

---

# Modelica 实例教程

作者 : Dr. Michael M. Tiller

译者 : 谢东平

及

 世冠科技



<b>正文前信息</b>	I
序言	III
<b>致谢</b>	V
技术方面	V
个人方面	V
贡献者	VI
工具	VII
<b>前言</b>	IX
Modelica 是什么?	IX
为什么选择 Modelica?	IX
我能用 Modelica 语言做什么?	IX
<b>献词</b>	XI
<b>第一部分 描述行为</b>	1
<b>第一章 基本方程</b>	3
第 1.1 节示例	3
第 1.1.1 节 简单的一阶系统	3
第 1.1.2 节 引入物理	7
第 1.1.3 节 电气方面的实例	11
第 1.1.4 节 机械示例	13
第 1.1.5 节 猎食者猎物系统	18
第 1.2 节回顾	25
第 1.2.1 节 模型定义	25
第 1.2.2 节 变量	26
第 1.2.3 节 方程	30
第 1.2.4 节 初始化	31
第 1.2.5 节 Record 类型定义	33
第 1.2.6 节 标注	34
<b>第二章 离散行为</b>	39
第 2.1 节示例	39
第 2.1.1 节 再探冷却	39
第 2.1.2 节 弹跳球	43
第 2.1.3 节 状态事件的处理	47
第 2.1.4 节 RLC 开关电路	57
第 2.1.5 节 速度的测量	59
第 2.1.6 节 滞回	68
第 2.1.7 节 同步系统	72

第 2.2 节回顾 . . . . .	76
第 2.2.1 节 事件 . . . . .	76
第 2.2.2 节 If 语句及 if 表达式 . . . . .	78
第 2.2.3 节 When 语句 . . . . .	78
<b>第三章 向量与数组</b>	81
第 3.1 节示例 . . . . .	81
第 3.1.1 节 状态空间 . . . . .	81
第 3.1.2 节 一维热传导 . . . . .	85
第 3.1.3 节 化学系统 . . . . .	89
第 3.2 节回顾 . . . . .	93
第 3.2.1 节 数组声明 . . . . .	93
第 3.2.2 节 数组的建构 . . . . .	94
第 3.2.3 节 数组函数 . . . . .	97
第 3.2.4 节 数组索引 . . . . .	104
第 3.2.5 节 循环 . . . . .	106
<b>第四章 函数</b>	107
第 4.1 节示例 . . . . .	107
第 4.1.1 节 多项式计算 . . . . .	107
第 4.1.2 节 插值 . . . . .	113
第 4.1.3 节 软件在环 (SiL) 控制器 . . . . .	121
第 4.1.4 节 非线性 . . . . .	123
第 4.2 节回顾 . . . . .	126
第 4.2.1 节 函数定义 . . . . .	126
第 4.2.2 节 控制流程 . . . . .	129
第 4.2.3 节 外部函数 . . . . .	130
第 4.2.4 节 函数标注 . . . . .	133
<b>第二部分 面向对象建模</b>	139
<b>第五章 包</b>	141
第 5.1 节示例 . . . . .	141
第 5.1.1 节 组织内容 . . . . .	141
第 5.1.2 节 引用包内内容 . . . . .	143
第 5.1.3 节 导入物理类型 . . . . .	144
第 5.2 节回顾 . . . . .	147
第 5.2.1 节 包的定义 . . . . .	147
第 5.2.2 节 查找规则 . . . . .	150
第 5.2.3 节 导入 . . . . .	151
第 5.2.4 节 Modelica 标准库 . . . . .	151
<b>第六章 连接器</b>	159
第 6.1 节简介 . . . . .	159
第 6.1.1 节 基于组件的建模 . . . . .	159
第 6.1.2 节 非因果连接 . . . . .	159
第 6.2 节示例 . . . . .	160
第 6.2.1 节 简单领域 . . . . .	160
第 6.2.2 节 流体连接器 . . . . .	162
第 6.2.3 节 框图连接器 . . . . .	163
第 6.2.4 节 图形连接器 . . . . .	164
第 6.3 节回顾 . . . . .	167
第 6.3.1 节 连接器定义 . . . . .	167
第 6.3.2 节 图形标注 . . . . .	167
<b>第七章 组件</b>	173
第 7.1 节示例 . . . . .	173
第 7.1.1 节 传热组件 . . . . .	173

第 7.1.2 节 电气部件 . . . . .	181
第 7.1.3 节 基本旋转组件 . . . . .	185
第 7.1.4 节 高级旋转组件 . . . . .	192
第 7.1.5 节 再探猎食者猎物模型 . . . . .	204
第 7.1.6 节 再探速度测量 . . . . .	213
第 7.1.7 节 框图组件 . . . . .	223
第 7.1.8 节 化学组件 . . . . .	234
第 7.2 节回顾 . . . . .	237
第 7.2.1 节 组件模型 . . . . .	237
第 7.2.2 节 系统模型 . . . . .	243
第 7.2.3 节 组件模型标注 . . . . .	245
<b>第八章 子系统</b>	<b>249</b>
第 8.1 节示例 . . . . .	249
第 8.1.1 节 齿轮总成 . . . . .	249
第 8.1.2 节 带迁移的掠食者猎物方程 . . . . .	254
第 8.1.3 节 直流电源 . . . . .	261
第 8.1.4 节 空间分布的传热 . . . . .	270
第 8.1.5 节 钟摆的间谐运动 . . . . .	276
第 8.2 节回顾 . . . . .	278
第 8.2.1 节 子系统接口 . . . . .	278
第 8.2.2 节 组件数组 . . . . .	280
第 8.2.3 节 修改语句 . . . . .	281
第 8.2.4 节 传值 . . . . .	282
<b>第九章 架构模型</b>	<b>285</b>
第 9.1 节示例 . . . . .	285
第 9.1.1 节 传感器比较 . . . . .	285
第 9.1.2 节 架构驱动方法 . . . . .	294
第 9.1.3 节 热力控制 . . . . .	304
第 9.2 节回顾 . . . . .	319
第 9.2.1 节 接口和实现 . . . . .	319
第 9.2.2 节 配置管理 . . . . .	320
第 9.2.3 节 可扩展连接器 . . . . .	325
<b>第三部分 索引及表格</b>	<b>327</b>
<b>参考文献</b>	<b>331</b>



# 正文前信息



本书的组织结构有一点特殊。这主要是因为，我预期大部分读者会阅读本书的 HTML 版本。因此，本书大量使用了超级链接以方便读者查找他们最需要的内容。大多数情况下，链接方便了本书电子版本的阅读。虽然对于出版的纸质版本，链接并不会起到太大作用。但我们已经尽了最大的努力保证所有版本格式的质量。

本书的价值在于提供了两条不同的学习“路径”。纵观全书，本书试图通过目录中的章节递进关系有逻辑的展现示例材料。即本书前几章主要专注于描述不同元件类型的数学特性，在后面章节中更多的讨论如何搭建模型的结构（例如，包、组件模型、子系统等）。其次，当阅读本书的电子版本时，对于描述的示例，我提供了相关的链接以便绕开正常的章节顺序。读者可以方便的在后续章节中继续阅读当前示例的更多细节（后续章节提供了示例扩展的 Modelica 语言特性）。我衷心的希望这种编排方法可以提高阅读体验而不会迷惑读者。

本书大部分章节分为三个部分。第一部分介绍本章节将要讨论的主题。第二部分通过广泛的示例展现说明第一部分中介绍的 Modelica 语言相关特性。注意，通常每个示例只涉及一个相关主题。所以，读者最好查看所有示例以获取主题的完整描述。第三部分中，主要对讲述的主题和示例中未展示的细节进行了回顾和总结，以此深入描述讨论的主题。



本书是许多讲座、书籍、讨论、会议等的结晶。因此，我也就不可能完全统计对本书作出贡献的所有人员。毫无疑问，这其中肯定会有未提及的人。首先，我要对那些被忽略的同仁致以诚挚的歉意。

## 技术方面

谈及 Modelica 时，首先要特别感谢 Hilding Elmquist。这不仅是因为他的技术远见，发现了符号处理技术可以解决工程问题的潜在应用价值。而且，得益于他的领导能力，Modelica 技术朝着开放的标准发展。Hilding 毫无疑问是“Modelica 之父”，对于 Modelica 技术取得的一切成果，他都受之无愧。

仅次于 Hilding 的是 Martin Otter，他为推动 Modelica 技术的发展不辞辛劳。我可以很坦诚的说 Martin 比我知道的任何人都努力。他不仅对 Modelica 技术的发展做出了不朽的贡献，而且他还承担着管理 Modelica 协会的工作任务，尽管这种工作通常情况下是费力不讨好的。我们需要记住的一点是，有伟大的技术想法并不等同于成功的秘诀。这还需要一些人一起将这些想法实现。我要感谢 Martin 的努力工作才成就了今天的 Modelica 协会。

当然，并不是 Martin 一个人在管理 Modelica 协会。Modelica 委员会和其成员同样在促进 Modelica 技术推广方面起到了极其重要的作用。最后，我要重申一下，Modelica 协会是一个致力于支持工程应用的开放标准组织。我要感谢所有成员为此做出的努力。

Modelica 协会并不是常设机构。但 Modelica 语言研发团队却总是存在。这些人在一年中开几次研讨会，以促进 Modelica 语言的发展。为此，他们无私的贡献了大量的时间和精力。我要感谢所有参与开发 Modelica 语言和 Modelica 标准库的同仁，感谢他们的辛勤付出。

## 个人方面

就我个人而言，我要感谢我的母亲、父亲、妻子、孩子和我的岳父岳母。感谢他们对我的坚定支持，才使得我在科学、工程应用和数学等领域能够全心投入。他们富有责任感的培养以及对我个人兴趣的坚定支持，才使我获得了充足的精力和时间参与到目前的项目中。

对某些人来说，提供一本免费的书似乎有些激进。这个项目能有这样的想法，是通过接触 Cory Doctorow 和 Lawrence Lessig 的工作而受到的启发。我要感谢他们关于知识共享方面的想法。这开阔了我的眼界。我还要感谢 Dietmar Winkler 与我多次讨论另类出版模式。我们经常会讨论 Doctorow 和 Lessig 的想法，以及如何将他们的想法应用在 Modelica 领域，创建更易于访问的内容。

回顾过去，我感到非常幸运。我工作过的几家公司都非常支持我参与 Modelica 项目。我第一次参与 Modelica 项目是在福特汽车公司工作期间。很感谢他们愿意赞助我参加各种与 Modelica 相关的活动。在福特之后，我进入了 Emmeskay（后来被 LMS 收购）工作。在 Emmeskay 公司工作期间我获益良多。我要特别感谢我的合作伙伴 Swami Gopalswamy 和 Shiva Shivashankar 给我加入 Emmeskay 的机会，并有幸与他们成为好朋友。在 Emmeskay 期间，我很荣幸能与 Michael Sasena 和 John Batteh 在多个 Modelica 相关的项目中共事。Emmeskay 是一个伟大的公司，这可以从工作在那里的团队成员反映出来。最后，我要感谢达索系统公司 (Dassault Systèmes) 给予了我与其他优秀人士一起工作的机会。尤其是我要感谢 Hilding Elmquist 和 Marc Frouin 鼓励我到那里工作。我还要感谢 Martin Malmheden、Dag Brück 和 Sandrine Loembe 与我在巴黎度过的所有美好时光。

## 贡献者

这个项目严格意义上讲是一次实验。目的是验证能否通过众筹的方法，在一个如 Modelica 这样较窄的技术领域寻找商机进行出版。我惊喜的看到这是可行的。这个项目有足够的资金支持。有鉴于此，我要感谢众筹项目的支持者。我要特别感谢以下人员极其慷慨的捐助：

- Hilding Elmquist
- Robert Norris
- Matthis Thorade
- Henning Francke
- Yang Ji
- Christoph Höger
- Philipp Mossmann
- John Batteh
- Dirk Zimmer
- Jan Brugård
- Swami Gopalswamy
- Peter Aronsson
- Michael P. Case
- Markus Groetsch
- Vicente Ramírez Perea
- Tisha Villanueva
- Adrian Pop
- Nimalendiran Kailasanathan
- Kevin Davies
- Peter Harman
- Dietmar Winkler
- Johan Rhodin

我还要感谢赞助商：

- 金牌赞助商
  - CyDesign
  - Wolfram Research
  - Modelon
  - Maplesoft
  - Dassault Systèmes
- 银牌赞助商
  - Ricardo Software
  - ITI
  - Modelica Association
  - Global Crown Technology
  - Siemens

- 铜牌赞助商
  - Suzhou Tongyuan
  - Open Source Modelica Consortium
  - DOFWare
  - Bausch-Gall GmbH
  - Technische Universität Hamburg/Harburg
  - Schlegel Simulation GmbH

本项目所有参与人和赞助商都在为开发出更高质量的 Modelica 教材这一共同目标不懈努力。换句话说，没有他们就不会有这个项目。

众筹项目的资金使我可以在这个项目上投入更多的时间，但是仍然有很多人在这个项目上帮助过我。首先，我要再一次感谢我的父亲对本书的初稿进行了校对。校对是一项无聊的工作，但却是必不可少的步骤。所以，我认为他值得为这项工作所作出的奉献而获得赞扬。此外，我还要感谢 Dietmar Winkler 和 Michael O' Keefe 对本书内容提供的反馈。Dietmar 还帮助我测试了本书电子版和 PDF 格式出版的问题。

我要感谢 Jeff Waters 作为“赞助商代表”所付出的努力。在编写本书期间，我与 Jeff 有过多次富有成效的讨论，以确保本书章节布局和图形设计满足赞助商的预期。

## 工具

编写本书需要很多不同的工具。通过使用这些工具极大的提高了我的工作效率。

本书使用 Sphinx 编写。Sphinx 是一个文档生成工具，支持多种格式的文件输出。通过使用 Sphinx 工具，使得我可以把工作的重点放在本书的内容上，而不必过多的考虑本书的编写格式。

为了编写本书，我需要一种方法来测试出现在本书中的模型，绘制生成的仿真结果以及生成相应的 JavaScript 代码，以实现本书在浏览器 HTML 中的集成仿真功能。OpenModelica 支持所有上述功能。除此之外，我还欠 Martin Sjölund 和 OpenModelica 团队一个大大的“人情”。因为在编写本书期间他们能快速的响应我的各种问题。很多时候我会在深夜（瑞典时间）与 Martin 通过 Skype 见面，而他都会非常谦和的帮助我。

本书的 HTML 版本通过 Emscripten 工具可以实现浏览器的集成仿真功能。Emscripten 允许将 C 和 C++ 语言的通用代码通过 LLVM 编译成 JavaScript。虽然我知道这是可能的方法。但是直到我看到 Tom Short<sup>1</sup> 在整合 OpenModelica 和 Emscripten 的工作后，才相信这种方式是可行的。正是他在这方面的工  
作才使得浏览器具有了集成仿真功能。

本书的编写使用了 Git 作为版本控制系统，托管给 GitHub<sup>2</sup> 进行管理。大多数人认为版本控制系统是一套非常神秘的备份系统，其实版本控制系统是合作的核心。我非常希望能看到它在工程上被广泛的应用。对于本书来说，GitHub 的“pull request”系统在整合审阅者的评审意见上非常有用。我要再次感谢 Dietmar Winkler 提醒我关于 Git 的许多不同功能。

我还使用了 Emacs 编辑器编写本书。虽然有许多优秀的编辑器支持各种语言和平台，Emacs 依然还是我许多项目开发过程中的好帮手。它自带的功能几乎可以支持我需要的所有文件类型。

在本书的编写过程中，好几个供应商给我提供了他们的专有工具。我没有太多的使用这些工具，在此我想感谢他们慷慨的为我提供了临时许可文件。具体来说，我要感谢 Dassault Systèmes、Maplesoft、Wolfram Research 和 ITI，他们分别给我提供了 Dymola、MapleSim、SystemModeler 和 SimulationX 的使用权。

本书的大部分内容是在 MacBook Air 上编写的。我的第一台电脑是 Apple //e。从那以后，我主要使用 PC 机和 Unix 工作站工作。最近，我在 Linux 机器上做了大量的开发工作。我经常会放弃使用 Macs 电脑，因为确信它不能支持我经常使用的那种命令行模式的开发工作。我不得不说我错的太离谱了。MacOSX（苹果操作系统）的生态系统几乎与我在 Linux 环境下使用的完全相同。我可以很容易的在

<sup>1</sup><https://github.com/tshort/openmodelica-javascript>

<sup>2</sup><http://github.com>

MacOSX 和 Linux 环境之间切换而不需要任何大的调整。MacBook Air 的电池续航能力和便携性让我的整个工作效率有了很大的提高。

本书的编写涉及到许多 HTML 布局、样式和嵌入 JavaScript 的测试和调试工作。这项工作主要使用 Firefox 来完成，偶尔也使用 Chrome。我要感谢 Mozilla 基金会和 Google 开发了如此卓越和标准兼容的浏览器。

本书的风格大量借鉴了 Sematic UI<sup>3</sup>这一 CSS 框架。

---

<sup>3</sup><http://www.semantic-ui.com>

您出于某些原因，机缘巧合地看到本书。若您没有任何 Modelica 方面的知识，对于 Modelica 语言您可能会有一些疑惑。我尝试着列出这些问题，希望可以引起您的兴趣并引导您进行更深入的探索。

## Modelica 是什么？

Modelica 是一种高级的陈述式语言用于描述事物的数学特性。它通常应用于工程领域，可以轻松的描述不同类型工程组件（例如弹簧、电阻、离合器等）的工作特性。此外，这些组件又可以方便的组合成子系统、系统，甚至架构模型。

## 为什么选择 Modelica？

Modelica 语言令人信服的原因主要有以下几点。第一，也是最重要的是它的技术实力。Modelica 编译器在后台运行复杂的算法，使得工程师可以将工作重点放在组件特性的数学描述上，以获得高性能的仿真速度。而工程师不需要具备某些相关领域的高深知识，例如微分代数方程组、符号运算、数值求解、代码生成、后处理等。

Modelica 语言在技术实现上能成功的关键是它支持广泛的建模形式，即 Modelica 语言支持在混合微分代数方程组中同时描述连续和离散特性。Modelica 语言支持在同一模型中同时使用因果（通常用于控制系统设计）和非因果（通常用于创建面向原理的物理模型设计）的建模方法。

最后，Modelica 语言另一个引人注目的事实是：Modelica 从一开始就是作为开放的语言标准设计的。Modelica 语言规范可以免费获得。这大大激励了工具供应商支持 Modelica 模型导入、导出的热情（因为不需要强制支付任何形式的版税）。

## 我能用 Modelica 语言做什么？

Modelica 绝对是一种理想的建模语言，它几乎可以用于所有工程领域的系统特性建模。单纯使用 Modelica 语言就可以完美的支持物理模型设计和控制模型设计。同时，Modelica 语言也是多领域的，因此它不会人为地限制其应用的工程领域或系统。综上所述，Modelica 语言提供了一整套用于建立集总参数模型的方法，几乎涉及所有的工程应用系统。



本书源于 Kickstarter (美国一家众筹网站) 的一个项目。目的是写一本符合知识共享授权，可供免费使用的书。我产生写作本书的想法主要是受到了 Cory Doctorow 和 Lawrence Lessig 的著作的鼓舞，他们的知识共享理念使我产生强烈的共鸣。我对应用我掌握的 Modelica 知识谋利毫无兴趣，我希望可以分享这些知识，让大家同样可以体验我在该领域探索时的快乐。令人开心的是，这个项目在 2012 年 12 月 4 日成功获得资助，我得以开始我这个创作具有完全知识共享授权的书籍的旅程。如果情况没有发生变化，本书原本将献给 Cory Doctorow 和 Lawrence Lessig，以感谢他们激励我从事这个项目。

但这本书并不是献给他们，而是 Aaron Swartz。Aaron 无疑是一位神童，在 14 岁时参与了 RSS 标准的开发。接下来他创建了自己的公司——Infogami，最终被 Reddit 兼并了。之后，他致力于知识共享授权方面的工作，反抗互联网审查制度。他既聪明又成功，而且非常无私，他非常关心技术发展以及人类的未来。他具有太多让我钦佩和向往的品质。可惜的是，Aaron Swartz 在 2013 年 1 月 11 日结束了自己的生命。

在我成功获得 Kickstarter 的项目资助后不到一个月，Aaron Swartz 就离世了，年仅 26 岁。Aaron 在自己年轻的生命里，取得了令人惊讶的成就。每当想到他以后可能达到的成就，就更加令人对他的死亡感到悲伤。尽管 Aaron 比我年轻近 20 岁，但他却帮助我，本项目以及那些将通过他参与创建的 RSS，知识共享及互联网技术阅读本书的人们开辟了道路。令人心碎的是，我才刚开始这个项目，他就去世了。因此，我决定将本书献给他。

由于已经有更优秀而且认识其本人的作家<sup>4</sup>为他写了传记<sup>5</sup>，因此我不会再详细介绍他的生平。但我必须说作为一个企业主、研究员和一名美国公民，我非常钦佩他所达到的成就。同时对那些决定把他作为案例以儆效尤的联邦检察人员（尤其是Carmen Ortiz<sup>6</sup>）对待他的方式感到非常愤怒。

本书献给 Aaron Swartz，是为了让人们记住他。更重要的是，让人们记住他用生命所坚持的事情。他过世已经一年多了，但我从没有忘记过他。我也希望可以通过本书提醒人们，记住他所代表的理念与精神。我们要从这个悲剧中吸取教训，也要确保我们的政府能够理解：像 Aaron Swartz 这样的人，可能有些激进和狂热，但不应该被起诉和被当做罪犯对待。他们应得到帮助和引导，如何用他们充沛的精力和才能来帮助社会。

“Aaron 走了。世界行者们，我们失去了一位智者。所有为真理抗争的黑客们，我们中的一员倒下了。所有的父母们，我们失去了一个孩子。让我们为他哭泣。”

—Tim Berners-Lee<sup>7</sup>



<sup>4</sup>[http://www.huffingtonpost.com/lawrence-lessig/aaron-swartz-suicide\\_b\\_2467079.html](http://www.huffingtonpost.com/lawrence-lessig/aaron-swartz-suicide_b_2467079.html)

<sup>5</sup><http://boingboing.net/2013/01/12/rip-aaron-swartz.html>

<sup>6</sup><https://petitions.whitehouse.gov/petition/remove-united-states-district-attorney-carmen-ortiz-office-overreach-case-aaron-swartz/RQNrG1Ck>

<sup>7</sup>[https://twitter.com/timberners\\_lee/status/290140454211698689](https://twitter.com/timberners_lee/status/290140454211698689)



# **第一部分**

## **描述行为**



## 基本方程

正如序言 ( III) 提到的，我们会以理解基本行为来开始我们对 Modelica 的探索。本章中，我们的重点在于展示如何编写基本方程。

### 第 1.1 节 示例

#### 第 1.1.1 节 简单的一阶系统

首先，让我们考虑一个非常简单的微分方程：

$$\dot{x} = (1 - x)$$

在这个方程中，只有一个变量  $x$ 。此方程可以用 Modelica 语言表述如下：

```
model FirstOrder
  Real x;
equation
  der(x) = 1-x;
end FirstOrder;
```

这段代码以关键词 model 开始，以表明模型定义的开始。后面紧跟着模型的名字 FirstOrder，再后面是我们感兴趣的变量声明部分。

从这个方程中，我们可以确定变量  $x$  是一个连续实数。在 Modelica 语言中，我们可以用 Real x; 语句来声明它。Real 类型只是众多数据类型的一种（在接下来的变量一节中，将会对数据类型进行更加详细的讨论）。

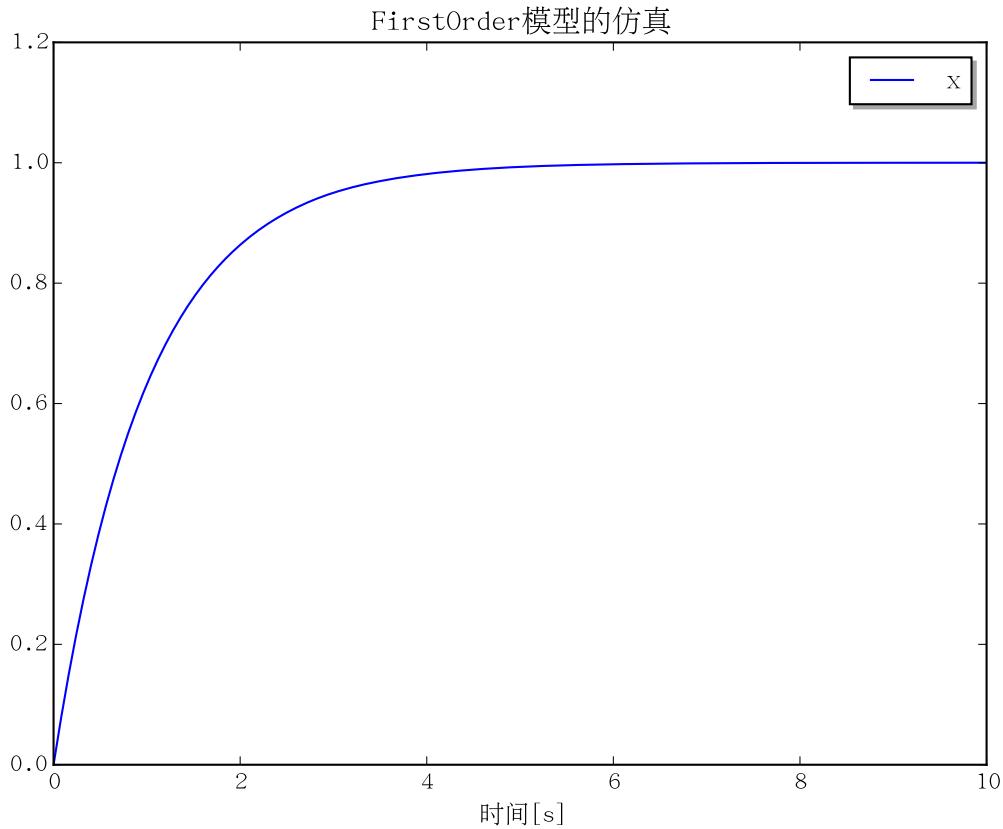
当所有的变量声明完成后，我们就可以编写描述模型特性的方程了。在这个例子中，我们可以用运算符 der 来表示变量  $x$  的一阶导数。所以：

```
der(x) = (1-x)
```

其所描述的含义等同于方程：

$$\dot{x} = (1 - x)$$

不同于大多数的编程语言，我们并不像运行“程序”那样将代码解析成逐条执行的指令。相反，我们使用 Modelica 编译器将代码转换成可以仿真的模型。这种仿真的过程基本上等同于求解方程（通常用数值方法）并获取其解的轨迹的过程。如下图所示：



通过上面的例子，让你对使用建模语言描述系统的数学特性有了一个初步的了解。我们并不需要描述如何去求解这个微分方程，我们关注的重点完全放在方程本身的特性上。在我们介绍更复杂的实例时，你将会发现很多涉及求解过程的繁琐工作都是由 Modelica 编译器自动处理的。

### 添加文档

现在，我们已经完成了这个简单数学方程的求解，让我们把重点转移到如何才能使模型更具可读性。我们考虑如下的模型：

```
model FirstOrderDocumented "A simple first order differential equation"
  Real x "State variable";
equation
  der(x) = 1-x "Drives value of x toward 1.0";
end FirstOrderDocumented;
```

注意在这个模型中引号中的文字。

理解这些引号中的文字模块非常重要，它们在计算机科学中被称为“字符串”，而不是注释。这些“描述性字符串”不像注释，不能在任意的地方插入。相反的，它们只能在特定的地方插入，以提供与其关联的模型元素的附加说明。

例如，第一个字符串“A simple first order differential equation”（一个简单的一阶微分方程）用来描述模型的属性。你可以注意到，它紧跟在模型名字后面。如果你希望对模型做相应的注释，文档语句必须插入在模型名字的后面。

在后面我们将看到，这种文档可以被很多工具所使用。例如，搜索模型时，搜索工具可以利用这些描述性的字符串进行识别匹配。这些文本也可以与模型的图示相关联。当然，这种文档对阅读模型的任何人都很有帮助。

通过上面的例子，我们可以看到，这些描述性的文本也可以放置在模型的其他位置。比如，在变量声明或方程后面添加相应的文档。

## 初始化

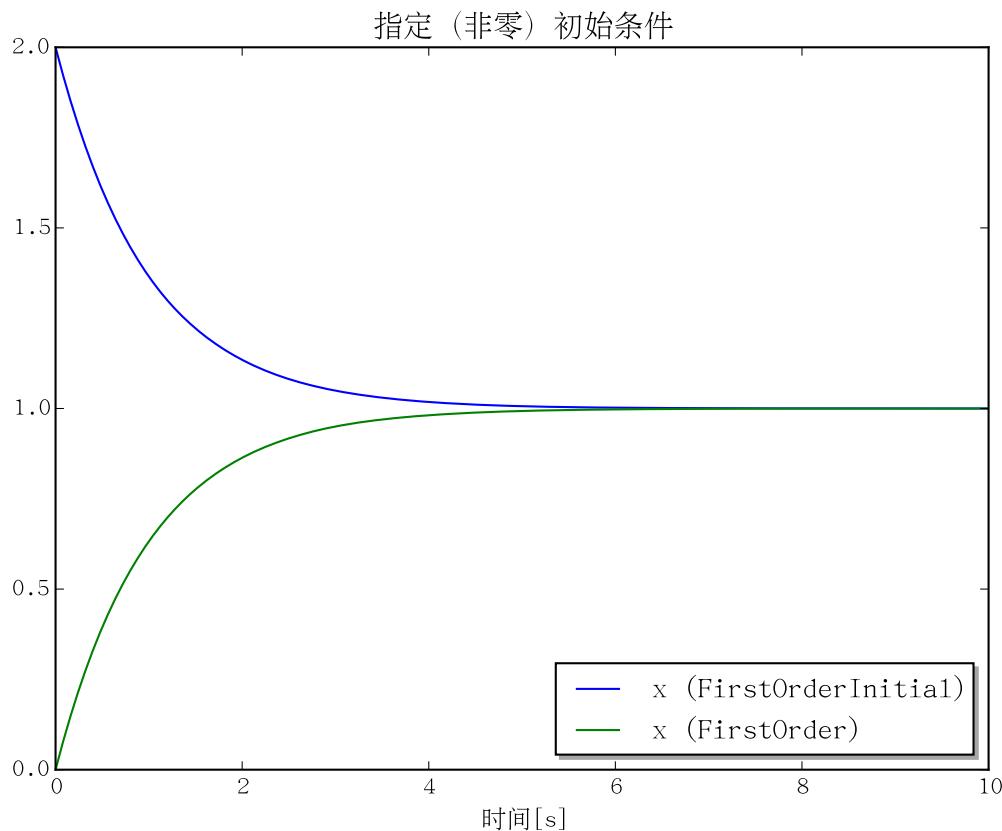
通过上面的例子可以看到，Modelica 语言允许我们使用微分方程去描述模型的特性。但是，设定初始条件和编写方程同样重要。

因此，Modelica 语言也提供用于描述方程组初始化的结构。例如，如果我们想给模型中变量  $x$  赋初值为 2，可以在模型中增加 initial equation（初始化方程）区域，如下所示：

```
model FirstOrderInitial "First order equation with initial value"
  Real x "State variable";
initial equation
  x = 2 "Used before simulation to compute initial values";
equation
  der(x) = 1-x "Drives value of x toward 1.0";
end FirstOrderInitial;
```

我们可以看到，上述模型和前面添加文档（4）小节所述模型唯一不同的地方在于，这里多了 initial equation 部分，即包含方程  $x = 2$ 。在前面的例子中，变量  $x$  的初始值在模型仿真开始时是不确定的。通常，这意味着变量  $x$  的初始值是它的 start 属性（其缺省值为零）。然而，每个工具都会使用自己特定的算法对最终的方程组进行表示。因此，我们最好明确的声明方程的初始化条件，就像上述模型声明的那样。通过在 initial equation 部分添加相应的方程，我们就明确的完成了变量  $x$  的初始化声明。

如下图所示，是否对变量进行初始化定义，其方程解的轨迹完全不同：



FirstOrderInitial 模型展示了初始化系统的典型方式：为系统状态提供明确的初始值。事实上，没有确定初始条件的微分方程组并不完整。FirstOrderInitial 模型可以用数学方程表示如下：

$$\dot{x} = (1 - x); \quad x(0) = 2$$

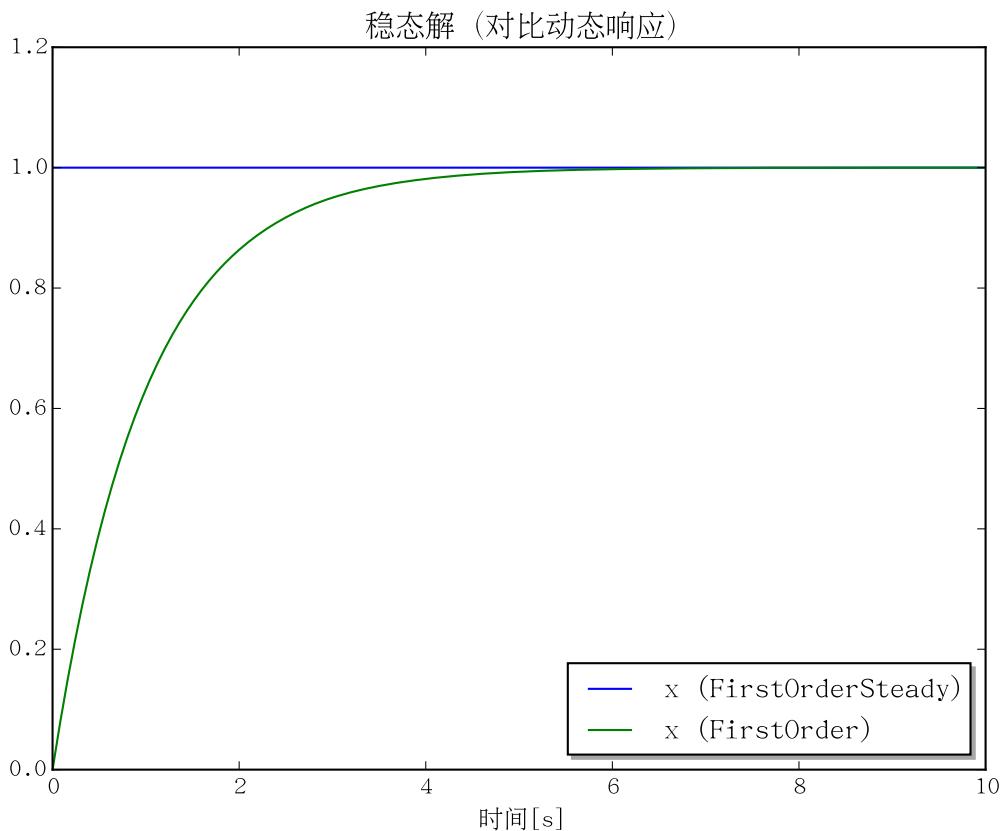
当然了，很多情况下，你也许需要更为复杂的初始化条件。initial equation 区域不仅可以包含显示方程的状态变量初始值。

例如，我们可能希望在仿真开始时， $x$  的导数为零。稍有点代数知识的就会知道，这可以通过指定初始

条件  $x(0) = 1$  来实现。但是，对于更复杂的系统，通过这种方式确定初始条件是很困难的。在这些情况下，可以在 Modelica 中直接添加初始条件  $\dot{x}(0) = 0$ ，如下所示：

```
model FirstOrderSteady
  "First order equation with steady state initial condition"
  Real x "State variable";
initial equation
  der(x) = 0 "Initialize the system in steady state";
equation
  der(x) = 1-x "Drives value of x toward 1.0";
end FirstOrderSteady;
```

仿真该系统会得到以下结果：



从这些结果中可以看到，因为没有破坏系统平衡的外部激励， $x$  的导数在仿真开始以及仿真过程中一直保持为 0。

本章节提供了 Modelica 初始化功能的概述，有关初始化功能更完整的内容将在后面的[初始化 \(31\)](#) 章节中介绍。

## 试验条件

当搭建模型时，建模人员可能希望为模型关联特定的试验条件。这可以通过应用 annotation（标注）来完成。标注中包含的信息与模型的属性没有直接联系。

例如，试验条件的描述包括仿真开始时间、结束时间以及容差范围等，这些信息并不描述模型的特性，只是提供进行模拟仿真的条件。试验条件通过特定的 experiment 标注保存在模型中。

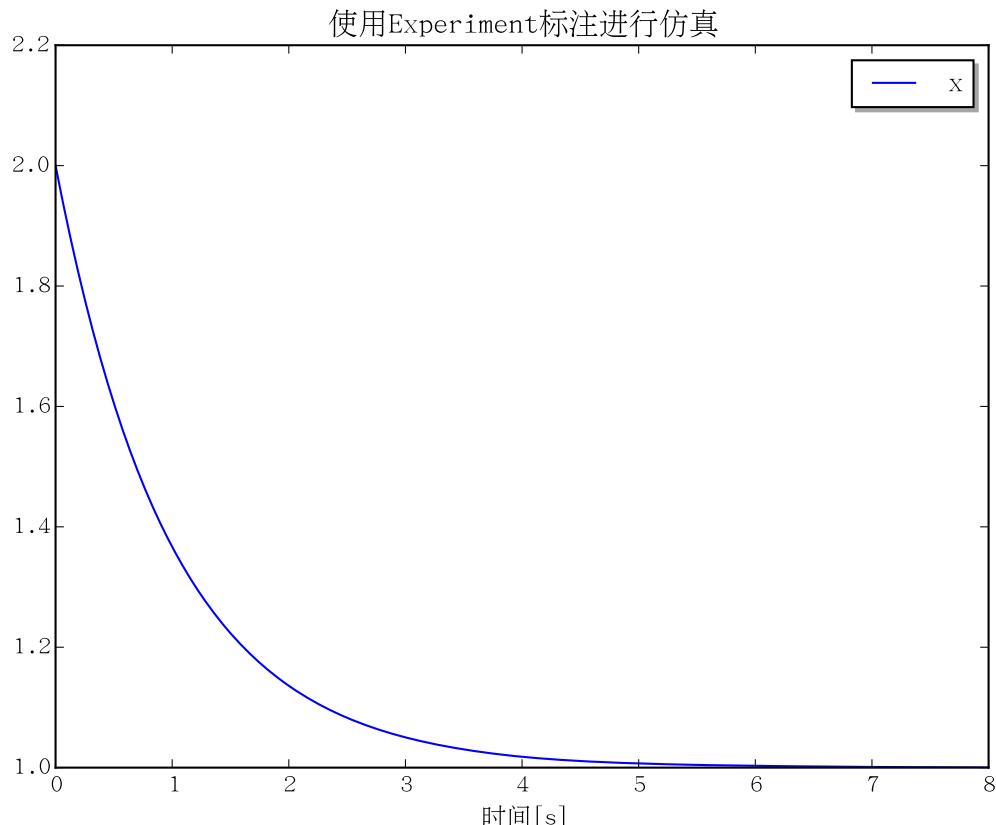
可以通过设定 4 个参数完成试验条件的设置，这 4 个参数都是可选的。如下所示为包含试验条件的一阶系统模型：

```

model FirstOrderExperiment "Defining experimental conditions"
  Real x "State variable";
initial equation
  x = 2 "Used before simulation to compute initial values";
equation
  der(x) = 1-x "Drives value of x toward 1.0";
  annotation(experiment(StartTime=0,StopTime=8));
end FirstOrderExperiment;

```

下图是使用这些试验条件的仿真结果:



