封面

摘要

目录

1、绪论

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，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，减少团队的学习、培养成本，甚至可以在很大的程度上增加开发效率。

2.1 约定

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，最好显示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.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。

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，拒绝循环依赖等；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，对调用而言是友好的。

4.不作多余的假设

约定背景：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。作为项目中基础模块的使用者，我们是自己的客户，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。

使用场景：当不是真正清楚自己是否需要某个看起来很炫酷的功能时，尽量不作多余的假设。

约定意义：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Martin Fowler，2001b）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。

良好实践：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，但实际上当前只有一个实现。这个时候可以果断地抛弃留下接口的念头，因为很有可能这个接口再也不会拥有多于一个的实现了。如果保持了简单性，完全可以通过重构来加入新的接口。这一原则可以归结为，尽量做到简单，但又不断绝复杂起来的可能性。

5.可读性很重要

在软件测试领域，有一种开发方法叫做测试驱动的面向对象开发。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）去探测对象的行为。当采用这种开发方法时，写出来的测试就如同自然语言一般，清晰易懂。例如，对于testIllegalLoginWithWrongPasswordAndFailed方法，说明了测试的内容及结果，该方法名就是最好的注释。

由于c语言根深蒂固的影响以及相关材料随意性，很多c风格的代码都难以读懂。著名的循环变量i，j就是该语言发扬光大。当然，现在依旧被广泛使用，但是python语言的发明者显然对这类命名深恶痛绝，直接从语言层面上消除了使用这种for循环的苗头。

汇编语言是机器的最爱，然而高级语言是面向作为开发者的人的，不好的命名和模块化让代码错失了与人交流的机会。在java这类完全面向对象的语言之中，包名是第一个“表述你自己”的机会，类名是第二个，此外是字段名、方法名，最后才是注释（包括类注释、字段注释、方法注释）。好的命名基本上可以做到零注释。值得注意的是，一些需要行内注释的地方，往往都是隐错存在的地方。

可读性的前提是良好的设计，如果类的设计不好，各个结构之间耦合严重，层次不清晰，就意味着可读性必然会受到影响。提高可读性的最好原则是，不放过前述任何一个可以解释程序做了什么的机会。如果能做到即使是局部变量的命名也不轻易的使用tempN的方式，那么代码的维护者（很可能就是你自己）在两个月后将从中受益。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。

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，提高其可理解性，降低其修改成本。在《重构-改善既有代码的设计》一书中，Martin Fowler提出了一系列重构的手法，用于在不改变软件可观察行为的前提下，调整其结构。重构虽然是极限编程下的一员“猛将”，但是基本上所有的开发方法都离不开重构。

在重构的世界里，满足设计模式不是最初设计的指导原则，而是一种重构以后软件应该达到的目标。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，接口的改变都势必影响到程序的可读性和可拓展性。重构的存在可以在关键的时刻挽救逐渐混乱的代码结构，让代码重新满足工程上的最佳实践——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”。代码“坏味道”的存在是程序存在设计失误、编码失误的信号。

重构不是高端的词汇，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 思想

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模块，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。考虑问题的时候，要更多地考虑“这个功能可以由下面的功能组成”，多过考虑“完成这个功能要先这么做，再那么做”。当然，人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，但是我们拥有重构的手法，使我们可以事后再进行“补救”，最终符合模块化的要求。模块化的代码更加可读，更好维护，更易得到重用的机会。

2.面向对象的思想

从形式上，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，就是面向对象。依据面向思想的方法，很容易构建出高内聚、低耦合的程序。

3.函数式编程的思想

函数式编程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格式。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：

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。

数据是不可变的。摒弃对数据的可写性，虽然多了需要四处复制数据的风险，但是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显然更加友好。

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。函数式编程相当于对编程施加了更多的限制。但使得调用更加简便、程序更易阅读和维护。在java 8以前java并不支持函数式编程风格，但在java 8中引入了lambda表达式和函数接口的特性完美支持了强类型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，大大减少了书写模板代码的工作量。

