUB);
```

幸运的是，各个模块已经得到了清晰的分离，仿真的步骤也十分明确，这提供了进一步分离程序的基础。为了使程序结构更加清晰，分离运算和界面逻辑，进一步提升程序的通用性，需要可以在前后台程序之间传递仿真模型的数据格式；为了保持程序的可拓展性，需要导出平台无关的“可交换数据”；为了存储模型到文件或者数据库中，需要模型的可持久化、可序列化版本。

这都要求我们对外提供一个统一的数据结构。通常，“在软件中添加新的层次就可以解决大多数问题”，通过在仿真程序中添加模型数据的解析程序，用于构建仿真模型并调用下层的仿真服务，就可以从程序中分离出模型数据。

3.3 节和 3.5 节指出，对于 web 程序而言，一个首先被考虑数据通讯方式是 json，因为它简单且能同时被前后端所理解。励磁模型表示为 json 如代码 5-12 所示，其中，config 是仿真计算的配置，components 是块组件的集合，lines 是连接线的集合。使用这种格式，我们就满足了所有要求。可以注意到，通过拓展 Block 的继承体系，

我们新增了示波器（scope）模块，用来记录其他模块的输出。

代码 5-12 励磁模型的 json 表示方法

```
var jsonModel = {
  "config": {
    "type": "fixed",
    "T": 0.01,
    "t": 0.0,
    "tt": 10
  },
  "components": {
    "s1": {
      "type": "step"
    },
    "j1": {
      "type": "joint",
      "lines": {
        "l1": "+",
        "l10": "-"
      }
    },
    .....（省略重复的配置）
    "b7": { //示波器模块
      "type": "scope"
    },
    "lines": {
      "l1": [
        "s1",
        "j1"
      ],
      .....（省略重复的配置）
      "l11": [
        "b5",
        "b7"
      ]
    }
  }
}
```

3.5 节指出，可以采用 groovy 编写一个上层的仿真驱动程序，来将 json 模型解析为 java 模型，并完成 config 所规定的仿真。其计算步骤为：

- （1）解析 json 为相应的 groovy 映射（见 3.5 节表 3-2 和代码 3-8）；
- （2）调用 java 程序，初始化仿真模型（见 5.6 节）；
- （3）验证模型，并计算（见项目源码）；
- （4）输出结果到用户界面（见 6.3 节）。

华 中 科技大学硕士学位论文

代码 5-13 演示了如何将 json 模型转换为 groovy 映射模型。代码 5-14 演示了将 groovy 映射模型进一步转换为 java 模型，它使用了工厂模式。代码 5-15 开发了基于 groovy 的上层驱动程序，它接收输入并使用下层服务完成仿真计算。

代码 5-13 json 模型转换为 groovy 映射

```
def model=new JsonSlurper().parseText(jsonModel)
```

代码 5-14 groovy 映射模型转换为 java 模型

```
class BlockFactory{
    static def create(Map info){
        def b
        switch(info.type){
            case 'step':
                b=new StepSource();break;
            case 'joint':
                b=new Joint();break;
            case 'inertia':
                b=new Inertia().config(info.k,info.t);break;
            case 'amplifier':
                b=new Amplifier(info.k);break;
            case 'limiter':
                b=new Limiter(info.upper,info.lower);break;
            case 'scope':
                b=new Scope();break;
            default:
                throw new IllegalArgumentException('no such
                    model');break;
        }
        return b;
    }
}
```

代码 5-15 仿真驱动程序

```
class Simulator {
    Config config
    Map<String,Block> components=[:]
    Map<String,Line> lines=[:]

    def initSystem(Map model){ // 解析仿真模型
        config=new Config() // 仿真配置
        config.config(model.config.T,
            model.config.t,model.config.tt)

        model.components.each{ // 仿真模块
            components[it.key]=BlockFactory.create(it.value)
        }

        model.lines.each{ // 连接线
```

```
def start=components[it.value[0]]
def end=components[it.value[1]]
def line=new Line(start,end)
lines[it.key]=line
if(end instanceof Scope){
    line.push(0,config.T)
}
}
model.components.each{ // 连接点
    if(it.value.type=='joint'){
        def joint=components[it.key] as Joint
        it.value.lines.each{
            joint.addLine(lines[it.key],
                it.value as char)
        }
    }
}

components.each{
    if(it instanceof Inertia){
        it.setConfig(config)
    }
}

def simulate(){
    config.iterate{i,k->
        lines.each{
            it.value.push(i,k)
        }
    }
}

private def checkValidation(){
    // TODO 验证模型是否合法
}

private def adjustLine(){
    // TODO 调整连接线的顺序以符合计算对顺序的限制, 见 5.10 节
}
}
```

至此, 重复的声明语句转移到了模型数据之中, 可以以编程的方式进行解析、传输和存储, 程序的调用也得到了极大的简化。同时, 程序对于既有的架构的修改是封闭的, 而对于添加新的模型是开放的, 这符合了 OCP (开闭原则) 的要求; 仿真程序不再依赖于 Block 实现类, 而是依赖于 Block 接口, 符合 DIP (依赖倒置原则) 的要求。满足上述面向对象的基本原则, 程序可以在不同的层次上进行复用和

拓展（7.1 节）。

但是，作为最终用户的调用接口，程序还不够简便，需要隐藏内部建模的过程，消除掉使用的复杂性。为了完成这一目标，下一章将进行程序 GUI（Graphic User Interface，图形用户界面）的开发。

我们将会看到，得益于对仿真业务的适当分层和分解，GUI 将表现出独立而又相似于后台程序的特征：从联系上看，两者之间仅通过 json 模型进行交互；从结构上看，二者的层次、组成基本一致。我们还将看到，近年来流行的计算机辅助可视化设计软件，是如何通过小增量迭代，最终演化而成的。

5.10 迭代的计算顺序

在实现仿真算法时（5.4 节），提到了对计算顺序的限制。在计算过程中，应当从电源出发，按照连接线的连接顺序一步步改变仿真块的状态。这引出了本节讨论的算法。出于实现上简单的考虑，只讨论单电源仿真系统的情况。

代码 5-16 记录连接线的 LinkedHashMap

```
def lines=[
    11:['s1', 'j1'],
    12:['j1', 'b1'],
    13:['b1', 'b2'],
    14:['b2', 'j2'],
    15:['j2', 'b3'],
    16:['b3', 'b4'],
    17:['b4', 'j2'],
    18:['b3', 'b5'],
    19:['b5', 'b6'],
    110:['b6', 'j1'],
    111:['b5', 'b7']]
```

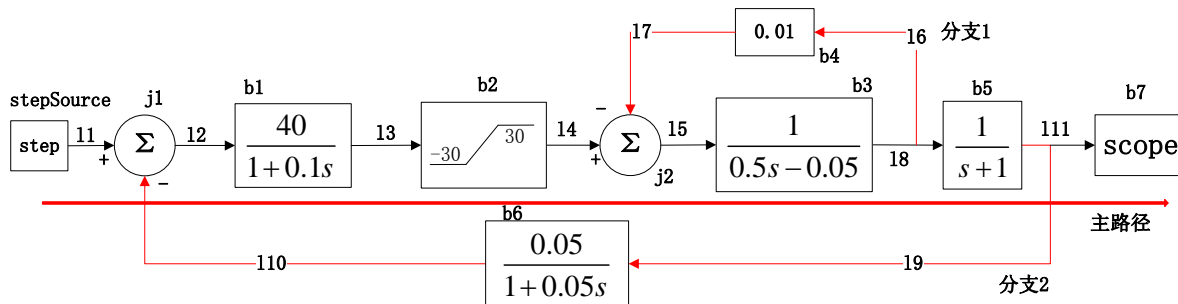


图 5-19 合理的运算顺序

以图 5-17 所示的励磁模型为例，代表其连接线的 groovy 结构如代码 5-16 所示。

连接线“11”包含了电源“s1”，因此对 lines 的遍历应该从“11”出发，可以得到一个合理的计算顺序。如图 5-19 所示，三条计算路径已经在图中标明。在仿真迭代中，应该首先更新主路径上仿真块的状态，然后分别更新分支 1 和分支 2 上仿真块的状态，可以满足计算的要求。算法如（伪）代码 5-17 所示：

代码 5-17 调整算顺序的算法