从图中可以看到，在 8 秒时仿真终止，因为求解器通过 experiment 确定了仿真的运行时间。

### 标注应用

虽然 experiment 标注在建立模型时被广泛的应用，但是也要注意到，一般情况下，建模工具是可以忽略任何或者全部标注内容的。

## 第 1.1.2 节 引入物理

虽然前面的章节让我们对模型的数学特性有了一定的了解，但是并没有包含与模型物理属性相关的内容。在本章节，我们将探索如何建立能体现其物理属性的模型。在此过程中，我们将重点介绍建模语言的一些特性，利用这些特性不仅可以将模型的物理特性和相应的工程领域结合起来，甚至还可以帮助我们避免一些错误。

让我们先从下面的例子开始:

```

model NewtonCooling "An example of Newton's law of cooling"
  parameter Real T_inf "Ambient temperature";

```

```

parameter Real T0 "Initial temperature";
parameter Real h "Convective cooling coefficient";
parameter Real A "Surface area";
parameter Real m "Mass of thermal capacitance";
parameter Real c_p "Specific heat";
Real T "Temperature";
initial equation
  T = T0 "Specify initial value for T";
equation
  m*c_p*der(T) = h*A*(T_inf-T) "Newton's law of cooling";
end NewtonCooling;

```

正如我们在前面章节 [简单的一阶系统 \(3\)](#) 中的实例讨论的那样, model 的定义包括变量和公式。

但是, 上述代码让我们第一次看到了关键字 parameter。一般来说, 关键字 parameter 用于表明变量的值是先验已知的 (即仿真开始之前)。更确切的说, 关键字 parameter 是用于指定变量的可变性。在[可变性 \(26\)](#) 一节中, 我们将会对这部分内容进行详细的讨论。至于现在, 我们可以将关键字 parameter 看作是一个必须提供赋值的变量。

在 NewtonCooling 实例中, 我们可以看到五个参数: T<sub>inf</sub>、T<sub>0</sub>、h、A、m 以及 c<sub>p</sub>。我们不需要费心的去解释这些变量的含义, 因为模型本身已经为每个变量添加了描述性的字符串。目前, 这些参数都没有赋值。但是我们很快会回到这个话题。迄今为止, 我们所看到的变量类型都是 Real。

让我们来研究一下这个模型的其余部分。下一个变量名是 T (数据类型也是 Real)。因为这个变量没有关键字 parameter 的限定, 因此, 它的值是通过模型中的方程确定的。

接下来, 我们看到的是两个 equation 部分。第一个是 initial equation 区域。该部分指定变量 T 如何进行初始化。从上面 initial equation 区域可以清楚的看到, 变量 T 的初始值是由参数 T<sub>0</sub> 决定的, 因此我们可以为它赋任何初值。

另一条方程是关于参数 T 的微分方程。数学上, 我们可以用以下方程进行描述:

$$mc_p \dot{T} = hA(T_{\infty} - T)$$

但是, 在 Modelica 语言中, 我们将它写成如下形式:

```
m*c_p*der(T) = h*A*(T_inf-T)
```

我们可以注意到, 本实例和 [简单的一阶系统 \(3\)](#) 的 FirstOrder 模型实例相比, 在公式的写法上并没有什么不同。

有一点值得注意的是, 在 NewtonCooling 实例中的方程, 等式左侧也包含相应的表达式。在 Modelica 中, **没有必要去明确某个方程是哪个单一变量的确定方程**。一个公式可以在等号的两边包含任意表达式。通过确定如何使用这些方程, 然后求解包含在等式中的变量是编译器的工作。

另一个区分 NewtonCooling 和 FirstOrder 模型实例的特点是我们可以独立的调整不同参数的值。此外, 这些参数的值和模型的物理特性, 材料或环境条件的可测量特性相关联。换句话说, 相比于 FirstOrder 实例中的简单数学关系, 这个版本稍微能体现实际系统的物理特性, 因为模型参数本身就是和物理属性相关联的。

现在, 我们还不能运行 NewtonCooling 模型, 因为缺少六个参数值。为了创建可以用于仿真的模型, 我们需要提供上述参数值, 即:

```

model NewtonCoolingWithDefaults "Cooling example with default parameter values"
  parameter Real T_inf=25 "Ambient temperature";
  parameter Real T0=90 "Initial temperature";
  parameter Real h=0.7 "Convective cooling coefficient";
  parameter Real A=1.0 "Surface area";
  parameter Real m=0.1 "Mass of thermal capacitance";
  parameter Real c_p=1.2 "Specific heat";
  Real T "Temperature";
initial equation
  T = T0 "Specify initial value for T";
equation

```

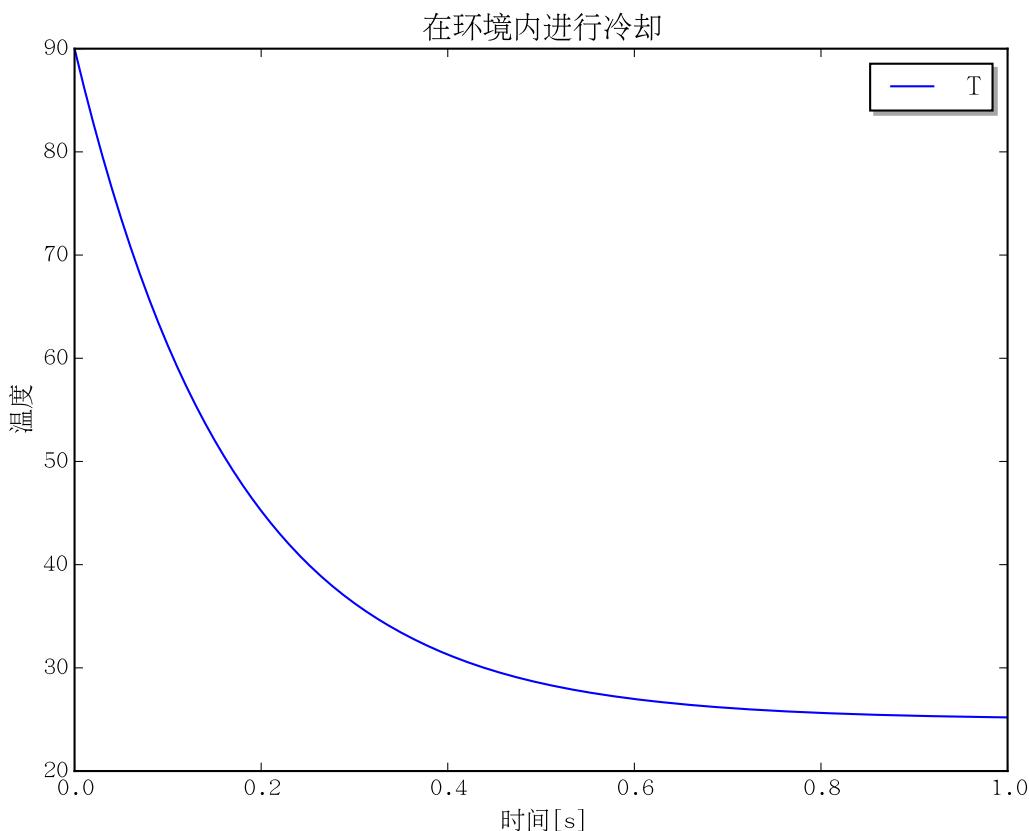
```

m*c_p*der(T) = h*A*(T_inf-T) "Newton's law of cooling";
end NewtonCoolingWithDefaults;

```

这里唯一的区别是，关键字 parameter 后面的变量都有了确定值。一种分析 NewtonCooling 模型不能仿真原因的方法在于，模型总共包含 7 个变量却只有一个方程（参考 [初始化 \(31\)](#) 以了解 initial equation 区域不能算作方程的原因）。但是，从概念上讲，NewtonCoolingWithDefaults 模型包含 7 个方程（6 个方程是对 parameter 中变量的初始化，另一个在方程区域）和 7 个未知量。

如果我们将 NewtonCoolingWithDefaults 模型进行仿真，可以得到变量 T 解的轨迹，如下所示：



## 物理单位

正如本节已经提到的那样，这些例子体现更多的物理属性。因为，模型包含各自对应于现实世界体系中各个属性的物理参数。然而，我们仍然缺少一些物理属性。尽管这些变量代表了物理量如温度，质量等方面，我们并没有明确为其规定任何物理类型。

正如你猜测的那样，变量 T 表示温度，在与变量相关的描述性文本中已经明确进行了表述。此外，即使不对此前的模型进行很深入的分析，大家也可以确定变量 T0 和 T\_inf 均须是温度。

但是对于其他变量，比如变量 h 或 A，它们表示什么？更重要的是，这些方程是否具有物理一致性？物理一致性表述的是方程等式两边是否具有相同的物理单位（即温度、质量、功率等）。

我们可以通过在变量声明的时候包含相应的物理单位，以更严格地表述不同变量的单位。如下所示：

```

model NewtonCoolingWithUnits "Cooling example with physical units"
  parameter Real T_inf(unit="K")=298.15 "Ambient temperature";
  parameter Real T0(unit="K")=363.15 "Initial temperature";
  parameter Real h(unit="W/(m2.K)")=0.7 "Convective cooling coefficient";
  parameter Real A(unit="m2")=1.0 "Surface area";
  parameter Real m(unit="kg")=0.1 "Mass of thermal capacitance";
  parameter Real c_p(unit="J/(K.kg)")=1.2 "Specific heat";

```

```

Real T(unit="K") "Temperature";
initial equation
  T = T0 "Specify initial value for T";
equation
  m*c_p*der(T) = h*A*(T_inf-T) "Newton's law of cooling";
end NewtonCoolingWithUnits;

```

你可以注意到，每个变量的声明部分都包含与变量相关联的物理单元文本 (unit="...")。这些附加文本的作用是为了指定与变量相关联的 unit (物理单位) 属性值。属性是每个变量所具有的特质。变量的属性集取决于变量的类型 (在接下来的章节将对变量部分进行更详细的讨论，参考 [变量 \(26\)](#))。

初看起来，似乎不是很明白，指定变量的物理单位属性 (比如 (unit="K")) 还不如简单在变量声明的后面添加描述性字符串 "Temperature"。事实上，人们甚至可以说指定变量的物理属性更糟糕。因为 "Temperature" 比单一的字母 "K" 更具有描述性。

然而，设置 unit (单位) 属性实际上有两个原因。第一个原因是，Modelica 语言规范定义了所有标准国际单位属性之间的关系 (例如 K、kg、m)。这包括由其他基本单位构成的复杂单位类型 (例如 N)。

另一个原因是，Modelica 语言还规定了如何进行复杂数学表达式单位计算的规则。在这种方式下，Modelica 语言标准定义了所有与 [单位属性检查](#) 相关的规则，以便检测 Modelica 模型中单位属性的错误或不一致。这对模型开发者来说是一个巨大的优势，因为添加物理单位属性不仅使得模型结构更清晰，而且还提供了出现错误情况下更好的诊断方法。

## 物理类型

但是，实际上在 NewtonCoolingWithUnits 实例中有一个缺陷，即我们必须对每个变量的 unit 属性重复定义。另外，如前所述，"Temperature" 比单一字母 K 更具有描述性。

幸运的是，对上述问题我们有一个简单的解决方法，即 Modelica 语言允许我们定义 derived types (派生类型)。到目前为止，我们所声明的变量类型只有 Real (实型)。Real 类型的问题是它可以是任何东西 (例如：电压、电流、温度)。我们想做的是如何稍微限制一下变量类型。这也就是派生类型产生的原因。要了解如何定义和声明派生类型，考虑下面的例子：

```

model NewtonCoolingWithTypes "Cooling example with physical types"
  // Types
  type Temperature=Real(unit="K", min=0);
  type ConvectionCoefficient=Real(unit="W/(m2.K)", min=0);
  type Area=Real(unit="m2", min=0);
  type Mass=Real(unit="kg", min=0);
  type SpecificHeat=Real(unit="J/(K.kg)", min=0);

  // Parameters
  parameter Temperature T_inf=298.15 "Ambient temperature";
  parameter Temperature T0=363.15 "Initial temperature";
  parameter ConvectionCoefficient h=0.7 "Convective cooling coefficient";
  parameter Area A=1.0 "Surface area";
  parameter Mass m=0.1 "Mass of thermal capacitance";
  parameter SpecificHeat c_p=1.2 "Specific heat";

  // Variables
  Temperature T "Temperature";
initial equation
  T = T0 "Specify initial value for T";
equation
  m*c_p*der(T) = h*A*(T_inf-T) "Newton's law of cooling";
end NewtonCoolingWithTypes;

```

你可以看到关于类型的定义 type Temperature=Real(unit="K", min=0);。其意思是 “” 让我们定义一个新类型 Temperature。这是内置 Real 类型的一个特例。其物理单位为开尔文 (K)。最小可能值则为 0”。

从这个例子中，我们可以看到，一旦我们定义了一个物理类型比如 Temperature，我们可以使用它为多个变量 (例如 T、T\_inf 和 T0) 进行声明，而无需为每一个变量指定 unit (单位属性) 或 min (最小值

属性)。另外，我们可以使用所熟悉的名字 Temperature 来代替国际单位 K。你可能想知道为什么创建了派生类型后，其他变量也可以获取其属性。如果需要进一步的了解，可以参考章节[内建类型 \( 26 \)](#)。

在这一点上，你会发现若要在每一个模型中都要定义变量 Temperature、ConvectionCoefficient、SpecificHeat、Mass，这会是一件极其乏味的事情。但是如果模型有这方面需要的话，还必须这样做。不过不用担心。我们会在后面关于[导入物理类型 \( 144 \)](#)的章节中讨论一个简单的解决方案。

### 第 1.1.3 节 电气方面的实例

现在，让我们回到工程应用方面。那些对电气系统比较熟悉的读者，考虑下面的回路：

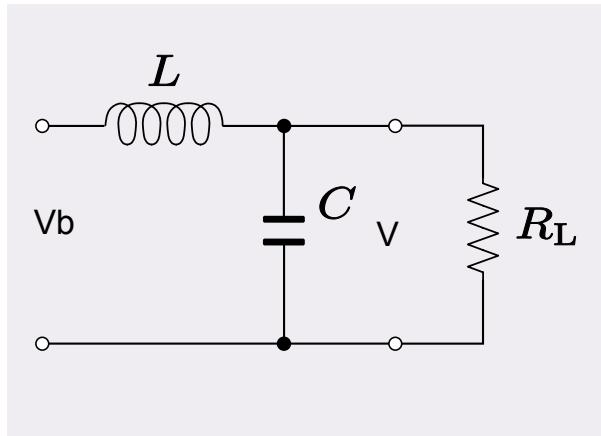


图 第一章.1: 低通 RLC 滤波器

假设我们要去求解以下变量： $V$ 、 $i_L$ 、 $i_R$  以及  $i_C$ 。我们可以使用下面与电感、电容以及电阻有关的方程来求解各个电流值  $i_L$ 、 $i_R$  及  $i_C$ 。

$$V = i_R R$$

$$C \frac{dV}{dt} = i_C$$

$$L \frac{di_L}{dt} = (V_b - V)$$

其中， $V_b$  表示电池电压。

因为我们只有 3 个方程，却要求解 4 个变量，因此我们需要一个附加的方程。附加方程根据基尔霍夫电流定律来得到，如下所示：

$$i_L = i_R + i_C$$

我们已经确定了求解这个问题所需的方程和变量。现在，我们直接把这些方程转换为 Modelica 语言，以此创建一个基本模型 (模型包括物理类型)。但是，在后面的章节[电气部件 \( 181 \)](#)中，我们还会回到这个电路。到时我们会演示如何通过拖放以及连接模型来创建电路。最后，我们会得出外观上类似于[低通 RLC 滤波器 \( 11 \)](#)的电路模型。

但是现在，我们将建立一个由简单变量和方程组成的模型。这可以表示为如下形式：

```
model RLC1 "A resistor-inductor-capacitor circuit model"
  type Voltage=Real(unit="V");
  type Current=Real(unit="A");
  type Resistance=Real(unit="Ohm");
  type Capacitance=Real(unit="F");
  type Inductance=Real(unit="H");
```

```

parameter Voltage Vb=24 "Battery voltage";
parameter Inductance L = 1;
parameter Resistance R = 100;
parameter Capacitance C = 1e-3;
Voltage V;
Current i_L;
Current i_R;
Current i_C;
equation
  V = i_R*R;
  C*der(V) = i_C;
  L*der(i_L) = (Vb-V);
  i_L=i_R+i_C;
end RLC1;

```

让我们一点点的来检查这个例子，并且加强理解各种语句的含义。让我们从模型定义部分开始：

```
model RLC1 "A resistor-inductor-capacitor circuit model"
```

我们可以看到，模型的名字是 RLC1。此外，它还包含关于这个模型的描述，即”A resistor-inductor-capacitor circuit model”。接下来，我们定义了需要用到的几个物理类型：

```

type Voltage=Real(unit="V");
type Current=Real(unit="A");
type Resistance=Real(unit="Ohm");
type Capacitance=Real(unit="F");
type Inductance=Real(unit="H");

```

每一行都通过关联特定的物理单位定义了一种指定了嵌入的 Real 类型的物理类型。然后，我们声明所有的 parameter 变量：

```

parameter Voltage Vb=24 "Battery voltage";
parameter Inductance L = 1;
parameter Resistance R = 100;
parameter Capacitance C = 1e-3;

```

这些 parameter 变量分别代表了不同的物理特性（这里分别表示电压、电感、电阻和电容）。最后定义我们想要求解的变量，即：

```

Voltage V;
Current i_L;
Current i_R;
Current i_C;

```

现在，所有的变量都已经声明了。我们要在模型中添加 equation 区域，去定义在生成解时使用的方程组。

```

equation
  V = i_R*R;
  C*der(V) = i_C;
  L*der(i_L) = (Vb-V);
  i_L=i_R+i_C;

```

最后，我们通过创建包含 model 名字的 end 语句去结束模型（例如，本例中模型名为 RLC1）：

```
end RLC1;
```

本实例和前面实例的区别在于，本实例包含更多的方程。在 NewtonCooling 例子中，有些方程左右两边都有表达式。其中还有一个微分方程（包含一个变量的微分）和简单的代数方程的混合方程。

这也进一步强调了，Modelica 不同于其他的建模环境。我们完全没有必要把方程变成所谓的“显式状态空间形式”。当然，我们可以把方程组重新整理成更加明确的形式，如下所示：

```

der(V) = i_C/C;
der(i_L) = (Vb-V)/L;

```

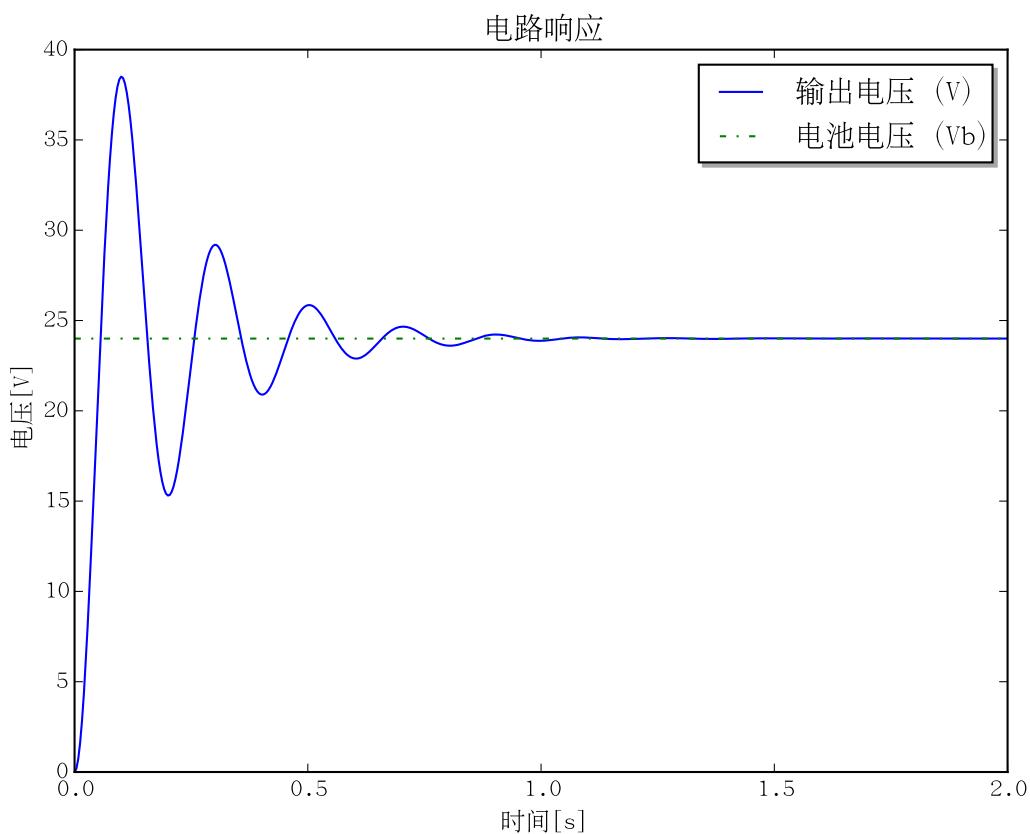
```
i_R = i_L - i_C;
V = i_R * R;
```

重要的是，在 Modelica 中我们没有必要进行这样的操作。相反的，我们可以自由的将方程组写成任何想要的形式。

最终，这些方程可能需要转换成类似显式状态空间的形式。但是，如果这样的转换是必须的，那么，这将是 Modelica 语言编译器的工作，而不需要模型开发者去做。这让模型开发者不再有必要去处理这些繁琐、费时且容易出错的任务。

将方程保存成其自有“教科书形式”这一功能非常重要。在我们后续章节中将会介绍到，这些方程将会“包含在”各个组件模型内。在这些情况下，我们（在创建组件模型时）无法明确知道用于求解每个方程所使用的变量。Modelica 语言编译器所执行的这些操作不仅使模型开发更快、更容易，而且也显著提高了模型的可重用性。

下图显示了模型 RLC1 的动态响应：

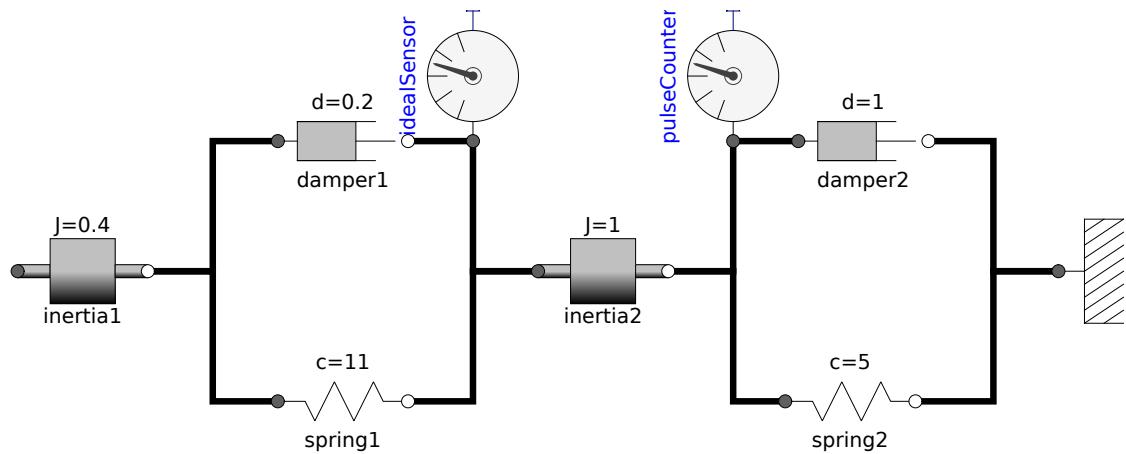


### 电气实例的扩展

如序言 ( III) 所述，这本书的结构基于超媒体的浏览方法。这可以使读者更方便地处理与其目标、兴趣相一致的材料。下一章将展示机械方面的模型，模型内的方程是基于机械系统原理获得的。但如果你希望看到包含更复杂的电气系统特性的实例，你可以直接跳到下面的 [RLC 开关电路 \( 57\)](#) 实例中。

### 第 1.1.4 节 机械示例

如果你对机械系统比较熟悉，这个例子可能有助于你理解前面所介绍的一些概念。我们期望建模的系统如下图所示：



值得指出的是以示意图的形式表达模型的意图是多么容易。假设通过适当的图示表达，专家就可以很快的理解系统的组成并对系统可能具有的特性有一定的了解。虽然现在我们专注于方程和变量的定义，我们最终将转换工作方式（在本书后续章节组件（173）将会讲解），**在开始建模的时候就以系统示意图的形式来搭建。**

现在，我们依然将重点放在如何用方程来表示这个简单的机械系统。每个与惯量相关的角位移  $\varphi$ ，及角速度  $\omega$  之间有这样的关系式： $\omega = \dot{\varphi}$ 。对于每个惯量，根据角动量守恒可以得到如下表达式：

$$J\dot{\omega} = \sum_i \tau_i$$

换句话说，即施加在惯量元素上的扭矩  $\tau$  总和等于转动惯量  $J$  和角加速度  $\dot{\omega}$  的乘积。

在上述方程中，我们唯一缺少的是扭矩值  $\tau_i$ 。从上面的示意图上我们可以看到，上述机械系统包含两个弹簧和两个阻尼器。对于弹簧，我们可以根据胡克定律来表示扭矩和角位移之间的关系，如下所示：

$$\tau = k\Delta\varphi$$

对每个阻尼器，我们可以用下方程表示其扭矩和相对角速度之间的关系：

$$\tau = d\Delta\dot{\varphi}$$

如果我们将所有的表达式放在一起，可以得到如下的系统方程：

$$\begin{aligned}\omega_1 &= \dot{\varphi}_1 \\ J_1\dot{\omega}_1 &= k_1(\varphi_2 - \varphi_1) + d_1 \frac{d(\varphi_2 - \varphi_1)}{dt} \\ \omega_2 &= \dot{\varphi}_2 \\ J_2\dot{\omega}_2 &= k_1(\varphi_1 - \varphi_2) + d_1 \frac{d(\varphi_1 - \varphi_2)}{dt} - k_2\varphi_2 - d_2\dot{\varphi}_2\end{aligned}$$

我们假设系统具有以下的初始条件：

$$\begin{aligned}\varphi_1 &= 0 \\ \omega_1 &= 0 \\ \varphi_2 &= 1 \\ \omega_2 &= 0\end{aligned}$$

这些初始化条件意味着系统的开始状态惯量元素没有转动（即  $\omega = 0$ ），但是在两个弹簧之间有一个非零的偏转。

将上述所有变量和方程放在一起，我们就可以用 Modelica 语言来描述这个问题，如下所示：

```

model SecondOrderSystem "A second order rotational system"
  type Angle=Real(unit="rad");
  type AngularVelocity=Real(unit="rad/s");
  type Inertia=Real(unit="kg.m2");
  type Stiffness=Real(unit="N.m/rad");
  type Damping=Real(unit="N.m.s/rad");
  parameter Inertia J1=0.4 "Moment of inertia for inertia 1";
  parameter Inertia J2=1.0 "Moment of inertia for inertia 2";
  parameter Stiffness k1=11 "Spring constant for spring 1";
  parameter Stiffness k2=5 "Spring constant for spring 2";
  parameter Damping d1=0.2 "Damping for damper 1";
  parameter Damping d2=1.0 "Damping for damper 2";
  Angle phi1 "Angle for inertia 1";
  Angle phi2 "Angle for inertia 2";
  AngularVelocity omega1 "Velocity of inertia 1";
  AngularVelocity omega2 "Velocity of inertia 2";
initial equation
  phi1 = 0;
  phi2 = 1;
  omega1 = 0;
  omega2 = 0;
equation
  // Equations for inertia 1
  omega1 = der(phi1);
  J1*der(omega1) = k1*(phi2-phi1)+d1*der(phi2-phi1);
  // Equations for inertia 2
  omega2 = der(phi2);
  J2*der(omega2) = k1*(phi1-phi2)+d1*der(phi1-phi2)-k2*phi2-d2*der(phi2);
end SecondOrderSystem;

```

像我们在低通滤波器 RLC1 例子中讲解的那样，让我们一步步的来讲解上述模型。

像往常一样，我们先从模型的名称开始：

```
model SecondOrderSystem "A second order rotational system"
```

接下来，我们介绍旋转机械系统的物理类型，即：

```

type Angle=Real(unit="rad");
type AngularVelocity=Real(unit="rad/s");
type Inertia=Real(unit="kg.m2");
type Stiffness=Real(unit="N.m/rad");
type Damping=Real(unit="N.m.s/rad");

```

然后，我们定义表示系统不同物理特性的各种参数：

```

parameter Inertia J1=0.4 "Moment of inertia for inertia 1";
parameter Inertia J2=1.0 "Moment of inertia for inertia 2";
parameter Stiffness k1=11 "Spring constant for spring 1";
parameter Stiffness k2=5 "Spring constant for spring 2";
parameter Damping d1=0.2 "Damping for damper 1";
parameter Damping d2=1.0 "Damping for damper 2";

```

对于这个系统，有四个非 parameter 变量。定义如下：

```

Angle phi1 "Angle for inertia 1";
Angle phi2 "Angle for inertia 2";
AngularVelocity omega1 "Velocity of inertia 1";
AngularVelocity omega2 "Velocity of inertia 2";

```

然后定义初始条件（我们很快会重温这一知识点）：

```
initial equation
  phi1 = 0;
```

```
phi2 = 1;
omega1 = 0;
omega2 = 0;
```

然后定义系统的动态响应方程:

```
equation
  // Equations for inertia 1
  omega1 = der(phi1);
  J1*der(omega1) = k1*(phi2-phi1)+d1*der(phi2-phi1);
  // Equations for inertia 2
  omega2 = der(phi2);
  J2*der(omega2) = k1*(phi1-phi2)+d1*der(phi1-phi2)-k2*phi2-d2*der(phi2);
```

最后，定义模型的结束。

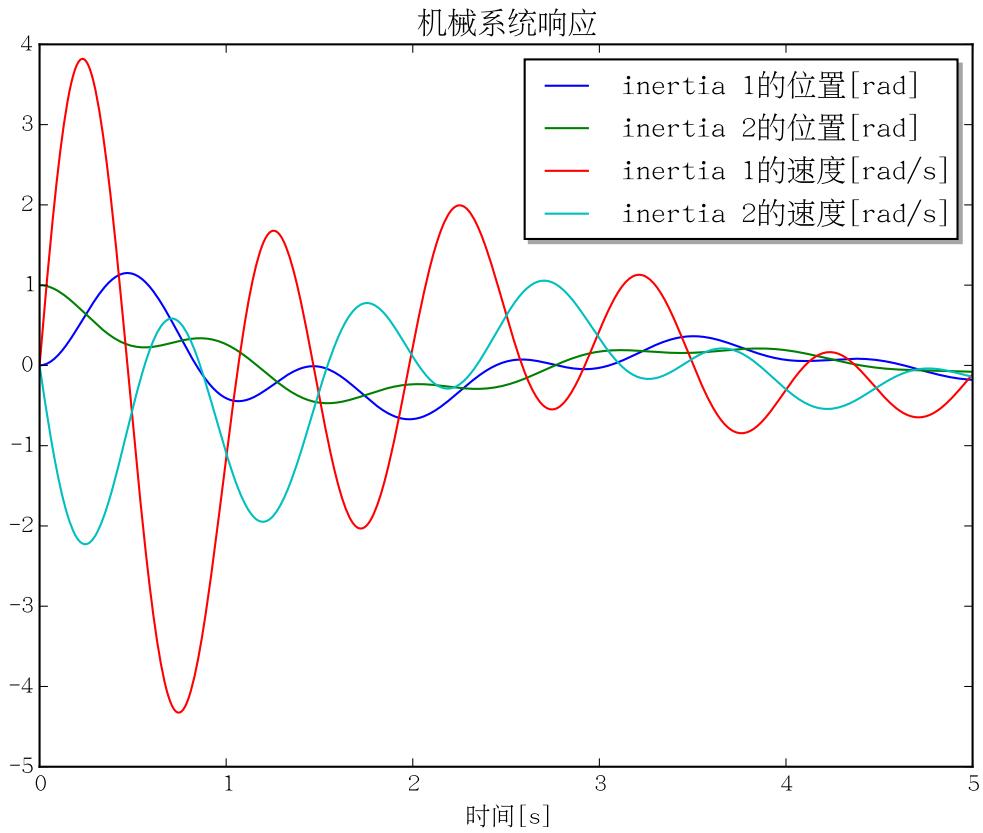
```
end SecondOrderSystem;
```

这个模型的唯一缺点是我们所有的初始化条件已经被“硬编码”到模型中。这也意味着，我们将不能指定任何其他组调用该模型的初始条件。我们可以克服这个问题，就像在Newton cooling examples (7) 例子中通过定义 parameter 变量来表示初始条件，如下所示:

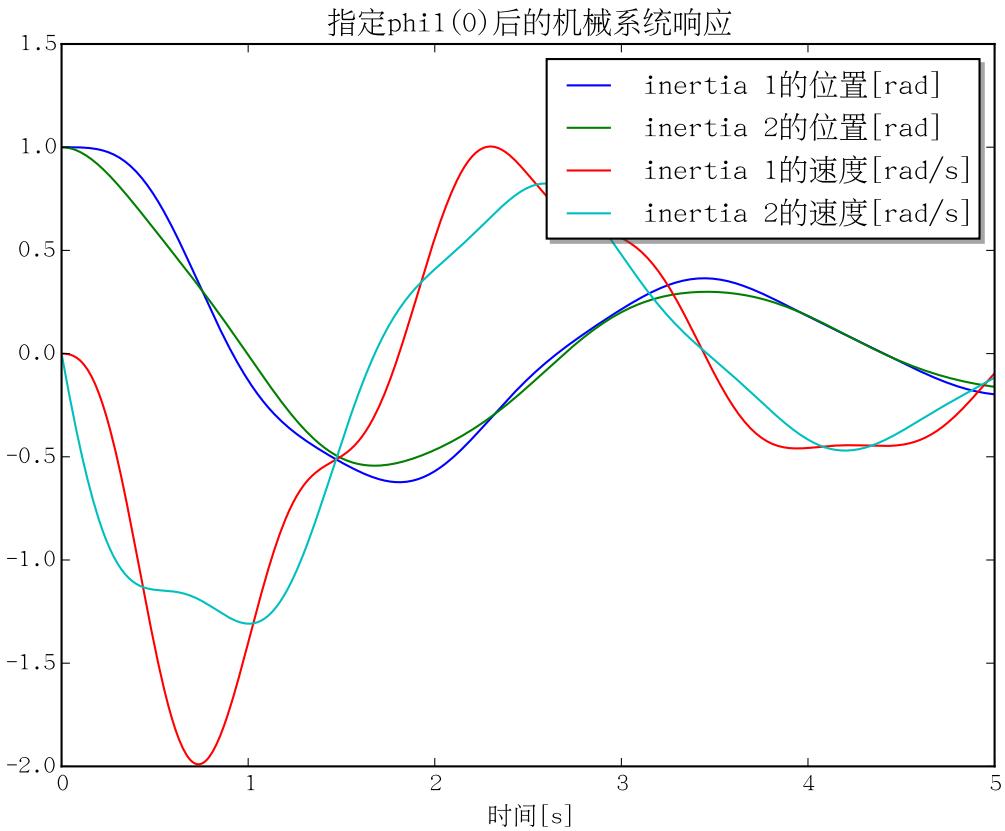
```
model SecondOrderSystemInitParams
  "A second order rotational system with initialization parameters"
  type Angle=Real(unit="rad");
  type AngularVelocity=Real(unit="rad/s");
  type Inertia=Real(unit="kg.m2");
  type Stiffness=Real(unit="N.m/rad");
  type Damping=Real(unit="N.m.s/rad");
  parameter Angle phi1_init = 0;
  parameter Angle phi2_init = 1;
  parameter AngularVelocity omega1_init = 0;
  parameter AngularVelocity omega2_init = 0;
  parameter Inertia J1=0.4 "Moment of inertia for inertia 1";
  parameter Inertia J2=1.0 "Moment of inertia for inertia 2";
  parameter Stiffness k1=11 "Spring constant for spring 1";
  parameter Stiffness k2=5 "Spring constant for spring 2";
  parameter Damping d1=0.2 "Damping for damper 1";
  parameter Damping d2=1.0 "Damping for damper 2";
  Angle phi1 "Angle for inertia 1";
  Angle phi2 "Angle for inertia 2";
  AngularVelocity omega1 "Velocity of inertia 1";
  AngularVelocity omega2 "Velocity of inertia 2";
initial equation
  phi1 = phi1_init;
  phi2 = phi2_init;
  omega1 = omega1_init;
  omega2 = omega2_init;
equation
  omega1 = der(phi1);
  omega2 = der(phi2);
  J1*der(omega1) = k1*(phi2-phi1)+d1*der(phi2-phi1);
  J2*der(omega2) = k1*(phi1-phi2)+d1*der(phi1-phi2)-k2*phi2-d2*der(phi2);
end SecondOrderSystemInitParams;
```

通过这种方式，该参数值即可以在仿真环境中更改（参数值通常会被用户编辑）。另外，我们也可以通过所谓的修改语句（modifications）来改变参数。

在这个最新版本的模型中你会看到，新设置的参数和之前硬编码参数的数值是一样的。因此，默认的初始条件和之前的完全一样。但是现在，我们有充分的自由去探索其他初始化条件的方法。例如，我们仿真 SecondOrderSystemInitParams 模型，可以得到角位移及角速度解的轨迹，如下图:



但是，如果将参数 `phi1_init` 的值修改为 1，可以得到以下的仿真结果：



### 机械示例的扩展

如果你想了解本实例更深的扩展，可以直接跳到包含更多旋转系统实例的速度的测量（59）一节里。否则，你可以继续下面涉及种群动力学方面的实例。

## 第 1.1.5 节 猎食者猎物系统

目前为止，我们已经介绍了热学、电学和机械方面的实例。实际上，这些都是工程实例。然而，Modelica 语言并不仅限于工程科学。为了强化这一点，本节将介绍基于猎食者和猎物之间关系的共同生态系统动力学模型。我们将使用[Lotka] (331)-[Volterra] (331) 方程去建立上述模型。

### 经典猎食者猎物系统

经典的猎食者猎物模型<sup>1</sup>涉及两个物种。一个物种叫做“猎物”。本节中，其物种种群用  $x$  表示。另一个物种称为“猎食者”，其物种种群用  $y$  表示。

有三个因素对猎食者猎物系统影响较大。首先是“猎物”物种的繁殖率。假设其繁殖率和种群成正比。如果你对化学反应比较熟悉，质量作用定律<sup>2</sup> [Guldberg] (331) 和其在概念上是完全相同的。如果你对质量作用定律不了解，可以理解为生存环境中潜在的配偶越多，种群的繁殖率就会越高。我们可以用数学公式表达如下：

$$\dot{x}_r = \alpha x$$

其中， $x$  表示猎物的种群， $\alpha$  表示种群繁殖率的比例常数， $\dot{x}_r$  表示由于繁殖引起的种群变化。

其次影响猎食者猎物系统的因素是猎食者的饥饿程度。如果没有足够的“猎物”来充饥，部分猎食者就会饿死。在建模饥饿程度时，要考虑竞争关系对其的重要影响。我们也是用比例关系描述上述模型。不同于繁殖率模型，此次是反比例关系，因为越多的猎食者越容易引发饥饿。在数学表达式上，这和繁殖率模型具有相同的表达形式：

$$\dot{y}_s = -\gamma y$$

其中， $y$  表示猎食者种群。 $\gamma$  表示种群饥饿的比例常数。 $\dot{y}_s$  表示由于饥饿引起的猎食者种群变化。

最后一个我们要考虑的因素是“捕食”，即猎食者对猎物的消耗。如果没有天敌，猎物会指数增长（至少从数学观点）。因此，捕食是保持猎物种群的重要因素。同样的，没有猎物，猎食者也会全部死光。因此，捕食平衡了这种效应，并且防止猎食者种群的消失。再次的，我们得到一个比例关系。但是，这个比例关系实际上是一个双线性关系。而且，概念上也类似于质量作用定律。这种比例关系在数学上很容易得到。事实上，猎食者发现并捕获猎物的机会与猎物种群、猎食者种群两者都存在着正比关系。由于这种特殊的影响因素，需要两个物种都参与。因此，这里的数学表达式与前述的繁殖率及猎食者饥饿程度的公式都在结构上有所不同，即：

$$\begin{aligned}\dot{x}_p &= -\beta xy \\ \dot{y}_p &= \delta xy\end{aligned}$$

其中， $\dot{x}_p$  表示由于捕食而导致的猎物种群减少。 $\dot{y}_p$  表示由于捕食而导致猎食者种群的增加。 $\beta$  表示捕获猎物概率的比例常数。 $\delta$  表示猎食者由于足够的营养以提高繁殖率可能性的比例常数。