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约定配置，从而减少甚至省略每次使用时的冗余配置。

在软件开发中，可以提前约定的地方包括项目架构的，编程接口等。在基于maven构建的项目中，源代码存在于目录src/java/main中，测试代码位于src/test/main中，就是一种约定。两个目录下的目录结构一直（同名的目录属于相同的包），绕开了java数据可见性的限制，将源代码和测试代码分开组织，方便代码的发布。有关于编程接口的约定，可以提前给用于配置的变量赋默认值。

对于javascript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，由于弱类型的原因，可以使用同一个函数统一进行处理。

当然，约定降低了代码的可读性，在程序中引入“约定优于配置”的代码时，要充分地写好文档注释，并提供测试样例，防止维护时摸不着头脑。

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。

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。如，用于存储的视图、业务计算的视图和展示的视图。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。

7.极限编程

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。极限编程适用于有工程背景的开发环境。当需求不明确或者经常变动时，应该使用极限编程的思想，尽快为客户提供原型，从而验证和调整需求，使软件开发工作符合实际情况。

3、软件环境搭建

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，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程序架构模式，采用了一种架构方式，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，提升了开发的可控性。架构方式如下表所示：

表 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的各个版本。 | 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对象注记法 | 高效的数据传输格式 |
| groovy json | groovy json处理api |  |
| spring+jackson | spring内置json处理api |  |
| 测试 | junit | 单元测试框架 |  |
| matlab/simulink | 用于验证计算的正确性 |  |
| 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 groovy plugin | groovy的eclipse插件 | 负责groovy代码的编译及配置 |
| eclipse maven plugin | maven的eclipse插件 | 集成maven功能 |
| eclipse git plugin | git的eclipse插件 | 集成git功能 |
| eclipse spring plugin | 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）项目的开发，以下简称“demo项目”。

前文提到过，上述架构基于经典的四层web应用架构提出：展示层（UI Layer）、web层、业务层（Business Logical Layer）、持久层（Persistence Layer）。限于实践的不足，无法应用于分布式、大吞吐量的环境之中。但是，在遵循面向接口、模块化、组件化、面向对象等良好编程实践和原则的条件下开发的程序，更加容易利用面向服务的架构、分布式等技术，拓展至更广泛的适用范围之中。对上述提及的非功能需求，本文不作过多的讨论。

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，因此对软件开发的讨论具有一般性。同时，即使涉及的技术在不断更新换代，但是要注意的是，本文涉及的软件工程的一般阶段已经十分成熟，接下来，有必要讨论作出上述选择的考量。

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）网站提供免费的git服务器，可以将项目提交到远端进行存储。

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。此外，git项目每次提交，要求给出当次提交的描述信息，便于项目开发成员进行交流。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git的gui工具，简化操作。



图 通过git bash命令行提交更改

文献《git权威指南》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，其github url为：

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

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。如图所示，在计算机1中开发并提交至本地版本库，然后push修改至远端服务器，再到计算机2中pull检出最新的版本并与本地版本进行合并。



图 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，提供了对第三方依赖的管理，并对代码的编译、测试，项目的发布等特定项目生命周期的任务提供支持，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项目约定结构如下图所示。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，这样测试类就能够访问源类的包可见域，而其他包不能访问，从而既保护了封装，又便于测试。此外，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 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插件通过作用于不同的阶段来完成对maven功能的拓展，同时，插件也通过坐标来进行定位。例如，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：

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

添加了对commons-maths3的依赖。通过使用maven的eclipse插件，在pom.xml发生变更时，maven可以自动检测到依赖的变化，并自动调整程序的依赖项。流程如下图所示：



图 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。

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> |

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> |

除maven以外，gradle集成了maven、ant等项目的构建功能，并提供了脚本化的配置方式，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如下所示：

|  |
| --- |
| apply plugin: 'java' // 插件配置  apply plugin: 'jetty'  apply plugin: 'eclipse-wtp'  sourceCompatiability = 1.8 // jdk配置  targetCompatiability = 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《maven详解》对maven的原理及使用作了深入细致的剖析。