```
输入：lines
输出：满足计算顺序的 lines

算法：找到电源所在的 line；

function findLine:
    寻找相连的 line;
    如果找到分叉的 line:
        分裂得到多条 line;
        跳转到 find line;
    否则:
        继续寻找直到没有相连的 line 为止
end findLine

从找到的 lines 中取出最长的一条作为主线路;
Function findRemain:
    对每条剩下的 line:
        调用 findLine 得到多条分支;
        从中取得最长的一条;
        如果没有剩余的 line:
            结束
    否则:
        跳转到 find remain 继续寻找;
end findRemain

按照主线路、分支 1、...、分支 n 的顺序组织 lines
返回 lines;
```

5.11 本章小结

在本章中，我们演示了如何从最初试验性质的代码开始，逐步通过小增量迭代的方式，构建仿真软件后台程序的方法。围绕仿真这一主题，程序历经从过程化到对象化的转变过程，从命令式编程建模到分离的 json 模型的迁移过程。遵循基本类似的开发方式，下一章将简要介绍仿真程序 GUI 的开发。

6. 仿真程序 GUI 开发简介

计算机懂得程序和数据结构，人懂得文字、图片、声音。开发应用程序的 GUI，本质上就是将数据在计算机和人之间进行表示、传输、处理和转换。在上一章中，模型分别被表示为了 java 代码（硬编码）、groovy 映射（脚本）和 json 模型（数据结构）三种形式。其中，json 正是独立于编程语言的、web 前端程序可以理解的形式。采用适当的图形表示，我们就可以将模型展示给用户，并接收他的输入，接着传递给仿真程序进行计算，最后向用户展示结果。

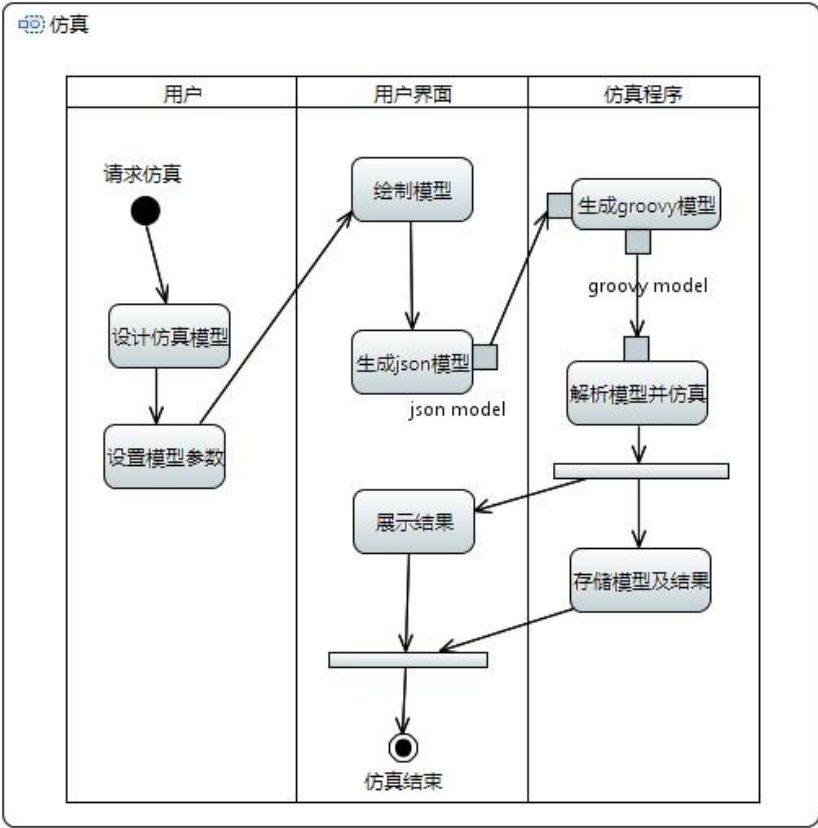


图 6-1 仿真的活动图

图 6-1 中标明了仿真活动的参与角色和各个角色的动作。用户需要可视化地设计模型并设置其参数；用户界面按照 5.9 节中的约定生成 json 数据模型并发送给后台程序；后台解析并进行计算，采取必要的处理，最后将仿真结果发回到用户界面并展示。

华 中 科技大学硕士学位论文

和 simulink 相似, demo 程序提供可视化的仿真模型搭建界面。对于 web 技术而言, 从新实现基础的绘图元素, 要熟悉很多 html 页面绘制图形的底层接口, 并自己编写绘图框架, 这是一个复杂的过程。为此, demo 程序采用 javascript 插件 snap.svg 作为绘图 api, 操作网页中的底层<svg>元素, 对仿真模型进行绘制, 并通过 javascript 对象封装对每个模型的配置和向 json 模型的映射过程。此外, 由于有对数学公式绘制的需求, 采用 MathJax 插件生成 svg 格式的数学公式。

svg (可缩放矢量图) 是使用遵循 xml 的语法描述二维矢量图形的一种数据格式。svg 底层支持了常见简单图形的绘制。并支持渐变、阴影等图形特效。还提供动画的操作。代码 6-1 和图 6-2 展示了使用 svg 表示从笛卡尔坐标(0, 0)到(50, 50)一条线段。

代码 6-1 svg 中的简单线段