全面考虑各种因素，可以得到每个种群的整体变化。用以下两个方程表示：

$$\begin{aligned}\dot{x} &= \dot{x}_r + \dot{x}_p \\ \dot{y} &= \dot{y}_p + \dot{y}_s\end{aligned}$$

根据上述两个方程，通过数学运算，可以重新整合方程等式的右侧，得到下方程组：

$$\begin{aligned}\dot{x} &= x(\alpha - \beta y) \\ \dot{y} &= y(\delta x - \gamma)\end{aligned}$$

根据前几章我们学到的知识，将上述方程转化成 Modelica 语言应该相当简单：

<sup>1</sup> [http://en.wikipedia.org/wiki/Lotka-Volterra\\_equation](http://en.wikipedia.org/wiki/Lotka-Volterra_equation)

<sup>2</sup> [http://en.wikipedia.org/wiki/Law\\_of\\_mass\\_action](http://en.wikipedia.org/wiki/Law_of_mass_action)

```

model ClassicModel "This is the typical equation-oriented model"
  parameter Real alpha=0.1 "Reproduction rate of prey";
  parameter Real beta=0.02 "Mortality rate of predator per prey";
  parameter Real gamma=0.4 "Mortality rate of predator";
  parameter Real delta=0.02 "Reproduction rate of predator per prey";
  parameter Real x0=10 "Start value of prey population";
  parameter Real y0=10 "Start value of predator population";
  Real x(start=x0) "Prey population";
  Real y(start=y0) "Predator population";
equation
  der(x) = x*(alpha-beta*y);
  der(y) = y*(delta*x-gamma);
end ClassicModel;

```

在上述代码中，只有一点我们还没有讨论，即变量 `x` 和 `y` 的 `start` 属性。正如我们在前面引入物理 (7) 章节里对 NewtonCoolingWithUnits 示例所介绍的那样，我们可以为变量指定各种的属性（在后面的章节内建类型 (26) 中将详尽地讨论变量的属性）。我们已经在前面章节讨论了变量的 `unit` 属性。这是我们第一次看到 `start` 属性。

细心的读者可能已经注意到程序中对变量 `x0` 和 `y0` 的声明以及其所代表的种群初始值。根据前面实例的讲解，有人可能认为这些初始化条件是以如下的方式被模型获取的：

```

model ClassicModelInitialEquations "This is the typical equation-oriented model"
  parameter Real alpha=0.1 "Reproduction rate of prey";
  parameter Real beta=0.02 "Mortality rate of predator per prey";
  parameter Real gamma=0.4 "Mortality rate of predator";
  parameter Real delta=0.02 "Reproduction rate of predator per prey";
  parameter Real x0=10 "Initial prey population";
  parameter Real y0=10 "Initial predator population";
  Real x(start=x0) "Prey population";
  Real y(start=y0) "Predator population";
initial equation
  x = x0;
  y = y0;
equation
  der(x) = x*(alpha-beta*y);
  der(y) = y*(delta*x-gamma);
end ClassicModelInitialEquations;

```

然而，对于 `ClassicModel` 示例，我们采用了更简洁的方法。我们可以在声明变量的 `start` 属性时直接指定其初始条件。在初始化 (31) 章节中将对这一功能做简单的介绍。

值得注意的是，这种方法既有优点也有缺点。方法的优点是比较灵活。其实，`start` 属性更多的是一种绑定关系的暗示。如果 Modelica 编译器将某个特定变量识别为状态（即某个变量需要一个初始化条件），并且模型的 `initial equation` 部分没有提供非充足的初始条件，这时 `start` 属性就可以作为替代，声明变量的初始条件。换句话说，你可以认为 `start` 属性是在必须具备初始条件的情况下“后备初始条件”。

你需要注意 `start` 属性所具备的一些缺点。首先，此属性只是一个完全可以忽略的提示工具。其次，其是否被忽略也很难预测。因为哪些变量作为状态在不同的工具里选择也会不同。

避免上述缺点我们可以使用 `fixed` 属性（在内建类型 (26) 章节中作进一步讨论）。变量的 `fixed` 属性被用来通知编译器 `start` 属性必须作为初始条件来使用。换句话说，如下的 `initial equation` 部分：

```

Real x;
initial equation
  x = 5;

```

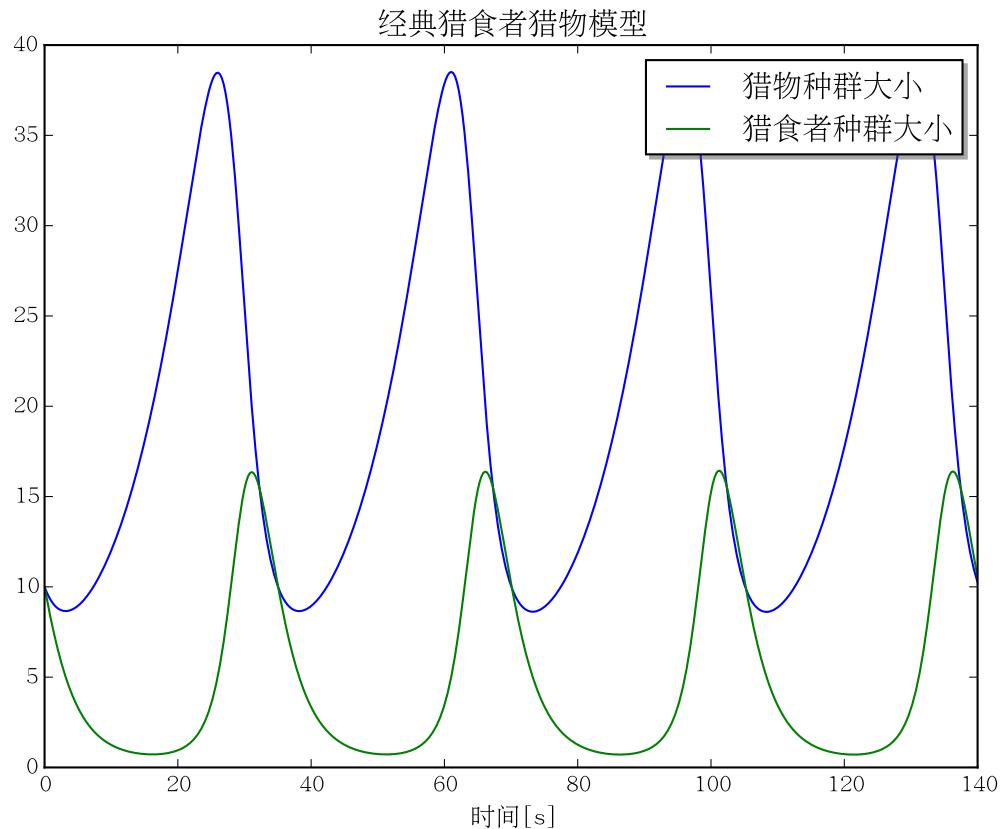
等同于下述变量声明中关于 `fixed` 和 `start` 属性的使用：

```
Real x(start=5, fixed=true);
```

最后，`start` 属性的另一个复杂之处是其功能被“重载”了。这就意味着，此属性有两种不同的用途。如果，所讨论的变量不是一个状态，而是一个“迭代变量”（即变量的解依赖于非线性方程组），那么 `start` 属性可以通过编译器作为初始的猜测值（即非线性求解方程首次迭代时使用的变量值）。

是否指定一个变量的 start 属性取决于你希望初始条件会得到多严格的。决定这点对于需要对语言有一定的经验。而这已经超出了本书的范围。然而，至少值得在这里指出不同选择的存在，以及简单解释一下权衡要素。

不管使用哪种初始化的方法，这些模型的仿真结果都是相同的。猎食者猎物系统的典型结果如下图所示：



通过上图，可以看到每个种群的周期性行为。最初，没有充足的食物来支撑较多的猎食者。存在的猎食者捕食任何可以找到的猎物。但即便如此，饥饿还是会发生并且导致猎食者种群数量的下降。在此期间，猎食者消耗猎物种群的速率如此之高，以致于猎物种群的繁殖速率不足以弥补因捕食而减少的种群数。因此，猎物种群的数量也在下降。

在某些点上，猎食者种群的数量变的如此之低，导致猎物种群的繁殖数量高于因捕食而减少的种群数。猎物种群的数量开始反弹。因为猎食者种群大小的回升需要较长的时间。所以在此期间，猎物种群的增长速度几乎不受猎食者种群的影响。最终，由于猎物的丰盛，猎食者种群的数量开始回升，直至系统恢复到猎食者和猎物种群的原始状态，然后整个循环再重演，无穷无尽。

该系统一次次的返回到相同的初始条件处（当然要忽略数值计算错误）。这一现象是系统最有趣的地方之一。特别是考虑猎食者猎物系统方程实际上是非线性的，这也更加令人注目了。

### 稳定状态初始化

让我们想象一下，这些物种种群的极端波动可能会导致一些不良的生态后果。在这种情况下，理解减少或消除这些波动的方法对整改系统将会很有帮助。一个简单的方法就是使两个种群维持在相对平衡的状态。但是，如何使用这些模型来帮助我们确定这样的“稳定”状态？

上述问题的答案就在于初始化条件。我们可以用能描述系统处于平衡状态的方程来初始化系统（相应的设置方法可以从[前面讲述的 \(6\) 实例 FirstOrderSteady 中获得](#)），而不是直接给猎食者和猎物种群指定初始值。幸运的是，Modelica 包含足够丰富的初始化方法，允许我们指定上述（或者其他）类型的初始化条件。

为了确保我们的系统开始的时候处于平衡状态，我们只需要简单的定义什么是平衡。从数学描述上来说，

系统如果满足以下两个条件，就可以说该系统是处于平衡状态的：

$$\begin{aligned}\dot{x} &= 0 \\ \dot{y} &= 0\end{aligned}$$

为了在 Modelica 模型中获取上述特征，我们只需在 initial equation 区域增加以下方程，如下所示：

```
model QuiescentModel "Find steady state solutions to LotkaVolterra equations"
parameter Real alpha=0.1 "Reproduction rate of prey";
parameter Real beta=0.02 "Mortality rate of predator per prey";
parameter Real gamma=0.4 "Mortality rate of predator";
parameter Real delta=0.02 "Reproduction rate of predator per prey";
Real x "Prey population";
Real y "Predator population";
initial equation
der(x) = 0;
der(y) = 0;
equation
der(x) = x*(alpha-beta*y);
der(y) = y*(delta*x-gamma);
end QuiescentModel;
```

上述模型和之前模型最大的区别就是包含突出显示的初始化方程。回到上述模型，你可能想知道那些初始化方程到底意味着什么？毕竟，我们需要求解的是变量  $x$  和  $y$ 。但是这些变量甚至都没有出现在我们的初始化方程内。系统是如何对这些变量求解的？

问题的答案在于理解函数  $x(t)$  和  $y(t)$  均是通过对带初始条件的微分方程进行积分来求解的。在仿真过程中，我们可以看到  $x$  和  $\dot{x}$  是通过下面的方程进行“耦合”的：

$$x(t) = \int_{t_0}^{t_f} \dot{x} \, dx + x(t_0)$$

(当然了，在  $y$  和  $\dot{y}$  之间也存在着类似的关系)

**但是**，在系统初始化的过程中（即在计算初始条件时），上述关系是不成立的。在这种情况下， $x$  和  $\dot{x}$  之间并不存在“耦合”关系（对  $y$  和  $\dot{y}$  并不适用）。换句话说，即便知道变量  $x$  和  $y$  是如何定义的也不能提供求解方程  $\dot{x}$  或  $\dot{y}$  的线索。在初始化问题上，我们可以认为  $x$ 、 $y$ 、 $\dot{x}$  和  $\dot{y}$  是四个相互独立的变量。

换一种说法，即在仿真的过程中，我们通过对  $\dot{x}$  进行积分来求解  $x$ 。因此，积分方程是用于求解  $x$  的方程。但是在初始化过程中，我们不能用这个等式，所以（对每个在仿真过程中求解的微分方程），我们需要一个额外的方程。

任何情况下，我们要明白是在初始化过程中需要四个不同的方程来得到唯一解。在我们的 Quiescent-Model 模型中，这四个方程如下所示：

$$\begin{aligned}\dot{x} &= 0 \\ \dot{x} &= x(\alpha - \beta y) \\ \dot{y} &= 0 \\ \dot{y} &= y(\delta x - \gamma)\end{aligned}$$

这些方程之间**并不相互矛盾**，理解这一点非常重要。如果你有编程背景，对前面两个方程可能会有疑问。“到底  $\dot{x}$  取值是多少？是零？还是  $x(\alpha - \beta y)$ ？”问题的答案是**都是**。没有理由认为这两个方程不可能同时为真。

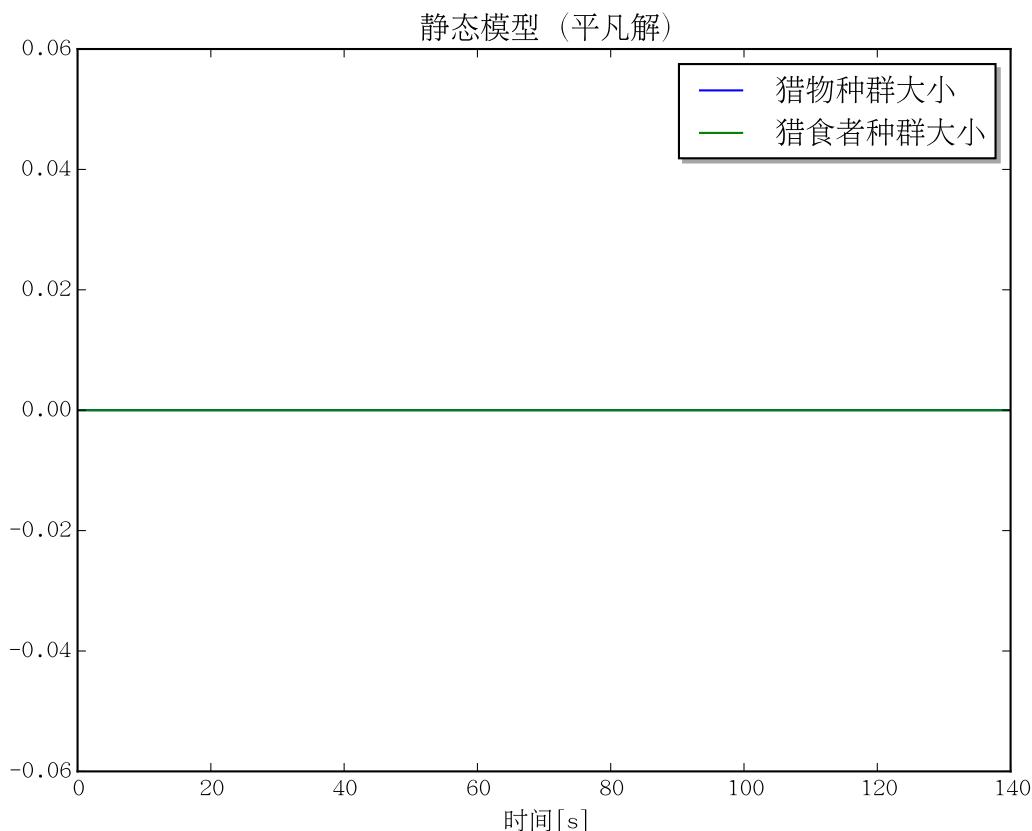
重要的是要记住，这些方程不具备赋值语句的功能。下面系统的方程与上述系统在数学表达上是完全一样的，并且清楚的展现了  $x$  和  $y$  是如何求解的：

$$\begin{aligned}\dot{x} &= 0 \\ \dot{y} &= 0 \\ x(\alpha - \beta y) &= \dot{x} \\ y(\delta x - \gamma) &= \dot{y}\end{aligned}$$

在这种格式下，我们更容易理解如何对  $x$  和  $y$  值进行求解。首先要注意的是，我们不能显式求解变量  $x$  和  $y$  的值。换句话说，如果没有变量  $x$  出现在等式的右边，我们就不能将这些方程的形式变换为  $x = \dots$ 。所以我们不得不接受，这个方程组同时包含变量  $x$  和  $y$ 。

但是，更加复杂的是该仿真系统是非线性的（这也是为什么我们不能用线性代数将其变换为显式方程组）。事实上，如果我们仔细研究这些方程，可以发现存在两种可能解。一个解是平凡解 ( $x = 0; y = 0$ )，另一个则是非零解。

如果我们试着仿真建立的 QuiescentModel 模型，会出现什么样的结果呢？仿真结果如下图所示：



根据第一种解，猎食者和猎物的种群数都变为了零。这种情况下系统没有繁殖、捕食以及饥饿。因为上述影响因素都和物种种群数成比例，而物种种群数都为零。所以系统没有变化。但这并不是一个很有趣的解决方案。

因为该系统是非线性的，因此系统方程有两个解。我们怎样才能使非线性求解器远离这个零根呢？如果你比较关注[经典猎食者猎物系统 \(18\)](#)模型的讨论，那么已经给你了答案的暗示。

还记得，前面提到过 start 属性是被重载了？在讨论[经典猎食者猎物系统 \(18\)](#)模型时，我们曾经指出，如果具有 start 属性的变量被选为迭代变量，start 属性的其中一个作用是提供初始化猜想值。在我们的 QuiescentModel 模型中，恰巧变量  $x$  和  $y$  就是迭代变量。因为该变量必须通过系统的非线性方程组来求解。这也就意味着，我们要对变量  $x$  和  $y$  的 start 属性值进行指定，以尽量“避开”系统的零解（或者说至少接近我们期望的非零解）。例如：

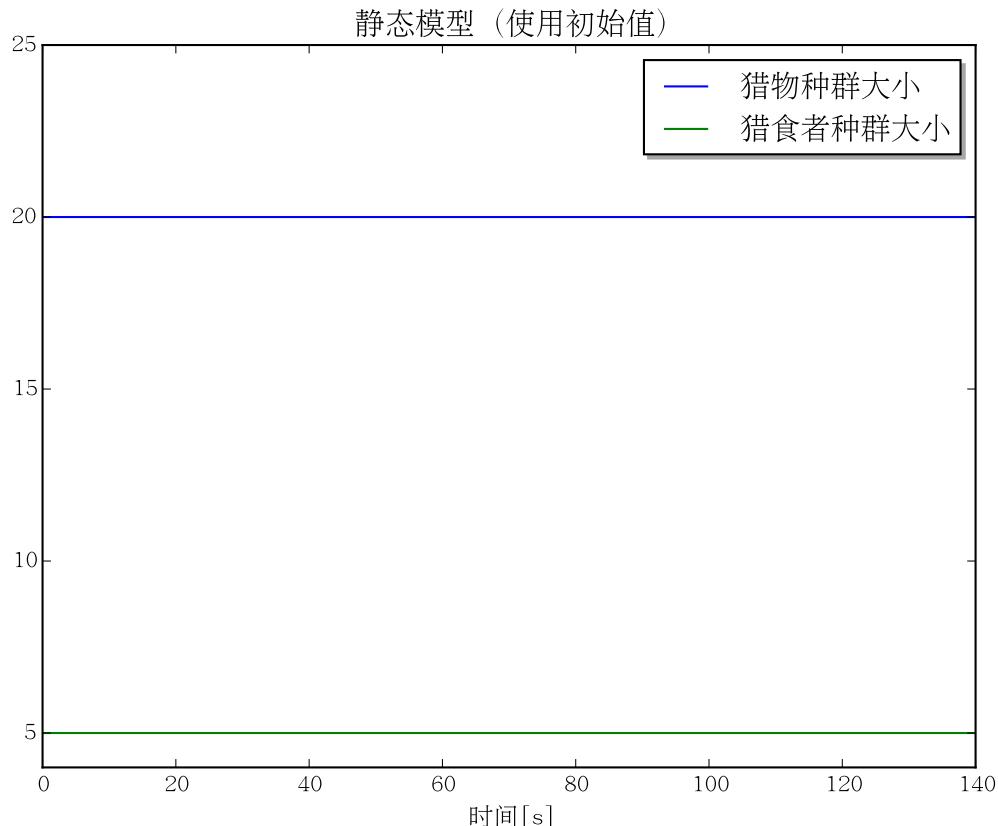
```
model QuiescentModelUsingStart "Find steady state solutions to LotkaVolterra equations"
parameter Real alpha=0.1 "Reproduction rate of prey";
parameter Real beta=0.02 "Mortality rate of predator per prey";
parameter Real gamma=0.4 "Mortality rate of predator";
parameter Real delta=0.02 "Reproduction rate of predator per prey";
Real x(start=10) "Prey population";
Real y(start=10) "Predator population";
initial equation
der(x) = 0;
der(y) = 0;
```

```

equation
  der(x) = x*(alpha-beta*y);
  der(y) = y*(delta*x-gamma);
end QuiescentModelUsingStart;

```

上述模型引导我们设置一组初始条件，使得仿真结果更加符合我们最初的目标（即猎食者和猎物种群大小都得到了非零解）。



值得指出的是（在[内建类型 \( 26 \)](#) 章节中将会讨论），start 属性的默认值是零。这也就是为什么当我们仿真 QuiescentModel 模型时，我们会恰巧准确的得到了系统的零解。因为这是我们初始的猜测，并且恰巧也是系统的精确解，因此系统就无需进行迭代或求解其他根了。

## 避免重复

基于猎食者猎物方程，我们已经讨论过几个不同的模型（ClassicModel, QuiescentModel 以及 QuiescentModelUsingStart 模型）。你有没有注意到这些模型间的共同点？如果你仔细观察，你会发现它们几乎所有的内容都相同。实际上模型之间几乎没有丝毫差别！

在软件工程中，有一种说法是“冗余是一切罪恶的根源”。这里的情况也不例外（其实很大程度上如此。因为 Modelica 代码其实也是属于软件）。目前为止，我们所写的代码维护起来将会特别恼人。这是因为我们发现的任何错误都必须在每个模型中修改。此外，我们所作的任何改进也必须应用到每一个模型上。目前，我们处理的模型数量相对较少，但这种“复制粘贴”的模型开发方法会导致大量只有轻微差异的模型。

那么，我们能做些什么来避免上述情况的发生呢？在面向对象的编程语言中，基本上存在两种机制来减少冗余代码。他们是组合（在后面的[组件 \( 173 \)](#) 章节中进行讨论）和继承。这里我们只对继承进行简单的介绍。

如果我们仔细观察 QuiescentModelUsingStart 模型，会发现和原来的 ClassicModel 模型之间几乎没有差别。事实上，唯一的不同之处如下所示：

```

model ClassicModel "This is the typical equation-oriented model"
  parameter Real alpha=0.1 "Reproduction rate of prey";
  parameter Real beta=0.02 "Mortality rate of predator per prey";
  parameter Real gamma=0.4 "Mortality rate of predator";
  parameter Real delta=0.02 "Reproduction rate of predator per prey";
  parameter Real x0=10 "Start value of prey population";
  parameter Real y0=10 "Start value of predator population";
  Real x(start=x0) "Prey population";
  Real y(start=y0) "Predator population";
equation
  der(x) = x*(alpha-beta*y);
  der(y) = y*(delta*x-gamma);
end ClassicModel;

```

```

model QuiescentModelUsingStart "Find steady state solutions to LotkaVolterra equations"
  parameter Real alpha=0.1 "Reproduction rate of prey";
  parameter Real beta=0.02 "Mortality rate of predator per prey";
  parameter Real gamma=0.4 "Mortality rate of predator";
  parameter Real delta=0.02 "Reproduction rate of predator per prey";
  Real x(start=10) "Prey population";
  Real y(start=10) "Predator population";
initial equation
  der(x) = 0;
  der(y) = 0;
equation
  der(x) = x*(alpha-beta*y);
  der(y) = y*(delta*x-gamma);
end QuiescentModelUsingStart;

```

换句话说，唯一的区别是增加了 initial equation 区域（原始的 ClassicModel 模型已经包含了变量 x 和 y 的非零 start 属性值）。理想情况下，我们可以通过简单的定义一个模型与另外模型间的差异，就可以由此避免任何冗余代码。事实证明，这也正是 extends 关键字的作用。考虑 QuiescentModelUsingStart 模型的以下版本：

```

model QuiescentModelWithInheritance "Steady state model with inheritance"
  extends ClassicModel;
initial equation
  der(x) = 0;
  der(y) = 0;
end QuiescentModelWithInheritance;

```

注意 extends 关键字的使用。从概念上讲，“扩展子句”只是简单的要求编译器将另外的模型（本例是 ClassicModel 模型）插入到所定义的模型中。通过这种方式，我们将从 ClassicModel 模型中复制（或“继承”）其包含的所有内容，而无需重复定义。因此，除了新加入的初始化方程外，QuiescentModelWithInheritance 模型和 ClassicModel 模型其他部分完全一样。

但是，有一种情况，如果我们不想完全精确地继承模型中的内容，该怎么办？例如，如果我们想要改变参数 gamma 和 delta 的值？

当使用 extends 功能时，Modelica 语言允许我们对模型加入相应的“修改语句”。模型对应的修改语句紧跟在所继承模型的名字后面，如下所示：

```

model QuiescentModelWithModifications "Steady state model with modifications"
  extends QuiescentModelWithInheritance(gamma=0.3, delta=0.01);
end QuiescentModelWithModifications;

```

还要注意的是，我们可以从 ClassicModel 中继承。但是，随后我们还必须重新定义初始化方程组以获得静止的初始条件。但是，我们不是通过继承 QuiescentModelWithModifications 模型，而是通过重复使用两个不同模型的内容来避免重复。

### 更多的种群动态

这就是本章所包含的所有实例。如果你想更深入的探索猎食者猎物方程，在下面再探猎食者猎物模型（204）章节中将会重新讲述如何通过示意图并连接在一起的图形化组件来构建复杂的种群动态模型。

## 第 1.2 节 回顾

本章第一部分通过实例介绍了一些语言特性。而这部分会回顾各种特性和概念，并对每一个主题提供更完整的讨论。

### 第 1.2.1 节 模型定义

model 定义是 Modelica 语言中最通用的定义方式。稍后在本书中（或者在本章中）将会介绍其他的方式（例如：record 定义）这些定义方式和 model 定义有着相同的语法结构，但在其包含的内容方面有所限制。

#### 模型定义的语法

正如我们在本章中看到的，模型定义首先是模型关键词。其次是模型的名字（或者是模型的描述）。模型的名字必须以字母开头，后面可以是字母、数字或者下划线。

#### 命名约定

尽管 Modelica 语言并没有对此进行严格要求，但一般约定模型名字以大写字母开头。大多数开发者使用所谓的“驼峰式”规则，即模型内每个单词首字母为大写

模型定义包含变量和方程（稍后讨论）。end 关键词以及紧随的模型名字的重复出现标志着模型的结束。//之后直至该行结束出现的任何文本，以及在分隔符/\* 和 \*/之间的文本都是模型的注释。

总之，模型定义具有如下一般形式：

```
model Some modelName "An optional description"
  // By convention, variables are listed at the start
  equation
    /* And equations are listed at the end */
end Some modelName;
```

#### 继承

正如我们在避免重复（23）这个部分提到的一样，我们可以在当前模型中添加 extends 子句，以此使用其他模型中的代码。另外值得注意的是，模型定义中可以包含多个 extends 子句。

每个 extends 子句必须包含被引用模型的名字。该子句也可以加上对于被引用模型的修改语句。对于继承其他模型的模型定义，你可以看到如下的通用语法

```
model Specialized modelName "An optional description"
  extends Model1; // No modifications
  extends Model2(n=5); // Including modification
  // By convention, variables are listed at the start
  equation
    /* And equations are listed at the end */
end Specialized modelName;
```

按照惯例，`extends` 子句通常列在模型定义最上方，在任何变量之前

在后面的章节中，我们会展示上述语法如何用来定义除了模型之外的其他实体。但是，我们目前的重点是模型。

## 第 1.2.2 节 变量

正如我们在前一节中所提到的，模型定义的特征之一就是包含变量声明。变量声明的基本语法就是在变量的“类型”（将在[内建类型 \(26\)](#) 章节中简短讨论）之后紧随变量的名称，

```
Real x;
```

相同类型的变量可以组合在一起使用下面的语法：

```
Real x, y;
```

变量声明也可以跟随在描述的后面。

```
Real alpha "angular acceleration";
```

### 可变性

#### 参数

模型中的变量声明默认假定为连续变量（连续型变量的解通常是连续的，但是也可能包含间断点）。然而，正如在[引入物理 \(7\)](#) 那节中提到的，能够在变量声明前加入 `parameter`（参数）限定词，以表明该变量是先验已知的。我们可以认为这个参数作为模型的“输入数据”，是不随时间变化的常数。

#### 常数

与 `parameter`（参数）限定符关系密切的是 `constant`（常数）限定词。当被置于变量声明前，`constant`（常数）限定符意味着变量的值是先验已知的常数，且不随时间变化。这两者之间的区别在于一旦模型被编译，`parameter` 限定词的值是可以随着模拟改变而改变的，但是 `constant` 限定词的值是不会改变的。对于模型开发者来说，`constant` 限定词的应用是保证终端用户不能更改该 `constant`。`constant` 限定词经常被用于代表物理常数。比如像数学中的 `pi` 或者重力加速度这样在大多数工程仿真中被假定为常数的量。

### 离散变量

另外一种可以放在变量声明前的限定词是 `discrete`（离散）限定词。目前还没有相关的例子可以演示 `discrete` 限定词的作用。但为了完整起见，它作为最后一个可变性限定词也列举在这里。

### 内建类型

目前为止，很多例子声明变量的时候引用 `Real`（实数）类型。顾名思义，`Real` 用于代表实数变量（实数变量在 Modelica 编译器中通常被编译为浮点数）。然而，`Real` 仅仅是 Modelica 语言的四大内置类型之一。

另外一种内置类型是 `Integer`（整数）类型。这种类型用于代表整数值。`Integer` 变量有许多用途，包括代表数组的大小（这种用途的例子将在[向量与数组 \(81\)](#) 这节中讨论）。

其他内置类型是 `Boolean`（布尔值）（用于代表值的 `true`（真）或 `false`（假））和 `String`（用于代表字符串）。

每一个内置类型都限定了变量值的可能值。显然，`Integer` 变量不能是 `2.5`。`Boolean` 或者 `String`（字符）不能是 `7`。`Real` 变量不能是“Hello”。

## 派生类型

正如前面介绍物理类型 (10) 的例子一样，内置类型可以进行“特殊化”。这个特性主要是用于修改与属性 (27) 相关的值，比如 unit。用于创建派生类型的语法是：

```
type NewTypeName = BaseTypeName(* attributes to be modified *);
```

BaseTypeName (基本类型名称) 一般为内置类型 (比如 Real (实数))。但是它也可以是另外一种派生类型。这意味着多层次的限定也是支持的。例如：

```
type Temperature = Real(unit="K"); // Could be a temperature difference
type AbsoluteTemperature = Temperature(min=0); // Must be positive
```

## 枚举类型

enumeration (枚举) 类型和 Integer 类型非常类似。enumeration 类型通常用于定义一组有限的特定值。事实上，枚举类型在语言上不是必须的。enumeration 类型的值总是可以用整数替代。但是 enumeration 类型却比 Integer 类型更安全和合理。

在 Modelica 中有两种内置的枚举类型。第一种是 AssertionLevel (断言级别)。其定义如下：

```
type AssertionLevel = enumeration(warning, error);
```

这些值的用处将在接下来的 assert (242) 这节中讨论。

另外一种内置枚举类型是 StateSelect (状态选择)，其定义如下：

```
type StateSelect = enumeration(never, avoid, default, prefer, always);
```

## 属性

在本章中到目前为止，已经提到了属性 (例如 unit (单位))。但是还没有详细讨论过，例如，一个既定的变量应该有何种属性？这主要依赖于变量的类型 (以及它所基于的基于内置类型或派生类型)。下面的表格列举了所有可能的属性以及与其类型 (例如属性可以赋什么类型的值) 以及其可以关联的类型，最后表格则简短描述属性的作用。

### Real 的属性

quantity 用以来描述这个变量所代表含义的文本。

Default: ""

**类型:** String

start start (初值) 这个属性有很多用途。start 属性的主要目的是 (正如在初始化 (31) 那节里广泛讨论过的) 为状态变量提供“备用”的初始状态。(参见 fixed 属性以获取进一步的细节)

start 属性的也可以用于变量，是迭代变量时的初始假想值。

最后，如果一个 parameter 没有明确指定的值，那么 start 属性的值可以作为 parameter 的默认值。

**默认值:** 0.0

**类型:** Real

fixed fixed (固定) 属性改变了 start 属性用于初始值的方式。正常来讲，start 属性被认为是“备用”的初始状态。这仅仅会在 initial equation 中没有指定足够的初始状态时使用。然而，如果 fixed 属性被设为 true，那么 start 属性就被明确地用于 initial equation。(即属性不再作为备用，而是作为严格的初始条件)。

此外，一种 fixed 属性的晦涩用法是用在“算出的参数”。在某些罕见的情况下，一个 parameter 不能显式初始化。因此，参数的需要得到一个通用方程作为其 initial equation。如果 parameter 一旦通过这种方式初始化，那么参数变量的 fixed 属性必须设置为 false。

**默认:** false (除了 parameter 变量，这里的默认值为 True)

**类型:** Boolean

**min** min (最小) 属性用来指定变量允许的最小值。这个属性可以通过各种方式被编辑器或者编译器使用，以通知使用者或者开发者关于潜在的无效输入或者解。

**默认值:**  $-\infty$

**类型:** Real

**max** max (最大) 属性用于指定变量允许的最大值。这个属性可以通过各种方式被编辑器或者编译器使用，去在出现可能是无效输入或者解时提醒使用者或者开发者。

**默认值:**  $\infty$

**类型:** Real

**unit** 正如在本章中广泛谈到的那样，变量是可以拥有与其相关的物理单位。这些单位的表示有其规则。而最终结果是 unit (单位) 属性可以用于检查模型，确保方程的物理一致性。“1”代表值没有物理单位。另一方面“” (默认没有给定值) 表明物理单位不确定。“1”和“”之间的区别在于前者明确指出量是无量纲 (无单位)。而后者指出量可能是有物理单位的，只是没有明确指出。

**默认值:** “” (即没有指定物理单位)

**类型:** String

**displayUnit** unit 属性描述了与变量相关值的物理单位。displayUnit (显示单位) 属性表达了当展示变量的值时偏向使用何种单位。例如，压力的国际单位是帕斯卡。标准大气压是 101,325 帕斯卡。当输入、显示或者绘制压力的时候，单位用巴更方便。

displayUnit 属性不影响变量的值或者模拟模型的方程它仅仅影响把它们转换成更方便的单位显示的那些值的显示。

**Default:** “”

**类型:** String

**nominal** nominal (额定) 属性用于指定变量的额定值。额定值一般用于在数字计算中执行各种类型的扩展，以避免四舍五入或者截断误差。

**默认值:** 0.0

**类型:** Real

**stateSelect** stateSelect (状态选择) 属性用于为 Modeliac 编译器提供线索，去确定某个既定的变量是否应该被选为状态量 (在需要选择的情况下)。在上节枚举类型 (27) 中我们讨论过，这个属性可能值是：never (从不)、avoid (避免)、default (默认)、prefer (偏向)、always (总是)

**默认值:** default

**类型:** StateSelect (枚举类型，请参阅枚举类型 (27))

## Integer 的属性

quantity 用以来描述这个变量所代表含义的文本。

**Default:** “”

**类型:** String

**start** 值得注意的是，Integer 变量可以用来表示状态变量或者迭代变量。在这种情况下，编译器就可以应用 start 属性。其用法和 Real 变量的情况一样 (见前面讨论的 Real 的属性 (27))。

对于 parameter，start 属性 (通常) 会作为 parameter 的默认值。

**默认值:** 0.0

**类型:** Integer

fixed 参见前节对Real 的属性 ( 27) 的讨论。

**默认:** false (除了 parameter 变量, 这里的默认值为 True)

**类型:** Boolean

min min (最小) 属性用来指定变量允许的最小值。这个属性可以通过各种方式被编辑器或者编译器使用, 以通知使用者或者开发者关于潜在的无效输入或者解。

**默认值:**  $-\infty$

**类型:** Integer

max max (最大) 属性用于指定变量允许的最大值。这个属性可以通过各种方式被编辑器或者编译器使用, 去在出现可能是无效输入或者解时提醒使用者或者开发者。

**默认值:**  $\infty$

**类型:** Integer

### Boolean 的属性

quantity 用以来描述这个变量所代表含义的文本。

Default: ""

**类型:** String

start 值得注意的是, Boolean 变量可以用来表示状态变量或者迭代变量。在这种情况下, start 属性就可以被编译器使用, 就如它在 Real 变量中的应用一样 (见前面讨论的Real 的属性 ( 27))。

对于 parameter, start 属性 (通常) 会作为 parameter 的默认值。

**默认值:** 0.0

**类型:** Boolean

fixed 参见前节对Real 的属性 ( 27) 的讨论。

**默认:** false (除了 parameter 变量, 这里的默认值为 True)

**类型:** Boolean

### String 的属性

quantity 用以来描述这个变量所代表含义的文本。

Default: ""

**类型:** String

start 从技术上讲, 一串 String (字符串) 可以作为一个状态变量 (甚至迭代变量)。但事实上从没有这种情况。因此对于 String 变量, start 属性的唯一实际应用是定义 (恰好是 String 类型的) parameter 的值。前提是参数没有给定明确的值。

Default: ""

**类型:** String

值得注意的是派生类型 ( 27) 保留了其来源的内置类型的属性。例如, 实数变量中的 min 属性拥有 Real 类型。但是必须明确指出, 属性自己不能拥有属性。换句话说, start 属性不能拥有 start 属性。

## 修改语句

目前为止，本书只介绍了有两种修改语句类型。第一种是用于修改属性值，例如：

```
Real x(start=10);
```

在这个例子中，创建了一个实数类型的变量 x。但是我们不打算直接使用这个变量。我们对 x 加入修改语句。在这里，我们修改 x 的 start 属性。这仅仅只是进入 x 的第一层做修改。在接下来的例子中，我们会在任意深度对 x 做修改。

另一种修改是在[避免重复 \(23\)](#) 那节中。此时修改语句是和扩展条款连接在一起的。例如：

```
extends QuiescentModelWithInheritance(gamma=0.3, delta=0.01);
```

在这里，修改语句作用在继承自 QuiescentModelWithInheritance 模型的部件上。和修改属性时一样，受修改的（模型）元件后带有括号。而所希望的修改则正正在括号中。

值得注意的是，修改语句可以任意嵌套。例如，我们可以修改来自于模型 QuiescentModelWithInheritance 变量 x 中的 start 属性，在 Modelica 中，这种修改语句如下：

```
extends QuiescentModelWithInheritance(x(start=5));
```

在这里，首先“进入”模型 QuiescentModelWithInheritance 修改（从 x 中“继承”的）内容。然后，再“进入”x 修改 start 属性的值。

Modelica 的核心主题之一是支持重用而避免“复制和粘贴”代码。而修改语句就是 Modelica 中支持重用的基本特性。在后面的章节中将会学习其他支持重用的特性。

## 第 1.2.3 节 方程

尽管方程可能是 Modelica 语言中唯一最重要的数学方面的体现，但是它仍然是最易于解释的。

### 基本方程

关于方程真的没有复杂的语义需要解释。所有的方程都是由左边一个表达式和右边一个表达式组成，中间被等号隔开。例如：

```
<left-hand expression> = <right-hand expression>;
```