3.3 后端架构

（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java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提供了一组新的采用函数式风格的流式api。

如下代码所示，判断列表中所有的元素都大于0。其中，“n -> n>0”实现了内置的Predicate<T>接口（这里是Predicate<Double>），由编译器在内部将这一语法糖转化为该接口的匿名内部类的实现。语法比起之前版本的for循环和迭代器有了进一步的简化。

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

demo项目主要完成了水轮机调速系统仿真程序的编制。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（java虚拟机）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。关于这点在demo项目的程序编制中会有说明。

（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《spring详解》介绍了spring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。



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="zte.hx.action" />  <mvc:annotation-driven /> |

其中，第一句指明了控制器bean的搜索包，第二句表明了通过注解进行配置。有如图所示的Person和PersonAction类及其实现，



图 简单的spring mvc控制器

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

当采用url：

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

访问该控制器时，spring框架会对结果进行转换得到与Person类对应的json，如下所示：

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

demo项目在前端页面和控制器之间反复使用上述特性传输数据，避免使用语法复杂和对浏览器不友好的页面脚本（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《jQuery权威指南》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，处理诸如语言类型，数组、集合操作，字符串操作等功能，以及赋予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：

|  |
| --- |
| 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。由于动态弱类型的特性，javascript语言有良好的字面量表示方法，表的对比表明，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 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 |

后端生成的遵循java bean规范的数据，可以通过spring的支持和json数据进行互相转换。文献《论文》说明了json数据在网络传输时的性能表现，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层面对json提供了支持。通过工具对象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 JsonTest.groovy

|  |
| --- |
| import groovy.json.JsonOutput // 导入相应的工具类  import groovy.json.JsonSlurper  def slurper = new JsonSlurper() // json转换为groovy对象  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

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”。然而，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文件是兼容的。正因为如此，二者相互调用无需额外操作，只要能在类路径下寻找到彼此即可——groovy代码可以和java代码无缝集成。

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）的编写；元编程（meta-programming）的特性支持groovy动态地对类进行拓展；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-evalue-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。



图 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。

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，水轮机调速系统的控制系统模型及仿真算法也是题中应有之义。

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，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。

仿真的步骤一般如下图所示：



图 仿真的一般步骤

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。

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/τ=1）和非实时仿真（t/τ≠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。

1.离散相似法

设一线性定常系统的状态方程如下所示：

(4-1)

(4-2)

其方框图如下所示：



图 系统方框图

在连续时间下，式(4-1)可表述为：

(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：

(4-4)

以左乘上式两端，并作拉普拉斯反变换，可得：

(4-5)

其中为转移矩阵，且有：

(4-6)

同时，我们也可以使用左乘(4-3)式两端，整理后得到：

(4-7)

利用微分计算规则，可以将上式改写为：

(4-8)

上式两端同时积分并整理，可得：

(4-9)

比较(4-5)式及(4-9)式，可以得到：

(4-10)

由拉普拉斯反变换可知，上式中矩阵指数可以展开成级数的形式并定义为：

(4-11)

由上述(4-9)式可求得该线性定常系统的准确解，当不易求解时，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：

(4-12)

当t=(k+1)T时，有：

(4-13)

由上述两式得到系统离散解的递推格式为：

(4-14)

当采用零阶保持器并且不进行补偿时，有：

(4-15)

上式带入(4-14)式可得：

(4-16)

令上式中，，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。

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。数值积分法本质上也属于离散相似法。

重写离散相似法中的系统如下：

(4-1)

(4-2)

由系统的结构可知：

(4-17)

对上述系统进行离散化处理，分别令t=kT和t=(k+1)T:

(4-18)

(4-19)

用(4-19)减去(4-18)式可得：

(4-20)

于是，对积分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

(4-21)

上式带入(4-20)可得：

(4-22)

简记为：

(4-23)