```
<svg width="200" height="200" style="border:solid 1px red">
  <line x1="0" y1="0" x2="50" y2="50" stroke="black"/>
</svg>
```

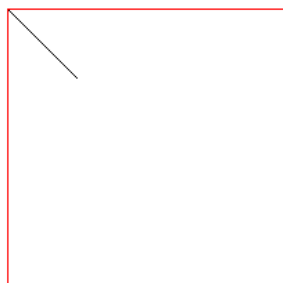


图 6-2 svg 中的直线

6.1 学习测试不可或缺

程序员的业务领域不知凡几, 涉及的技术更是不胜其数。程序员的工作, 虽然归根结底可以看做使用操作系统或者编程语言提供的接口调用底层的硬件功能, 但掌握不同风格的调用接口、调用风格、api 架构等, 需要不断吸收新的知识。学习测试就是程序员掌握新技术的利器。正规的学习测试通常采用测试框架, 利用其断言技术测试程序的边界条件是否满足要求。在对 svg.snap 等 web 前端插件的学习中, 可以依靠脚本代码的快速响应特点, 边修改代码边查看网页的变化。

华 中 科技大学硕士学位论文

在仿真项目的 `snapTest` 和 `mathjaxTest` 文件夹中，分别演示了对两种插件的学习测试。关于 `snap.svg` 和 `mathjax`，可以通过官方网站查询相关资料。图 6-3 和图 6-4 是对这两个插件的简单应用



图 6-3 snap 绘制的基本形状

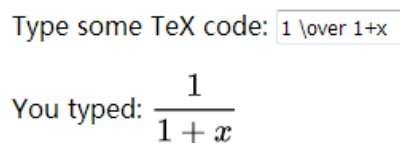


图 6-4 在网页中输出数学公式

在 `demo` 程序的 GUI 的长达两个月（2017.1.16至 2017.3.16，除去双休日，每日约三小时）开发过程中，学习测试占用了约 60%的时间。通过不断测试 `snap` 插件和 `mathjax` 插件提供的功能、特性，逐渐掌握他们的使用方式，为 GUI 的开发打下了基础。可见，学习测试至关重要。编写程序不同于推导学术理论和发明新式算法，需要实事求是，亲身实践，反复验证，“没有调查就没有发言权”。

6.2 前端程序的基本结构

在网页开发中，前端的代码一般采用过程式的风格。通过函数封装一个个基本的回调代码，给按钮、表单等界面元素添加事件响应等动态行为。`jQuery` 框架将这一基本的开发过程封装到极致^[30]，极大地方便了前端程序员对 `dom` 的操作。但是，对于框架程序的开发而言，分层分块的结构显得更加清晰和便于管理。`javascript` 并不从语言层面直接支持面向对象的开发范式，而是支持原型链（`prototype chain`）式的继承^[31, 32]。为了方便程序的拓展，在 GUI 代码中引入了如图 6-5 所示的原型式继承结构。

其中略去了对 `snap` 的依赖。可以看到，这个结构和后台的继承体系（图 5-12）极为相似。需要注意的是，前端的模型仅仅封装了参数和绘制逻辑，只负责程序的“展示部分”。`Block` 之下是和后台相对应的仿真模块，它们通过实现 `Block` 规定的接口对可视化建模提供支持。`Line` 代表了 `Block` 之间的连接线，并记录了仿真模型的拓扑结构。`Model` 对 `Block` 和 `Line` 进行管理，包括调用 `snap` 进行绘制和模型的 CRUD（Create, Retrieve, Update, Delete）操作，以及导出 `json` 格式的模型。

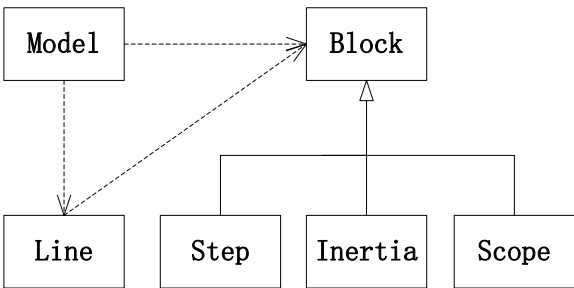


图 6-5 GUI 的程序结构

6.3 程序的用户界面

在长期的学习测试的基础上，通过不断引入新的特性，搭建仿真程序的可视化建模界面。最终得到界面的 demo 版本如图 6-6 所示。在程序的后台 web 层，基于 spring mvc 框架编写代码 6-2 所示的控制器。使用代码 6-3 访问该控制器，并传入生成的 json 模型，就可以运行仿真程序。

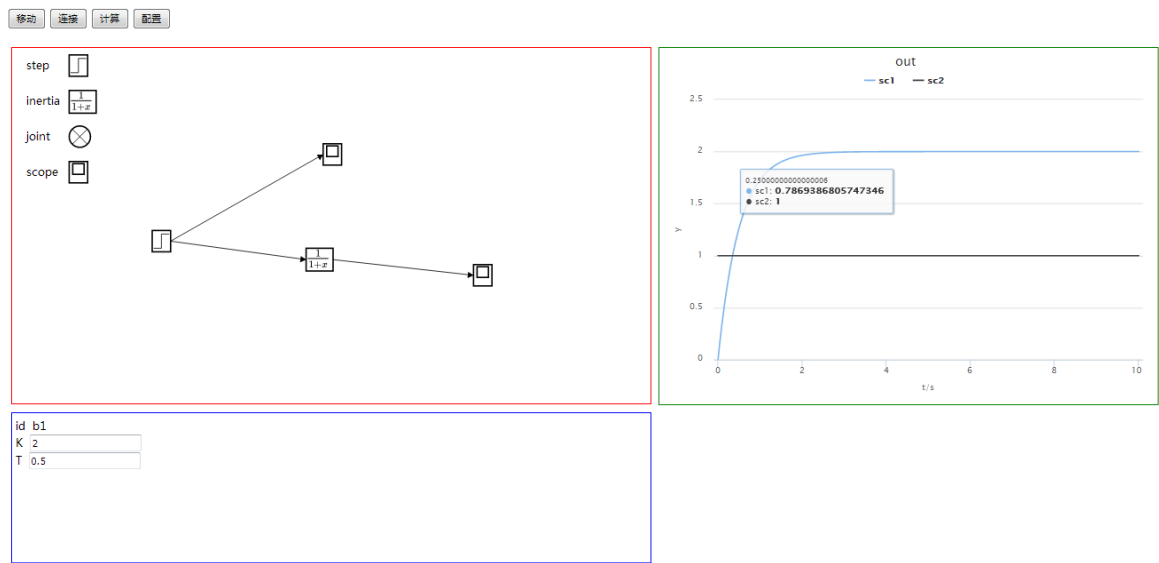


图 6-6 GUI 的 demo 版本及运行结果

代码 6-2 基于 spring mvc 的后台控制器程序

```
@Controller
public class SimulationController {
    @ResponseBody
    @PostMapping(path = "/simulate")
    @SuppressWarnings("unchecked")
    public LineData simulate(
        @RequestParam("model") String jsonModel) {
        JsonSlurper js = new JsonSlurper();
        Object obj = js.parseText(jsonModel);
```

华 中 科技大学硕士学位论文

```
Simulator simulator = new Simulator();
simulator.initSystem(Map<String, Object>) obj);
simulator.simulate();

Map<String, Scope> scopes =
    (Map<String, Scope>) simulator.findOutputs();

LineDataBuilder ldb =
    LineDataBuilder.createBuilder("out", "t/s", "y");
double[] time = (double[]) simulator.getTime();

scopes.forEach((name, scope) -> {
    ldb.addSeries(name, time,
        LangUtil.toPrimitiveDoubleArray(scope.getData()));
});
return ldb.build(); // 返回 json 数据供前端绘图
}
}
```

代码 6-3 调用后台控制器运行仿真程序

```
var url='/Simulation/simulate.action';
var jsonModel=model.toJsonModel();
$.post(url,{model:jsonModel},function(data){
    // code to draw result chart
});
```

控制器代码 6-2 中混入了较多的逻辑，包括解析 json 模型、创建仿真器、解析仿真结果等，这违反了 SRP（单一职责原则），但是，由于控制器处于代码层次结构较为顶层的位置，虽然引入了很多其他依赖，但是本身被依赖的地方较少（只被前端代码依赖），可以放心地修改，而不会对其他模块产生较大的影响，因而可以暂时忍受这段冗长的代码。此外可以看到代码中的 @SuppressWarnings 注解，由于 java 调用 groovy 时存在的类型问题，编译器警告程序中含有较多的“未经检查的类型转换”，这是容易出现类型安全问题的地方，编写代码时应予以关注。groovy 调用 java 代码时也存在类型相关的问题。

下面结合图 6-6，说明程序的运行过程。首先，在左上角的区域（程序中为红色方框）中，拖拽式地建立了一个简单的惯性环节，并为阶跃源和惯性环节添加了示波器。然后单击“配置”按钮，点选惯性环节，可以看到在左下的区域（程序中为蓝色方框）中显示了惯性环节的配置信息，此处，为其设置 $k=2$ 的放大倍数和

$T=0.5$ 时间常数。在程序中，采用默认的仿真配置，即采取 10s 的总仿真时长和 0.01s 的仿真步长。最后，点击计算按钮，运行仿真程序。

在右上角区域（程序中为绿色方框）所示的结果显示模块中，程序解析并使用 highcharts 插件绘制了阶跃电源和惯性环节的输出。在开环状态下，惯性环节输出最终的稳定值为 2，在 $t=0.5s$ 时，输出达到了稳定状态的 0.707 倍。

值得一提的是，绘制曲线所使用的 highcharts 插件，其内部也使用了 svg 的技术，只不过无论是从图形精美程度还是功能完善程度上来看，都远胜于我们 GUI 的 demo 版本☺。

6.4 一次性脚本代码的使用

时间回到 19 世纪末，当时的程序还存在于穿孔卡上，编程就是打孔。这种打孔程序是真正的“一次性程序”。至今，这样的一次性程序还有其价值。就像 shell 脚本一样，通过一个解释引擎，将输入的“文本卡带”解释为程序执行，这样的编程方式给程序带来了极大的便利^[33, 34]。类 Unix 操作系统的“小内核”、“小程序”给业界带来了太多惊喜。python 和 Unix 类似，从设计之初就奉行“简洁至上”的信条^[35, 36]。让程序员欣喜地发现短短代码却可以同时做到简洁易懂和功能强大。

在 windows 操作系统中，软件基本朝着集成化、多功能化的方向发展。以 eclipse 集成开发环境为例，不仅自身支持基本的 java 程序开发环境，还以插件的方式对 spring 框架、tomcat 服务器提供窗口化的支持。相比于这种瑞士军刀般的 windows 程序，Unix 及 Linux 等系统则支持另外一种工作方式：即所有的程序都是小巧的。一个程序一般只完成一类特定的任务，并且考虑各种情况来做到尽善尽美。在使用中，一个任务被分解为多个过程，每个过程由一个“小程序”负责，用户数据在这些程序组成的“管道”之间传递。究竟哪种方式合适见仁见智。但是，作为程序员，应该抵御国内市场现状的影响，尝试接受“Unix 式”的处理方式。这更符合程序员的思维方式，因为程序员一段时间总是只专注于一个任务。此外，git、maven、gradle、svn、jetty、derby 在实现上都是支持这种处理模式的。程序员不能被界面蒙蔽了双眼而看不到实质的逻辑。目前流行的 android 操作系统^[37]，基于一

华 中 科技大学硕士学位论文

种 linux 的内核而构建，用户的操作可以被专门的程序响应，不难注意到，这就是内核（kernel）特征的外在表现。

目前，Linux 操作系统默认安装了 python，以提高脚本程序的开发效率。在 demo 项目中，编写前台程序的时候，为了获得程序更加迅速的响应，除了避免使用服务器页面（如 jsp）之外，还采用了 notepad 文本编辑器编写 html 测试页面和 js 脚本。这样，当程序编写完成以后，需要将程序部署到项目之中。此外，为了节省网络流量，还需要将 js 文件以一定的顺序压缩，去掉多余的空格、换行符、注释等元素。这两个任务都可以使用 python 来方便的完成。模拟 maven 构建项目的各个阶段，可以将这两个任务划分为 clean、prepare、compress、publish 四个阶段。如代码 6-4 所示。

代码 6-4 使用 python 完成代码的压缩和部署