通过本章中的实例，读者可以反复接触每个例子中的这种格式。上面展示的唯一真实存在的语法偏差是关于方程的描述应该如下，例如：

```
V = i*R "Ohm's law";
m*der(v) = F "Newton's law";
```

正如我们先前指出的那样，在 Modelica 语言中，方程的左边和右边是表达式，而不是赋值。换句话说（对比大多数的程序语言），方程的左边不一定要包含变量（正如上面牛顿第一定律中看到的那样）。

### 初始方程

正如在本章中看到的许多例子一样。在一个模型中指定方程用于求解初始状态是可能的。关于初始化的整个主题将会在下章中详细讨论，标题就是[初始化 \(31\)](#) 目前在本章中要说的是如果方程仅仅用于求解初始化，equation 区域必须冠以 initial (初始) 关键词。

```
initial equation
  x = 0; // Only used to solve for initial conditions
```

## 带条件的方程

下一章将讨论如何应用 if (如果) 命令表示有条件的行为。值得提前指出的是方程可以是有条件的。条件方程有两种形式。第一种是平衡形式。例如:

```
if a>b then
  x = 5*time;
else
  x = 3*time;
end if;
```

在平衡的情况下，方程的数量总是一致的（上述代码中都为 1）。但是要使用哪条方程这点可以改变。这个很重要，因为要在 Modelica 中模拟一个模型，变量的数量必须等于方程的数量。而且，在模拟的过程中方程的数量必须是固定的。

另外一种条件方程的类型是方程的数量是不平衡的。这意味着方程的数量在 if 和 else 的两侧可能是不相等的（不像此前的平衡情况那样）。

但是请记得，方程的数量在模拟过程中不能改变。那么是如何做到 if 和 else 两侧的方程数量是不同的呢？只有一种情况模拟过程中条件表达式的值不能改变。为了确保条件表达式永不改变，条件表达式中所有的变量必须拥有所谓的参数级别的可变性。

还记得讨论可变性（26）的时候，带有 parameter（参数）限定词的变量在模拟过程中不能改变的事实吗？如果带有 parameter（参数）限定词的变量在模拟过程中不能改变，且表达式中的所有变量都拥有这个参数级别的可变性，那么整个表达式一定也拥有参数级别的可变性（亦即表达式的值在模拟过程中不能改变）。

关于这个问题，你一定会质疑，这种不平衡的情况有用吗？再次提前指出，其中一种应用是在带条件的初始方程，例如：

```
..
parameter Boolean steady_state;
initial equation
  if steady_state then
    der(x) = 0;
    der(y) = 0;
..
..
```

换句话说，如果布尔参数 steady\_state（稳定状态）为真，那么初始方程是有效的。但是如果参数为假，它们就无效。这里的条件表达式具有参数级别的可变性是因为，表达式仅仅包含一个变量，而这个变量是个参数。

这些就是本节所要讨论的，离散和有条件的行为将在下一章（39）中讨论

### 第 1.2.4 节 初始化

#### 概述

正如我们先前在稳定状态初始化（20）这节中提到过的，初始化是由一个模型中的两个方程和模型中初始条件给定的状态变量来描述的。在 Modelica 中，初始条件是通过结合普通方程（在 equation（方程）区域描述）和初始化方程（在 initial equation（初始方程）区域中描述）来计算的。

对于初学者来说第一个混淆的来源是理解需要多少初始条件。这个问题的答案是简单的。为了有一个适定的初始化问题（就是没有过多或者过少的初始方程），需要如之前在系统中提到的在 initial equation（初始方程）中具有相同数量的方程。注意，应当避免有太少的初始方程，因为工具可以增加我们给定的初始方程的数量直到问题适定，但是如果初始方程太多那问题就不能求解。

当然，提到初始方程的数量等于状态的数量只回答了一个问题，却很快产生了另一个问题。例如，要如何决定需要多少状态？对于本章中的模型，这个答案很简单。目前为止每个例子中的出现在 der(...) 操作中的状态都是变量。换句话说，例子中不同的变量表示一个状态。

## 常微分方程

需要注意的是，我们进行微分的每个变量并不总是状态。这章中，目前看到的所有模型都是常微分方程 (ODEs)。当处理 ODEs 时，每个不同的变量都是状态。也就是说，每个初始变量需要一个初始方程。但是在接下来的章节中我们将会遇到所谓的微分代数方程 (DAEs)。在这些方程中，仅仅只有一些微分变量可以认为是状态。

事实上理解初始化不需要详细讨论 DAEs。实际上，所有的 Modelica 工具都要执行所谓的指标约简 (index reduction)。但是指标约简算法本身是相当的复杂（因为现在不讨论），而效果却相当简单。指标约简将 DAEs 转化为 ODEs。换句话说，Modelica 编译器将包含 Modelica 代码的 DAE 问题转化为相对简单的 ODE 形式。

因此忽略 DAEs 和指标约简的讨论。我们假定问题已经转化为 ODE，然后再讨论初始化。在这种情况下，只需要理解模型中的每个状态都需要初始化。模型将具有如下的 ODE 形式：

$$\begin{aligned}\dot{\vec{x}}(t) &= \vec{f}(\vec{x}(t), \vec{u}(t), t) \\ \vec{y}(t) &= \vec{g}(\vec{x}(t), \vec{u}(t), t)\end{aligned}$$

$t$  是当前模拟的时间。 $\vec{x}(t)$  是系统时间为  $t$  时的状态值。 $\vec{u}(t)$  是系统时间为  $t$  时的外部输入。

注意，变量上部的箭头仅仅指的是这个变量是个矢量而不是标量。这个问题中出现的唯一的变量是  $\vec{x}$ 。 $\vec{x}$  代表了系统的状态。另外需要注意的是两个函数无论是  $\vec{f}$  还是  $\vec{g}$  都不依赖于变量  $\vec{y}$ 。

思考后，你会发现  $t$  和  $\vec{u}(t)$  都是系统外部的量。所以，我们没有必要计算或者控制这些量。之所以认为  $\vec{x}$  是系统状态，原因在于它是计算  $\dot{\vec{x}}(t)$  和  $\vec{y}(t)$  时所需要（从系统中获取）的唯一信息。

回到最初的初始化的主题，在一个正常的时间步内，将  $\dot{\vec{x}}(t)$  积分来求解  $\vec{x}(t)$ 。换句话说：

$$\vec{x}(T) = \int_{t_i}^T \dot{\vec{x}}(t) dt + \vec{x}(t_i)$$

这只要有前一个时间步就可以进行下去。当没有前面的时间步，那么插入方程的  $\vec{x}$  的值将成为模拟的第一个  $\vec{x}$  值。换句话说，这也就是初始条件。

可以想象，通过指定如下方程来指定初始条件：

$$\vec{x}(t_0) = \vec{x}_0$$

$t_0$  是模拟的起始时间。 $\vec{x}_0$  是具体的初始值。为状态提供具体的值是指定初始条件最常见的情况。因此需要处理这种情况。但是这个方法在 [稳定状态初始化 \(20\)](#) 中是不起作用的。那种情况下我们不需要为状态提供具体的初始值。相反，我们要为  $\dot{\vec{x}}(t_0)$  提供初始值。那如何处理这两种情况呢？

## 初始方程

答案是假定在模拟开始时点我们需要求解以下问题：

$$\begin{aligned}\dot{\vec{x}}(t_0) &= \vec{f}(\vec{x}(t_0), \vec{u}(t_0), t_0) \\ \vec{y}(t_0) &= \vec{g}(\vec{x}(t_0), \vec{u}(t_0), t_0) \\ \vec{0} &= \vec{h}(\vec{x}(t_0), \dot{\vec{x}}(t_0), \vec{u}(t_0), t_0)\end{aligned}$$

注意到引进了新方程  $\vec{h}$ 。这个新方程代表了 initial equation 中的任何方程。 $\vec{h}$  将  $\vec{x}$  和  $\dot{\vec{x}}$  作为参数，这允许表达很多可能的初始值。为了给状态定义一个明确的值，定义  $\vec{h}$  如下：

$$\vec{h}(\vec{x}(t_0), \dot{\vec{x}}(t_0), \vec{u}(t_0), t_0) = \vec{x}(t_0) - \vec{x}_0$$

如果希望从稳态解开始仿真，则应该定义  $\vec{h}$  如下：

$$\vec{h}(\vec{x}(t_0), \dot{\vec{x}}(t_0), \vec{u}(t_0), t_0) = \dot{\vec{x}}(t_0)$$

当然，对不同的状态，我们也可以对其各自使用上述不同的形式，乃至很多其他的形式，去描述初始条件。因此在写初始方程的时候，需要明白的是方程需要为上面所示的一般通用形式。而且你不能使用的方程数不能超过系统中的状态。

## 结论

正如在本章中所演示的一样，Modelica 中 initial equation 结构允许应用多种方式去表达系统初始化。最终，所有形式都是可以计算系统的初始值。但是在描述这些值的计算时，我们有很大的自由度。

这是 Modelica 所擅长的领域。Modelica 内初始化拥有第一等的地位。而这种灵活性也会为许多真实应用场景提供好处。

### 第 1.2.5 节 Record 类型定义

本书的前些章节，我们介绍了 model 类型的定义概念。虽然目前为止我们还没有看到任何其他的类型，Modelica 语言其实还包括 record 类型。像 model 类型一样，record 类型可以有自己的变量，但是不允许包含方程。record 类型主要用于数据的分组。但是，在下面的标注 (34) 章节中你会发现它们在描述与标注相关的数据时也非常有用。

#### 语法

record 类型的定义看起来基本上类似于 model 类型，只是不包含任何的方程：

```
record RecordName "Description of the record"
  // Declarations for record variables
end RecordName;
```

像 model 定义那样，类型定义的开始和结束都会带有所定义的 record 类型名称。对所定义的 record 类型的说明以字符串的形式紧跟类型名的后面。与 record 类型相关联的所有变量都在 record 类型定义内声明。

以下是 record 类型定义方面的实例：

```
record Vector "A vector in 3D space"
  Real x;
  Real y;
  Real z;
end Vector;

record Complex "Representation of a complex number"
  Real re "Real component";
  Real im "Imaginary component";
end Complex;
```

#### Record 类型的构造函数

现在，我们已经知道了如何定义一个 record 类型。那应该如何创建 record 类型呢？如果我们想声明一个变量，恰巧这个变量属于 record 类型，变量声明的本身就会创建一个 record 类型实例，而且我们还可以通过修改语句指定 record 类型内的变量值，例如：

```
parameter Vector v(x=1.0, y=2.0, z=0.0);
```

但在某些情况下，我们可能希望创建一个 record 类型而不是一个变量（例如，在表达式中使用，作为参数传递给函数或者在修改功能中使用）。对每个 record 类型定义，都会自动生成与 record 类型名称完全相同的函数名。这个函数称为“记录构造函数”。记录构造函数输入与 record 类型内部定义相匹配的变量，并返回一个 record 类型实例。所以，在上述 Vector 定义实例中，我们也可以通过记录构造函数初始化 parameter 变量，如下所示：

```
parameter Vector v = Vector(x=1.0, y=2.0, z=0.0);
```

在这种情况下，变量 v 的值通过表达式 Vector(x=1.0, y=2.0, z=0.0) 调用记录构造函数进行赋值。

## 第 1.2.6 节 标注

回顾一下，在`试验条件`(6)章节中，我们讨论了通过annotation功能定义模型仿真开始和结束的时间。通过annotation功能包含上述信息是一种方法，并且不包含与模型特性相关的信息。在试验条件下，我们可以定义特定模型如何进行仿真的信息。但是，在Modelica语言中，标注广泛应用于提供各种与模型相关的附加信息。例如，在`后续的章节中`(167)，我们可以看到标注可以用来描述图形外观组件和连接器。现在，更重要的是要理解标注只是额外的数据，除了描述模型的物理特性，它可以“附加”在Modelica不同的元素上。

在本节中，我们将首先讨论annotation在Modelica模型中可以出现的地方。接下来，我们将解释如何使用`Record`类型定义(33)来描述标注所包含的内容。最后，我们将介绍一些包含在Modelica规范内的“标准”标注。

### 标注的位置

标注可以在Modelica模型的不同位置上出现。我们将讨论每个可能出现的位置，并讲述每种情况下的语法。

#### 标注的声明

标注的声明紧跟在一个声明的后面，并且在标示符;的前面。下面是一个包含标注的简单声明：

```
parameter Real length "Rod length" annotation(...);
```

从上述程序段中可以看到，标注紧跟在描述性字符串的后面，并在标示符;的前面。另外，括号内的...只是用来放置标注数据(35)的占位符。具体内容将在本书后面的章节中讨论。

#### 语句标注和方程标注

另外，也可以将标注和方程相关联，例如：

```
T = T0 "Specify initial value for T" annotation(...);
```

标注总是紧跟在声明和方程定义的结束处，并且在标示符;的前面。

#### 标注的继承

在章节`修改语句`(30)和`避免重复`(23)中，我们已经简单的对关键词extends进行了讨论。如下所示，我们也可以将annotation和extends相关联：

```
extends QuiescentModelWithInheritance(gamma=0.3, delta=0.01) annotation(...);
```

正如我们在前面的每个例子中观察到的那样，annotation的声明总是在标示符;的前面结束。

#### 模型的标注

与模型标注相关的标注数据直接在模型自身定义的时候声明。这正正是在前面章节里对`试验条件`(6)讨论的时候，我们所看到的标注，即：

```
model FirstOrderExperiment "Defining experimental conditions"
  Real x "State variable";
  initial equation
    x = 2 "Used before simulation to compute initial values";
  equation
    der(x) = 1-x "Drives value of x toward 1.0";
```

```
annotation(experiment(StartTime=0,StopTime=8));
end FirstOrderExperiment;
```

请注意，与我们前面描述的标注位置不同，该标注并没有附加到任何部分上。这表明它是模型本身的标注。

## 标注数据

### 一般的语法

标注的使用语法与前面章节修改语句（30）讨论的语法相同。这就意味着，标注中即可以包含对变量的赋值，如：

```
annotation(Evaluate=true);
```

也可以包含对一些内部变量的修改，如

```
annotation(experiment(StartTime=0,StopTime=8));
```

## 用户标注

标注就是为了方便模型开发者将任意数据添加到模型中而设计的。例如，如果用户想要将部分数据与给定的模型定义相关联，他们可以为模型添加如下的标注：

```
annotation(PartNumber="FF78-E4B879");
```

标注数据的一般原则是，读取模型的工具，在写入模型的时候，**必须保留标注信息**。该工具不必（一般来说，无需）理解读入的标注信息，但是，必须将这些数据保存。

## 多个标注

假设，用户想要同时指定编号和试验标注，他们可以采用下述方式定义标注：

```
annotation(PartNumber="FF78-E4B879",
           experiment(StartTime=0,StopTime=8));
```

注意，这两条标注信息可以并存。可以将标注可视化为如下树状结构来理解标注的定义：

- PartNumber="FF78-E4B879"
- experiment
  - StartTime=0
  - StopTime=8

## 命名空间

这就引入另一个关于标注的原则。**只要标注的名称不同**，我们就可以一次加入多个。由于这个原因，名称的选取很重要。我们应避免与其他名称的潜在冲突。例如，在加入零件号时，最好要将其放一个特别的变量内。这个变量最好应该和你的公司或者应用情况相关，而且足够特别。例如：

```
annotation(XogenyIndustries(PartNumber="FF78-E4B879"),
            experiment(StartTime=0,StopTime=8));
```

在本例里，变量 XogenyIndustries 可以为某个特定组织或目的创建出一个“命名空间”。如果其他组织想要一个不同的部件号与同一个模型相关联，他们就能通过在自己的标注中的层次中做到这一点，如：

```
annotation(XogenyIndustries(PartNumber="FF78-E4B879"),
    AcmeEquipment(PartNumber="A23335-992"),
    experiment(StartTime=0,StopTime=8));
```

有时候，Modelica 的工具供应商会加入他们自己的特殊标注（如在 Modelica 的标准库中）。按照惯例，工具供应商使用以两个下划线为前缀的名字，如：

```
annotation(XogenyIndustries(PartNumber="FF78-E4B879"),
    __ModelicateTechnologies(enableCoolFeature10=true),
    AcmeEquipment(PartNumber="A23335-992"),
    experiment(StartTime=0,StopTime=8));
```

### 解释

请记住，标注数据是任意的。这使得模型可与任意数据相关联的。数据的意义在一般情况下并没有定义在 Modelica 标准里。正如我们将马上看到的一样，存在几个“标准”标注（我们会在整本书中描述）。这些标注都记录在标准里。但是，如果用户添加了标准外的标注，工具会假设用户有一些方法（使用某些 Modelica 的工具、编译器或其它可以读取 Modelica 的技术）去提取并解释其标注数据。

也就是说，虽然你可以对模型添加（非标准）标注数据，但工具只需要保存该标注，而毋须进行解析。

### 记录

记录 Modelica 标注时常常假设标注和某个 Record 类型定义（33）相关联。我们将在下个主题看到这个技巧的几个例子。强烈建议大家使用这种方法来记录预期的标注数据。其实，这种技术如此的流行而且实用，以致于有人建议，让其实际上成为语言本身的一部分。

### 简单标注

这部分介绍了 Modelica 的几个“标准标注”。这如前面所讨论，标注通常可以包含任意信息。这些标注不会被工具改变，而且可能会在某些时候得到解析。标准标注的语法和意义在 Modelica 标准都有描述。这样，这些标注就能 Modelica 工具得到一致以及普遍的解析。

我们将（尽可能地）按照 record 定义的方式去描述标准标注。这些 record 定义不正式存在。它们只是一种表达包含在标注中数据的简明方式。

Documentation

#### 类型：模型标注

Modelica 的 Documentation 标注可用纯文本或 HTML 表示模型相关文档。文档是由两部分组成。第一部分是有关模型的信息。第二部分则是关于模型修订历史的信息。The structure of the Documentation annotation is described by the following record definition: Documentation 标注的结构可以用下列记录定义表示：

```
record Documentation
    String info "Documentation in text or HTML format";
    String revision "Revision information in text or HTML format";
end Documentation;
```

当在标注里面嵌入 HTML 时，HTML 代码必须由 <html> 标记包围，如：

```
model MyWidget
// ... declarations
annotation(
    Documentation(
        info="<html><h1 class=\"heading\">Introduction</h1><p>...</p></html>"));

```

```
// .. equations
end MyWidget;
```

这里的 MyWidget 模型包含 HTML 文档。文档在 `<html>` 标记内，所有用于定义属性的引号都用 \” 进行转义，以避免意外地终结 info 字符串。

experiment

**类型：**模型标注

experiment 标注用于指定模型仿真方式的信息。标注数据可以用 record 形式表示如下：

```
record experiment
  Real StartTime "Time at which the simulation should start";
  Real StopTime "Time at which the simulation should stop";
  Real Interval(min=0) "Time interval between results";
  Real Tolerance(min=0) "Solver tolerance to use";
end experiment;
```

Evaluate

**类型：**声明标注（适用于参数）

Evaluate 标注指示 Modelica 编译器，指定 parameter 值在编译的时候可以转化为 constant。换句话说，标注表示该用户不需要在每次仿真都改变 parameter 的值。

这种标注背后的动机在于，它允许 Modelica 编译器在编译期间对 parameter 进行一些之前不可能的假设。这些假设可能限制方程组，让其变得更容易解。因为参数不再会像一般情况那样有一个取值范围。

Evaluate 标注只是个 Boolean 变量（True 值表示 parameter 值可以转化为 constant）。这在标注可以使用如下：

```
parameter Real x annotation(Evaluate=true);
```

HideResult

**类型：**声明标注

HideResult 标注用于表明，分析者并不关心该变量的解。通过把 HideResult 值设置为 True，模型开发者告诉 Modelica 编译器并不需要在任何产生的结果里储存被标注的值。这可以节省模拟时间和磁盘空间。因为这样避免了写入根本不会被读取的数据。

HideResult 标注使用如下：

```
Real z "Uninteresting variable" annotation(HideResult=true);
```



### 离散行为

到目前为止，我们看到的例子均为纯粹连续的。这意味着，系统不曾受到突然的扰动。本章中，我们将重点放在如何描述所谓“离散行为”上。许多不同的工程用例均描述了类似的行为。而我们将在本章中通过不同的例子对其进行探索。

一般情况下，当谈论离散行为时，我们通常会提到“事件”(event)。事件是任何在系统内触发了某种不连续性的东西。微分方程通常会有连续的解。但产生事件时，方程则会带来各种不连续性。

发生在某个特定时间的事件是最简单的一类事件。这种事件自然被称作“时间事件”(time event)。由于这种事件和时间紧密相关，我们甚至可以在时间发生前就知道发生时间。以给定频率被激活的数字时钟所触发的变化就是时间事件的一个例子。

我们会遇到的另一类时间就是所谓的“状态事件”(state event)。这类事件都通常更难以处理，原因是无法先验地知道发生此类事件的时间。和时间事件不同，我们必须等到某信号越过特定阈值。一般而言，我们不知道什么时候信号会越界。此外，确定事件发生的精确时刻是比较昂贵的。

在本章中，我们将看到一系列有关这两类事件以及 Modelica 中处理事件的语言特性的例子。这些语言特性可用于表述事件的产生条件以及对其的响应。

## 第 2.1 节 示例

### 第 2.1.1 节 再探冷却

#### 改变环境条件

我们将从一个简单的例子开始介绍了时间事件。我们将重新审视在前面物理类型(10)中介绍的热学模型。不过，这次我们将对该系统添加一个扰动。具体地说，我们将仿真开始的半秒后让环境温度的突然下降。修正后的模型可以如下：

```
model NewtonCoolingDynamic
  "Cooling example with fluctuating ambient conditions"
  // Types
  type Temperature=Real(unit="K", min=0);
  type ConvectionCoefficient=Real(unit="W/(m2.K)", min=0);
  type Area=Real(unit="m2", min=0);
  type Mass=Real(unit="kg", min=0);
  type SpecificHeat=Real(unit="J/(K.kg)", min=0);

  // Parameters
  parameter Temperature T0=363.15 "Initial temperature";
  parameter ConvectionCoefficient h=0.7 "Convective cooling coefficient";
  parameter Area A=1.0 "Surface area";
  parameter Mass m=0.1 "Mass of thermal capacitance";
  parameter SpecificHeat c_p=1.2 "Specific heat";

  // Variables
```

```

Temperature T_inf "Ambient temperature";
Temperature T "Temperature";
initial equation
  T = T0 "Specify initial value for T";
equation
  if time<=0.5 then
    T_inf = 298.15 "Constant temperature when time<=0.5";
  else
    T_inf = 298.15-20*(time-0.5) "Otherwise, increasing";
  end if;
  m*c_p*der(T) = h*A*(T_inf-T) "Newton's law of cooling";
end NewtonCoolingDynamic;

```

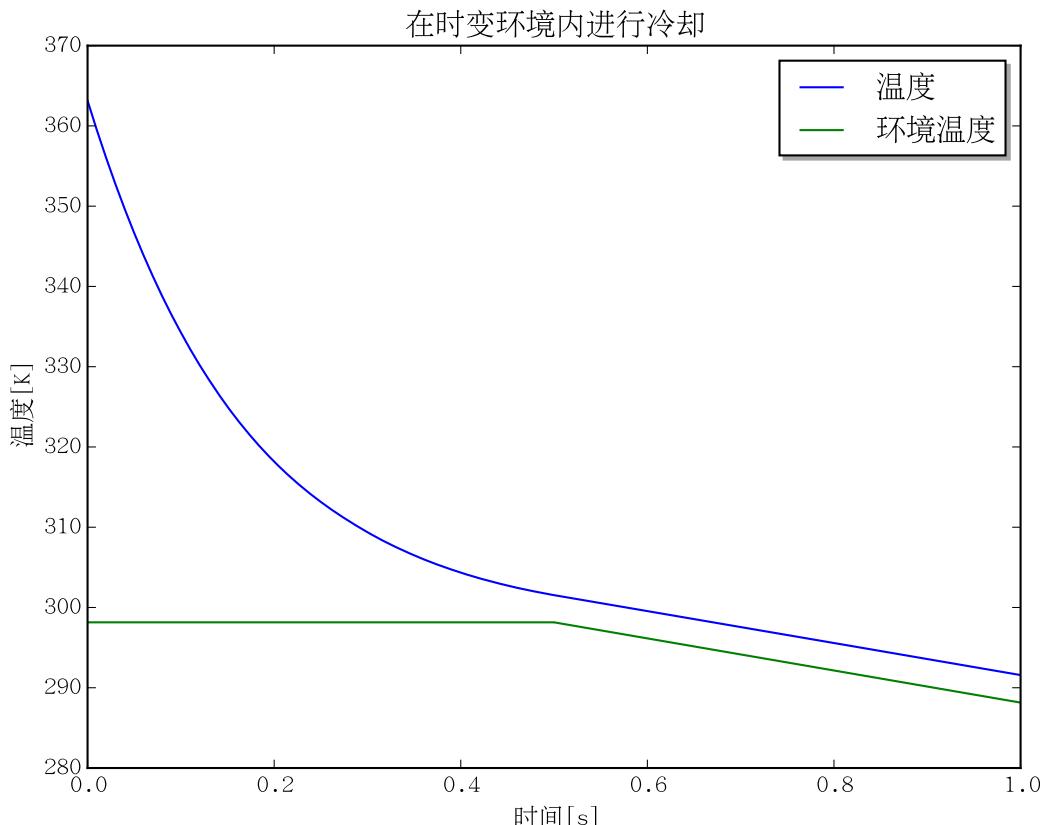
高亮行为 if 语句。这条特定的 if 语句为 `T_inf` 的计算提供两条不同的等式。

### 时间变量

你会注意到这个模型中没有声明变量 `time`。之所以如此是因为 `time` 是一个内置在所有 Modelica 模型内的变量。

两个方程中实际使用哪个依赖于条件表达式 `time<=0.5`。正因为这个表达式仅仅取决于 `time`，而不是在模型中的任何其他变量，我们可以把这两个公式之间的转换称为“时间事件”。关键的一点是，对这些方程进行积分时，我们可以告诉对方程组进行积分的求解器精确地停在第 0.5 秒，然后继续使用不同的公式进行求解。在下一节我们介绍的经典的弹跳球 (43) 例子里，我们会看到对于其他状态的事件而言，这是不可能的。

但是现在，让我们继续考虑我们的冷却例子。如果我们对这个模型进行一秒钟的仿真，我们将得到以下的温度轨迹：



正如你在这些结果中看到的，环境温度确实在半秒后开始减少。在研究的温度本身的动态响应时，我们看到两个不同的阶段。第一阶段是趋向平衡态的初始瞬态响应（从而和环境温度相匹配）。第二阶段是随

在环境温度的减小的进行的追踪。

### 初始瞬态

值得注意的是，这是建模中一个非常常见的问题。你常常要模拟的是系统对一些扰动的响应（如在此例中环境温度的下降）。但是，如果系统开始时不处于某种平衡态下，系统响应将包括某种形式的初始瞬态（图示）。为了清楚地区分这两种响应，我们要避免它们之间有任何重叠。**要做到这一点最简单的方法就是从平衡态开始仿真**（参见我们之前在稳定状态初始化（20）的讨论）。这完全避免了初始瞬态，允许我们完全专注于我们所感兴趣的扰动。

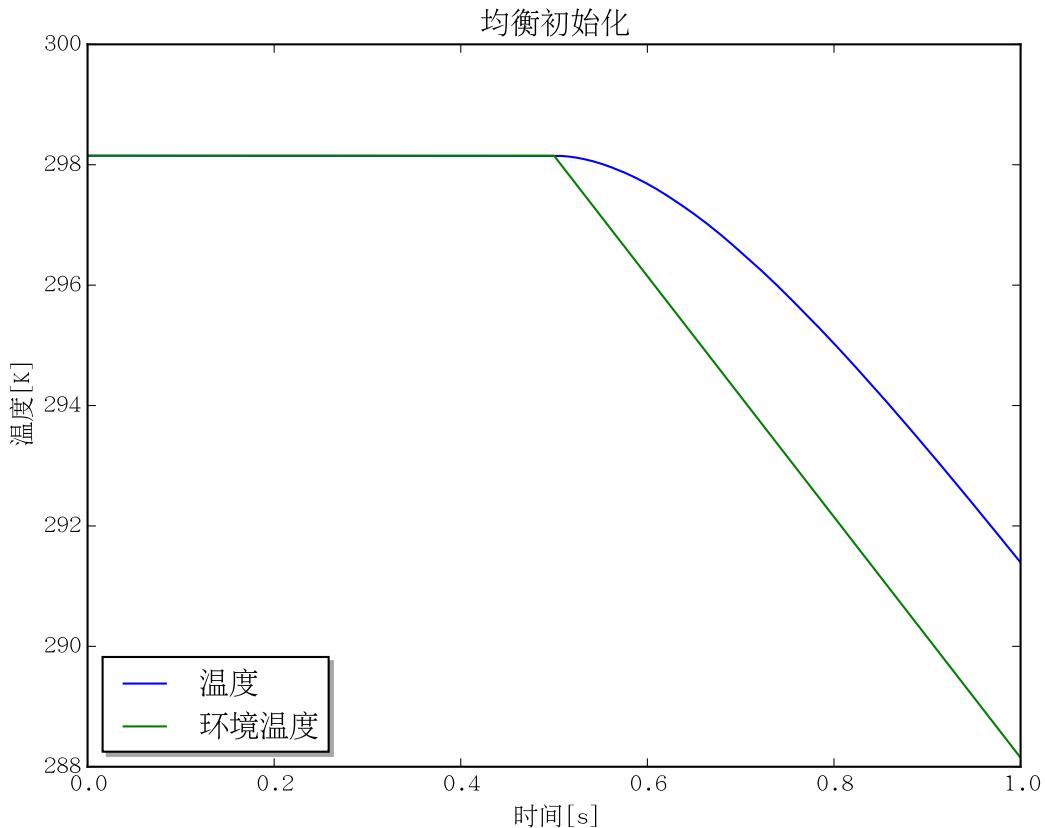
正如我们在初始化（31）中的讨论所了解到的一样，我们可以通过添加初始方程解决初始瞬态的问题。我们用初始方程找到这样的一个 T 值，使得我们的系统从平衡态开始仿真，即：

```
model NewtonCoolingSteadyThenDynamic
  "Dynamic cooling example with steady state conditions"
  type Temperature=Real(unit="K", min=0);
  type ConvectionCoefficient=Real(unit="W/(m2.K)", min=0);
  type Area=Real(unit="m2", min=0);
  type Mass=Real(unit="kg", min=0);
  type SpecificHeat=Real(unit="J/(K.kg)", min=0);

  parameter ConvectionCoefficient h=0.7 "Convective cooling coefficient";
  parameter Area A=1.0 "Surface area";
  parameter Mass m=0.1 "Mass of thermal capacitance";
  parameter SpecificHeat c_p=1.2 "Specific heat";

  Temperature T_inf "Ambient temperature";
  Temperature T "Temperature";
initial equation
  der(T) = 0 "Steady state initial conditions";
equation
  if time<=0.5 then
    T_inf = 298.15 "Constant temperature when time<=0.5";
  else
    T_inf = 298.15-20*(time-0.5) "Otherwise, increasing";
  end if;
  m*c_p*der(T) = h*A*(T_inf-T) "Newton's law of cooling";
end NewtonCoolingSteadyThenDynamic;
```

我们唯一改变了的是初始化的方程。相比从某一固定温度开始系统的仿真，我们现在让仿真开始时温度的变化为零（至少在一开始没有外加扰动影响的时候）。现在的温度响应不再包括任何初始瞬态，我们可以只专注于系统对干扰的响应。



## 简洁性

if 语句的一个问题是，它们可以让相对简单的行为变化看起来相当复杂。我们可以用好几种另外的语法结构以更少的代码获得相同的行为。

第一种方法是使用一个 if 表达式。相对于包含方程组“分支”的 if 语句，if 表达式只包含表达式分支。此外，语法上 if 表达式也更为简洁。倘若我们使用 if 表达式，我们的 equation 部分会被简化为：

```
equation
  T_inf = 298.15 - (if time<0.5 then 0 else 20*(time-0.5));
  m*c_p*der(T) = h*A*(T_inf-T) "Newton's law of cooling";
```

此外，我们可以使用许多 Modelica 的内置函数，如 max 来表示在环境温度的变化，例如：

```
equation
  T_inf = 298.15 - max(0, 20*(time-0.5));
  m*c_p*der(T) = h*A*(T_inf-T) "Newton's law of cooling";
```

## 事件

我们已经看到了几种方式来表达在系统行为的突然变化。但要特别指出的是，我们不仅仅描述了环境温度的变化，我们也规定了何时 (when) 它会发生变化。

考虑最后一个例子：我们的系统初始状态是一个平衡态。在仿真的开始，系统没有显著的动态。因为系统中并没有真正变化，积分器是不可能积累显著的积分误差。所以，为了最小化完成仿真所需的时间，可变步长的积分器会在这样的情况下提高其步长。

但这样做有其风险。风险在于，系统中突然的扰动可能会让积分器“措手不及”。如果发生了这样的扰动，积分器对于大步长不会导致显著积分误差的假设，便不再成立了。

那么问题就变成，积分器如何知道何时它可以使用一个大的时间步长，何时不能。通常情况下，这些积分方案使用一种试错法。积分器试图使用大步长，然后估算由该步长引入的误差。如果误差小于某个阈值，则积分器接受新算出的状态（或可能尝试更大的步长）。但倘若在该步长引入了过多的错误，那么积分器便尝试较小的步长。但它们不知道为了符合误差阈值要求需要多少的步长。这意味着它们将盲目地尝试更小的步长，直到符合条件。

但 Modelica 不止是关于相关系统的积分。Modelica 编译器研究问题的结构。在我们的所有例子中，编译器都可以看到系统有一个明显的行为变化。不仅如此，它可以看到这个行为改变是一个时间事件。亦即一个这样的一个事件，其发生时间被先验地、不需要任何关于解的轨迹的信息就被确定。

因此，Modelica 语言编译器会告诉底层的积分器，在第 0.5 秒时系统行为会突然发生变化，然后它会指导积分器一步不多地积到该时点。结果，在时间步长内从不会发生急剧的变化。相反，该积分器会简单地在该事件的另一侧重新开始积分。这完全避免为了最小化步长内的误差而去盲目搜索截止时间。相反，该积分器会自动恰好积到该点，然后在该点后重新启动。

Modelica 语言有众多能优化仿真进行方式的特性。这是一个其中一个例子。读者可以在关于事件（76）的紧接小节中找到对于事件处理的进一步讨论。在接下来的章节中，我们还可以看到另一种更复杂事件的例子。这种事件会依赖于解中的变量的值。

## 第 2.1.2 节 弹跳球

### 对弹跳球的建模

在前面的例子（39）里，我们看到了某些事件和时间的关系。这些所谓“时间事件”（time event）只是其中一种“事件”（event）。我们将在本节中研究另一类事件，即状态事件（state event）。状态事件是依赖于解的轨迹的一类事件。

处理状态事件相对而言更为复杂。不同于能够先验地知道发生时间的时间事件，状态事件依赖于解的轨迹。因此，我们无法完全避免去“搜索”发生事件的时点。

为了看到状态事件的效果，我们考虑一个弹跳球在平坦的水平面上弹跳时的行为。球在平面以上时会在重力作用下加速。而球最终与平面接触时，它会遵从以下关系从表面反弹：

$$v_{\text{final}} = -ev_{\text{initial}}$$

其中  $v_{\text{final}}$  为球在碰撞瞬间（在垂直方向）的速度  $v_{\text{initial}}$  为球在碰撞前的速度。而  $e$  则是恢复系数。恢复系数是对球在碰撞后所保留动量占比大小的度量。

如果我们把以上内容转化为 Modelica，可能会是以下的样子：

```
model BouncingBall "The 'classic' bouncing ball model"
  type Height=Real(unit="m");
  type Velocity=Real(unit="m/s");
  parameter Real e=0.8 "Coefficient of restitution";
  parameter Height h0=1.0 "Initial height";
  Height h;
  Velocity v;
  initial equation
    h = h0;
  equation
    v = der(h);
    der(v) = -9.81;
    when h<0 then
      reinit(v, -e*pre(v));
    end when;
end BouncingBall;
```

在这个例子中，我们采用参数  $h_0$  指定球在平面上的初始高度，参数  $e$  指定恢复系数。变量  $h$  和  $v$  分别表示高度以及垂直速度。

本例有趣的地方在于模型中的公式。特别是其中的 when 语句：

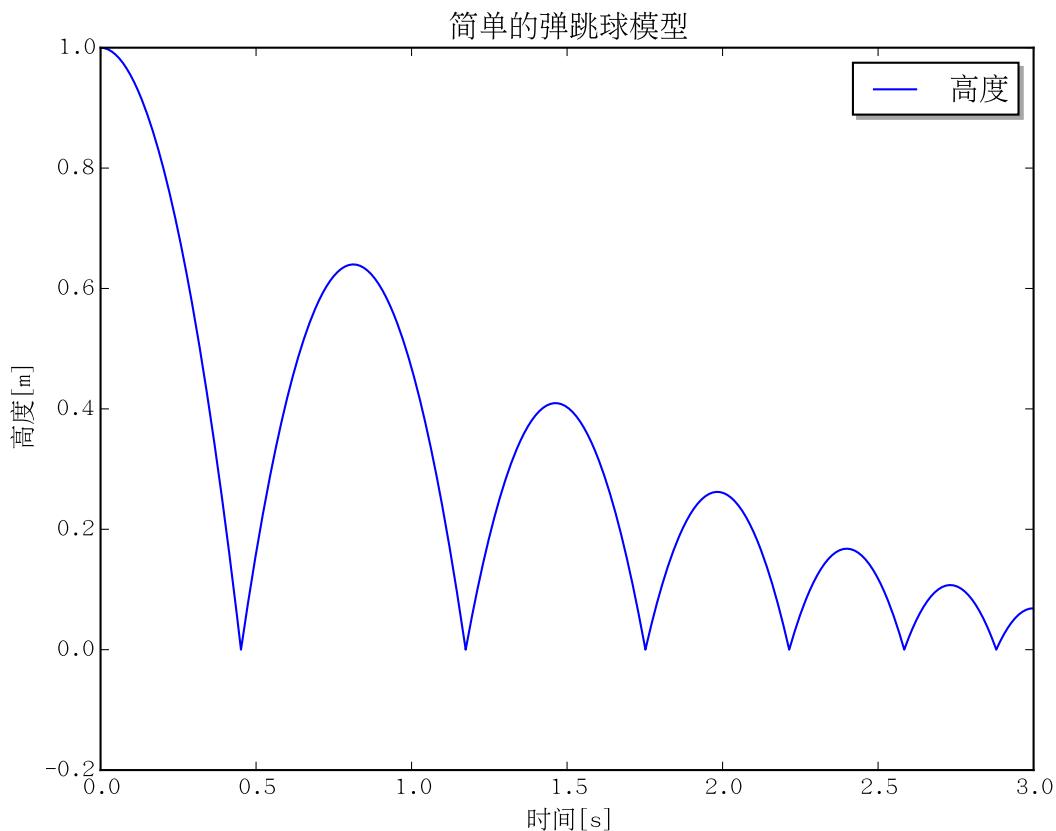
```

equation
  v = der(h);
  der(v) = -9.81;
  when h<0 then
    reinit(v, -e*pre(v));
  end when;

```

when (当) 语句由两部分组成。第一部分是条件表达式，表示事件发生的时刻。在本例中，事件发生在“当”高度  $h$  第一次降低到 0 以下时。when 表达式的第二部分描述了在事件出现时所发生的事情。在本例中， $v$  的值被 reinit（重新初始化）操作符重新初始化了。我们可以用 reinit 操作符为状态设定新的初始条件。从概念上讲，你可以认为 reinit 当成是在仿真过程中插入的 initial equation。但是，reinit 只会改变一个变量的值，而且这种改变总是显式的（也就是说它没有 initial equation 那么灵活）。在本例中，reinit 语句会将  $v$  的值重新初始化，使之与在碰撞前的  $v$ （也就是 pre( $v$ )）方向相反，大小变为原来的  $e$  倍。