由e(k)近似将上式带入(4-1)和(4-2)，可得：

(4-24)

(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

1、离散相似法

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

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：

再分别对上述两个环节进行离散化处理。

在这里，我们选取如下的两个模块作为基本模块：



图 积分环节



图 惯性环节

对于积分环节，其微分方程表述如下：

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：

可知，使用零阶保持器进行离散化，积分环节的差分方程为：

(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：

(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。

2.数值积分法

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所示：



图 基本仿真流程

4.3控制系统仿真程序的实现

依照上个小节的流程和算法，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，从而难以进行测试、调试和改进。从第一章的讨论可知，重复的代码是每个程序可以加以优化的地方。

采用离散相似法进行仿真计算时，复杂的系统最终都转化为了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可以通过模块化封装的思路，“分治”地处理成可以进行复用的针对一组类似系统的通用型程序。这正是基于matlab的simulink仿真程序成功的原因之一。在java这种对于面向对象的开发范式提供良好支持的语言当中，如何较好地对仿真程序进行封装，已经有过比较多的探讨。

文献《控制系统数字仿真技术》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开发带来较大的难度。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。

simulink则采用后缀为.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，将绘制的信息渲染到用户视图上，将仿真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。

由于simulink出色的封装和简单的使用接口，出现了很多基于simulink的仿真程序。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和C#）调用simulink的方式。很显然，这种方式舍弃了程序的性能，而且开发出来的程序加入了对matlab核心程序的依赖，难以部署。仅仅为了用户界面就作出上述妥协是不可取的。使用基于matlab的ui程序或开发新的simulink插件包是更好的方法。



图 简单的simulink惯性环节

代码 图所示模型生成的.mdl文件片段

|  |
| --- |
| Block {  BlockType Constant  Name "Constant"  Position [150, 100, 180, 130]  }  Block {  BlockType Scope  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 Sum  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 TransferFcn  Name "Transfer Fcn"  Position [260, 97, 320, 133]  Denominator "[0.1 1]"  } |

通过对相关资料的研究，从客观的要求上来看，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，如表所示。

表 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200ms内完成时长为300s，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）是对程序进一步的要求，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，采取一种快速原型的开发方法，通过反复迭代与重构，首先就着重实现程序的核心仿真计算功能，再逐步地添加其他辅助功能。在迭代过程中可以看出，采用文件（.mdl文件或本文采用的文件）进行外部化配置不是实现上的“偶然抉择”，而是重构的“必然结果”。

4.4水轮机调速系统模型

1 线性模型

水轮机调速系统可以使用控制模型进行描述。其主要结构如下图所示：

图 调速系统结构

如上图所示，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，应对上述环节分别采用相应的模型，单独进行描述。

文献《水轮机自动调节》给出了一种微机调速器模型。

压力引水系统：



图 压力引水系统模型

执行机构经过简化处理，可以近似看成一个随动系统：



图 随动系统

引水系统及水轮机系统模型：



图 引水系统及水轮机模型

发电机及负载模型：



图 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所示。

图 调速系统线性化模型

需要注意的时，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，引入如下的模型。

4.5 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，容易使用程序语言实现。缺点在于保持器数量越多，算法的精度越低。针对该算法的上述特点，可以很方便的使用它来实现初期的原型开发，在需要确保精度的场景中，再考虑使用数值积分方法中更为精确的龙格-库塔法等方法。

本章提出了将敏捷建模、快速迭代的开发方法运用到工程领域软件的开发之中。探讨了这一过程的可能性和方法论。在带有研究性质的项目之中，对需求的理解不是一成不变、一蹴而就的，对研究对象的了解是不断加深、日渐成熟的。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：

（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。

有助于将开发的注意力集中在最终的产品上。然而，正如文献指出的，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数学模型。下一章中将探讨如何运用前述思想、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

敏捷的关键在于迅速得到可以运行的软件即原型，并通过对原型的迭代，添加增量型的变化，仅在必要的时候开发刚好满足需要的文档。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，再给出面向对象的实现，并不断加入新的功能和选项，最终得到完整的程序。