```
from pack import compress
import shutil

import logging

cwd = Path('.') # 当前目录
basedir = cwd.absolute().parent # 工作目录
snapdir = basedir.joinpath('snapTest')
outdir = cwd.joinpath('out') # 输出目录
logging.basicConfig(level=logging.INFO) # log

def clean(): # 清理产生的 python 缓存代码
    for f in cwd.iterdir():
        name = str(f.name)
        if 'cache' in name.lower():
            for ff in f.iterdir():
                os.remove(str(ff))
                print('deleting {}'.format(str(ff)))
            f.rmdir()

def prepare(): # 生成输出文件夹
    if not outdir.exists():
        outdir.mkdir()

def compress_snap(): # 压缩 js 文件
    files=['block.js', 'blockconfig.js',
'amplicifier.js', 'homopoly.js',
'integrator.js', 'line.js', 'model.js']
    jsfiles = [str(snapdir.joinpath(file)) for file in
files]
```

华 中 科技大学硕士学位论文

```
js_out_file = str(outdir.joinpath('snap-util.js'))
compress(jsfiles, outfile=js_out_file)

def publish(): # 复制到项目文件夹 util 下
    for f in outdir.iterdir():
        if f.is_file:
            logging.info('publishing %s', str(f))
            js_pub_dir = basedir.joinpath(
'Simulation/src/main/webapp/js/util')
            shutil.copy(f, js_pub_dir)

if __name__ == '__main__':
    clean()
    prepare()
    compress_snap()
    publish()
    input()
```

可以看到，使用 `python` 进行目录和文件操作十分简便。`python` 支持的列表解析使迭代的语法更加的轻便。另外，这类代码与项目的主体并不冲突，只是定制地使用了操作系统的功能，因此编写的时候不用处理恼人的依赖，完成任务以后，这些代码也没有维护的需要，丢掉即可。这类代码更多的应该是 `Unix` 与 `Linux` 系统管理员的一种福利^[38]。

6.5 对程序开发的思考

作为程序与用户直接交互的一层，前端的用户界面直接表达了开发团队对于需求的理解，良好清晰、制作精良的界面会直接为最终的软件产品加分，在瞬息万变的市场中，这一点尤其重要。但是，界面依托于后台业务逻辑而存在，界面再炫酷，后台逻辑混乱，思路不清晰，也无济于事。这就是所谓内部质量决定外部质量。所以，在项目初期，必须仔细研究、求证需求，以保证业务的完成。敏捷方法主张使用用户故事来促进和客户、用户的面对面交流^[39]，以澄清对需求理解上的模糊之处。

在敏捷开发中，拥抱变化，代码能实时地响应需求的变化，即使在临发布时，也是欢迎变化的。要达到这种要求并不容易。在传统的开发过程之中，任务的蛋糕被横着切为数层，上下层“分开来吃”，各层之间通过接口进行交涉，交流成本较大。而敏捷的蛋糕竖着切，每个成员既能独当一面，还要“吃完自己的”以后考虑“尝尝别人的”，这谈何容易。首先，绝对不是随便来个人编程就可以的，那种培

训班三个月出师的就不行。敏捷团队人员少而精，首重协作，其次素质，其次奉献。敏捷绝不仅仅是一种开发方法，更是对精神品质的要求。

在开发的过程中，敏捷鼓励结对编程的方式。文献^[40]描述了以这种方式开发保龄球计分程序的过程。在编程过程中，结对的二人紧密合作，积极交流，互相启发，增加的效率远超过结对所消耗的人力。敏捷开发的现场可以是一对对结对编程的伙伴热烈讨论交流的场景，但绝不应该是为了便于监督而强制凑合起来的圆桌会，也绝不应该是集合所有团队成员走过场式的问责会。这两者只会成为团队士气、效率和人力资源的杀手。程序开发是一种创造性活动，程序员用代码刻画了自己的严密思维和精彩逻辑；又是一种高度的实践，需求和代码本身都来源于对生产实际和计算机技术的实践。那么这种活动需要些什么呢？也不多：一桌、两椅、一台过得去的电脑、以及专心致志的两人而已。

现代的会议应该是敏捷的会议。敏捷的会议，应该显得敏捷。Scrum 敏捷方法^[41-43]提倡四个会议，并且为了保证程序员的工作环境，除此以外的会议尽量不在团队全员中展开。其中，迭代计划会、评审会和回顾会一般每个迭代过程只开一次，总用时不超过八个小时（一次迭代周期一般是 4 周）。例行的每日站会不超过 20 分钟。会议不应该是老生常谈，也不该是夸夸其谈，更不该流于形式。

由于缺乏有效的团队工作实践，本文不可能作过多的探讨。但是，置身于敏捷团队之中，所有成员怀揣同一个目标，团队之于每个人，如同膀臂之于手指，挥洒自如。作为敏捷团队的一员，应该足以自豪吧！

6.6 本章小结

本章简要介绍了 demo 项目 GUI 的开发过程，并针对一个简单的惯性环节演示了系统的使用。到本章为止，整个 demo 项目的开发就接近尾声了，因此，6.5 节从开发过程出发对软件开发过程作了一些思考。

在第七章中，将把 demo 程序用于 4.5 节的线性水轮机模型的仿真之中。此外，还将适当拓展程序，讨论如何利用 demo 程序的仿真架构解决上述模型的参数辨识问题。

7. 仿真程序的应用

本章的讨论基于 4.4 节中给出的仿真模型。为了阅读的方便，重新绘制为如图 7-1 所示的模型。

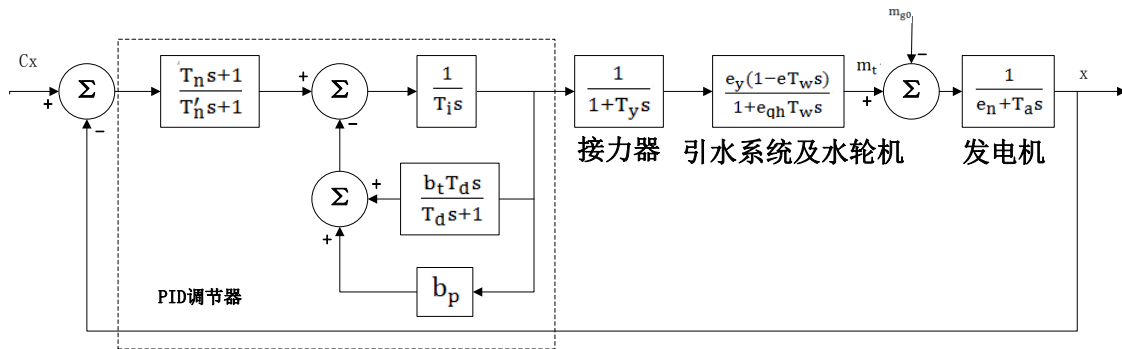


图 7-1 线性化的水轮机调速系统仿真模型

7.1 仿真软件的拓展

为了满足对水轮机调节系统线性化模型仿真的要求，需要对第五、六章讨论的仿真模块进行拓展。观察图 7-1，可知尚需要积分器、放大器和一次齐次式，才能构建起仿真模型。可以将这三种仿真模块抽象为图 7-2 至 7-4 所示的三种元素，并且具备重用的价值。



图 7-2 积分器



图 7-3 放大器

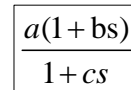


图 7-4 一次齐次传递函数

当拓展仿真软件时，需要拓展后台的基本仿真模块和 GUI 的绘制模块。为简单起见，选择放大器的拓展代码作为实例，模型及界面的分别如代码 7-1 至 7-2 所示。

代码 7-1 放大器的拓展程序