假设  $h_0$  为正值，重力不懈的拉扯确保球最终将贴近平面。在  $h_0$  为 1.0 的情况下运行仿真，我们可以看到模型有以下行为：

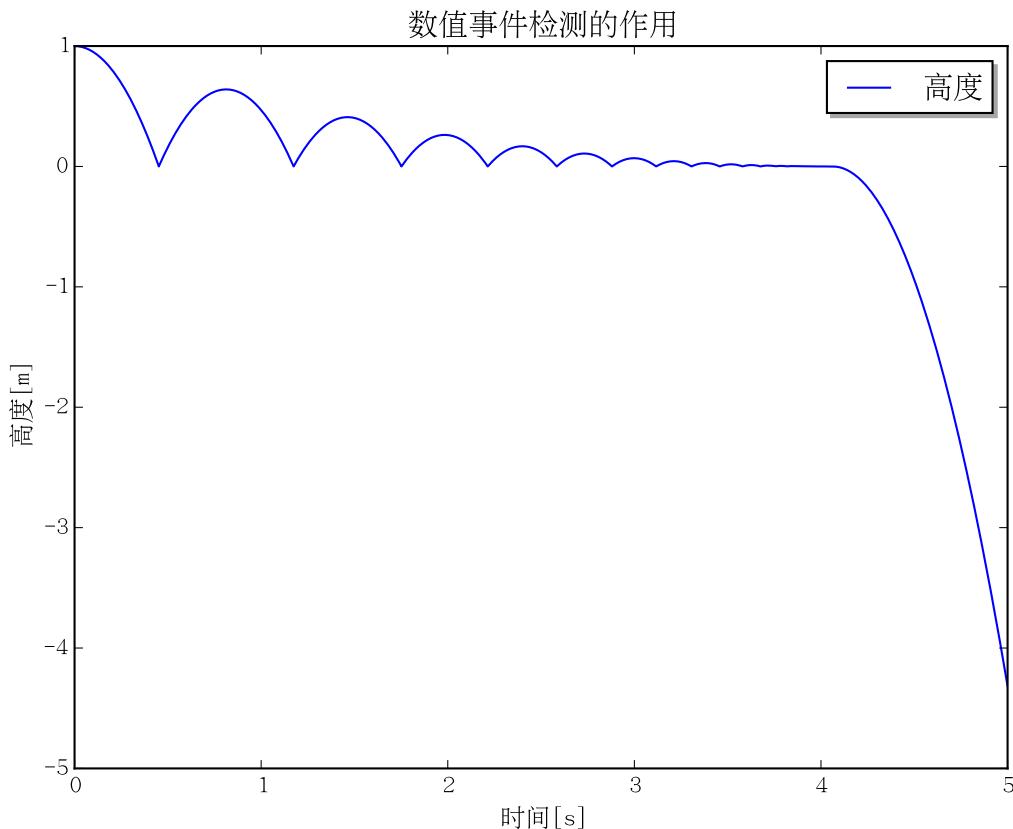


在此图中，我们看到在 0.48 秒左右，球第一次和表面发生了碰撞。这是因为条件  $h < 0$  在该时刻首次为真。需要注意的是，是什么使之成为状态事件（这个条件表达式引用了其他变量而不是变量 time，而非我们在前面冷却例子（39）内见到的样例。）

因此，仿真进行时假定球是在自由下落。一直到它发现在解的轨迹中某个特定的时间步长内，条件表达式  $h < 0$  的值发生变化。倘若出现了这样的时间步长，求解器必须确定条件式的值变真的确定时刻。一旦该时刻被确定，求解器通过处理在该时间内影响了系统状态的 when 语句（如任何 reinit 语句）以计算该系统的状态。然后求解器以计算出的状态为初值重启积分计算。本例弹跳球内 reinit 语句的作用是，计算一个碰撞后让球重新（开始）上升的新值  $v$ 。

但要记住的是，一般情况下大多数的 Modelica 模型的解是由数值方法得出的。正如我们将很快看到，这会对我们考虑的离散行为产生深远的影响。这是因为，在所有的事件（时间或状态事件）的核心都是条件表达式，例如本例中的  $h < 0$ 。

倘若我们模拟弹跳球的时间稍长，这种含义就会变得很清楚。在这种情况下，大多数 Modelica 语言的工具将提供这样的一个解：



显然，我们在看到轨迹后马上知道有些问题了。但到底是什么有问题呢？

### 数值精度

正如我们前面的暗示，答案就在于对条件表达式  $h < 0$  的数值处理上。由于数值精度的问题，我们无法得知我们在开始计算本步时是到底在一个刚刚发生的事件之后，还是在事件即将发生之时。

为了解决这个问题，我们必须引入一定量的迟滞（死区）。这意味着在这种情况下，一旦条件  $h < 0$  为真，我们必须离状态“足够远”才能让事件再次发生。换句话说，当  $h$  小于零时会发生事件。但在我们再次触发事件前，我们要求  $h$  必须大于某个  $\epsilon$ 。换句话说， $h$  并不是仅仅简单地大于零， $h$  必须大于  $\epsilon$ （其中  $\epsilon$  是由求解器通过检查一系列缩放因子得到的）。

之前仿真里的问题是，每次球反弹时， $h$  的峰值都会有些下降。这里峰值是指，球即将开始再次下降时  $h$  的值。最终， $h$  峰值不足以超过该临界值  $\epsilon$ 。反过来，这意味着 when 语句从未触发以及 reinit 语句不会再重置  $v$ 。结果是球在继续不停地自由下落。

所以一个显然的问题就是，如何实现我们真正的目标行为（球永远不会降到地面以下）。因此，我们必须对模型进行如下的一些小改动：

```
model StableBouncingBall
  "The 'classic' bouncing ball model with numerical tolerances"
  type Height=Real(unit="m");
  type Velocity=Real(unit="m/s");
  parameter Real e=0.8 "Coefficient of restitution";
  parameter Height h0=1.0 "Initial height";
  constant Height eps=1e-3 "Small height";
  Boolean done;
  Height h;
  Velocity v;
```

```

initial equation
  h = h0;
  done = false;
equation
  v = der(h);
  der(v) = if done then 0 else -9.81;
  when {h<0,h<-eps} then
    done = h<-eps;
    reinit(v, -e*(if h<-eps then 0 else pre(v)));
  end when;
end StableBouncingBall;

```

应该指出的是，许多方法都可以用以解决这个问题。这里提出的解决方案仅是其中之一。在这种方法中，我们实际上创建了两个表面。其中一个在 0 的高度上，另一个在一个高度 -eps (略低于 0)。球“正常地”弹起时只会触发 when 语句的首个条件。然而，如果球在触地后不能足够高地回弹，从而“穿过”中的第一表面。我们检测到这点（以及球已经穿过的事实），并设置 done 标志符。done 标志符的效果实际上就是关闭重力。

注意这里 when 语句的语法：

```

when {h<0,h<-eps} then
  done = h<-eps;
  reinit(v, -e*(if h<-eps then 0 else pre(v)));
end when;

```

特别要注意，这个语句不只是一个条件表达式，而是两个。更准确地说，它实际上有向量形式的条件表达式。在本书的后面我们会介绍向量与数组 (81)，但在本章我们必须指出 when 既可以包含一个标量条件表达式，也可以包含矢量条件表达式。

如果一个 when 语句包含矢量条件表达式，那么当任意一个条件表达式为真时便会触发 when 语句内的语句。请仔细注意这断解释的语法。人们很有可能把如下形式 Modelica 代码：

```

when {a>0, b>0} then
  ...
end when;

```

理解为“当 a 大于零或 b 大于零”。**非常重要**的是要避免一个十分常见的错误，即误认为以下两个 when 语句是等价的：

```

when {a>0, b>0} then
  ...
end when;

when a>0 or b>0 then
  ...
end when;

```

这两者并不等价的。为了理解上述差异，我们把条件表达式改为如下形式：

```

when {time>1, time>2} then
  ...
end when;

when time>1 or time>2 then
  ...
end when;

```

还记得我们之前对 when 语句中矢量表达式的陈述：当任意一个条件表达式为真时便会触发 when 语句内的语句。假设我们从 time= 0 开始运行仿真，直到 time=3，那么 when 语句：

```

when {time>1, time>2} then
  ...
end when;

```

会被两次触发。第一次在当 time>1 为真时。另一次在 time>2 为真时。与此相反，在这种情况下：

```
when time>1 or time>2 then
  ...
end when;
```

仅存在一个条件表达式，且这个表达式仅仅一次变为真（当 time>1 为真并保持为真）。or 操作符实际上屏蔽了第二项条件 time>2，使得第二项条件在此情况下甚至可以不存在。换句话说，该条件仅一次变为真。其结果是，在 when 语句中的语句仅被触发一次。

关键是要记住的是，对于 when 语句，向量条件表示任何，而非逻辑或。此外，该语句仅在条件变为真的瞬间被激活。我们会在本章的后面部分讨论对比 if 和 when ( 78) 之间的重要区别时，进一步探讨最后这句话的含义。

### 第 2.1.3 节 状态事件的处理

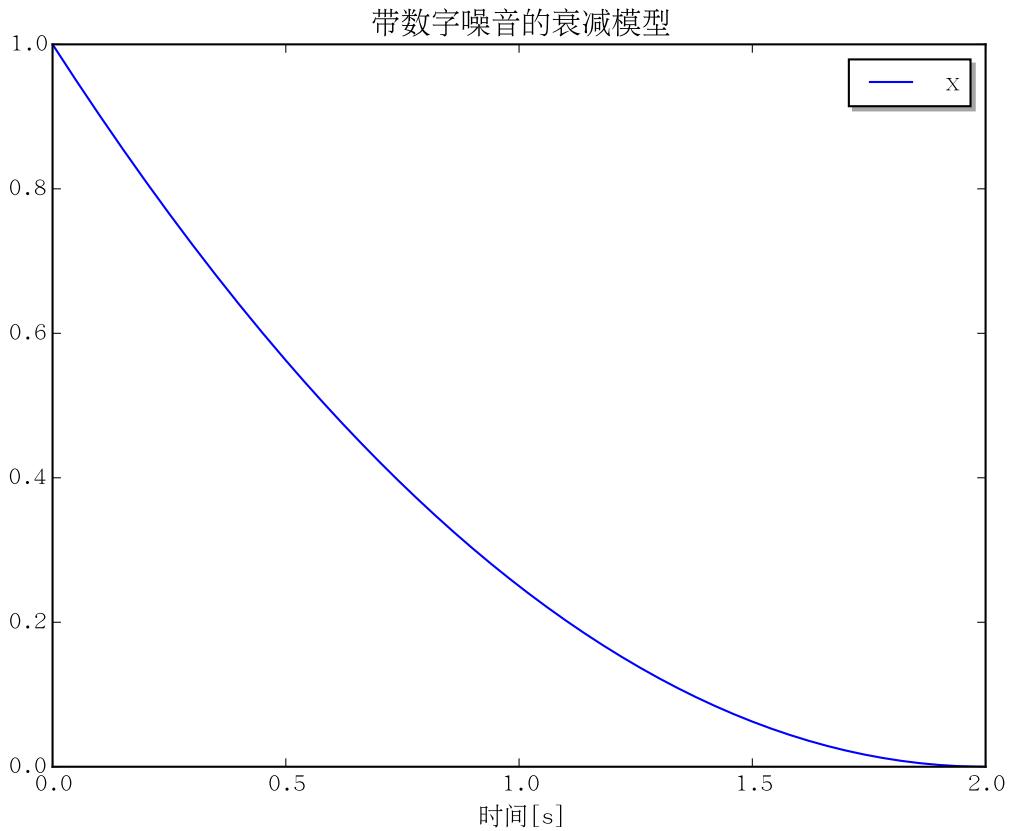
我们已经介绍了时间事件 ( 39) 和状态事件 ( 43)。现在让我们来看看与状态事件相关的一些重要的困难。出人意料的是，这些困难甚至会在简单的模型里出现。

#### 基本衰减模型

请考虑以下简单得几近不值一提的模型：

```
model Decay1
  Real x;
initial equation
  x = 1;
equation
  der(x) = -sqrt(x);
end Decay1;
```

如果试图对此模型进行 5 秒的仿真，我们会发现仿真大概用 2 秒后就停止了。结果是如下的轨迹：



再一次，数值问题混了进来。尽管在数学上  $x$  的值应该不可能的降到低于零，但在使用数值积分方法时，少量错误可能混入结果，使得  $x$  低于零。发生这种情况时， $\sqrt{x}$  的表达式产生浮点异常，结果仿真终止。

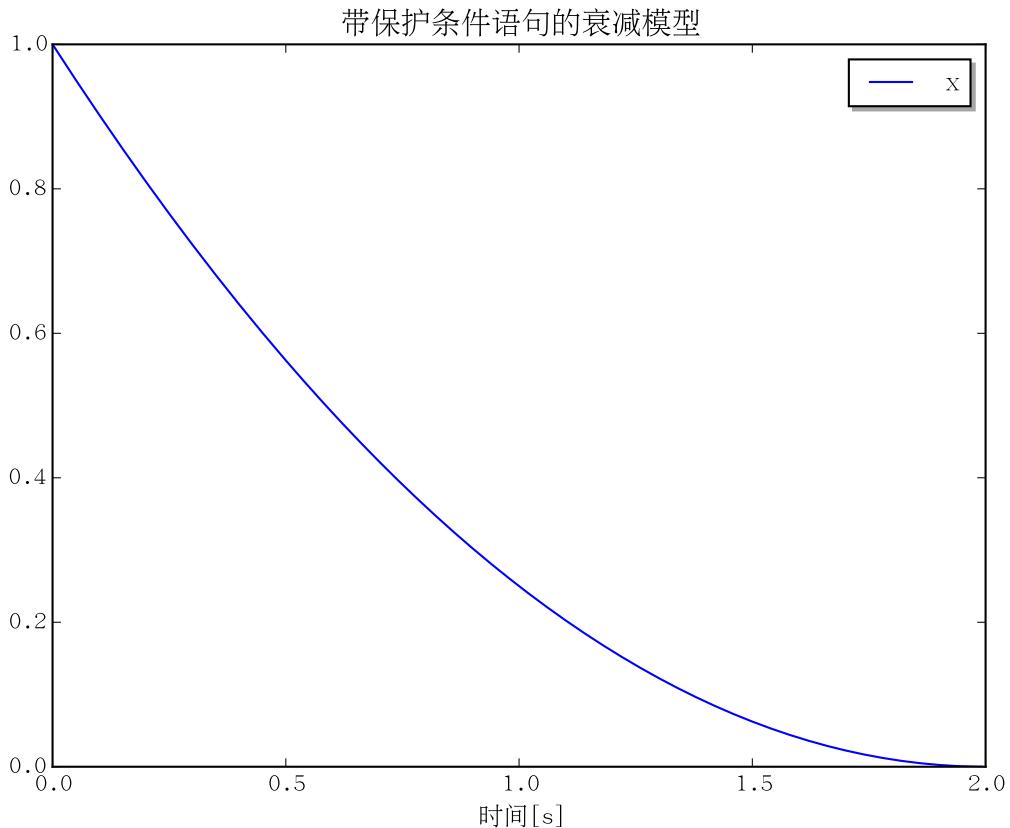
### 保护表达式

为了避免这种情况，我们可以引入一个 if 表达式防止计算负数的平方根，如下：

```
model Decay2
  Real x;
initial equation
  x = 1;
equation
  der(x) = if x>=0 then -sqrt(x) else 0;
end Decay2;
```

这个模型进行仿真，我们得到如下的轨迹<sup>1</sup>：

<sup>1</sup> 这个模型不会总会仿真失败。失败的出现取决于仿真引入多少积分误差。而积分误差则又取决于所使用的数值容差。



再一次地，仿真失败。但是为什么呢？它出于相同的原因失败了：对负数的求平方根而导致的数值异常。大多数人在看到像这样（或者例如被零除）的浮点异常错误信息时，他们都非常困惑，毕竟他们都像我们一样写下了保护表达式。他们自然认为是当  $x$  小于零时  $\sqrt{x}$  根本不可能会被计算。**但这个假设是错误的。**

## 事件和条件表达式

### 事件在行为上的作用

若有如下 if 表达式：

```
der(x) = if x>=0 then sqrt(x) else 0;
```

$\sqrt{x}$  被以负参数调用是完全有可能的。其原因与这是状态事件有关。请记住，时间事件的发生时间是可以预知的。但是，状态事件则不然。为了确定事件何时会发生，我们必须在解的轨迹寻找以确定条件（例如： $x>=0$ ）何时为假。

要了解的重要一点是，直到事件发生时，行为都不会改变。换句话说，这个 if 表达式的两个值代表了两种类型的行为， $\text{der}(x)=\sqrt{x}$  和  $\text{der}(x)=0$ 。由于  $x$  最初大于零，最初的行为便是  $\text{der}(x)=\sqrt{x}$ 。求解器将继续使用该方程，直到它确定了由  $x>=0$  所表示的事件发生的时间。而为了确定该事件发生的时间，求解器必须越过条件表达式的值发生变化的点。这意味着，在试图准确确定条件  $x>=0$  从真假的变化的时间这一过程中，虽然  $x$  已经为负，求解器仍会继续使用公式  $\text{der}(x)=\sqrt{x}$ 。

大多数用户开始是认为每次  $\text{der}(x)$  被计算时，if 表达式也会被计算（特别是在 if 表达式中的条件表达式）。但愿前面的章节已经很清楚地阐释了，这并非事实。

我们可以减少在尝试和重试积分步长中所花费的时间。这是得益于 Modelica 可以从 if 表达式提取出所谓的“过零”(zero-crossing) 函数。这些函数之所以被称为过零函数，因为它通常被构造成为事件发生的位置有根存在的形式。例如，如果我们有以下 if 表达式：

```
y = if a>b then 1 else 0;
```

这里的过零函数是  $a - b$ 。之所以选择这个函数的原因是，它在  $a > b$  成立的时刻正好是由正转为负。

回想我们在前面的公式

```
der(x) = if x>=0 then sqrt(x) else 0;
```

在此过零函数显然是  $x$ 。这是因为事件在  $x$  自己穿过零时发生。

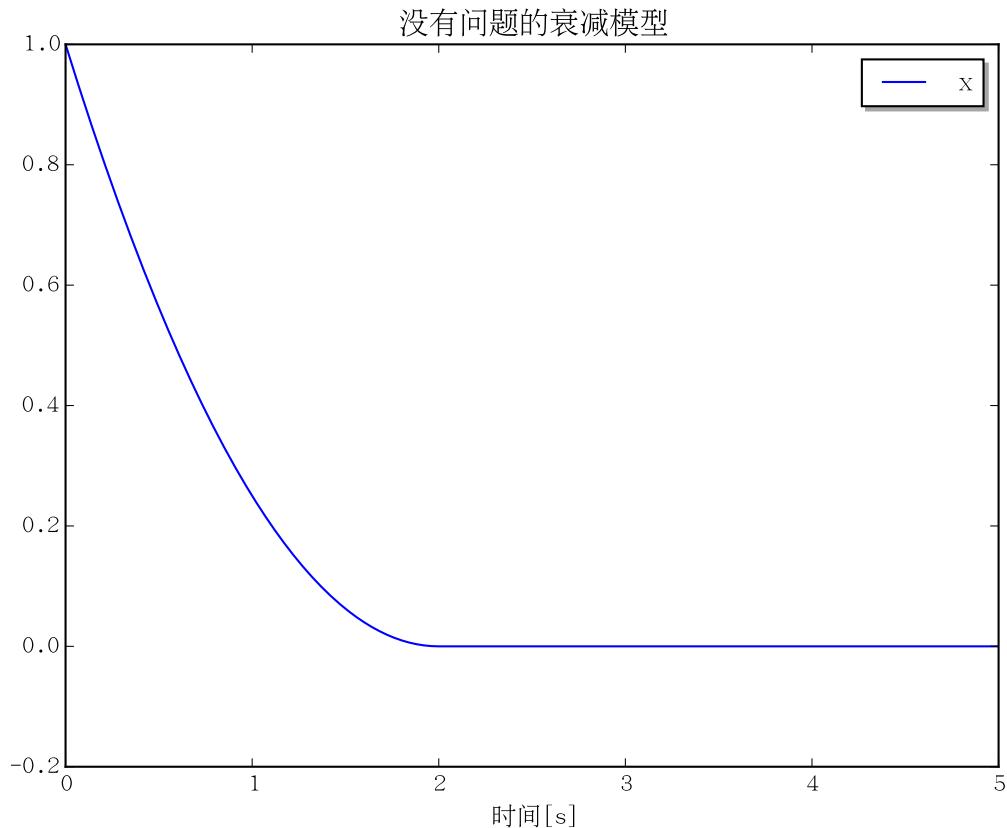
Modelica 编译器收集在模型中所有的过零函数以给予积分器使用。在积分时，积分器检查是否有任何的过零函数变了号。如果这些过零函数变了号，那积分器便使用在该步长中计算出的解去内插过零函数，以此计算交叉函数变号的时刻。而这就是事件发生的时间点。这种方法非常有效率。因为求根算法可以使用更多的信息来帮助他们识别为根的位置（如过零函数的导数）。加上由于它不涉及采取额外的积分步骤，而仅从触发事件的积分步长开始计算内插函数的值，这种求解算法本身非常花销非常低。

## 事件的抑制

但经过这一切，如何避免我们在 Decay1 和 Decay2 模型中所看到的问题，仍然是不明确的。答案是一个特殊的操作符，noEvent。noEvent 操作符抑制了这种特殊的事件处理方式。相反，它提供大多数用户一开始预期中的行为。也就是，在每一个  $x$  的取值都会进行条件表达式的计算。我们可以从下面的模型中看到 noEvent 操作符的作用：

```
model Decay3
  Real x;
initial equation
  x = 1;
equation
  der(x) = if noEvent(x>=0) then -sqrt(x) else 0;
end Decay3;
```

结果如下：



现在仿真毫无问题地完成了。这是因为 noEvent 确保了  $\sqrt{x}$  不会在  $x$  值为负时被调用。

这似乎有点奇怪，我们必须显式地加入 noEvent 操作符才能得到我们认为最直观的行为。为什么不把默认行为设为最直观的一个呢？答案是性能原因。使用条件表达式生成事件提高了仿真的性能。因为这给予求解器何时要为行为突变做准备的相关线索。大多数时候，这种方法不会导致任何问题。我们在本章中提出的例子都是为了强调这个问题，但这些例子并不能代表大多数情况。出于这个原因，noEvent 不是默认的，而是必须显式地使用。但是应当指出的是，noEvent 操作符应该只用于行为变化是平滑过渡的时候。否则它会带来性能问题。

## 抖振

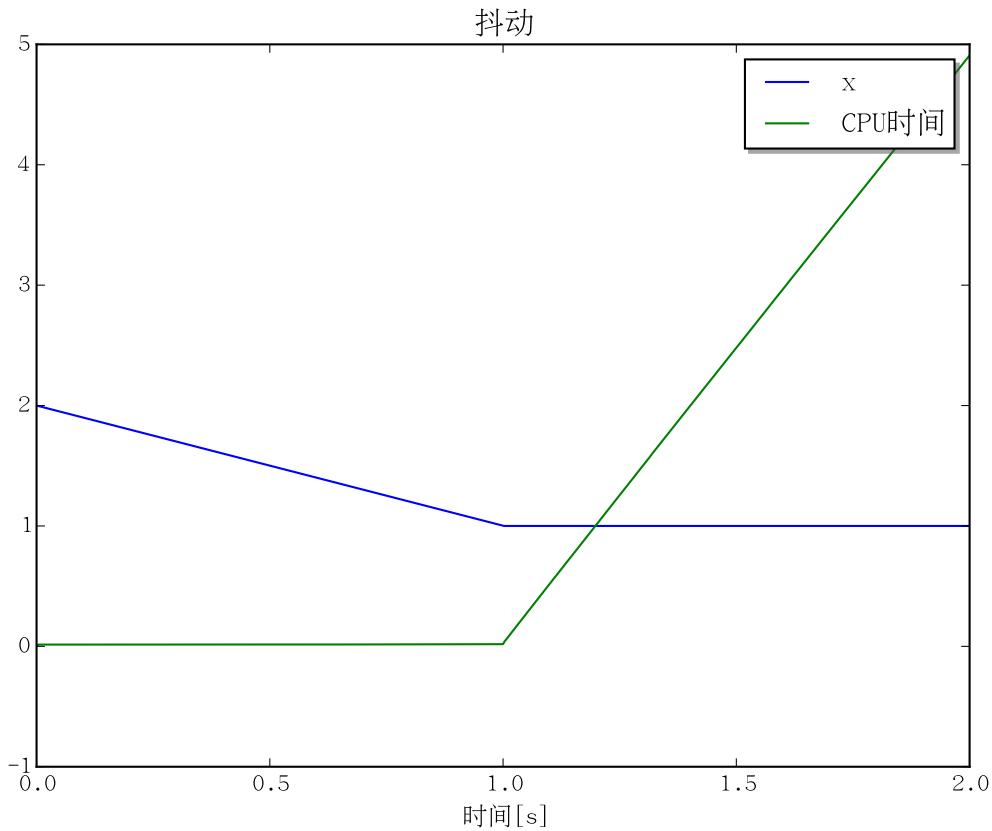
用 Modelica 语言时，你早晚会遇到一个被称为“抖动”（chattering）常见现象。考虑下面的模型：

```
model WithChatter
  Real x;
initial equation
  x = 2;
equation
  der(x) = if x>=1 then -1 else 1;
end WithChatter;
```

实际上，这个模型的行为是，对  $x$  的任何初值，状态  $x$  都会线性渐近到 1。在数学上讲，一旦  $x$  值到达 1，它的值便不再会变更。因为，任何从 1 的偏移，不论正负，都将立即导致它返回到 1。

但我们不会用一个严格的数学方法去求解这些方程。相反，我们将使用浮点数表示法表示变量，而用数值积分器计算结果。因此，我们只能满足于有限的精度和积分误差。这些误差的净效应是， $x$  的轨迹不会完全保持 1 而将偏离略微上方和下方。而每次发生这种情况都会生成一个事件。

对此模型进行仿真给了我们如下结果：

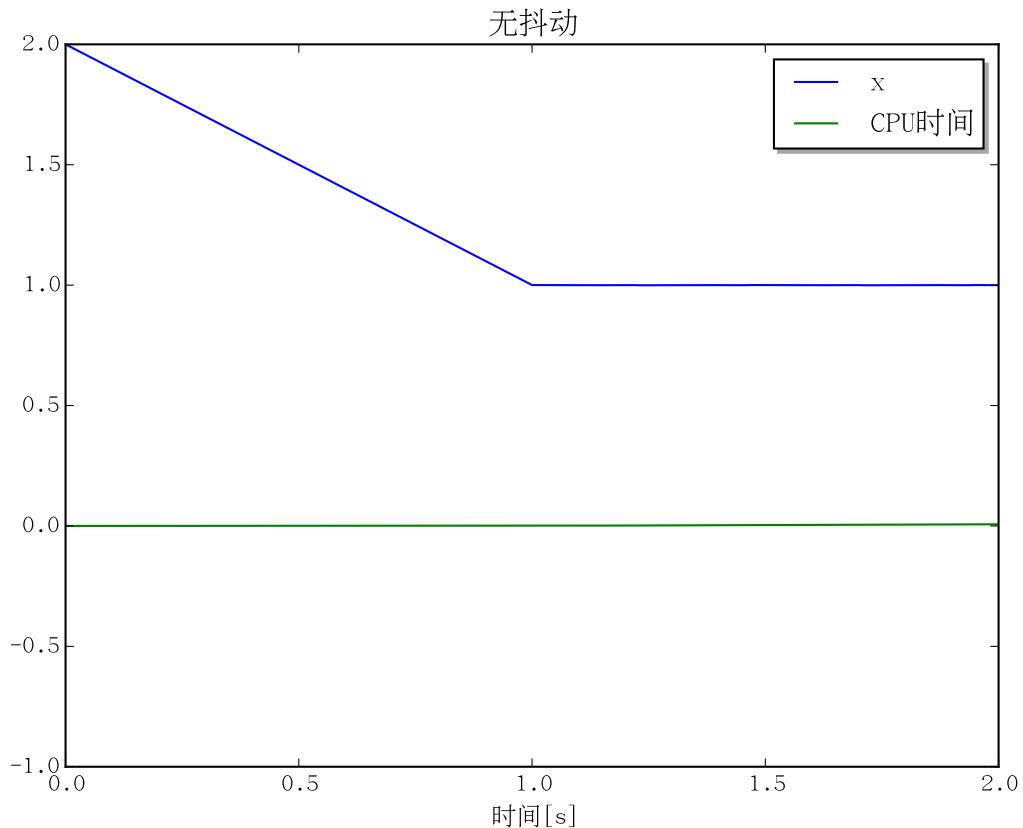


这种模型可能会引起一种被称为“抖动”的现象。抖动其实只是由于求解器在大量事件产生时，由于过度地缩短时间步长所引致的仿真速度下降。如果我们观察在仿真过程中所使用的 CPU 时间，抖动对仿真性能的影响其实是显而易见的。当  $x$  接近 1 时，CPU 时间便开始急剧上升。原因是在幕后事件造成大量的非常小的时间步长而显著地增加所执行的计算的数量。这是因为在幕后，事件造成大量非常小的时间步长，从而让所执行的计算量也急剧增加。WithChatter 这个例子之所以重要，在于本例有一个显而易见的解析解，但即使如此，本例仍然受到事件的高频率产生而减低了仿真性能。

这是另一种 noEvent 运算符有所助益的情况。因为我们知道，这 if 表达式并不引入任何行为上的突然变化，我们可以用 noEvent 操作符把条件表达式包裹起来。如下：

```
model WithoutChatter
  Real x;
initial equation
  x = 2;
equation
  der(x) = if noEvent(x>=1) then -1 else 1;
end WithoutChatter;
```

这样做以后，我们会得到几乎相同的解，但仿真性能更佳：



请注意  $x$  到达 1 的前后，CPU 占用率的斜率没有明显的变化。这与在 WithChatter 的情况下有着显著差异。

在现实中，像这样的方程是罕见的。在本例中，我们使用一个极端情况试图清楚地显示抖动所带来的影响。这里所描述的行为并不是特别现实或物理的。在这种情况下，我们夸大了抖动产生的效果以清楚地表明仿真性能受到的影响。

抖动在现实世界中的更典型的例子包括，在某个稳定点周围生效的条件表达式（Decay2 就是一个很好的例子）。在这种情况下就会发生抖动，因为该系统往往自然稳定到其中在条件表达式的发生点或者其附近。而由于精确度和数值计算上的考虑，与条件式相关联的事件会被频繁触发。倘若系统有许多部件在这样的平衡点附近，这个效果便会被加剧了。

### 速度与准确性

希望讨论至今大家已经清楚，为什么在某些情况下有必要抑制事件的产生。不过，大家可能会问，为什么不直接跳过事件，每次都计算条件表达式？那么，让我们花一些时间来探讨这个问题，并解释为何就整体而言把条件表达式与事件进行关联是个非常好的主意<sup>2</sup>。

如果没有事件检测，积分器只会直接跳过事件。而如果这种情况发生，积分器会错过重要的行为改变。这将会影响仿真的准确性有显著的影响。这是因为大多数积分程序的准确性是基于对被积函数及其导数连续性的假设。倘若这些假设不再成立了，我们需要让积分程序知道这点。如此，积分器便可以考虑这些行为的改变。

这就到事件出场的时候了。事件强迫积分器在行为发生变化的时刻停止，然后在发生行为变化后的时点重新启动。其结果是在牺牲模拟速度的前提下获得更高的精度让我们看一个具体的例子。请考虑下面这个简单的 Modelica 模型：

<sup>2</sup> 特别感谢 Lionel Belmon 对我原来讨论的质疑，以及找出了我的几个未经证实的假设。因为如此，现在的解释有了很大的改进，而且还加上了仿真结果以支持我的结论。

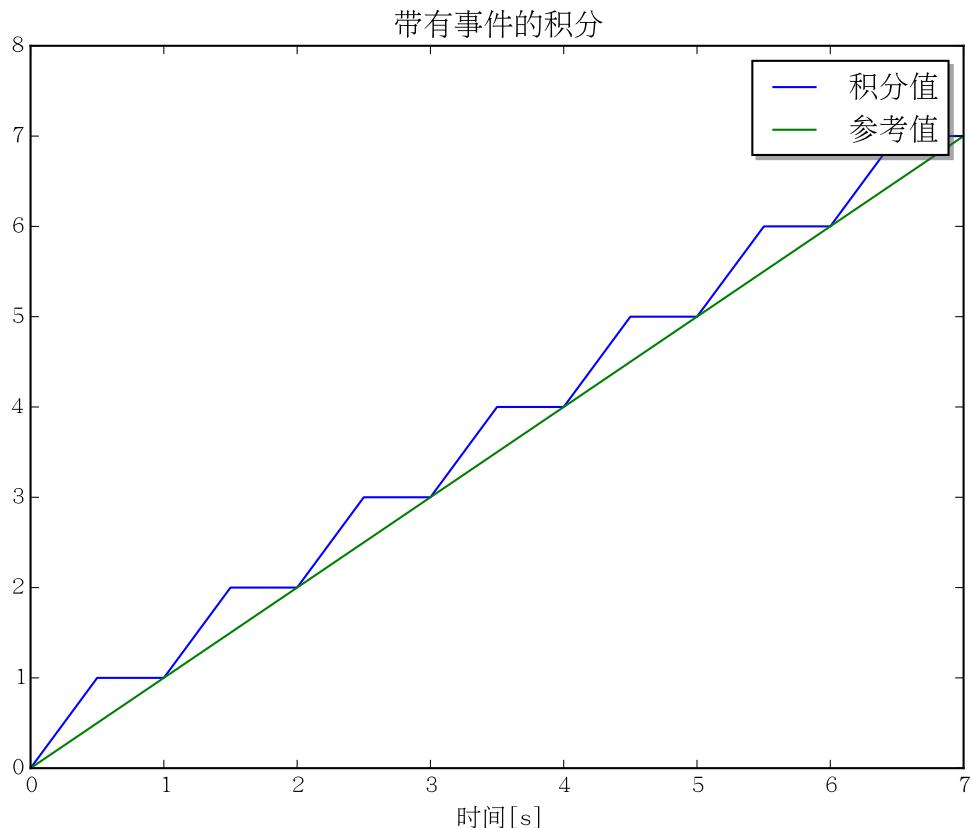
```

model WithEvents "Integrate with events"
parameter Real freq = 1.0;
Real x(start=0);
Real y = time;
equation
  der(x) = if sin(2*Modelica.Constants.pi*freq*time)>0 then 2.0 else 0.0;
end WithEvents;

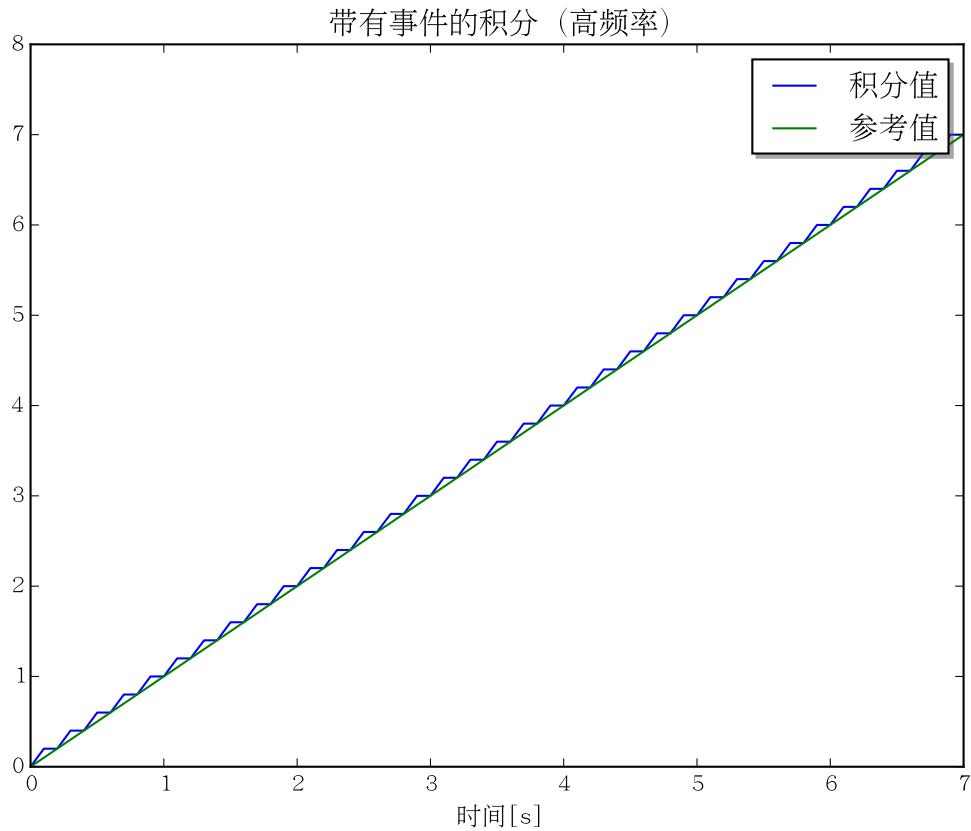
```

我们可以从这个系统看到在一半的时间里  $x$  的微分为 2。而在另外一半时间里  $x$  的微分则为 0。所以，在每一个周期中，“ $x$ ”平均的微分应该是 1。这意味着在每一个周期结束时， $x$  和 “ $y$ ” 应该相等。

倘若我们使用 `WithEvents` 对模型进行仿真，我们会得到如下的结果：



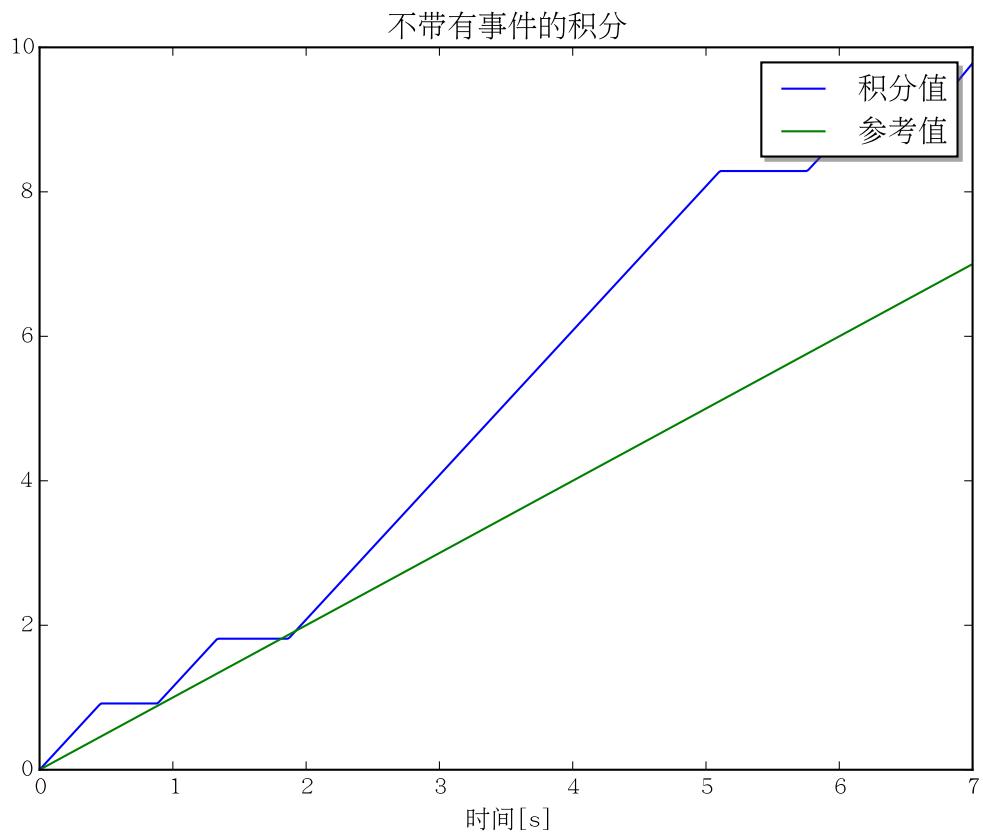
请注意在每个周期结束时， $x$  和  $y$  的轨迹如何在一点汇合。这是潜在的积分精度的视觉指示。即使我们增加基本周期的频率，我们可以看到这个特点仍然存在：