5.1 最初的groovy代码

首先，采用最简单的惯性环节，输入同样采用最简单的阶跃函数，系统的方框图和程序的计算流程如图所示。



图 简单的惯性环节



图 程序计算流程

采用离散相似法离散化后的模型为：

式中，T为仿真步长。

相应的代码如下：

代码 惯性环节仿真计算inertiaTest.groovy

|  |
| --- |
| def start=System.currentTimeMillis()  def dt=0.2 //仿真间隔  def totalTime=10 //seconds  def e=Math.E;  def n=totalTime/dt  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=0.01  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 ${end-start} ms"  println out[1..<10] //输出结果打印  def path='C:\\out.txt'  def pw=new PrintWriter(path)  (0..<n).each{k->  pw.println("${time[k]} ${out[k]}")  }  pw.flush()  pw.close() |

程序的输出如下图所示：



图 程序输出

值得注意的是，上述程序运行耗时为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，这是二者配置的差异带来的性能差异。

程序输出打印到文本文件中如图所示。



图 文本输出

下面需要验证上述计算的正确性。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模型如图所示。



图 simulink惯性环节模型

上述仿真采用和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Scope将仿真结果以矩阵的形式保存在工作空间中，在图所示的配置下，输出为<1001×2 double>类型的矩阵，命名为inertiaOut。



图 simulink仿真配置

编写如下的matlab函数，绘制图形对每次仿真的结果进行对比。在本例中使用compare(inertiaOut)得到如图所示的对比曲线图。

代码 绘制曲线仿真结果进行对比的函数

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



图 dt=0.01s仿真结果对比



图 dt=0.2s时的仿真结果对比

可以看出，当仿真步长增大10倍、并且是时间常数的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，仿真的过程式代码也无法被复用，但至少，我们实实在在看到了程序运行的结果。这多少给了我们一些信心。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 最初的java代码

上个小节中的仿真程序的java版本如代码所示。代码完成了和代码完全相同的功能，仿真的主体程序如方法simulate所示。

代码 仿真程序的java版本

|  |
| --- |
| package zte.hx.simulation.demo;  import zte.hx.simulation.util.PrintUtil;  import zte.hx.util.TestUtil;  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.printTo("C:\\out.txt", w -> { // 打印  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];  double[] in = new double[n];  for (int i = 0; i < n; ++i) {  time[i] = i \* dt;  in[i] = 1.0;  }  double K = 1.0;  double T = 1.0;  out = new double[n];  out[0] = 0;  double c1 = 1 / (T + dt); // 缓存系数  double c2 = K \* dt / (T + dt);  for (int k = 1; k < n; ++k) { // 迭代  out[k] = c1 \* out[k - 1] + c2 \* in[k];  }  }  } |

这里我们可以直观的感受到，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。从开发的经验来看，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，才能更好地发挥其优势。

此外，从运行的结果（如图）来看，java代码仿真10s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。当然，显而易见的，仿真的一系列算法本身的复杂度采用大O标记可以表示为O(n)，模型再复杂，其复杂度只和迭代总步长相关，因此复杂度基本保持不变。后文的试验还会再讨论到程序的性能问题。但是，程序的性能优化一般放在开发的最后阶段，在开发过程中，不用太过关注。



图 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。借助现代ide的功能，封装极大提高了代码的可读性。本小节中的TestUtil.timeIt()方法和PrintUtil.printTo()方法都运用了到了语言的“闭包”特性。其实现如下所示。

代码 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 {  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 辅助打印函数printTo，使用groovy实现

|  |
| --- |
| class PrintUtil {  static def path='C:\\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，文档和设计不会事先告诉开发人员有这么一段代码（如果有，很可能属于“过度设计”了），而是在开发过程中不断遇到类似的逻辑，于是提炼并重构的“模块”。事实上，代码1源于对性能测试的一个简单但又不断重复的代码片段，代码2则是demo项目开发过程中预计到的将会不断出现的一段代码。两段代码都用于测试阶段。如果有需要“复制粘贴”的地方，在重复代码把事情弄得更糟糕之前，不如先考虑一下类似上面的“全局函数”。