```
public class Amplifier
    extends BaseBlock implements LinearBlock {
    double k;

    Amplifier(double k) {
        this.k = k;
    }
}
```

华 中 科技大学硕士学位论文

```
        @Override
        public void next(double input) {
            next = k * input;
        }

        @Override
        public void setInitValue(double initValue) {
            next(initValue);
            moveOn();
        }
    }
}
```

代码 7-2 放大器界面元素的拓展程序

```
var Amplifier=Block.plugin(function(Block,Snap){
function Amplifier(){
    this.id="amp"+(++_idx); // 内置的 id
    this.block=Block.amplifier.use().attr({id:this.id});
    this._rect=Block.amplifier.select("rect");
    this._k=1.0; // 放大系数
    this.type='amplifier'; // 代表类型的字符串
}

Amplifier.prototype=new RectangleBase;
Amplifier.prototype.constructor=Amplifier;

var _idx=0,
    proto=Amplifier.prototype,
    config=Block._.config;

Block.createAmplifier=function(){
    return new Amplifier;
};

Block.amplifier=null;

Block._predefAmplifier=function(svg){ // 绘制方式
    var paper=svg.paper;
    var rect=paper.rect(0,0,40,32)
        .attr({stroke:'black',fill:'white',strokeWidth:2});
    var txt=paper.text(13,20,'A');
    var g=paper.g(rect,txt);
    Block.amplifier=g.toDefs();
};

proto.toModel=function(){ // 转换为 json 模型
    return {type:this.type,k: this._k};
};

proto.getConfig=function(){ // 配置，在界面上显示模型的参数
    var configs[];
```

```

        configs.push(config('id','id',this.id,
                               configTypes.TEXT_TYPE));
        configs.push(config('_k','k',this._k,
                               configTypes.INPUT_TYPE));
        return configs;
    };

    proto.updateConfig=function(){ // 更新模型参数
        var configs=this.config.configs;
        this._k=configs[1].value;
    };

    return Amplifier;
});

```

可以看到，得益于前面章节的设计，对后台模型的拓展代码 7-1 显得十分简便。只需要拓展相应的接口（如 **BaseBlock**）或原型链（如 **RectangleBase**），并实现父类的契约中限定的方法就行了。对于如图 7-4 所示的较为复杂的一次齐次线性传递函数，在实现中将其分解为若干简单模型的组合。现简单推导如下。

$$\frac{a(1+bs)}{1+cs} = a \times \frac{\frac{b}{c} + \frac{b}{c} \times cs + 1 - \frac{b}{c}}{1+cs} = a \times \left(\frac{b}{c} + \frac{1 - \frac{b}{c}}{1+cs} \right)$$

于是，图 7-4 所示的模块就可以转化为图 7-5 所示的等价的复合模块，其中每个模块都已经被实现。这样，所有的问题都回归到 4.3 节所讨论的积分和惯性环节这两个基本模块之上了，简化了程序的实现方式。具体的实现参考 demo 项目的源代码。

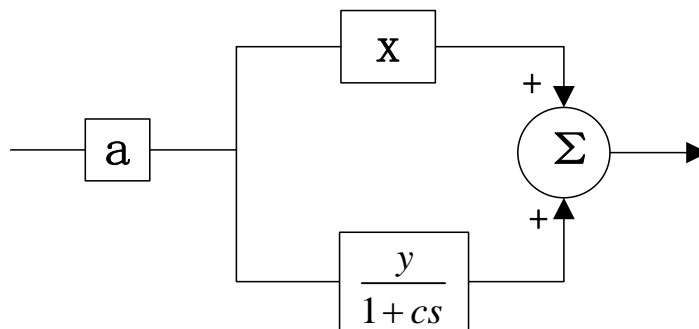


图 7-5 一次齐次式模块的等效模块，其中 $x = \frac{b}{c}$, $y = 1 - \frac{b}{c}$

7.2 调速系统暂态仿真

仿真模型如图 7-1 所示，参考文献^[25]的处理方式，选取各个参数如下面的清单

所示。

清单 7-1 水轮机线性模型参数设置

```
% 设置调速器的参数

bt = 0.8                % 暂态转差系数
Td = 3.36              % 暂态反馈时间常数
Tn = 0.0                % 微分时间常数
bp = 0.0
Ty = 0.2
Ti = 0.05
% 机组及引水系统参数

Tw = 1.0                % 水流加速时间常数
Ey = 1.0
Eh = 1.5
Eqy = 1.0
Eqh = 0.5
Ta = 5                  % 机组惯性时间常数
En = 1.0                % 机组自平衡系数