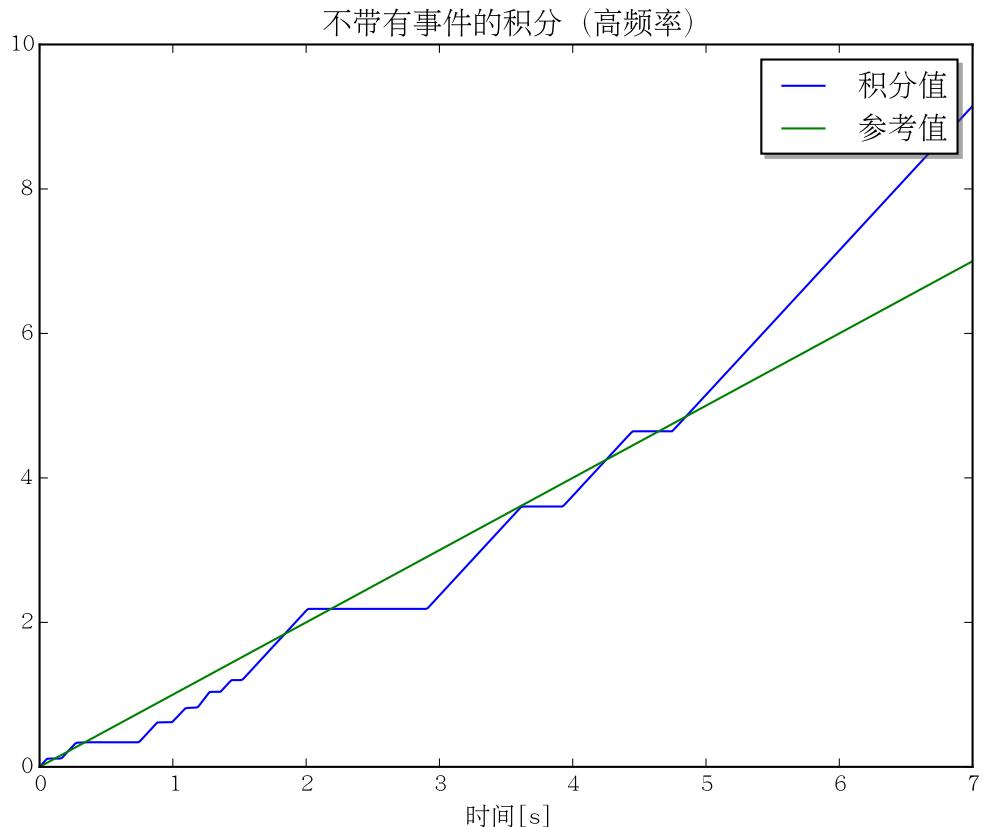
不过，在使用完全相同的积分参数的情况下，现在让我们考虑通过 noEvents 操作符以抑制事件产生的  
情况：

```
model WithNoEvents "Integrate without events"
parameter Real freq = 1.0;
Real x(start=0);
Real y = time;
equation
  der(x) = noEvent(if sin(2*Modelica.Constants.pi*freq*time)>0 then 2.0 else 0.0);
end WithNoEvents;
```

在这种情况下，积分器不能得知行为变化。尽管它会尽量准确地进行积分，但如果积分器没有明确知道这些行为改变发生，那么它会继续盲目地使用错误的微分值，并在行为变化后很长时间继续进行外推。如果使用相同的积分器设置对使用了 WithNoEvents 的模型进行仿真，我们可以看到两者的结果有如何显著的不同：



请注意积分器何其迅速地引入了些非常显著的错误。



在这些例子中使用的积分器设定是为了说明 noEvent 操作符对精度可能造成的影响而设置的。然而，选择这些设置的目的确实是强调这些差异。如果使用更典型的设置，结果上的差异很可能不这么会引人注目。此外，使用 noEvent 的影响无法预测或量化。因为，这些影响在使用不用的求解器时会显著有所不同。但核心的问题名曲：使用 noEvent 操作符可能对仿真结果的精度有显著影响。

#### 第 2.1.4 节 RLC 开关电路

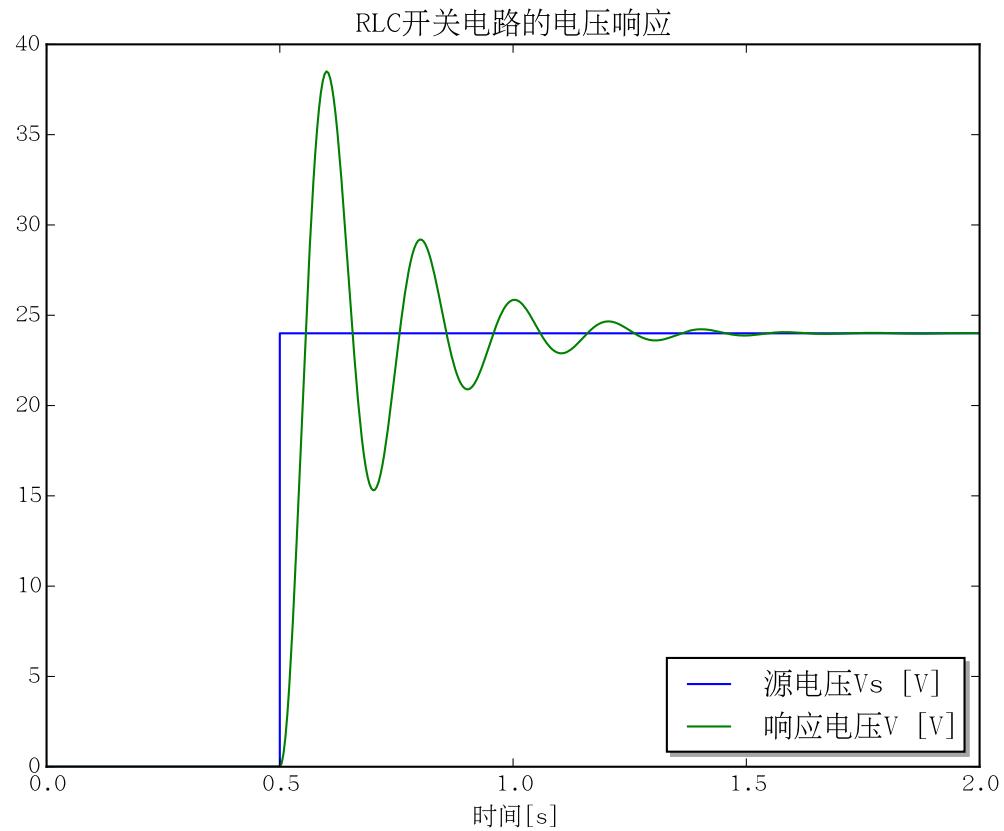
在本节中，我们将介绍另一个模型。正如本章前面部分（39）介绍的传热模型一样，这个模型里包含时间事件。这次，我们将展示如何能够模拟在第一章中介绍的那个（11）RLC 电路的开关行为。在介绍了前面的例子后，这个模型应该不会让大家感到意外。介绍这个例子仅仅是为了在电气模型背景下介绍时间事件。

RLC 开关电路模型如下：

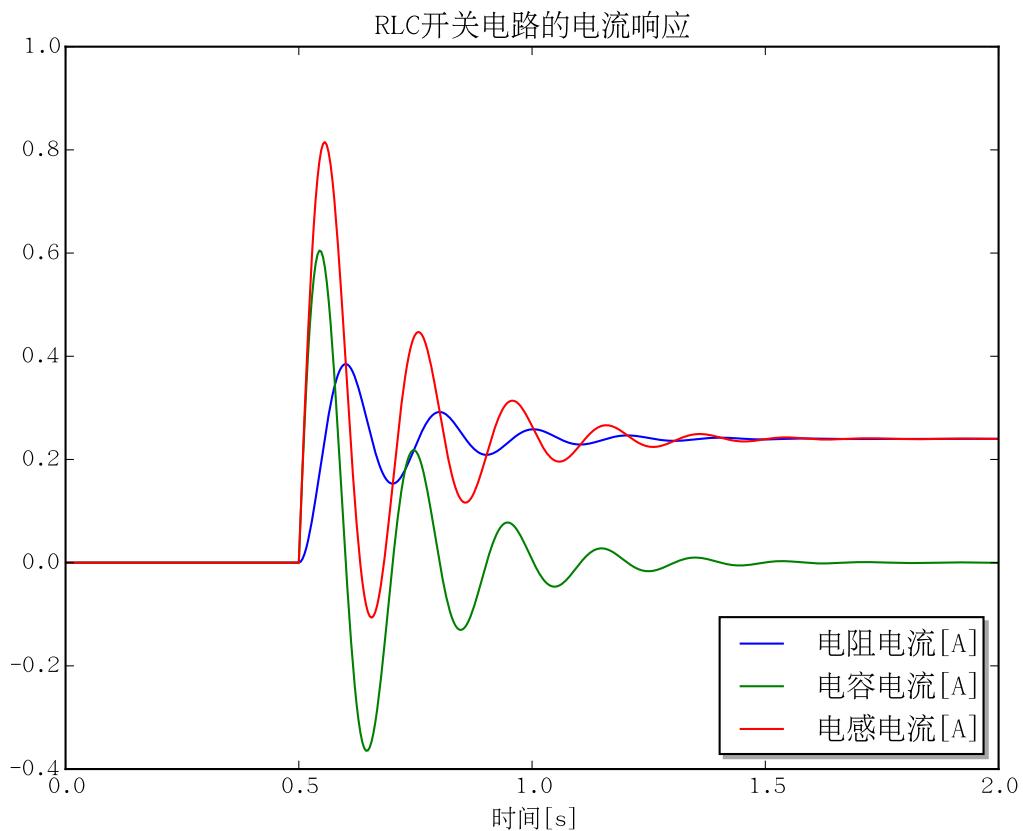
```
model SwitchedRLC "An RLC circuit with a switch"
  type Voltage=Real(unit="V");
  type Current=Real(unit="A");
  type Resistance=Real(unit="Ohm");
  type Capacitance=Real(unit="F");
  type Inductance=Real(unit="H");
  parameter Voltage Vb=24 "Battery voltage";
  parameter Inductance L = 1;
  parameter Resistance R = 100;
  parameter Capacitance C = 1e-3;
  Voltage Vs;
  Voltage V;
  Current i_L;
  Current i_R;
  Current i_C;
equation
  Vs = if time>0.5 then Vb else 0;
  i_R = V/R;
  i_C = C*der(V);
  i_L=i_R+i_C;
  L*der(i_L) = (Vs-V);
end SwitchedRLC;
```

时间事件在这个模型里是通过 if 表达式引入的。条件表达式里唯一随着时间变化的量就是 time。这意味着条件表达式会触发时间事件。而当底层的数值求解器在对系统公式进行积分时，它可以先验地知道事件的发生时间。

在下图我们可以看到在开关式电压供应下模型的电压响应：



此外，从下图我们可以看到电感、电阻和电容等元件的电流响应：



希望到这里，读者会对用以产生事件和扰动的基本机制感到直观而熟悉。

### 第 2.1.5 节 速度的测量

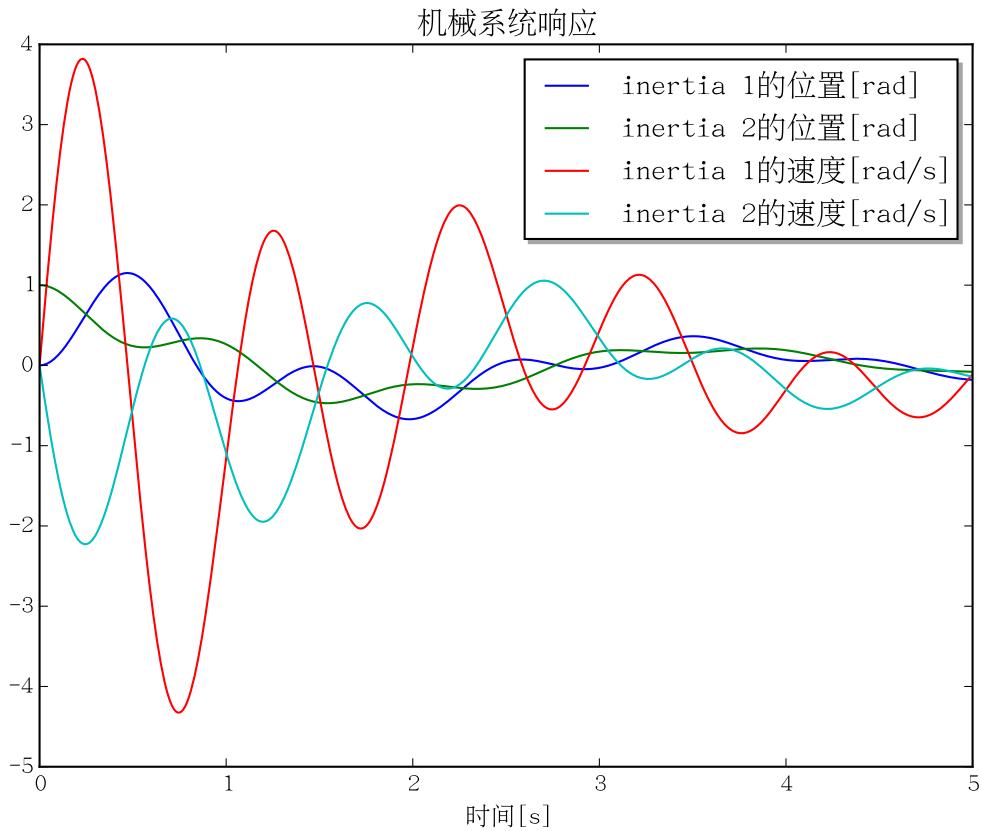
#### 基准系统

在许多应用中，我们需要对连续行为与离散行为之间的相互作用进行建模。在本节中，我们将看到用于测量旋转轴速度的技术。在这里的讨论里，我们将重用前面在[基本方程 \(3\)](#) 中讨论过的机械中的例子 (13)。

```
model SecondOrderSystem "A second order rotational system"
  type Angle=Real(unit="rad");
  type AngularVelocity=Real(unit="rad/s");
  type Inertia=Real(unit="kg.m2");
  type Stiffness=Real(unit="N.m/rad");
  type Damping=Real(unit="N.m.s/rad");
  parameter Inertia J1=0.4 "Moment of inertia for inertia 1";
  parameter Inertia J2=1.0 "Moment of inertia for inertia 2";
  parameter Stiffness k1=11 "Spring constant for spring 1";
  parameter Stiffness k2=5 "Spring constant for spring 2";
  parameter Damping d1=0.2 "Damping for damper 1";
  parameter Damping d2=1.0 "Damping for damper 2";
  Angle phi1 "Angle for inertia 1";
  Angle phi2 "Angle for inertia 2";
  AngularVelocity omega1 "Velocity of inertia 1";
  AngularVelocity omega2 "Velocity of inertia 2";
  initial equation
    phi1 = 0;
    phi2 = 1;
    omega1 = 0;
    omega2 = 0;
  equation
    // Equations for inertia 1
    omega1 = der(phi1);
    J1*der(omega1) = k1*(phi2-phi1)+d1*der(phi2-phi1);
    // Equations for inertia 2
    omega2 = der(phi2);
    J2*der(omega2) = k1*(phi1-phi2)+d1*der(phi1-phi2)-k2*phi2-d2*der(phi2);
end SecondOrderSystem;
```

我们将在新模型中增加一个 `extends` (扩展) 条款，以重用旧的模型。这个语句实质上是导入被扩展模型的一切内容。我们将在之后讨论[包 \(141\)](#) 时进一步介绍 `extends` 条款。现在不妨暂时把这个语句的作用当成把另一个模型的内容直接复制到当前模型内。

记得 `SecondOrderSystem` 模型的解如下：



在此情况下，我们仅仅是把计算出来的解绘制出来了。但在实际系统中，我们不可能直接知道一个轴的旋转速度。相反，我们必须进行测量。不过，测量误差会引入误差。而不同的测量技术则会引入不同类型的误差。在本节中，我们将介绍如何能够模拟各种不同的测量技术。

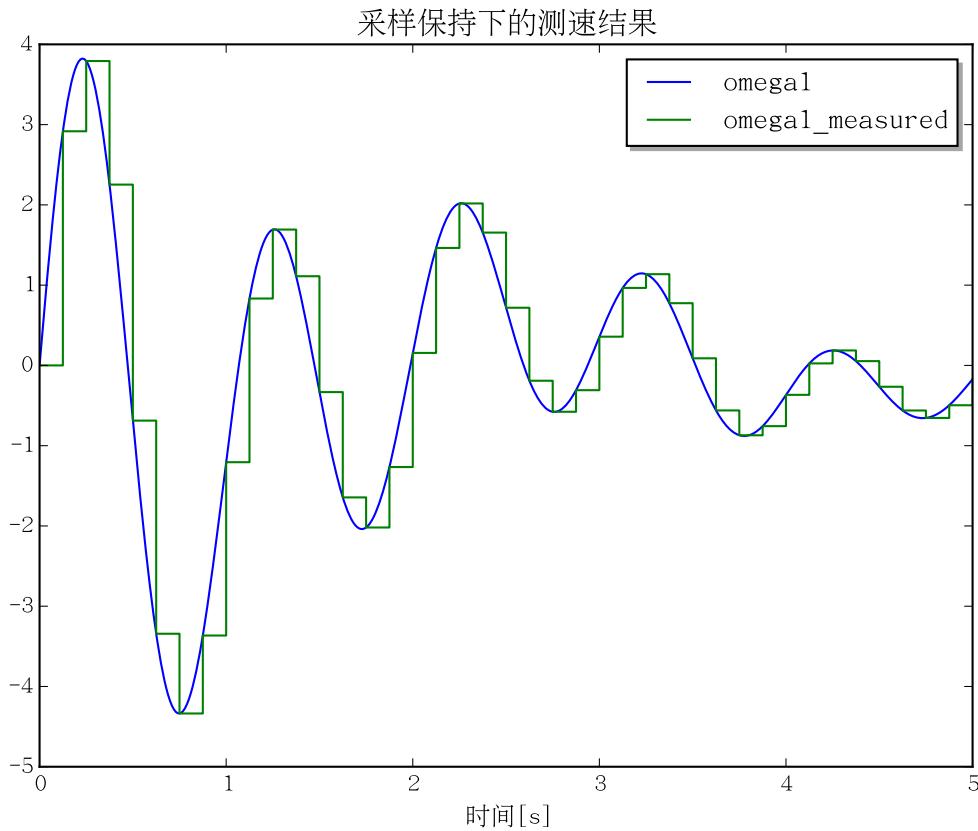
### 取样保持

我们将考察的第一种测量方法是用采样和保持的方法。某些速度传感器具有测量系统旋转速度的电路。而这些传感器并非提供连续的速度值。传感器在给定时间点对速度进行取样，然后将其保存在某个地方。这种方法被成为“取样保持”(sample and hold)。下面的模型展示了如何实现采用采样保持的方法测量角速度  $\omega_1$ :

```
model SampleAndHold "Measure speed and hold"
  extends BasicEquations.RotationalSMD.SecondOrderSystem;
  parameter Real sample_time(unit="s")=0.125;
  discrete Real omega1_measured;
equation
  when sample(0,sample_time) then
    omega1_measured = omega1;
  end when;
end SampleAndHold;
```

请注意，变量  $\omega_1$ \_measured 的声明内有 discrete (离散) 这个限定词。这个特殊的限定词表示指定变量不具有连续的解。相反，这个变量的值（只）会在仿真过程中离散跃变。加入 discrete 关键字并不是必需的，但此关键字有其用处。它可以提供了模型意图的进一步信息，帮助编译器检查模型的正确性。（例如用户不能求该变量的导数。）

现在让我们来研究这个模型生成的解:



值得注意的是，解中测得的值是分段恒定的。这是因为只有当 `when` 被激活时，`omega1_measured` 的值才会被设置。`sample` 函数是一个特殊的内建函数。此函数在其第一个参数（在本例中值为 0）所示时刻首次为真，之后在恒定间隔里所变为真。恒定间隔的长度则有第二个参数决定（本例中为变量 `sample_time`）。

### 间隔测量

在前面的例子中，我们其实没有对速度进行估测，而是直接报告了变量 `omega1` 在特定时刻的值。换句话，在我们在采样时刻得到 `omega1` 的采样值是完全准确的。不过通过“保持”(holding) 测量值（而非继续跟踪 `omega1` 变量），我们便引入了误差。

在余下的例子中，我们重点放在用于估计旋转轴速度的技术。在这些例子中，我们不再会在测量中直接使用实际速度。相反，我们将立足于物理系统所产生的事件，尝试使用这些事件去重建旋转速度的估计值。

这些我们要处理的事件产生自连接在旋转轴的离散元件。例如，一种典型生产这些事件的方法是使用“齿轮编码器”。齿轮编码器包括一个在旋转轴上的齿轮。在齿轮的两侧，我们将放置光源和光传感器。随着齿轮的齿通过在光源的前面，齿轮会挡住光线。其结果是，光传感器的信号将给出一个近似方波信号。这些方波的前缘便是我们需要响应的事件。

我们研究的第一种测算方法是通过测量事件之间经过的时间间隔来计算轴的转速。既然我们知道，每当轴旋转  $\Delta\theta$  角度时这些事件便会发生，我们可以如此测算转速：

$$\hat{\omega} = \frac{\Delta\theta}{\Delta t}$$

这种速度测算方法可用 Modelica 表示为：

```
model IntervalMeasure "Measure interval between teeth"
  extends BasicEquations.RotationalSMD.SecondOrderSystem;
  parameter Integer teeth=200;
```

```

parameter Real tooth_angle(unit="rad")=2*3.14159/teeth;
Real next_phi, prev_phi;
Real last_time;
Real omega1_measured;
initial equation
  next_phi = phi1+tooth_angle;
  prev_phi = phi1-tooth_angle;
  last_time = time;
equation
  when {phi1>=pre(next_phi),phi1<=pre(prev_phi)} then
    omega1_measured = tooth_angle/(time-pre(last_time));
    next_phi = phi1+tooth_angle;
    prev_phi = phi1-tooth_angle;
    last_time = time;
  end when;
end IntervalMeasure;

```

其中 tooth\_angle（齿角）表示  $\Delta\theta$ 。注意 tooth\_angle 并不需要用户指定。用户使用 teeth 参数指定齿轮的齿数。tooth\_angle 参数的值会由参数 teeth 计算（注意，虽然我们在这里手工输入了  $\pi$  的值，我们将在本书的常数（152）中学习如何在以后避免这种情况）。

让我们来仔细看一下这个模型中的 when 语句：

```

equation
  when {phi1>=pre(next_phi),phi1<=pre(prev_phi)} then
    omega1_measured = tooth_angle/(time-pre(last_time));
    next_phi = phi1+tooth_angle;
    prev_phi = phi1-tooth_angle;
    last_time = time;
  end when;

```

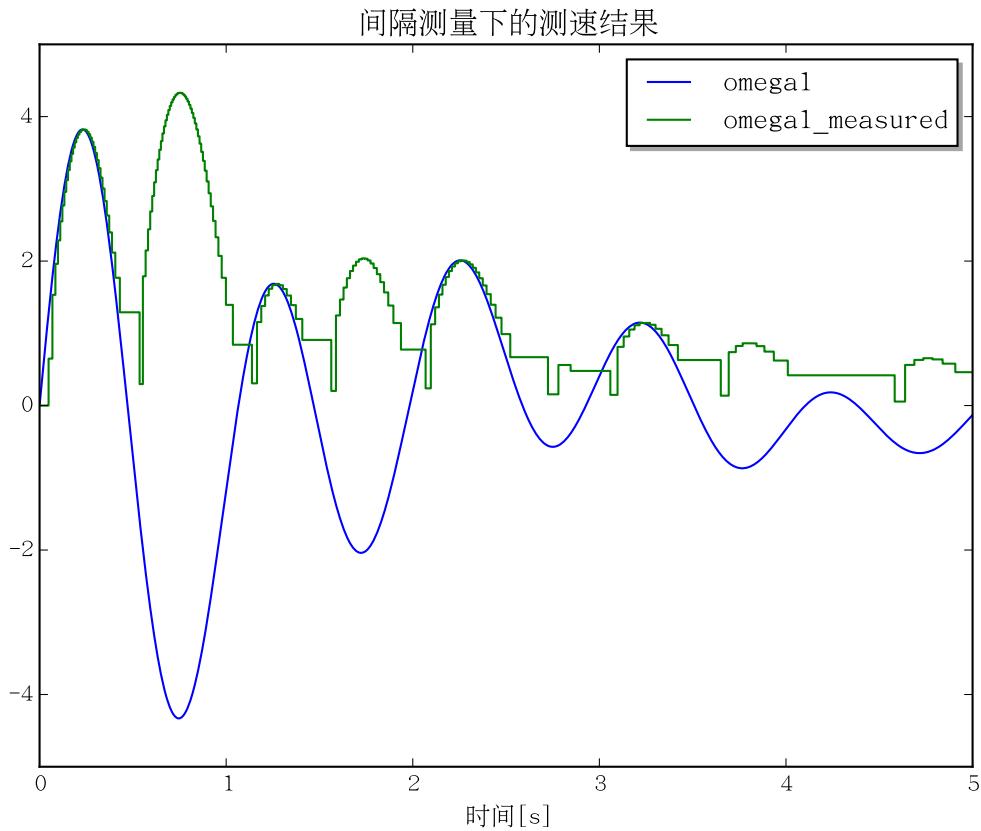
在这里，我们使用了之前在弹跳球（43）例子中用过的矢量表达式语法。请记住，如果其中的任何条件的变为真，那么 when 语句便会被激活。在这里，一旦角度 phi1 大于 next\_phi 或小于 prev\_phi，那么 when 语句便会被激活。

另外需要注意是在整个 when 语句里使用的 pre 操作符。模型内发生事件时，变量值可能会非连续地改变。在一个事件中，当我们试图计算所有变量受事件影响后的取值时，pre 操作符允许我们引用变量在事件之前的价值。这里的 pre 操作符在该模型中的作用是引用 next\_phi、prev\_phi 以及 last\_time 在之前（事件前）的值。由于所有变量都受到在 when 内部语句的影响，因此 pre 操作符是必须的。所以，例如 last\_time（没有 pre 操作符）指代的是 last\_time 在事件结束时的取值。而 pre(last\_time) 指代的则是 last\_time 在任何事件发生前的取值。

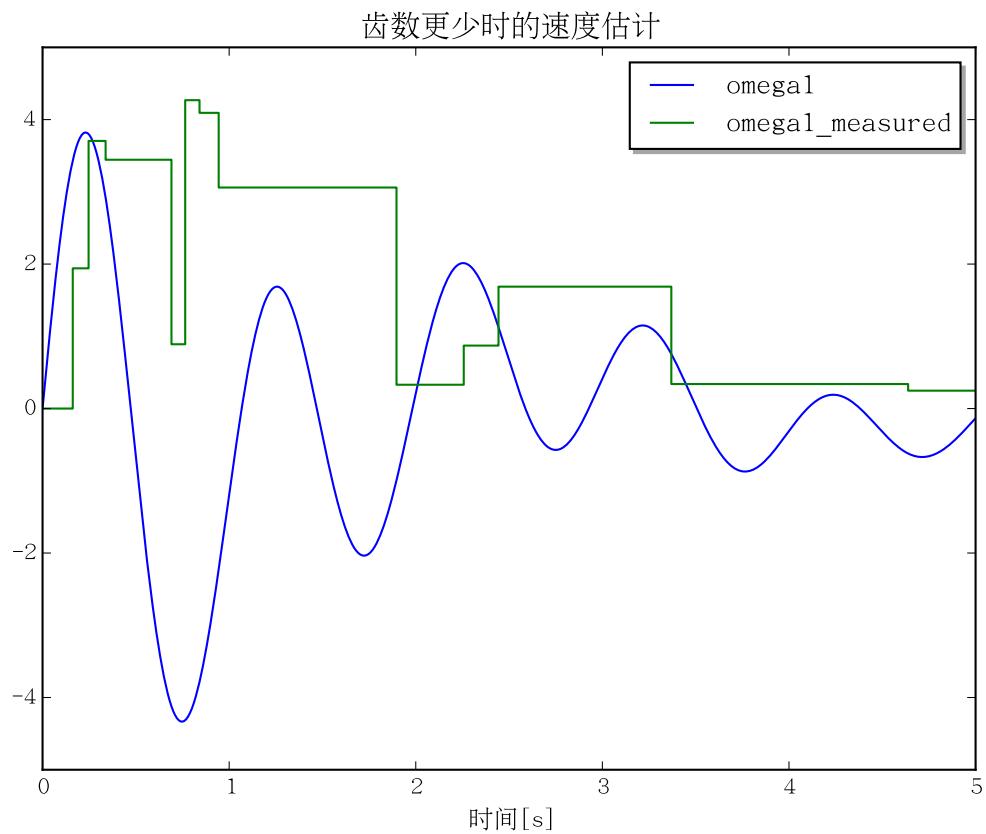
### pre 操作符的使用

一般而言，如果一个变量的变化是由被激活的 when 语句造成的，那么在 when 语句使用的条件表达式与上述会被更改的变量相关联时，你几乎总是需要使用 pre 操作符来指代这个变量（正如我们在前例中做的一样）。这清楚地表明，你是回应 when 语句触发之前系统的状态。

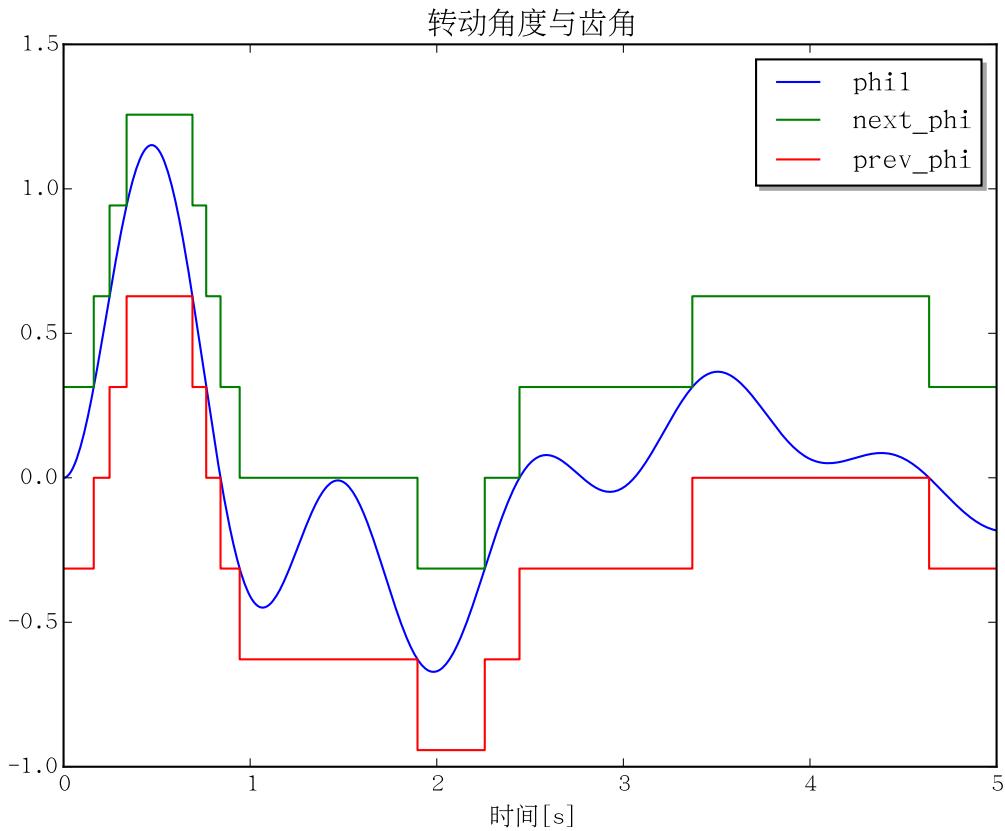
让我们看看用上述方法给出的速度测算值：



从以上结果我们可以马上看到这个测算算法的两个重要特性。首先，测算结果是无符号的。也就是说，我们无法使用类似齿轮编码器的装置得知轴转动的**方向**。其次，低转速和旋转方向的改变可以显著降低测算的准确性。结果对齿轮的齿数也非常敏感。如果我们把编码器中齿的数量即参数 teeth 减少为 20，我们将得到十分不同的结果。



为何测量信号如此地不准确呢？为了明白这一点，大家可以观察下列图表，对比变量  $\text{phi1}$  和下列相邻齿的角位置  $\text{next\_phi}$  以及  $\text{prev\_phi}$ 。



在这个图里，我们可以清楚地看到，较低的转速以及反转都会产生不规则的事件。这会从而引入显著估计误差。

### 脉冲计数

上述时间间隔测量技术需要能在硬件中断时执行速度计算的硬件。另一种估测速度的方法需要对（固定）时间间隔内发生的事件数目进行计数，并以此作为速度的估算值。使用这种方法，在事件发生时仅仅会进行事件总数的求和运算，实际的计算则只会在定期更新时进行。

基于本节前面的例子，我们似乎会很自然地用以下形式在模型里实现这种估测技术：

```
model Counter "Count teeth in a given interval"
  extends BasicEquations.RotationalSMD.SecondOrderSystem;
  parameter Real sample_time(unit="s")=0.125;
  parameter Integer teeth=200;
  parameter Real tooth_angle(unit="rad")=2*3.14159/teeth;
  Real next_phi, prev_phi;
  Integer count;
  Real omega1_measured;
initial equation
  next_phi = phi1+tooth_angle;
  prev_phi = phi1-tooth_angle;
  count = 0;
equation
  when {phi1>=pre(next_phi),phi1<=pre(prev_phi)} then
    next_phi = phi1+tooth_angle;
    prev_phi = phi1-tooth_angle;
    count = pre(count) + 1;
  end when;
  when sample(0,sample_time) then
    omega1_measured = pre(count)*tooth_angle/sample_time;
```

```

count = 0 "Another equation for count?";
end when;
end Counter;

```

然而，此模型有一个问题。注意 count 变量实际上有**两个**方程。如果我们尝试编译这样的模型就会导致方程数量超过变量数量。(也就是说问题是奇异的。)

那么，我们应该如何处理这种情况呢？之所以需要两个不同的方程，是因为在更新 count 时，我们需要对不同的事件作出不同的反应。我们可以尝试用如下形式将需求表达为一个 when 语句,:

```

when {phi1>=pre(next_phi),phi1<=pre(prev_phi),sample(0,sample_time)} then
  if sample(0,sample_time) then
    omega1_measured := pre(count)*tooth_angle/sample_time;
    count :=0;
  else
    next_phi := phi1 + tooth_angle;
    prev_phi := phi1 - tooth_angle;
    count :=pre(count) + 1;
  end if;
end when;

```

但是，这种代码很快就会变得难以阅读。幸运的是，我们可以通过把所有的 when 陈述中放在 algorithm (算法) 区域以解决这个问题。

algorithm 区域的特性是对于任何在其内部进行赋值的变量而言，它都仅被视为是一个等式。譬如说，这个特性让 count 变量可以被多次赋值。使用 algorithm 区域时，读者必须明白赋值的顺序不再无足轻重。如果有冲突出现时（例如在同一个 algorithm 区域内一个变量被赋了两个值），那么只有最后一个会被使用。对于 algorithm 区域我们还需要注意另外一点，就是它并不支持一般意义的等式。相对地，大家必须使用赋值语句。

在这个方面上，algorithm 区域的工作方式与大多数编程语言非常相像。在算法区域的语句是顺序执行的，且每个语句不被解析为等式而是作为给变量赋值的表达式。具有某些编程背景的人们可能会觉得 Modelica 面向公式的一面既迷惑又陌生。对于他们而言，熟悉的赋值语句可能是很诱人的。但需要注意，之所以要避免 algorithm 区域的一大原因是，它会干扰 Modelica 语言编译器进行的符号运算。这既会导致仿真性能不佳，又会让你在建模时丧失灵活性。因此，大家最好尽可能地使用 equation 区域。

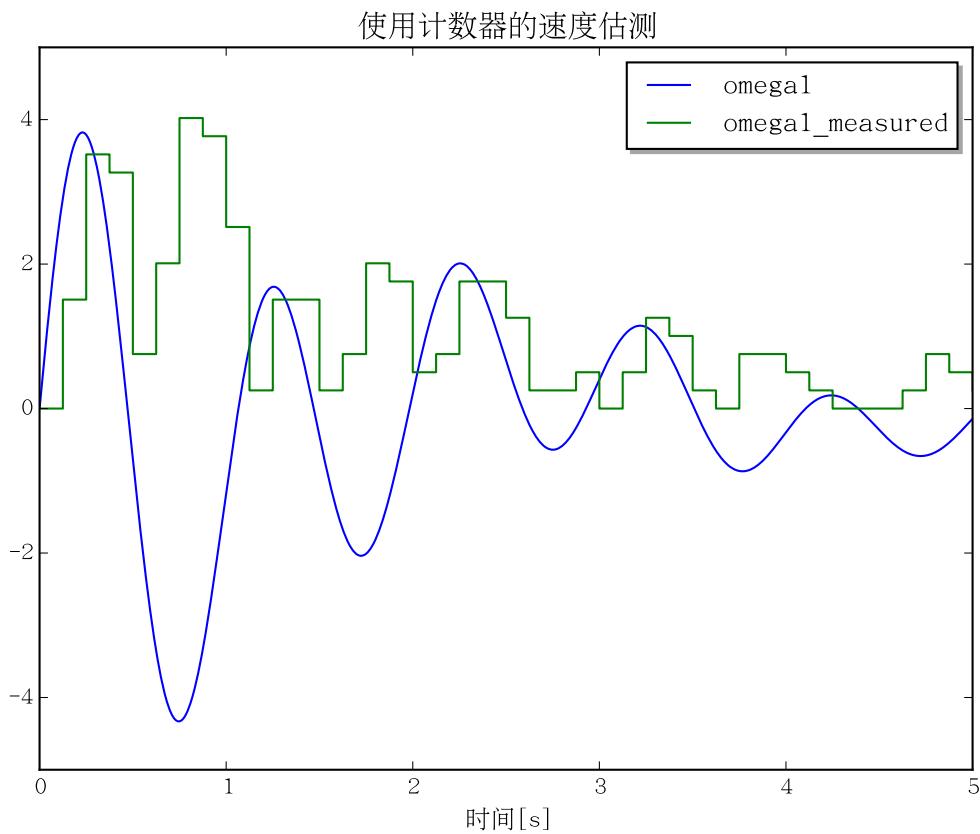
在本例里 algorithm 区域的使用并不会带来显著的影响。以下是将前述测算方法用 algorithm 区域重构后的结果：

```

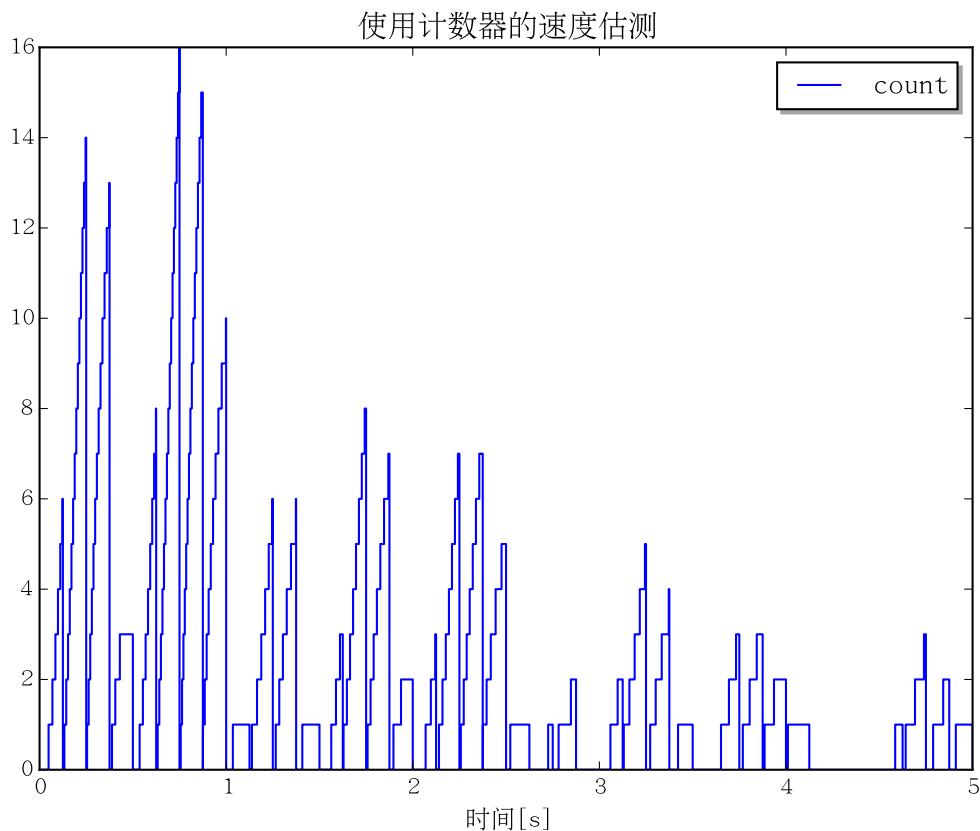
model CounterWithAlgorithm "Count teeth in a given interval using an algorithm"
  extends BasicEquations.RotationalSMD.SecondOrderSystem;
  parameter Real sample_time(unit="s")=0.125;
  parameter Integer teeth=200;
  parameter Real tooth_angle(unit="rad")=2*3.14159/tooth;
  Real next_phi, prev_phi;
  Integer count;
  Real omega1_measured;
initial equation
  next_phi = phi1+tooth_angle;
  prev_phi = phi1-tooth_angle;
  count = 0;
algorithm
  when {phi1>=next_phi,phi1<=pre(prev_phi)} then
    next_phi := phi1 + tooth_angle;
    prev_phi := phi1 - tooth_angle;
    count := pre(count) + 1;
  end when;
  when sample(0,sample_time) then
    omega1_measured := pre(count)*tooth_angle/sample_time;
    count :=0;
  end when;
end CounterWithAlgorithm;