下面，是我们迈出增量式、迭代式开发的第一步了，为了说明开发的思路，接下来的两步都会迈得足够小。

5.3 第一次迭代

考虑给“最初的模型”（如图）加入一个新的环节，如图所示。



图 第一步迭代的模型

采用离散相似法离散化后的模型为：

其中，T为仿真步长。这里，我们取，。

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，虽然模型结构上的顺序为：电源——惯性环节——积分环节，但是由于离散模型是“无后效性的”，并没有给计算添加新的限制。也就是说，计算过程中先计算还是都是可行的。

代码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 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).eachWithIndex{a,k-> // 迭代  …  x2[k]=(x2[k-1]+c2\*x1[k]).trunc(4)  }  … |

与simulink仿真对比的结果为：



图 简单串连环节的仿真结果

5.4 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，就得到如图所示的模型。



图 简单的闭环模型

依旧采用离散相似法，得到离散化的模型为：

其中，T为仿真步长。这里，我们同样取，。

由模型可知，引入一个求和模块之后，对算法产生了较大的影响，这体现在对计算过程引入了新的限制。在迭代的过程中，欲求，须先求误差，而欲求err(k)，需要知道在k时刻的状态，这相当于对求解顺序施加了限制，计算顺序和上面的公式相同。由此可知，加入某些模块对迭代时的算法会产生影响，在开发通用的仿真程序时，必须考虑到这一点。

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 闭环后新增的代码片段

|  |
| --- |
| ...  def err=[] // 误差  err[0]=u[0]-x2[0]  ...  def c11=e\*\*(-10\*T)//buffered coefficient  def c12=1-e\*\*(-10\*T)  def c21=2\*T  (1..n).each{k ->  x2[k]=x2[k-1]+c21\*x1[k-1]  err[k]=u[k]-x2[k] // iteration  x1[k]=c11\*x1[k-1]+c12\*err[k]  y[k]=x2[k]  }  ... |

与simulink仿真对比的结果如图所示。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 简单闭环系统的仿真结果

5.5 是时候模块化了

连续三个系统的仿真计算，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。前面代码中的省略的部分，都是可以封装模板代码；而不同的代码，也可以抽出共性用“类模板”来建模；剩下的看似没有共同点的代码，可以通过一些编程手段来刻画。不断地重构，直到重复被尽可能地消除掉。

这一个小节依旧以5.1节中最为简单的系统为例。从物理结构来看，这个系统包括三个部分：输出源（Source）、惯性块（Block）和这两者之间的连接线（Line）。从对象的观点来看，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，输出源的数据变化规律已知，并且通过连接线传到惯性环节，惯性环节就像一个黑箱子，将输入进行转化后输出。



图 简单的惯性环节

从上面的描述中，我们提炼出了Source、Block、Line三个基本模型，就像面向对象分析中那样。但是我们显然比这做得更多，因为我们已经有了前面的三次尝试，所以，得到上面的结论更加自然。使用类进行建模。如类图所示，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。

仿真程序如代码所示。这段程序依旧很简洁，但是和过程化的程序相比，已经显得颇为复杂，而且还有几个明显的缺陷：

（1）程序依旧不能复用。Line类中的start和end对next方法做了假定，Simulation类没有普适性；

（2）程序显得松散而不严谨。当两个类具有相同的方法签名时，像groovy这样的语言认定两个类具有相同的接口，但是这个约定降低了可读性，且不同类型的模块接口并不是完全一致的，仿真算法与模块的类型相关。



图 程序类图

代码 模块化的仿真程序

|  |
| --- |
| class Config{ // 仿真配置  double T=0.01  double totalTime=10  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] |