$$E = \left( E_{qy} * \frac{E_h}{E_y} \right) - E_{qh} = 1.0$$

```

使用仿真 demo 程序对模型进行仿真，并和文献^[25]所采用的 matlab 模型（如图 7-6）仿真的结果进行对比，其结果如图 7-7 至 7-11 所示。

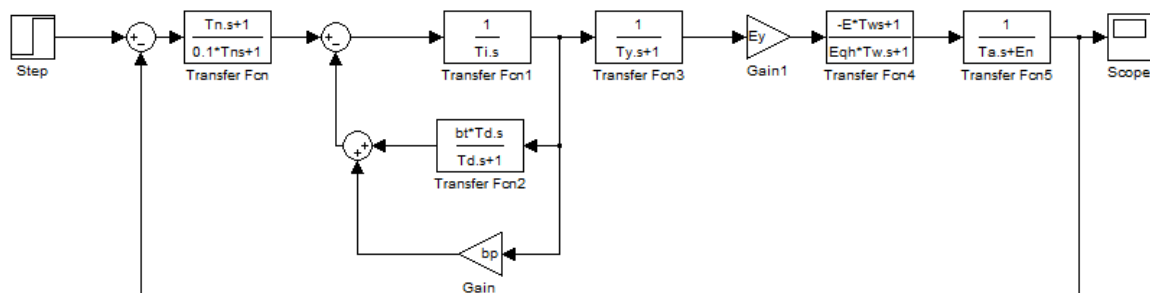


图 7-6 simulink 中建立的调速系统模型

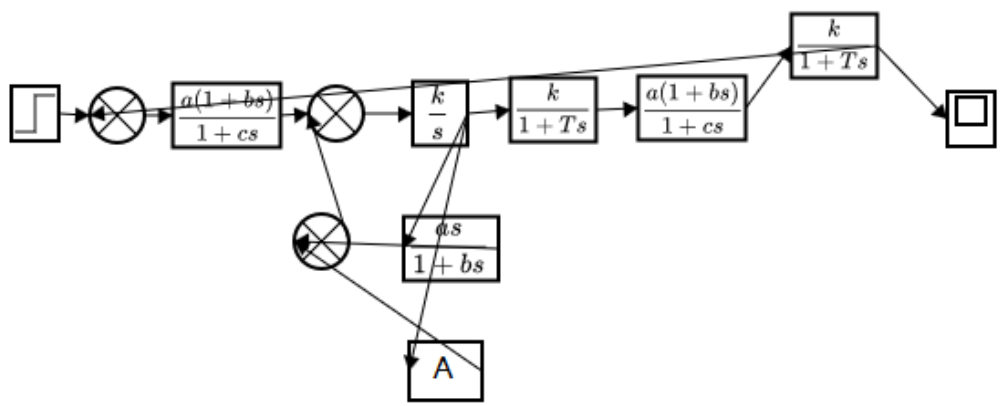


图 7-7 demo 程序中建立的调速系统模型

id	hm1_1
a	2.688
b	3.36

图 7-8 $\frac{as}{1+bs}$ 模块的配置

id	hm2
a	1
b	-1
c	0.5

图 7-9 右侧 $\frac{a(1+bs)}{1+cs}$ 模块的配置

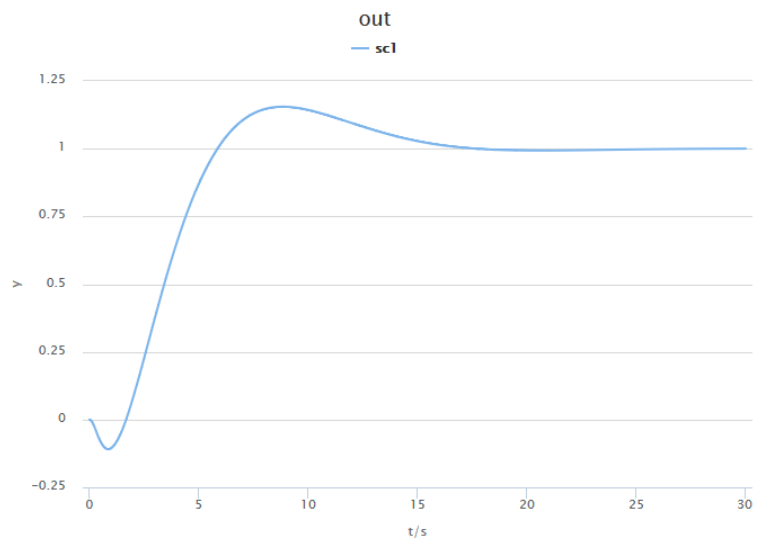


图 7-10 demo 程序的输出

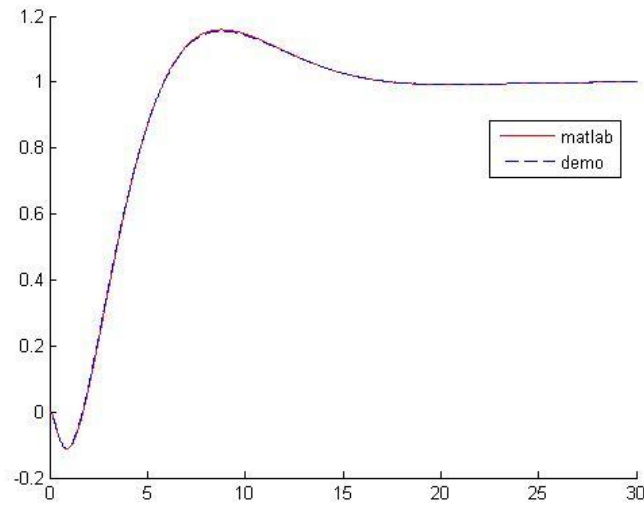


图 7-11 demo 程序和 simulink 程序输出的对比

7.3 调速系统参数辨识

本文采用引力搜索算法^[44]（Gravitational Search Algorithm, GSA）对水轮机调速系统进行参数辨识。本节中所有的程序都运行于“计算机 2”（表 3-1）之上，“计算机 2”的性能较优，关于性能的讨论也都基于这台机器的配置。

7.3.1 引力搜索算法

GSA 和 PSO 算法^[45]、GA 算法^[46]类似，属于给定求解范围的全局随机搜索算法。这类算法可以提炼出一个通用的调用接口，如代码 7-3 所示。函数接口 **NaturalLaw** 用于计算一组坐标的适应度，**Range** 类指定一个维度上的搜索范围并且在该范围内生成随机坐标。粒子群算法（PSO）、遗传算法（GA）也可以统一到这个接口上来。这样便于使用多态特性和策略模式对代码进行统一。

代码 7-3 全局随机搜索算法的一种通用接口

```
public double[] search(NaturalLaw nl, List<Range> ranges);

@FunctionalInterface
public interface NaturalLaw{ // 计算适应度的函数接口
    double judgeFitness(double[] cordinate);
}

public class Range{ // 代表搜索范围的类
    double lower;
```

华 中 科技大学硕士学位论文

```
double upper;

// other method omitted.
}
```

借助这样一个统一的接口，并假设求解极小值，计算 $\sqrt{3}$ 的代码变得十分简洁，如代码 7-4 所示，对 $\sqrt{3}$ 的计算转换为了求解到 0 的距离为 $\sqrt{3}$ 的点。

代码 7-4 使用搜索算法搜索 $\sqrt{3}$ 在区间[0, 5]的根

```
double[] result = search(coordinate -> {
    double root = cordinate[0];
    return Math.abs((root * root - 3));
}, Arrays.asList(new Range(0, 5)));
```

在 demo 项目的 `hust.hx.algorithm.gsa` 包下，包含了经典 GSA 算法和一种改进 GSA 算法^[44]的实现，两种算法的接口如代码 7-5 所示，代码中提供了对迭代次数 `lifeSpan` 和每一维度的搜索范围 `spaces` 的配置方式，算法的其他相关参数使用默认值即可，具体可参考源代码 `ClassicGSA.java` 和 `WeighedGSA.java`。

代码 7-5 两种 GSA 算法的实现

```
// 经典 GSA 算法的构造方法
ClassicGSA(NaturalLaw law, List<Range> spaces);

// 加权 GSA 算法的构造方法
ClassicGSA(NaturalLaw law, List<Range> spaces);

// 配置方法
public void configure(int lifeSpan, int creatureCount);

// 设置逼近方式，是搜索极大值还是极小值
public void setMode(Mode mode);
enum Mode {
    SMALL_BETTER, BIG_BETTER
}

// 启动计算的方法
public void rockAndRoll();

// 最佳位置和最佳适应度
public double[] bestOne()
public double bestFitness()
```

在项目测试包 `hust.hx.algorithm.gsa` 下，包含了使用 `groovy` 对 GSA 算法的测试，测试文件为 `gsa_tests.groovy`，该程序对典型的测试函数的极值进行了求解，采用的

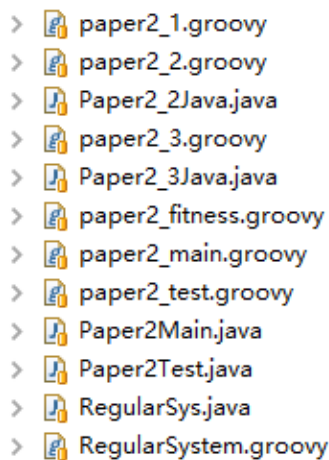
测试方法和文献^[44]相一致。

7.3.2 调速系统的代码实现

这个小节把 7.2 节的仿真结果作为已知条件，对图 7-7 所示系统的进行参数辨识，并选取 b_t 、 T_d 、 T_y 、 T_i 作为待辨识的参数，以迭代输出 out 和仿真结果输出 $real$ 的差之平方和为目标函数计算适应度：

$$fitness = \min \left[\sum_{i=1}^n (real_i - out_i)^2 \right]$$

为了配合参数辨识算法，调速系统仿真模型采用类似代码 5-10 的编程方式实现。所有测试程序代码包含在 demo 项目的测试包 hust.hx.simulation.demo.block 下。如图 7-12 所示，其中，RegularSystem.groovy 和 RegularSys.java 分别代表调速系统的 groovy 和 java 实现，paper2_n.groovy 和 Paper2_nJava.java 代表对 n 个参数的参数辨识。