```

测算方法的仿真结果见下图：



同样，我们可以看到这种方法并不能确定旋转的方向。由下图，我们可以对每个采样间隔内发生事件的数量有一个大概的感觉：



在一般情况下，间隔内计数越高，测算结果就越准确。

## 结论

本节说明如何可以通过 when 这一语言特性对发生在系统中的物理事件作出响应。这类事件及其对系统的影响，对比我们之前讨论过的连续时间动力学特性，其实同样重要。由于完整系统通常包括连续的和不连续行为，Modelica 对上述事件的发现和应对能力是其适合用于系统建模的重要原因。。

## 第 2.1.6 节 滞回

在本节中，我们将讨论滞回。这是为了理解某些特定的建模类型所必须的一个重要概念。还记得前面在状态事件的处理 (47) 的讨论中，我们看到了抖动的例子。在那些例子里，我们可以使用 noEvent 运算符来解决抖动的问题。这是因为前述抖动的情况纯粹是由数值噪音产生的。那与行为的突然变化毫无关系。

在本节中，我们将考虑一个稍微极端的例子。请考虑以下模型：

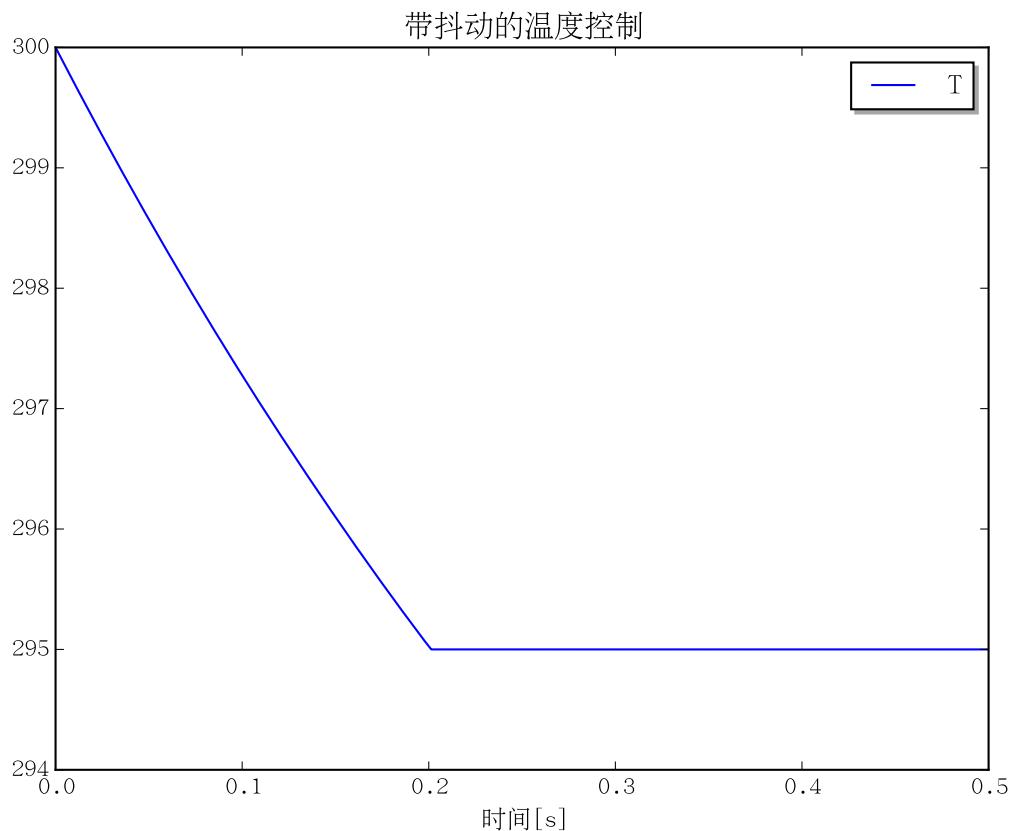
```
model ChatteringControl "A control strategy that will 'chatter'"
  type HeatCapacitance=Real(unit="J/K");
  type Temperature=Real(unit="K");
  type Heat=Real(unit="W");
  type Mass=Real(unit="kg");
  type HeatTransferCoefficient=Real(unit="W/K");
  Boolean heat "Indicates whether heater is on";
  parameter HeatCapacitance C=1.0;
  parameter HeatTransferCoefficient h=2.0;
  parameter Heat Qcapacity=25.0;
  parameter Temperature Tamb=285;
  parameter Temperature Tbar=295;
```

```

Temperature T;
Heat Q;
initial equation
T = Tbar+5;
equation
heat = T < Tbar;
Q = if heat then Qcapacity else 0;
C*der(T) = Q-h*(T-Tamb);
end ChatteringControl;

```

倘若我们对模型进行仿真，便会得到下列结果：



然而，从开始仿真到产生上述结果却需要很长的时间。读者可以通过观察加热器在仿真的输出，以进一步了解性能不佳的原因。

你会看到，在 0.2 秒左右之后，加热器不断地打开和关闭。这发生得如此频繁，以致于你要将图放大多次后才能看到这些转换。大量的状态转换让结果在正常比例下像一个填充满的矩形。

这实际上是在控制系统中的现实问题。如果你仔细观察家里电炉的工作方式便会发现，电炉不会在高于或低于设定的室内温度时就不停地打开关闭。相反，电炉会等到温度变得高于或低于设定温度一定额度后，才会开始作出反应。

围绕着设定温度周围引入的“带”就叫做滞回。ChatteringControl 模型的问题就在于它并没有任何滞回。相反，该模型不停地开关加热器以响应微乎其微的温度变化。

要建模滞回必须考虑一个棘手的问题，也就是滞回是“有状态的”。那么要确定系统的行为我们必须知道系统的历史。因此，我们并不能简单地使用 if 语句。其原因是 if 语句除去考虑系统的当前状态外，并不会考虑别的因素。为了实现滞回，我们需要用到 when 语句。考虑下列模型：

```

model HysteresisControl "A control strategy that doesn't chatter"
type HeatCapacitance=Real(unit="J/K");
type Temperature=Real(unit="K");
type Heat=Real(unit="W");

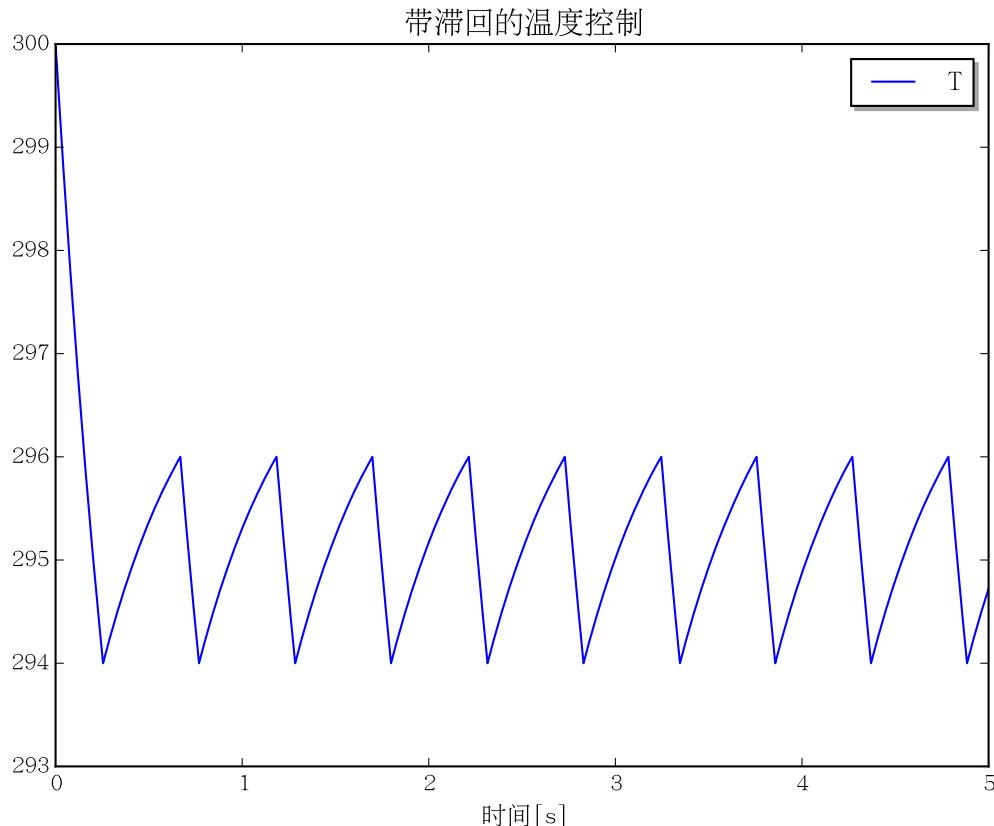
```

```

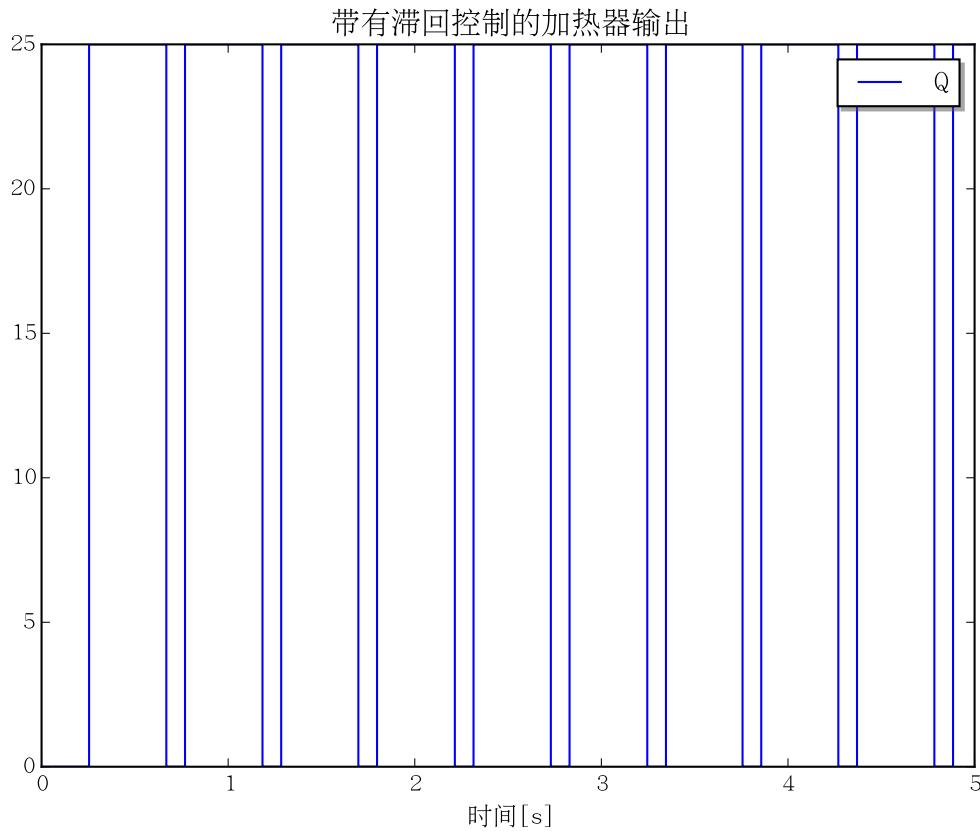
type Mass=Real(unit="kg");
type HeatTransferCoefficient=Real(unit="W/K");
Boolean heat(start=false) "Indicates whether heater is on";
parameter HeatCapacitance C=1.0;
parameter HeatTransferCoefficient h=2.0;
parameter Heat Qcapacity=25.0;
parameter Temperature Tamb=285;
parameter Temperature Tbar=295;
Temperature T;
Heat Q;
initial equation
  T = Tbar+5;
  heat = false;
equation
  Q = if heat then Qcapacity else 0;
  C*der(T) = Q-h*(T-Tamb);
  when {T>Tbar+1,T<Tbar-1} then
    heat = T<Tbar;
  end when;
end HysteresisControl;

```

仔细观察上面的 when 语句，我们可以知道仅当  $T > T_{\text{bar}} + 1$  或者  $T < T_{\text{bar}} - 1$  变为真时，系统才会有响应。请注意，若上述表达式变为假，系统并不会有响应。这就是为何 if 语句在此并不适用。使用 if 语句或者 if 表达式时，只要条件表达式的值发生了变化，系统的行为就会改变。而使用 when 语句时，仅仅当条件为真时，when 语句内的代码才会被激活。如果我们对该模型进行仿真并观察其温度，那么我们会看到温度保持在期望温度的滞环带内。



更重要的是，我们观察系统输出的热量时便会发现，与先前例子不同的是，解热器的开与关之间有些许的时间间隔。



通过使用 algorithm 区域可以让实现滞回的逻辑变得更显然（正如我们前面在速度估测方法（65）的讨论一样）。

```

model HysteresisControlWithAlgorithms "Control using algorithms"
  type HeatCapacitance=Real(unit="J/K");
  type Temperature=Real(unit="K");
  type Heat=Real(unit="W");
  type Mass=Real(unit="kg");
  type HeatTransferCoefficient=Real(unit="W/K");
  Boolean heat "Indicates whether heater is on";
  parameter HeatCapacitance C=1.0;
  parameter HeatTransferCoefficient h=2.0;
  parameter Heat Qcapacity=25.0;
  parameter Temperature Tamb=285;
  parameter Temperature Tbar=295;
  Temperature T;
  Heat Q;
initial equation
  T = Tbar+5;
  heat = false;
equation
  Q = if heat then Qcapacity else 0;
  C*der(T) = Q-h*(T-Tamb);
algorithm
  when T<Tbar-1 then
    heat :=true;
  end when;
  when T>Tbar+1 then
    heat :=false;
  end when;
end HysteresisControlWithAlgorithms;

```

注意这两个条件表达式是如何被分成两个独立的 when 语句的。如此这般，热源开闭的缘由也就显而易见了。由于这两个 when 语句都是对同一个变量 heat 进行赋值，因此两句都是在 algorithm 区域里定义的。

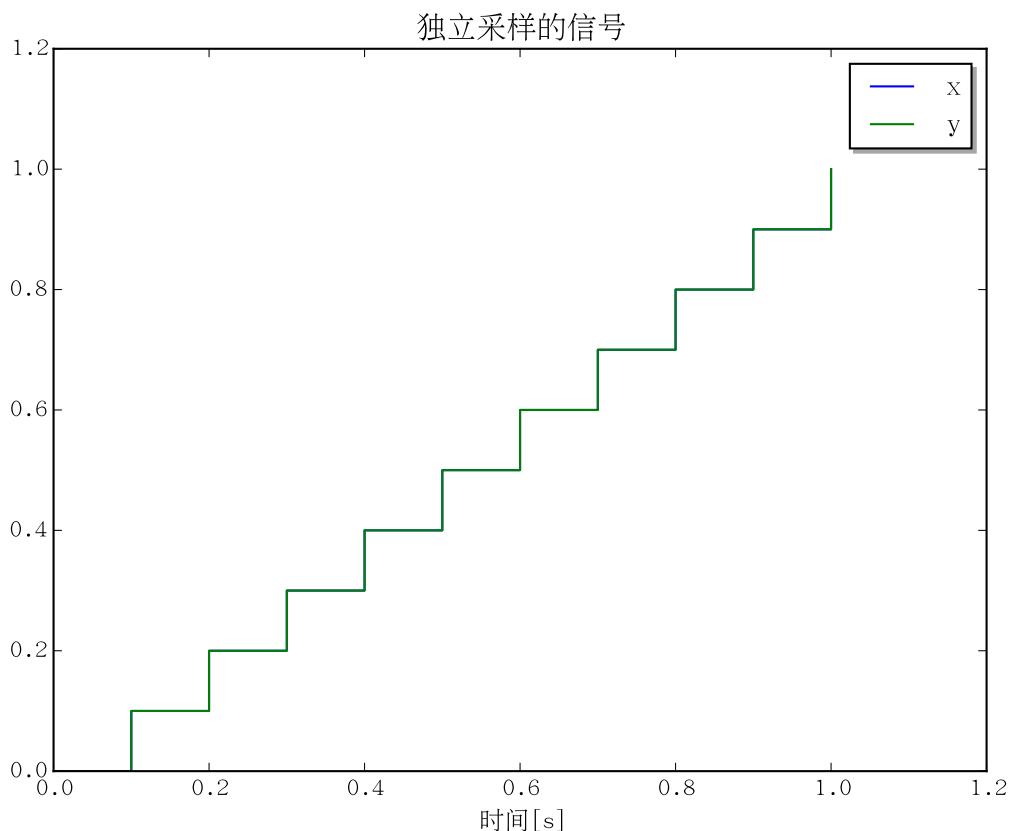
### 第 2.1.7 节 同步系统

Modelica 语言 3.3 版引入了一个新功能以解决有关非确定性离散行为的问题[Elmqvist] ( 331 )。本节将先介绍这些问题在 3.3 版本之前的表现。然后，我们将用例子展示这些新特性能如何帮助解决这类问题。先考虑下列模型：

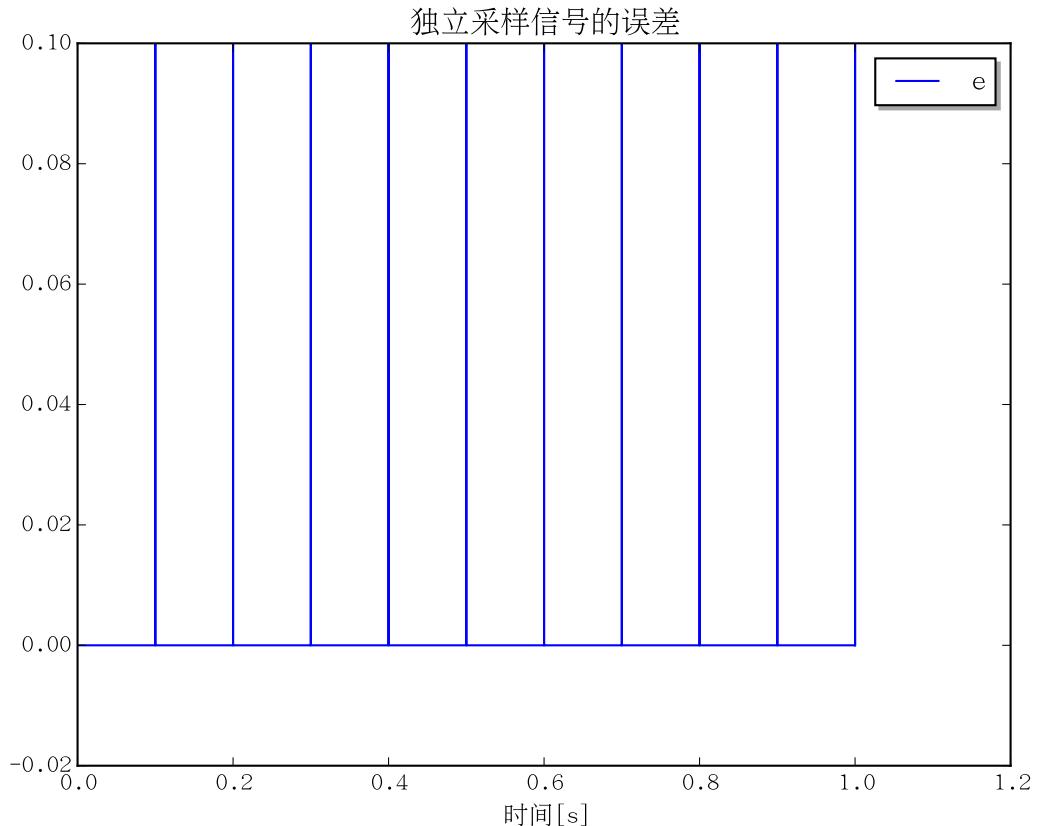
```
model IndependentSampling "Sampling independently"
  Real x "Sampled at 10Hz via one method";
  Real y "Sampled at 10Hz via another method";
  Real e "Error between x and y";
  Real next_time "Next sample for y";
equation
  when sample(0,0.1) then
    x = time;
  end when;

  when {initial(), time>pre(next_time)} then
    y = time;
    next_time = pre(next_time)+0.1;
  end when;
  e = x-y;
end IndependentSampling;
```

倘若你仔细观察，你会发现 x 和 y 都在离散的时间点上被计算。此外，两变量的采样时点都是在仿真开始时以及之后的每 0.1 秒。但问题是，它们真的一样吗？为了更容易解决这个问题，我们加入变量 e 以计算两者间的差。



对模型进行仿真，我们会得到如下的 x、y 轨迹。当然，两个轨迹看上去是一样的。但要真正确定它们之间是否存在任何不同，我们绘制了误差值 e：



现在，让我们考虑一下模型：

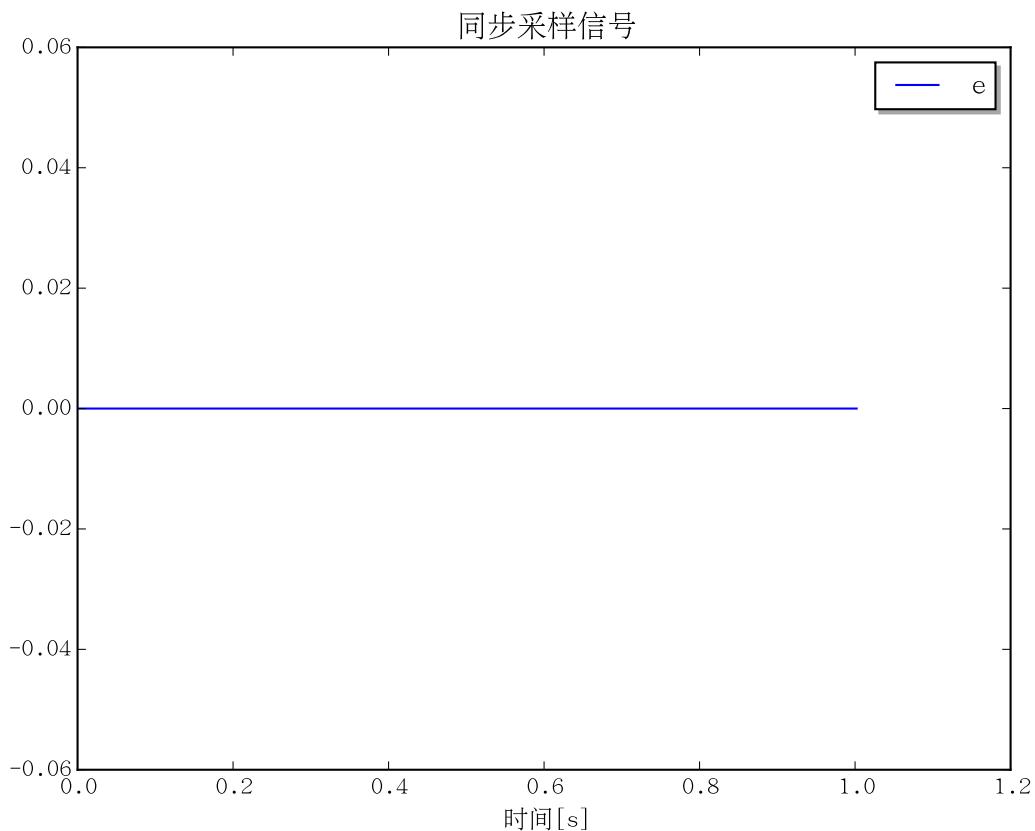
```
model SynchronizedSampling "A simple way to synchronize sampling"
  Integer tick "A clock counter";
  Real x, y;
  Real e "Error between x and y";
equation
  when sample(0,0.1) then
    tick = pre(tick)+1;
  end when;

  when change(tick) then
    x = time;
  end when;

  when change(tick) then
    y = time;
  end when;

  e = x-y;
end SynchronizedSampling;
```

在这里，我们设置了一个共同的信号以触发两个变量的赋值操作。通过这种方式，我们可以确定，当 tick 信号变为真时，无论是 x 还是 y 都将被赋予一个值。显然，运行这个模型后我们可以看到，误差永远为零：



在这样的做法里，每个信号都是基于一个共同的“节拍”（或时钟）进行取样的。这是一个很好的避免确定性问题的方式。但是，如果你有两个不同频率的信号，且已知在特定的时点两者会被同时取样，那么情况又是如何呢？请考虑如下例子：

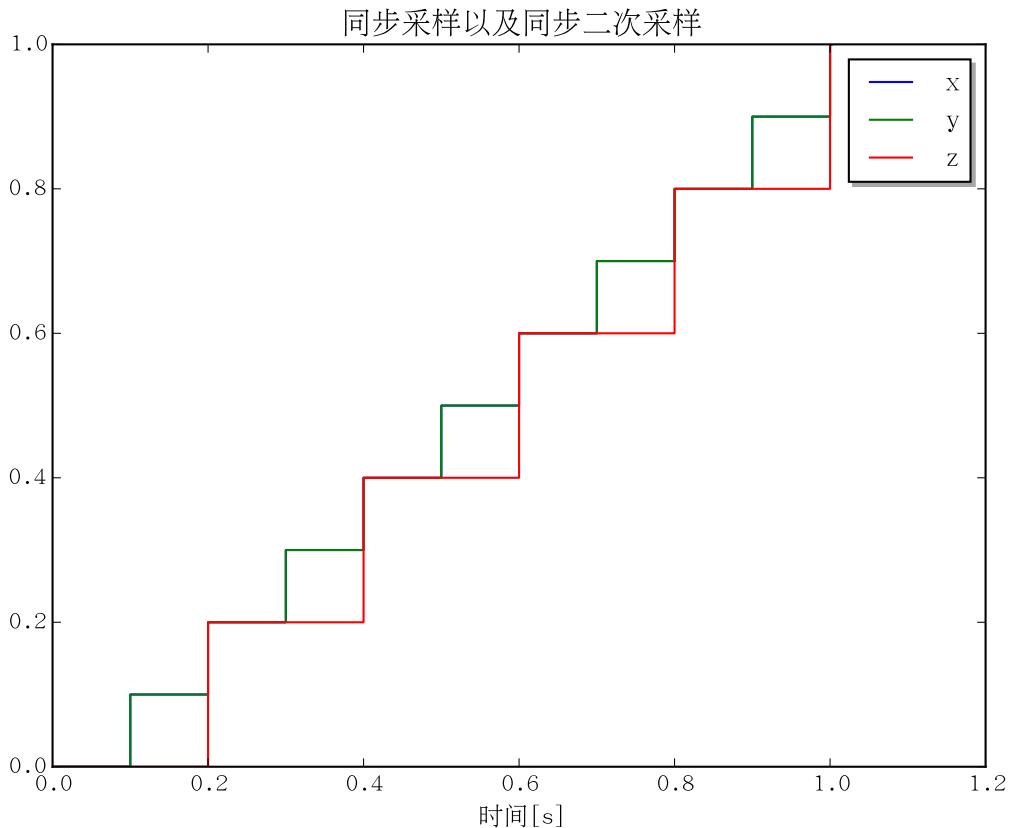
```
model SubsamplingWithIntegers "Use integers to implement subsampling"
  Integer tick "Clock counter";
  Real x, y, z;
equation
  when sample(0,0.1) then
    tick = pre(tick)+1;
  end when;

  when change(tick) then
    x = time;
  end when;

  when change(tick) then
    y = time;
  end when;

  when mod(tick-1,2)==0 then
    z = time;
  end when;
end SubsamplingWithIntegers;
```

在这种情况下，变量 `tick` 是计数器。每次此变量改变时，我们会更新 `x` 和 `y` 的值。所以，直到这一点都是与先前的模型相同的。然而，我们增加了第三个信号 `z`。而该信号仅在当 `tick` 值是为奇时才会进行采样。所以相比之下，`x` 和 `y` 的采样次数为其的两倍。但我们可以肯定的是，每次 `z` 更新时，`x` 和 `y` 是在完全相同的时点进行更新。对模型进行仿真我们会得到一下结果：



这是 Modelica 语言在 3.3 版本之前所采用的方法。但 3.3 版本引入了一些新的功能以帮助我们表述这些情况。

请考虑以下模型：

```
model SamplingWithClocks "Using clocks to sub and super sample"
  Real x, y, z, w;
equation
  x = sample(time, Clock(1,10));
  y = sample(time, Clock(1,10));
  z = subSample(x, 2);
  w = superSample(x, 3);
end SamplingWithClocks;
```

现在我们不再使用 when 语句。相反，我们会使用一个增强版的 sample（取样）函数。这个 sample 函数的首个参数是待取样的表达式，第二个参数则是取样的时间间隔。让我们逐行阅读并讨论下面的代码。首先，我们有：

```
x = sample(time, Clock(1,10));
```

注意，我们已经不再使用 0.1，而且不再能看到表示为实数值的时钟间隔。相反，我们使用 Clock 表达式为 x 定义一个有理数值的时钟间隔。这一点很重要，因为这种做法允许我们对不同时钟进行精确比较。让我们看下一行：

```
y = sample(time, Clock(1,10));
```

再一次，我们看到在时钟间隔的有理表示。这意味着在实践中，Modelica 语言编译器可以确信这两个时钟 x 与 y 是相同的。因为两者均是用可确切比较的整数量进行定义的。也就是说，在执行仿真时，我们可以确信这两个时钟会同时触发。

如果我们想创建一个正好是 x 两倍慢的时钟，那么我们可以使用 subSample 运算符来完成。在 z 的定义里我们可以看到这种情况：

```
z = subSample(x, 2);
```

Modelica 语言编译器可以在幕后对这些时钟间的关系进行推理。编译器知道每隔  $\frac{1}{10}$  秒时钟 x 就会触发。因此 Modelica 语言编译器可以使用 subSample 运算符所提供的信息推论出，每隔  $\frac{2}{10}$  秒时钟 z 就会触发。理论上，这意味着 z 也可以被定义为：

```
z = sample(time, Clock(2,10));
```

但通过使用 subSample 运算符相对 x 去进行对 z 的定义，我们可以确保无论 x 是如何定义的，z 的触发频率总是 x 的一半。

类似地，我们可以使用 superSample 运算符定义另一个触发频率为 x3 倍的时钟 w。

```
w = superSample(x, 3);
```

同样，我们可以直接使用 sample 以对 w 进行如下定义：

```
w = sample(time, Clock(1,30));
```

但使用 superSample 我们可以确保 w 的采样速度总是 x 的三倍、z 的六倍（因为 z 也是基于 x 定义的）。

Modelica 语言的同步时钟特性是相对较新。因此，并非所有的 Modelica 语言编译器都支持这些特性。要了解更多有关这些同步功能及其应用，可以参考[Elmqvist] ( 331) 以及／或者 3.3 版以后的 Modelica 规范。

## 第 2.2 节 回顾

### 第 2.2.1 节 事件

在关于 [基本方程](#) ( 3) 的第一章里，我们看到了描述连续行为的例子。在该章中介绍的方程在任何时候都有效，而且这些方程的解总是连续的。在本章中，我们讨论了 Modelica 语言用以描述离散行为的种种方法。事件导致所有 Modelica 内离散行为的根本原因。

#### 条件表达式

事件的产生有以下两种可能。首先，时间可以用条件语句产生。在本章的前面几例，我们看到了条件表达式可以触发事件。倘若这些条件表达式仅仅涉及变量 time，我们就将其成为“时间事件”。变量 time 是一个内建的全局变量，而且是所有模型的“输入”。

如果事件是由涉及解内变量的条件表达式产生的，我们就将其成为“状态事件”。时间事件和状态时间的重要区别在本章内的 [第一个例子](#) ( 39) 以及 [第二个例子](#) ( 43) 中分别作了讨论。

创建条件表达式可以使用关系运算符 ( $>$ 、 $>=$ 、 $<$ 、 $<=$ 、 $=$ ) 以及逻辑运算符 (not、and、or)。而正如我们在 [事件的抑制](#) ( 50) 的讨论结果，我们可以在条件表达式外加上 noEvent 运算符以抑制其事件的生成。

通常情况下，产生事件的条件表达式出现在一个 if 语句或 if 表达式的里面。但也应注意的是，即使是简单的变量赋值，例如：

```
Boolean late;
equation
  late = time >= 5.0 "This will generate an event";
```

也可以触发事件的产生。

#### 分段构造

处理条件表达式时有一种特殊情况。在某些情况下，条件表达式是一种有效的创建构造分段表达式的方法。例如：

```
x = if (x<0) then 0 else x^2;
```

Modelica 编译器很难可靠地确定函数是否连续，或者函数是否有连续导数。出于这个原因，Modelica 自带了 smooth 操作符以明示这种情况。例如，用以下形式使用 smooth 运算符：

```
x = smooth(if (x<0) then 0 else x^2, 2);
```

表示该表达式是连续的，而且在两次微分后仍将保持连续。当然，在本例里，表达式是无论进行多少次微分仍然是连续的，只是 smooth 需要指定一个上界而已。

## 事件与函数

除了由条件表达式生成，事件还可以通过 Modelica 内的某些函数产生。

### 事件生成的函数

以下是当其返回值不连续时就会产生事件的函数的列表。

函数	描述
div(x,y)	省去小数部分的代数商。
mod(x,y)	x/y 的模数
rem(x,y)	代数除法的余数
ceil(x)	不小于 x 的最小整数
floor(x)	不大于 x 的最大整数 (返回 Real)
integer(x)	不大于 x 的最大整数 (返回 Integer)
initial()	初始化时为 true，否则为 false
terminal()	仿真结束时为 true，否则为 false
sample(t0,dt)	在 t0 时刻以及以后的每 dt 秒生成一个事件
edge(x)	仅在 x 变为 true 的一瞬间为 true
change(x)	每当 x 改变时为 true

### 不会产生事件的函数

以下是不产生事件的函数的列表。

函数	描述
abs(x)	x 的绝对值
sign(x)	x 的符号 (返回 -1、0 或 1 )
sqrt(x)	x 的平方根
min(x,y)	x 和 y 中的最小值
max(x,y)	x 和 y 中的最大值

### 与时间有关的操作符

下面的操作符提供了用于产生事件的信号的特殊信息：

函数	描述
pre(x)	在事件的发生时保存了 x 在事件前的取值
previous(x)	在一个时钟节拍内保存了 x 在上个时钟节拍的取值
hold(x)	任何时间都保存了 x 在上个时钟节拍的取值
sample(expr,clock)	在一个时钟节拍内 expr 的取值
noEvent(expr)	抑制 expr 内产生的事件
smooth(expr,p)	表示 expr 至少 p 次可导

### 有关时钟的运算符

下面的运算符用于创建取样时钟（在固定间隔内触发的事件产生器）：

函数	描述
Clock(i,r)	每隔 $\frac{i}{r}$ 秒触发的时钟。其中 i 与 r 都是 Integer 类型的。
Clock(dt)	每隔 dt 秒触发的时钟。其中 dt 是 Real。
subSample(u,s)	采样速度为 u 采样速度 $\frac{1}{s}$ 倍的时钟，其中 s 为 Integer。
superSample(u,s)	采样速度为 u 采样速度 s 倍的时钟，其中 s 为 Integer。

注意 Clock 的构造函数被重载了（也就是说可以采用不同类型的参数）。值得重申的是，Modelica 语言的同步时钟特性是相对较新。因此，并非所有的 Modelica 语言编译器都支持这些特性。要了解更多有关这些同步功能及其应用，可以参考[Elmqvist] ( 331) 以及／或者 3.3 版以后的 Modelica 规范。

### 第 2.2.2 节 If 语句及 if 表达式

虽然其十分直观的，但我们仍然值得简短介绍一下 if 语句以及 if 表达式的语法。因为 if 表达式解释起来最简单，所以让我们先从它开始介绍。if 表达式有以下形式：

```
if cexpr then expr1 else expr2;
```

其中 cexpr 是条件表达式（其结果是一个 Boolean 值。）。expr1 则是 if 表达式在 cexpr 结果为 true 的取值。相对地，expr2 则是 if 表达式在 cexpr 结果为 false 的取值。

if 语句有如下的一般语法：

```
if cond1 then
  // Statements used if cond1==true
elseif cond2 then
  // Statements used if cond1==false and cond2==true
// ...
elseif condn then
  // Statements used if all previous conditions are false
  // and condn==true
else
  // Statements used otherwise
end if;
```

重要的是要注意，当一个 if 语句出现在 equation 区域里，无论在 if 语句的哪个分支里（对于有 elseif 的情况也适用），方程的数量均必须一致。一个例外是在 initial equation 或 initial algorithm 内的 if 乃至不需要 else 子句。因为在这些情况下，分支内的方程数量不需要相等。另外一个值得注意的例外是，在函数 ( 107) 内使用的 if 同样也没有要求方程的数目是在两个分支上相同

Note: 请注意，if 语句以及 if 表达式内的条件表达式有可能会产生事件 ( 76)。

### 第 2.2.3 节 When 语句

通过使用 when，我们可以表达我们所感兴趣的条件判断以及对这些条件判断的回应。在本节中，我们将回顾 when 语句背后的主要思想。when 语句的一般形式如下：

```
when expr then
  // Statements
end when;
```