我们可以进行重构，使用继承体系和外部化配置来改善程序结构，使用强类型和接口来明确程序边界。有人指出，可以利用groovy的可选类型来提供强类型的限制，但是，编程经验告诉我们，强弱类型联合起来使用，更加让人迷惑，哪里使用强类型，哪里使用弱类型都会让人纠结不已。还是将可选类型的特性作为对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，但是后面可以看出，groovy还有其他的用途。

如前所述，上述程序还存在不足，但是它实现了基本的模块，并且强调了仿真中的核心步骤。下个小节将引入类层次结构来进行建模。

5.6 类层次结构

在前面的代码中，阶跃电源可以抽象为电源模型，它为系统提供模型，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。以上的描述可以作为我们改进程序的出发点。改善后的程序结构如图所示。



图 程序的继承结构

可以看出，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）卡如下所示：

表 Block的CRC卡

|  |  |
| --- | --- |
| 仿真块 | |
| 建模的基本模块 |  |

表 Line的CRC卡

|  |  |
| --- | --- |
| 连接线 | |
| 记录系统结构 | 仿真块 |
| 传递上个块的输出，并促使块的转态改变 | 仿真块 |

表 Joint的CRC卡

|  |  |
| --- | --- |
| 连接点 | |
| 记录系统结构 |  |
| 汇聚多个输入 | 连接线 |

表 Simulation的CRC卡

|  |  |
| --- | --- |
| 仿真 | |
| 进行一次仿真计算 | 连接线、连接点、仿真块 |

如上所示，通过引入继承结构，程序显得更加复杂，但我们得到了下述好处：消除了重复，使程序结构更加清晰，进而使代码容易阅读；程序拓展更加容易，只需要添加新的实现类即可新增功能。通过结合继承结构和CRC卡，可以迅速弄清程序的结构。因而，将这些图表制品加入到文档之中，可以使系统更好维护。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，这样减少了文档的维护成本，并且避免过度开发文档所耗费的不必要的精力。

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，其关键的方法如下：

代码 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，并推动其连接的模块的状态发生改变（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。这使得

这个小节在系统中引入了继承结构，下个小节将使用一个例子说明其应用。

5.7 仿真实例：ExciterTest

采用文献中的励磁系统模型，如图所示。模型中的各个模块对应在代码中的变量也已经标出。



图 励磁系统模型

代码 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 l1 = new Line(stepSource, j1);  Line l2 = new Line(j1, b1);  Line l3 = new Line(b1, b2);  Line l4 = new Line(b2, j2);  Line l5 = new Line(j2, b3);  Line l6 = new Line(b3, b4);  Line l7 = new Line(b4, j2);  Line l8 = new Line(b3, b5);  Line l9 = new Line(b5, b6);  Line l10 = new Line(b6, j1);  j1.addLine(l1, Joint.ADD); // 配置连接点  j1.addLine(l10, Joint.SUB);  j2.addLine(l4, Joint.ADD);  j2.addLine(l7, Joint.SUB);  List<Line> lines = Arrays.asList(l1, l2, l3, l4, l5, l6, l7, l8, l9, l10);  TestUtil.timeIt(() -> {  config.iterate((i, k) -> {  lines.forEach(l -> l.push(i, k));  });  });  }  } |
|  |

可以发现，程序特点在于对象和对象之间的信息传递，通过各个对象的协作完成仿真。可以注意到，依照从l1到l10的顺序调用push方法符合5.4节中关于计算顺序的讨论。

照例通过和matlab结果进行对比以验证程序的正确性。如图，可以看出，对于T=0.01s的两条仿真曲线基本吻合，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，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 励磁系统仿真结果比较

在这里，我们优先考虑正确性，延后考虑精度的问题。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的问题。现在，使整个程序迅速运行起来才是当务之急。不得不一再指出，有了一个正确、简单的程序，再进行重构，远比过早考虑精度、性能等问题更加高效。

最后指出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。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。下个小节我们考虑如何通过模型的外部化配置，来统一程序的数据接口，并进一步提高仿真程序的通用性。

6、仿真程序的应用