```
> paper2_1.groovy
> paper2_2.groovy
> Paper2_2Java.java
> paper2_3.groovy
> Paper2_3Java.java
> paper2_fitness.groovy
> paper2_main.groovy
> paper2_test.groovy
> Paper2Main.java
> Paper2Test.java
> RegularSys.java
> RegularSystem.groovy
```

图 7-12 参数辨识的测试代码

如代码 7-6 所示，在测试程序编写的过程中，首先采用了 groovy 实现。groovy 语言的语法更加简洁，强大的 api 和函数式编程带来的效率提升让人无法抗拒。但是，在程序的运行过程中，特别是搜索算法的迭代过程中，groovy 程序带来的性能损失也让人扼腕叹息：鱼和熊掌果然不可得兼。

代码 7-6 辨识程序的 groovy 实现 paper2_3.groovy

```
def fitness(List origin, List output){ // 适应度函数
    def res=0.0
```


华 中 科技大学硕士学位论文

```
        for(int i=0;i<output.size();++i){
            res+=(output[i]-origin[i])**2
        }
        return res
    }

    // 原始参数 (bt, td, ty, ti)
    def rs=new RegularSystem(0.8, 3.36, 0.2, 0.05)
    rs.simulate()
    def origin=rs.output // 原始输出

    def btRange=GsaRange.of(0.001, 1)
    def tdRange=GsaRange.of(0.001, 5)
    def tyRange=GsaRange.of(0.001, 1)
    ClassicGSA u=new ClassicGSA({coordinate->
        def bt=coordinate[0]
        def td=coordinate[1]
        def ty=coordinate[2]
        def sys=new RegularSystem(bt,td,ty,0.05)
        sys.simulate()
        def output=sys.output
        return fitness(origin,output)
    },btRange,tdRange,tyRange)
    u.configure(1000,50)
    TestUtil.timeIt{ u.rockAndRoll(); }
```

不难发现，GSA 每次迭代中计算所有粒子的适应度，容易导致性能瓶颈。使用 java 程序性能监控软件 JProfiler，对 groovy 版本程序的热点（hotspot）进行监测发现，大部分的性能降低来自于算法迭代时使用 groovy 调用 java 代码时编译器生成的“多余代码”。使用代码 5-4 所示的计时辅助函数 timeIt，在 paper2_test.groovy 和 Paper2Test.java 中的进行测试也显示，groovy 运行一次仿真平均需要 120ms，java 则为小于 20ms。在 1000 次迭代，50 个搜索粒子的仿真条件下，这意味着 6000s 和 600s 的区别。

显然，600s 还是无法接受的运行时间啊！为了进一步提高程序的运行效率，减少响应时间，需要引入多线程，以充分利用多核的硬件条件。使用 java 提供的 Executors 多线程框架，改进后的程序如代码 7-7 所示。可以看到，为了使每个子线程具有独立的仿真系统副本，采用了 ThreadLocal 接口，以保证每个线程计算的时候互不干扰。此外，在每次迭代开始时，使用 sys.reset()方法重置仿真参数，以进行新的仿真。这种复用对象的方式，也进一步降低了新建对象带来的性能开销。如果不采取复用的策略，就意味着 1000 次迭代和 50 个粒子会产生至少 50000 次 new 操

华 中 科技大学硕士学位论文

作！最后，需要注意的是，计算结果必须使用新的 List 进行拷贝，不然主线程引用子线程使用的内存，会产生空指针错误。

采用多线程以后的程序变得臃肿难看，但是在运行时的速度提升却是显而易见的：计算最多在 2min 内就可以完成☺。

代码 7-7 采用多线程提升参数辨识的速率

```
RegularSys rs = new RegularSys(0.8, 3.36, 0.2, 0.05);
rs.simulate();
List<Double> origin = (List<Double>) rs.getOutput();

ThreadLocal<RegularSys> tr = new ThreadLocal<RegularSys>() {
    @Override
    public RegularSys initialValue() {
        return new RegularSys(0.1, 0.1, 0.1, 0.1);
    }
};
ExecutorService es = Executors.newFixedThreadPool(4);

ClassicGSA u = new ClassicGSA((coordinate) -> {
    double bt = coordinate[0];
    double td = coordinate[1];
    double ty = coordinate[2];
    List<Double> output = null;

    Future<List<Double>> f =
        es.submit(new Callable<List<Double>>() {
            @Override
            public List<Double> call() throws Exception {
                RegularSys sys = tr.get();
                sys.reset(bt, td, ty, 0.05);
                sys.simulate();
                List<Double> out = sys.getOutput();
                List<Double> res = new ArrayList<>(out); // 拷贝
                return res;
            }
        });
    try {
        output = f.get();
    } catch (InterruptedException | ExecutionException e) {
        e.printStackTrace();
    }
    return fitness(origin, output);
}, Arrays.asList(Range.of(0.001, 1),
    Range.of(0.001, 5), Range.of(0.001, 1)));

u.configure(1000, 50);
u.rockAndRoll();

try {
```

```

        es.awaitTermination(1, TimeUnit.SECONDS);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    es.shutdown();

```

7.3.3 参数辨识的结果

对三个参数 b_t 、 T_d 、 T_y 的辨识结果显示在表 7-1 中。结果基本上是精确的。

表 7-1 辨识的参数选取及辨识结果

参数	准确值	搜索范围	搜索结果	误差百分比
b_t	0.8	[0, 1]	0.8	0.0%
T_d	3.36	[0, 5]	3.360	0.0%
T_y	0.2	[0, 1]	0.200	0.0%
适应度	$\text{fitness} = \min[\sum_{i=1}^n (\text{real}_i - \text{out}_i)^2] = 5.15\text{e-}19$			

但是，对四个参数 b_t 、 T_d 、 T_y 、 T_i 的辨识始终无法收敛，最优适应度曲线（best-fitness ever）始终呈现震荡的趋势，无法收敛。辨识结果如图 7-13 所示。

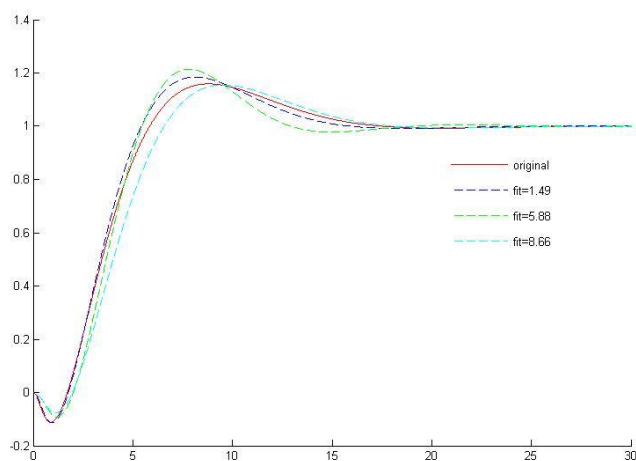


图 7-13 四个参数的辨识结果

8. 全文总结及工作展望

8.1 全文总结

本文从实践出发，以经典水轮机调速系统仿真为切入点，以 java 技术为平台，以 eclipse 作为集成开发环境，探讨了开发工程领域软件的指导思想及可行实践，完成了仿真 demo 项目的设计、开发、测试、使用及拓展，并应用项目的软件制品进行了水轮机调速系统的暂态仿真及参数辨识。

本文主要完成了以下几个方面的工作：

- (1) 结合编程实践提出了一系列程序开发工作中有价值的开发思想和指导原则。并将这一系列的思想及原则运用到 demo 项目的开发之中，优化了开发过程和程序结构，提高了程序的可读性及可拓展性。
- (2) 总结了软件开发过程之中涉及的各个阶段及可选用的技术及开源辅助工具，探讨了选取这些技术及工具的考量，提出了一套 java 平台下可选的环境搭建方案，并且可以被推广到其他的开发平台。在 demo 项目中利用上述环境进行开发，大大提高了开发效率，降低了主要业务以外的其他工作成本。
- (3) 较为深入探索了控制系统仿真软件的 oo 设计方案及实现方法。对比了面向过程和面向对象的仿真程序之间的差别，在此基础上，开发了可读性更强、可拓展性更好的面向对象版本。
- (4) 创造性的使用了基于敏捷的快速迭代开发方法。并通过不断迭代引入新的仿真特性，最终完成了 demo 项目的开发。在开发过程中，还探索了脚本语言 groovy 和 python 在软件开发过程中的使用。针对不同的任务，灵活地使用了适合的语言加以完成，效果良好。
- (5) 进行了仿真程序 GUI 的开发，在界面的实现上引入了一组具备拓展性的继承结构，并在此过程中较为深入的探索了支持这种结构的 javascript 的语言特性，最终为程序提供了类似于 SIMULINK 但是基于 web 技术的、支持可视化建模的用户界面。揭示了可视化建模程序背后的实现原理。

- (6) 应用开发的程序对水轮机调速系统进行了可视化建模和暂态仿真，拓展仿真程序完成了调速系统的参数辨识。进一步演示了程序的使用及拓展。

8.2 工作展望

限于作者的知识和实践水平，本文开发的程序只能称为一个 demo，和真正的商用程序相比还有很大的差距。本文尚存在下述不足之处：

- (1) 本文所采用的仿真模型精度较低，计算量较大。这与采用离散相似法这种线性近似方法有关。采用多点数值积分算法会使这一情况得到改善。同时由于多点数值积分方法具有较高的精度，可以适当地增加仿真步长，从而不仅可以满足精度的要求，也能减少程序的运算次数，并最终从算法的改进出发提升程序的性能。
- (2) 由于时间的限制，demo 程序的 GUI 没有尽善尽美，还存在诸多可以改进的地方。
- (3) 7.3 节中对于四个参数的辨识结果不收敛，出于时间的原因，未能做进一步的研究。

所幸遵循本文论述的一系列原则和方法，通过较多的努力，这些问题都是能够解决的。

华 中 科技大学硕士学位论文

参考文献

- [1] Subramaniam V. Practices Of An Agile Developer. Oreilly Vlg Gmbh & Co, 2006
- [2] Freeman S, Pryce N. Growing Object-Oriented Software, Guided by Tests. Journal of Object Technology, 2009(3)
- [3] Rossum G V. Python Programming Language.In: 1991
- [4] Fowler M. Refactoring: Improving the Design of Existing Code.In: 2002: 256
- [5] Beck K. Extreme Programming explained. 2000: 292
- [6] Object-oriented analysis and design, with applications =. 中国电力出版社, 2003
- [7] Wampler D, Payne A. Programming Scala. Oreilly Vlg Gmbh & Co, 2009
- [8] Bauer C, King G. Hibernate in Action. Guide to Web Development with Java, 2008: 137-184
- [9] Walls C, Breidenbach R. Spring in Action. 中文版, 2005(May)
- [10] 李刚. Struts 2权威指南. 电子工业出版社, 2007
- [11] 温昱. 一线架构师实践指南. 电子工业出版社, 2009
- [12] 蒋鑫. Git权威指南. 机械工业出版社, 2011
- [13] 许晓斌. Maven实战. 机械工业出版社, 2011
- [14] 周志明. 深入理解Java虚拟机. 机械工业出版社, 2013
- [15] 陶国荣. jQuery权威指南. 机械工业出版社, 2013: 132
- [16] 龚建华. JSON格式数据在Web开发中的应用. 办公自动化, 2013(20): 46-48
- [17] 王小强, 程耕国. 基于Ajax和Json的批量数据传递. 软件导刊, 2010(05): 187-188
- [18] Surhone L M, Tennoe M T, Henssonow S F. Apache Derby. Betascript Publishing, 2010
- [19] 韩璞, 罗毅, 周黎辉. 控制系统数字仿真技术. 中国电力出版社, 2007
- [20] 黄莉, 李咸善, 袁喜来. 基于Simulink的水电机组模块化建模与仿真. 水电自动化与大坝监测, 2007(05): 14-17
- [21] 景微娜, 左信. 基于Matlab和Visual Basic仿真软件开发. 系统仿真学报, 2008, 20(6): 1459-1461
- [22] 董锡君, 罗志军, 洪兴昌. 基于Simulink和C/C++混合编程的战术导弹稳定控制系统仿真. 系统仿真学报, 2002, 14(9): 1229-1231
- [23] 董继维, 汪斌, 卢琴芬等. 基于Simulink和VC++混合编程的高速列车牵引传动系统仿真软件. 机电工程, 2011, 28(12): 1519-1522
- [24] 吴中习, 周泽昕. 《电力系统分析综合程序》用户程序接口(PSASP/UIP)的开发和应用. 电网技术, 1996(2): 15-20
- [25] 程远楚, 张江滨. 水轮机自动调节. 中国水利水电出版社, 2010
- [26] Martin R C. Clean Code. Refactoring, 2011, 29(2): 24-25
- [27] Martinowler, 福勒, 徐家福. UML精粹:标准对象建模语言简明指南. 清华大学出版社, 2005
- [28] Cohn M. User Stories Applied: For Agile Software Development. Addison Wesley Longman Publishing Co., Inc., 2004
- [29] Erichgamma, 伽玛, Helm等. 设计模式:可复用面向对象软件的基础. 机械工业出版社, 2007
- [30] 陶国荣. jQuery权威指南. 机械工业出版社, 2013: 132
- [31] Flanagan D F P. JavaScript: The Definitive Guide. 东南大学出版社, 2011: 1-4
- [32] Zakas, Nicholas C. Professional JavaScript for Web Developers.In: 2012: 14
- [33] 贝奇, 陈葆钰. UNIX操作系统设计. 机械工业出版社, 2012
- [34] 史蒂文斯. UNIX环境高级编程. 人民邮电出版社, 2014
- [35] Lutz M, Ascher D, Willison F. Learning Python. 东南大学出版社, 2008
- [36] Beazley D, Jones B K. Python Cookbook. 东南大学出版社, 2014
- [37] 李刚. 疯狂Android讲义. 电子工业出版社, 2015
- [38] Gift N, Jones J. Python for Unix and Linux System Administration. 开明出版社, 2009
- [39] Cohn M. User Stories Applied: For Agile Software Development. Addison Wesley Longman Publishing Co., Inc., 2004

华 中 科技大学硕士学位论文

- [40] Martin R C. Agile Software Development: Principles, Patterns, and Practices. Prentice Hall PTR, 2003: 43-46
- [41] Schwaber K, Beedle M. Agile Software Development with Scrum. PTR, Pages: 158, Year of Publication: 2001, ISBN: 0130676349, 2001, 63(2): 365-396
- [42] Schwaber K. Scrum development process.In: 1995: 117-134
- [43] Rising L, Janoff N S. The Scrum Software Development Process for Small Teams. IEEE Software, 2000, 17(4): 26-32
- [44] 徐遥, 王士同. 引力搜索算法的改进. 计算机工程与应用, 2011, 47(35): 188-192
- [45] 唐俊. PSO算法原理及应用. 计算机技术与发展, 2010, 20(2): 213-216
- [46] Goldberg D E. Genetic Algorithm in Search, Optimization, and Machine Learning. 1989, xiii(7): 2104-2116