#### 对比 if 和 when

在前面对滞回 ( 68) 的讨论里，我们简要地讨论了 if 语句和 when 语句之间的区别。在 when 语句内的代码仅会在触发条件表达式为真的一瞬间被激活。在所有其它的时候，when 语句不会有任何影响。而 if

陈述或 if 表达式只要在条件表达式为真时，它们就有效。倘若 if 陈述或 if 表达式包含了 else 子句，那么总会有一个分支有效。

## 表达式

大多数时候，expr 表达式会是个条件表达式，而且通常会涉及关系运算符。when 语句常用的条件表达式有例如 `time>=2.0`、`x>=y+2`、`phi<=prev_phi` 等。回忆前面在讨论间隔测量 (61) 测算算法时，对于同时出现在 expr 以及 when 表达式里的变量，你几乎总要在给这些变量加上 pre 操作符。

在弹跳球 (43) 的例子里，我们遇到过 expr 并非有一个（标量）条件表达式，而是由条件表达式向量组成的情况。请回忆前面关于有向量形式条件语句的 when 语句的讨论：当向量中的任何一个条件变为真时，when 语句就会被激活。

## 语句

when 语句的作用是为变量定义新的取值。我们可以用两种方法定义新的取值。第一种方法是通过将具有以下形式的公式：

```
var = expr;
```

在这种情况下，var 将会等于 expr 的取值。而 expr 里的 pre 操作符用于指代变量在事件前的取值。任何以这种方式赋值的变量均为离散变量。这意味着，这些变量的值仅在事件进行时发生变化。换句话说，变量将是分段常数函数。请注意，以这种方式赋值的变量不可能在仿真的任何时间间隔内都保持连续。

虽然严格而言并非必须，但如果你想明确地将变量标记为离散，你可以用 discrete 限定词作为其前缀（正如我们在本章前面的取样保持 (60) 所看到的例子一样）。添加 discrete 限定词可以确保该变量的值必须由 when 语句来确定。

另一种在 when 变量里为变量赋值的方法，正如在弹跳球 (43) 的例子一样，是通过使用 reinit 操作符。在这种情况下，when 语句内部的代码将会有如下形式：

```
reinit(var, expr);
```

在使用 reinit 操作符时，变量 var 必须是一个状态。换句话说，此变量必须是微分方程求解的结果。在这样的变量里使用 reinit 会停止积分过程并改变该状态（以及其它在同一个 when 语句里添加了 reinit 的）的值。紧接着，积分实际上使用了一套新的初始条件重新开始。而没有使用 reinit 操作符进行重新初始化的其他状态会保持不变。

## algorithm 区域

最后要注意 when 语句如何与 Modelica 的“单赋值规则”产生关系。Modelica 规范中的这条规则规定了每一个变量的值都正好对应一个求值的方程。正如速度的测量 (59) 以及滞回 (68) 小节里提到的，有时我们需要用多个赋值语句来描述系统行为（或者说这样可以表达得更为清晰）。在这些情况下，如果所有的赋值语句都被只被放在 algorithm 区域，那么这些语句实际上会被看成是一个等式。但是，这样做会减弱编译器执行符号运算的能力，结果可能至少会影响仿真的性能和模型的可重用性中的一个。

值得注意的是，如果在初始化期间需要 algorithm 区域这种语义，Modelica 包含了一种 initial algorithm 区域来实现这种功能。initial algorithm 区域的作用可以类比前面在初始化 (31) 讨论过的 initial equation 区域。正如 initial equation 一样，initial algorithm 只会在初始化阶段确定初始化条件时有效，但这种区域支持多次赋值。前述关于符号操作的警告仍然有效。



## 向量与数组

### 第 3.1 节 示例

#### 第 3.1.1 节 状态空间

##### ABCD 形式

回顾前面关于常微分方程 (32) 的讨论，我们可将微分方程表述为如下形式：

$$\begin{aligned}\dot{\vec{x}}(t) &= \vec{f}(\vec{x}(t), \vec{u}(t), t) \\ \vec{y}(t) &= \vec{g}(\vec{x}(t), \vec{u}(t), t)\end{aligned}$$

在这种形式下， $x$  表示系统中的状态， $u$  表示系统的任何外部指定的输入，而  $y$  表示该系统的输出（亦即可能并非状态的变量，但这些变量可通过状态和输入的值求得。）

这些方程有一个有趣的特例。这个特例发生在是当函数  $\vec{f}$  以及  $\vec{g}$  和向量  $\vec{x}$  还有  $\vec{u}$  线性相关时。在这种情况下，方程可以改写为：

$$\begin{aligned}\dot{\vec{x}}(t) &= A(t)\vec{x}(t) + B(t)\vec{u}(t) \\ \vec{y}(t) &= C(t)\vec{x}(t) + D(t)\vec{u}(t)\end{aligned}$$

此问题中的矩阵是所谓“ABCD”矩阵。“ABCD”形式是很有用的，因为如果系统用这种形式表达的，那么我们就可以在系统上进行一些有趣的计算。例如，我们可以使用  $A$  矩阵计算系统的自然频率。而使用这些矩阵的不同组合，我们可以确定对系统控制相关的几个极其重要的性质（例如可观察性和可控性）。

请注意，ABCD 形式允许这些矩阵随时间变化。有一种稍为更专门的形式，除去线性以外，也不随时间变化：

$$\begin{aligned}\dot{\vec{x}}(t) &= A\vec{x}(t) + B\vec{u}(t) \\ \vec{y}(t) &= C\vec{x}(t) + D\vec{u}(t)\end{aligned}$$

这种形式通常被称 LTI（线性时不变）形式。LTI 形式是很重要的。因为这种形式除了具有与 ABCD 的形式一样的特殊性质外，还可以以一个非常简单的形式实现“模型交换”。以前，当用户（采用手工方法或者用建模软件）推导给定系统的行为方程后，其中一个将这些方程导入其他工具的方法，就是把方程变为 LTI 形式。这意味着，该模型可以用带有数字或表达式的一系列矩阵进行交换、共享或者发布。如今，新技术如 Modelica 语言和 FMI<sup>1</sup> 为模型交换提供了更好的选择。

##### LTI 模型

如果有人提供了我们一个 LTI 形式的模型，我们要如何用 Modelica 描述它呢？这是其中一种可能的方法：

<sup>1</sup><http://fmi-standard.org>

```

model LTI
  "Equations written in ABCD form where matrices are also time-invariant"
  parameter Integer nx=0 "Number of states";
  parameter Integer nu=0 "Number of inputs";
  parameter Integer ny=0 "Number of outputs";
  parameter Real A[nx,nx]=fill(0,nx,nx);
  parameter Real B[nx,nu]=fill(0,nx,nu);
  parameter Real C[ny,nx]=fill(0,ny,nx);
  parameter Real D[ny,nu]=fill(0,ny,nu);
  parameter Real x0[nx]=fill(0,nx) "Initial conditions";
  Real x[nx] "State vector";
  Real u[nu] "Input vector";
  Real y[ny] "Output vector";
initial equation
  x = x0 "Specify initial conditions";
equation
  der(x) = A*x+B*u;
  y = C*x+D*u;
end LTI;

```

第一步，是在模型里声明 nx、nu、ny 等参数。这些参数分别代表了状态、输入和输出的数量。然后，我们定义矩阵 A、B、C 以及 D。因为我们正在创建一个线性时不变表示下的模型，所以所有这些矩阵都可以被定义为参数。由于矩阵 A、B、C 和 D 的定义紧接着 [ 以及 ]，因此我们知道它们均是数组。而由于 [] 内有标示了两个维度，我们知道以上的数组均是矩阵。最后，我们看到 x0、x、u 以及 y 的变量声明。这些变量也都是数组。不过，由于这些变量仅有一维，因此它们都是向量。

本模型的另一个特点是其参数均有默认值。对于 nx、nu、ny 等参数，默认的假设是状态、输入和输出的数量均为零。而对于矩阵则默认其元素均为零。除非另有规定，对于初始条件我们同样假设所有的状态在仿真开始取零值。我们将很快看到，这些假设如何可以让我们通过直接覆盖参数值来写编写简单的模型。

## 向量方程

该模型的其余部分现在看起来应该非常熟悉了。必须指出，模型中的方程皆是矢量方程。Modelica 语言中方程可以包括标量或数组。对于矢量方程唯一的要求是，方程的两侧需要有相同的维数和以及每个维度大小均相等。因此，在本例的 LTI 模型里，我们有以下的初始化方程：

```

initial equation
  x = x0 "Specify initial conditions";

```

这个方程是一个向量方程，内容是 x 的每一个元素在仿真开始时等于其在 x0 的对应元素值。实际上，这些向量中的每组对应元素会自动展开为一系列的标量方程。

还有另外一点有助于保持这些方程可读性。Modelica 语言包含了关于函数向量化（103）的一些特殊的规则。概括地说，这些规则规定了，倘若你有一个函数可以对标量进行运算，那么你就可以立刻用这个函数可以进行向量运算。如果你尝试用该函数进行向量运算，Modelica 语言会自动将函数应用在向量中的每个元素上。因此，LTI 模型内的 der(x) 表达式就是一个表示 x 中每个元素微分的向量。

最后，许多代数运算符（如：+ 以及 \*）在应用于向量或矩阵时有着特殊的意义。运算符的定义设计得与常规的数学符号对应。所以，在 LTI 模型里，表达式 A\*x 对应了矩阵与向量的积。

## LTI 例子

考虑到这一点的所有，让我们重新审视我们几个以前的例子，看看他们如何能在长期激励形式表示使用我们的 LTI 模型。讨论了上述的内容后，让我们重新检视前述的数个例子。目的是看看这些可以如何使用 LTI 模型表示为 LTI 形式。注意，我们会再次使用继承（通过使用 extends 关键词）以重用 LTI 模型的代码。

让我们从前面介绍的简单的一阶系统（3）开始。使用 LTI 模型，我们可以将模型重写为：

```

model FirstOrder "Represent der(x) = 1-x"
  extends LTI(nx=1,nu=1,A=[-1], B=[1]);
equation
  u = {1};
end FirstOrder;

```

在我们扩展继承 LTI 时，我们只需要指定与默认值不同的参数即可。在这里，我们指定模型有一个状态和一个输入。然后，我们定义 A 和 B 为  $1 \times 1$  矩阵。最后，由于有一个输入，我们需要为这个输入提供一个方程。该输入一般而言可以随时间变化，因此我们不把它表示为参数，而表示为方程。请注意，在方程中有：

```
u = {1};
```

表达式 {1} 是一个向量源代码文本。这表示，我们用其元素组成的列表来构建向量。在这里，向量仅有一个元素 1。但我们建立一个用逗号分隔开的表达式列表去创建较长的向量，例如：

```
v = {1, 2, 3*4, 5*sin(time)};
```

值得一提的是，我们在 extends 语句内除了设置参数值，也可以包含等式。因此，我们可以完全避免 equation 区域，而将模型简化为：

```

model FirstOrder_Compact "Represent der(x) = 1-x"
  extends LTI(nx=1,nu=1,A=[-1], B=[1], u={1});
end FirstOrder_Compact;

```

一般来说，加入 equation 区域可以使代码有点更具可读性。但也有一些情况下，向 extends 语句加入等式作为对模型的修改会更为方便。

现在，让我们来关注也是前面讨论过的冷却模型 (7)。我们可以把模型用 LTI 形式改写如下：

```

model NewtonCooling "NewtonCooling model in state space form"
  parameter Real T_inf=27.5 "Ambient temperature";
  parameter Real T0=20 "Initial temperature";
  parameter Real hA=0.7 "Convective cooling coefficient * area";
  parameter Real m=0.1 "Mass of thermal capacitance";
  parameter Real c_p=1.2 "Specific heat";
  extends LTI(nx=1,nu=1,A=[-hA/(m*c_p)],B=[hA/(m*c_p)],x0={20});
equation
  u = {T_inf};
end NewtonCooling;

```

这个模型非常类似于前一个。然而，在这种情况下我们并不是把数字直接输入矩阵里。相反，我们用输入带有参数 m、c\_p 等等的表达式。这样的话，当这些物理参数改变时，矩阵 A 和 B 的值也会相应改变。

我们可以采取类似的做法把前面的机械示例 (13) 改写为 LTI 形式。

```

model RotationalSMD
  "State space version of a rotational spring-mass-damper system"
  parameter Real J1=0.4;
  parameter Real J2=1.0;
  parameter Real k1=11;
  parameter Real k2=5;
  parameter Real d1=0.2;
  parameter Real d2=1.0;
  extends LTI(nx=4, nu=0, ny=0, x0={0, 1, 0, 0},
             A=[0, 0, 1, 0;
                 0, 0, 0, 1;
                 -k1/J1, k1/J1, -d1/J1, d1/J1;
                 k1/J2, -k1/J2-k2/J2, d1/J2, -d1/J2-d2/J2]);
equation
  u = fill(0, 0);
end RotationalSMD;

```

同样，我们从物理参数得到 A 的值。在本例里要注意 A 的构造。数学上，矩阵 A 被定义为：

$$A = \begin{vmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ -\frac{k_1}{J_1} & \frac{k_1}{J_1} & -\frac{d_1}{J_1} & \frac{d_1}{J_1} \\ \frac{k_1}{J_2} & -\frac{k_1}{J_2} - \frac{k_2}{J_2} & -\frac{d_1}{J_2} & -\frac{d_1}{J_2} - \frac{d_2}{J_2} \end{vmatrix}$$

在构造 A 时，我们可以注意到其前两列可以更容易表示为一个零矩阵以及一个单位矩阵的组合。换句话说，将矩阵表示为子矩阵的组合可能更为清晰，即：

$$A = \left| \begin{array}{c|cc|cc} & \begin{vmatrix} 0 & 0 \\ 0 & 0 \end{vmatrix} & \begin{vmatrix} 1 & 0 \\ 0 & 1 \end{vmatrix} \\ \begin{vmatrix} -\frac{k_1}{J_1} & \frac{k_1}{J_1} \\ \frac{k_1}{J_2} & -\frac{k_1}{J_2} - \frac{k_2}{J_2} \end{vmatrix} & \begin{vmatrix} -\frac{d_1}{J_1} & \frac{d_1}{J_1} \\ \frac{d_1}{J_2} & -\frac{d_1}{J_2} - \frac{d_2}{J_2} \end{vmatrix} \end{array} \right|$$

在 Modelica 语言，我们可以如下地用子矩阵构建 A 矩阵：

```
model RotationalSMD_Concat
  "State space version of a rotational spring-mass-damper system using concatenation"
  parameter Real J1=0.4;
  parameter Real J2=1.0;
  parameter Real k1=11;
  parameter Real k2=5;
  parameter Real d1=0.2;
  parameter Real d2=1.0;
  parameter Real S[2,2] = [-1/J1, 1/J1; 1/J2, -1/J2];
  extends LTI(nx=4, nu=0, ny=0, x0={0, 1, 0, 0},
    A=[zeros(2, 2), identity(2);
      k1*S+[0,0;-k2/J2], d1*S+[0,0;0,-d2/J2]],
    B=fill(0, 4, 0), C=fill(0, 0, 4),
    D=fill(0, 0, 0));
  equation
    u = fill(0, 0);
  end RotationalSMD_Concat;
```

本节里，我们并没有把 Lotka-Volterra 方程表示为 LTI 形式。Lotka-Volterra 方程虽然是时不变系统，但是同时却是非线性的。值得一提的是，Modelica 并没有在 LTI 模型内执行对线性和时不变这两个属性的约束。因此，用上述方法其实可以描述非线性或者时变的模型。但倘若如此就会引起一定的混乱，因为 LTI 这个术语意味这方程是线性时不变的。

## 使用部件

在至今所有的例子中，我们已经（通过 extends）用继承来重用 LTI 模型的公式。一般而言，**把方程作为子组件**是一个相比之下好得多的代码复用方法。为了说明这种方法，我们将前面讨论过的 electrical examples ( 11) 重写为 LTI 形式。不过这次，我们会建立为 LTI 模型创建一个命名实例。

```
model RLC "State space version of an RLC circuit"
  parameter Real Vb=24;
  parameter Real L=1;
  parameter Real R=100;
  parameter Real C=1e-3;
  LTI rlc_comp(nx=2, nu=1, ny=2, x0={0,0},
    A=[-1/(R*C), 1/C; -1/L, 0],
    B=[0; 1/L],
    C=[1/R, 0; -1/R, 1],
    D=[0; 0]);
  equation
    rlc_comp.u = {Vb};
  end RLC;
```

请注意，这一次我们没有使用 extends 或任何形式的继承。相反，我们实际上声明了一个类型为 LTI 而名为 rlc\_comp 的变量。一旦我们介绍完 Modelica 语言中描述不同行为的所有基础知识，我们就会将注

意力转向如何将这些方程整理成可重用的组件 (173)。但现在，这不过是对后面的（重要）内容“先睹为快”而已。

我们在这个 RLC 例子中看到的是，我们现在有一个叫做 rlc\_comp 的变量，而此部件拥有所有 LTI 模型的参数和变量。所以，例如我们可以看到用于指定输入 u 的方程写作：

```
rlc_comp.u = {Vb};
```

请注意，我们所提供的这个方程中变量 u 是 rlc\_comp 里面的变量。正如我们将在后面看到的，我们可以用层次结构来管理在描述复杂系统时产生的复杂度。在这里，使用 . 操作符可以让我们引用层级结构里的变量。同样，我们会在介绍组件 (173) 时对此进行彻底讨论。

### 第 3.1.2 节 一维热传导

我们前面关于状态空间 (81) 的讨论引入了矩阵和向量。上面讨论的焦点主要是数组的数学性质。在本节中，我们将考虑如何用数组来表述更实际的问题，即变量的一维空间分布。我们将观察 Modelica 与数组有关的语言特性，以及这些特性如何帮助我们更简略地表达系统行为。

我们的问题关于一个简单的热传导问题。考虑如下所示的一维棒：



#### 推导方程

让我们考虑此杆在每个离散截面的热平衡。首先，我们可以得到第  $i$  个截面的热容。这可以表示为：

$$m_i C T_i$$

其中  $m$  为第  $i$  节的质量， $C$  为比热容（材料属性），而  $T_i$  则是第  $i$  节的温度，我们可以进一步将质量描述为：

$$m_i = \rho V_i$$

其中  $\rho$  是材料的密度，而  $V_i$  则是第  $i$  节的体积。最后，第  $i$  节的体积如下：

$$V_i = A_i L_i$$

其中  $A_i$  为第  $i$  节的截面积（假设其为恒定），而  $L_i$  为第  $i$  节的长度。对于这个例子，我们假定该杆是由相等大小的部分组成。在这种情况下，我们可以定义各段长度  $L_i$  如下：

$$L_i = \frac{L}{n}$$

我们也将假定全杆的截面积为恒定。因此，每节的质量可以写为：

$$m = \rho A L_i$$

在这种情况下，各部分的热容量会是：

$$\rho A L_i C T_i$$

反过来，这意味着在任何时间该节中的热量增长是：

$$\rho A L_i C \frac{dT_i}{dt}$$

其中我们假定  $A$ 、 $L_i$  还有  $C$  不会随着时间变化。

完成了以上热容。此外，我们将考虑两种不同形式的热传递。热传递的第一种形式，我们将考虑每节在特定环境温度  $T_{amb}$  下的对流。

$$q_h = -hA(T_i - T_{amb})$$

其中  $h$  为对流系数。另一个形式的热传递是相邻部分之间的热传导。这里有两个影响因素，其中之一是与  $i-1$  元素的热传导（如果这个元素存在），另外则是与  $i+1$  元素的热传导（前提也是该元素存在）。这些因素可以分别表示为：

$$q_{k_{i \rightarrow i-1}} = -kA \frac{T_i - T_{i-1}}{L_i}$$

$$q_{k_{i \rightarrow i+1}} = -kA \frac{T_i - T_{i+1}}{L_i}$$

利用上述关系，我们对首个元素的热平衡方程有：

$$\rho A L_i C \frac{dT_1}{dt} = -kA \frac{T_1 - T_2}{L_i} - hA(T_1 - T_{amb})$$

类似地，最后一个元素的热平衡方程为：

$$\rho A L_i C \frac{dT_n}{dt} = -kA \frac{T_n - T_{n-1}}{L_i} - hA(T_n - T_{amb})$$

最后，所有其他元素的热平衡有：

$$\rho A L_i C \frac{dT_i}{dt} = -kA \frac{T_i - T_{i-1}}{L_i} - kA \frac{T_i - T_{i+1}}{L_i} - hA(T_i - T_{amb})$$

## 实现

我们首先定义各物理量的类型。定义类型可以让变量有正确的单位。而在特定的软件会支持对方程的进行量纲检测。我们的类型定义如下：

```
type Temperature=Real(unit="K", min=0);
type ConvectionCoefficient=Real(unit="W/K", min=0);
type ConductionCoefficient=Real(unit="W.m-1.K-1", min=0);
type Mass=Real(unit="kg", min=0);
type SpecificHeat=Real(unit="J/(K.kg)", min=0);
type Density=Real(unit="kg/m3", min=0);
type Area=Real(unit="m2");
type Volume=Real(unit="m3");
type Length=Real(unit="m", min=0);
type Radius=Real(unit="m", min=0);
```

我们也将定义几个参数以描述所模拟的棒。

```
parameter Integer n=10;
parameter Length L=1.0;
parameter Radius R=0.1;
parameter Density rho=2.0;
parameter ConvectionCoefficient h=2.0;
parameter ConductionCoefficient k=10;
parameter SpecificHeat C=10.0;
parameter Temperature Tamb=300 "Ambient temperature";
```

在上述参数给定的情况下，就可以用以下列参数定义式计算每节的面积和体积：

```
parameter Area A = pi*R^2;
parameter Volume V = A*L/n;
```

最后，这个问题里唯一的数组是各部分的温度（因为实际上这是唯一沿杆长而改变的量）：

```
Temperature T[n];
```

以上就是我们需要做出的所有声明。现在让我们考虑所需的各种方程。首先，我们需要指定杆的初始条件。我们将假定  $T_1(0) = 200$ 、 $T_n(0) = 300$ ，而其他所有元素的初始温度则为上述条件的线性内插值。下列方程便体现了上述的条件：

```
initial equation
  T = linspace(200,300,n);
```

其中 `linspace` 操作符用于产生一个  $n$  值的数组。而这  $n$  个值介于 200 和 300 之间线性改变。回想之前的状态空间 (81) 例子，我们可以加入其两边表达式均为向量的方程。本例的方程便是此类方程的又一个例子。

最后，我们介绍每部分的温度随时间变化的方程：

```
equation
  rho*C*der(T[1]) = -h*(T[1]-Tamb)-k*A*(T[1]-T[2])/(L/n);
  for i in 2:(n-1) loop
    rho*C*der(T[i]) = -k*A*(T[i]-T[i-1])/(L/n)-k*A*(T[i]-T[i+1])/(L/n);
  end for;
  rho*C*der(T[end]) = -h*(T[end]-Tamb)-k*A*(T[end]-T[end-1])/(L/n);
```

第一个方程对应的第 1 个元素的热平衡，而最后一个方程则对应第  $n$  个。中间的方程则对应了其他所有元素。注意这里使用了 `end` 作为下标。如果某维度的下标为一个表达式时，表达式内的 `end` 表示该维度的大小。在这里，我们用 `end` 表示最后一个元素。当然，我们在这里也可以用  $n$  来表示，但一般而言，当某个维度的大小不曾与某个特定变量相关时，`end` 可能会非常有用。

另外，请注意模型内使用的 `for` 循环。`for` 循环让循环变量在一定范围的值里变化。在本例里，循环变量为  $i$ ，而值的变化范围则是从 2 到  $n - 1$ 。`for` 循环的基本语法如下：

```
for <var> in <range> loop
  // statements
end for;
```

其中，`<range>` 是一个含有不同值的向量。而一种方便的产生数列的方式是使用范围操作符：`:`。范围操作符前的值是数列的初值，而在操作符后的值则是数列的终值。因此，以表达式 `5:10` 例，表达式会生成一个由 5、6、7、8、9、10 组成的向量。注意，生成的向量包括了用于指定范围的两个值。

当 `for` 循环用在等式区域时，循环内每个方程都会在 `for` 循环每次迭代时产生一个新的方程。因此，在本例对应每个在 2 和  $n - 1$  之间的  $i$ ，我们会一共生成  $n - 2$  个方程。

综合以上内容，完整的模型如下：

```
model Rod_ForLoop "Modeling heat conduction in a rod using a for loop"
  type Temperature=Real(unit="K", min=0);
  type ConvectionCoefficient=Real(unit="W/K", min=0);
  type ConductionCoefficient=Real(unit="W.m-1.K-1", min=0);
  type Mass=Real(unit="kg", min=0);
  type SpecificHeat=Real(unit="J/(K.kg)", min=0);
  type Density=Real(unit="kg/m3", min=0);
  type Area=Real(unit="m2");
  type Volume=Real(unit="m3");
  type Length=Real(unit="m", min=0);
  type Radius=Real(unit="m", min=0);

  constant Real pi = 3.14159;

  parameter Integer n=10;
  parameter Length L=1.0;
  parameter Radius R=0.1;
  parameter Density rho=2.0;
  parameter ConvectionCoefficient h=2.0;
  parameter ConductionCoefficient k=10;
  parameter SpecificHeat C=10.0;
```

```

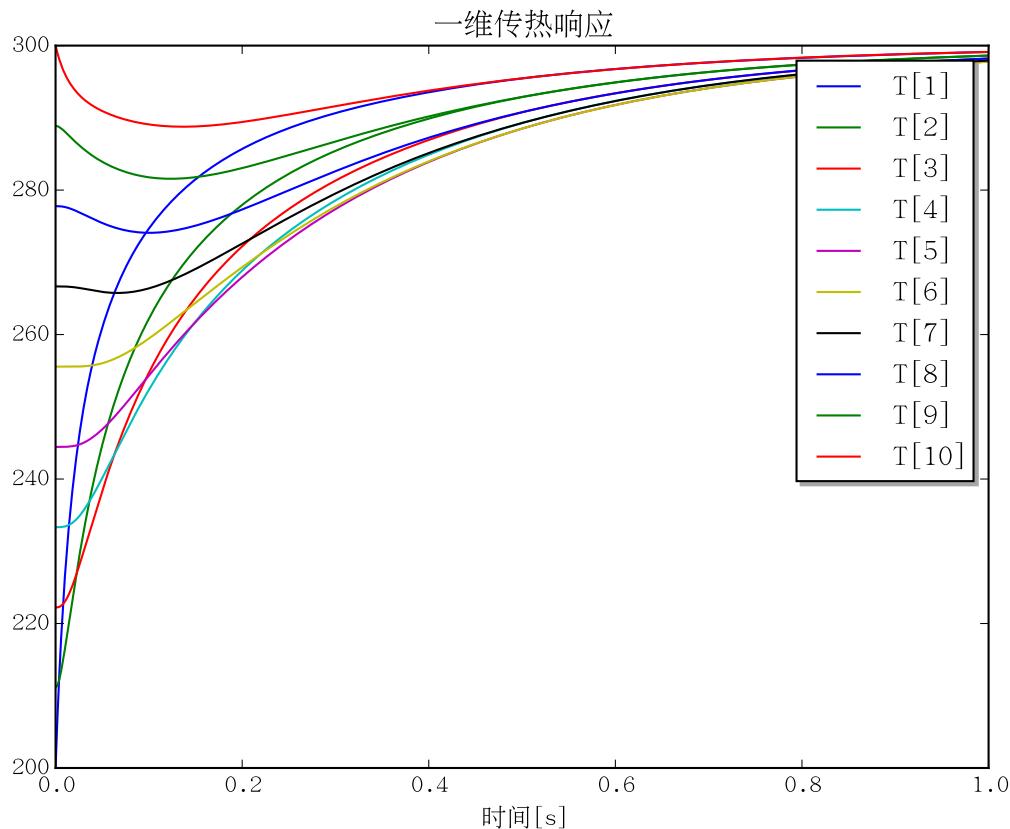
parameter Temperature Tamb=300 "Ambient temperature";
parameter Area A = pi*R^2;
parameter Volume V = A*L/n;

Temperature T[n];
initial equation
  T = linspace(200,300,n);
equation
  rho*V*C*der(T[1]) = -h*(T[1]-Tamb)-k*A*(T[1]-T[2])/(L/n);
  for i in 2:(n-1) loop
    rho*V*C*der(T[i]) = -k*A*(T[i]-T[i-1])/(L/n)-k*A*(T[i]-T[i+1])/(L/n);
  end for;
  rho*V*C*der(T[end]) = -h*(T[end]-Tamb)-k*A*(T[end]-T[end-1])/(L/n);
end Rod_ForLoop;

```

Note: 请注意，我们用常量形式在模型里包含了  $\pi$ 。在这本书的后面，我们将讨论如何正确导入常用的常数（152）。

对模型进行仿真就会得到各节点温度的解，如下：



注意温度在开始是如何线性分布的（正如我们在 initial equation 区域所指定的一样）。

### 其他可选方法

实际上，有好几种方法可以生成我们需要的方程。而每种方法在不同的情景下有其优点与缺点。我们将在下面一一介绍这些可能的方法。而选择哪种方法可以让方程感觉上最容易理解，则是取决于模型开发者自己。

我们可以用一个数组特性让这些方程变得更为简单。这个特性叫做数组解析（译注：原文为 array

comprehension，类似 Python 的 list comprehension）。数组解析将 for 循环倒了过来。也就是说，我们在单一的等式后加上了循环变量在循环时如何取不同值的信息。在我们的例子中，我们可以使用数组解析将方程表达为如下方式：

```
equation
  rho*V*C*der(T[1]) = -h*(T[1]-Tamb)-k*A*(T[1]-T[2])/(L/n);
  rho*V*C*der(T[2:n-1]) = {-k*A*(T[i]-T[i-1])/(L/n)-k*A*(T[i]-T[i+1])/(L/n) for i in 2:(n-1)};
  rho*V*C*der(T[end]) = -h*(T[end]-Tamb)-k*A*(T[end]-T[end-1])/(L/n);
```

我们还可以在数组解析内加上一些 if 表达式以删去不存在的热平衡效应。在这种情况下，我们可以把 equation 简化为一个（虽然占用了多行的）方程：

```
equation
  rho*V*C*der(T) = {-h*(T[i]-Tamb)
    -(if i==1 then 0 else k*A/(L/n)*(T[i]-T[i-1]))
    -(if i==n then 0 else k*A/(L/n)*(T[i]-T[i+1])) for i in 1:n};
```

回顾前述几个例子，Modelica 语言支持矢量方程。在这些情况下，只要方程左右均是大小同样的向量，我们就可以使用一个（向量）方程来表示多个标量方程。我们可以利用这个特性来简化方程式，结果如下：

```
equation
  rho*V*C*der(T[1]) = -h*(T[1]-Tamb)-k*A*(T[1]-T[2])/(L/n);
  rho*V*C*der(T[2:n-1]) = -k*A*(T[2:n-1]-T[1:n-2])/(L/n)-k*A*(T[2:n-1]-T[3:n])/(L/n);
  rho*V*C*der(T[end]) = -h*(T[end]-Tamb)-k*A*(T[end]-T[end-1])/(L/n);
```

注意，倘若大家使用一定范围的下标去存取向量变量，如 T，那么结果就会得到这些下标对应元素所组成的向量。例如，表达式 T[2:4] 等价于 {T[2], T[3], T[4]}。下标表达式并不需要是一个范围表达式。例如，表达式 T[{2,5,9}] 等价于 {T[2], T[5], T[9]}。

最后，让我们考虑重构这些方程的最后一个方法。想象一下，我们引入了另外两个向量变量：

```
Heat Qleft[n];
Heat Qright[n];
```

然后我们就可以写出以下的两个方程（再次使用矢量方程），用以定义到前后两节分别的热损失。

```
Qleft = {if i==1 then -h*(T[i]-Tamb) else -k*A*(T[i]-T[i-1])/(L/n) for i in 1:n};
Qright = {if i==n then -h*(T[i]-Tamb) else -k*A*(T[i]-T[i+1])/(L/n) for i in 1:n};
```

这使我们可以用矢量方程表达每个部分的热平衡，而无须包含任何下标。

```
rho*V*C*der(T) = Qleft+Qright;
```

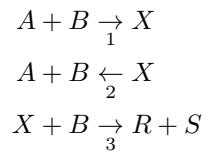
## 结论

在本节中，我们看到了使用向量变量以及向量方程式来表示一维传热的多种方法。当然，这些向量的相关功能可用于类型广泛的不同问题。本节目的是为大家介绍几种特性，以演示开发者在使用向量时拥有的不同选项。

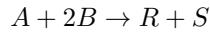
### 第 3.1.3 节 化学系统

在本节中，我们会考虑几种不同的方式去描述化学系统的行为。首先，我们会在不使用数组功能的前提下进行建模。然后，我们会用向量描述相同的行为。最后，我们将使用枚举再次实现相同的模型。

在我们所有的例子里，我们建立的模型均是基于以下的反应系统<sup>2</sup>:



必须注意  $X$  不过是反应的中间结果。总反应可表示为:



使用质量作用定律，我们可以将这些化学方程式转化为一下数学方程:

$$\begin{aligned}\frac{d[A]}{dt} &= -k_1[A][B] + k_2[X] \\ \frac{d[B]}{dt} &= -k_1[A][B] + k_2[X] - k_3[B][X] \\ \frac{d[X]}{dt} &= k_1[A][B] - k_2[X] - k_3[B][X]\end{aligned}$$

其中  $k_1$ 、 $k_2$ 、 $k_3$  分别是第一、二、三个反应的反应系数。这些方程是通过在考虑每种物质的变化，以及涉及该物质的每个反应后所推导得到的。因此，例如第一个反应  $A + B \rightarrow X$  是把  $A$  分子和  $B$  分子转化为  $X$  分子。我们可以看到  $-k_1[A][B]$  这项出现在  $A$  的平衡方程里。这项表示了  $A$  应为该反应而减少的量。这些平衡方程的每一项均是以类似的方式推导出来的。

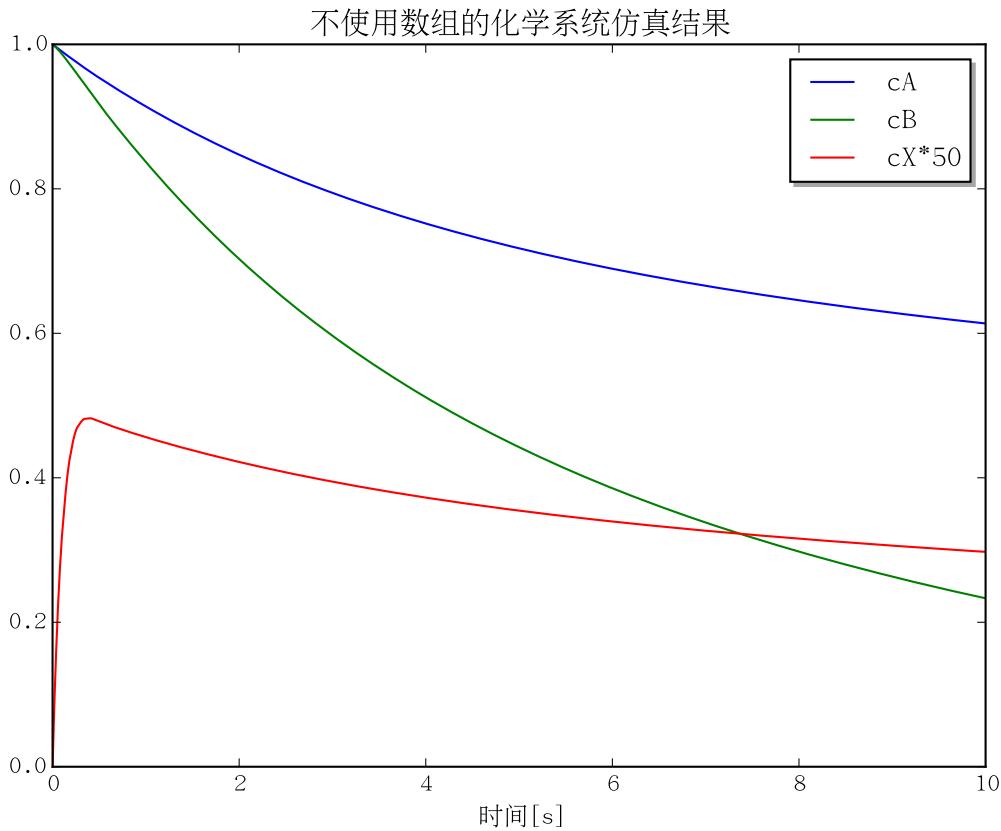
## 不使用数组

首先，让我们采取完全不使用数组的方法。在这里，我们直接把浓度  $[A]$ 、 $[B]$  以及  $[X]$  用变量  $cA$ 、 $cB$ 、 $cX$  表示如下:

```
model Reactions_NoArrays "Modeling a chemical reaction without arrays"
  Real cA;
  Real cB;
  Real cX;
  parameter Real k1=0.1;
  parameter Real k2=0.1;
  parameter Real k3=10;
initial equation
  cA = 1;
  cB = 1;
  cX = 0;
equation
  der(cA) = -k1*cA*cB + k2*cX;
  der(cB) = -k1*cA*cB + k2*cX - k3*cB*cX;
  der(cX) = k1*cA*cB - k2*cX - k3*cB*cX;
end Reactions_NoArrays;
```

通过这种方法，我们建立了每种物质的平衡方程。对模型进行仿真后，我们会得到以下结果:

<sup>2</sup><http://library.wolfram.com/examples/chemicalkinetics/>



### 使用数组

另一种对化学系统进行建模的方法是使用向量。用这种方法，我们将化学物质  $A$ 、 $B$ 、 $X$  分别用索引 1、2、3 标记。浓度则被映射到向量变量  $C$ 。我们同时也将反应系数储存在反应系数向量  $k$  里。

进行了上述变换后，所有的方程都转化为向量方程：

```
model Reactions_Array "Modeling a chemical reaction with arrays"
  Real C[3];
  parameter Real k[3] = {0.1, 0.1, 10};
  initial equation
    C = {1, 1, 0};
  equation
    der(C) = {-k[1]*C[1]*C[2] + k[2]*C[3],
               -k[1]*C[1]*C[2] + k[2]*C[3] - k[3]*C[2]*C[3],
               k[1]*C[1]*C[2] - k[2]*C[3] - k[3]*C[2]*C[3]};
end Reactions_Array;
```

反应方程是非线性的，所以这些方程不能被转换成完全线性的形式。但是，我们可以通过使用矩阵向量积进行进一步简化。换句话说，方程：

$$\begin{aligned}\frac{d[A]}{dt} &= -k_1[A][B] + k_2[X] \\ \frac{d[B]}{dt} &= -k_1[A][B] + k_2[X] - k_3[B][X] \\ \frac{d[X]}{dt} &= k_1[A][B] - k_2[X] - k_3[B][X]\end{aligned}$$

可以转化为下面的形式:

$$\frac{d}{dt} \begin{Bmatrix} [A] \\ [B] \\ [X] \end{Bmatrix} = \begin{bmatrix} -k_1[B] & 0 & k_2 \\ -k_1[B] & -k_3[X] & k_2 \\ k_1[B] & -k_3[X] & -k_2 \end{bmatrix} \begin{Bmatrix} [A] \\ [B] \\ [X] \end{Bmatrix}$$

而上述方程可以用以下的 Modelica 形式进行表示:

```
der(C) = [-k[1]*C[2], 0,
           k[2];
           -k[1]*C[2], -k[3]*C[3], k[2];
           k[1]*C[2], -k[3]*C[3], -k[2]]*C;
```

这种方法的问题是，我们必须时刻留意各个索引（如 1、2 或 3）分别对应哪种物质（如 A、B 及 X）。

## 使用枚举

为了解决数字和名称来回映射的问题，我们的第三种方法则是利用 Modelica 的 enumeration 类型。枚举类型可以让我们定义一个名称的集合。然后，我们可以用这个名称的几何对应数组的下表。我们将定义以下的枚举:

```
type Species = enumeration(A, B, X);
```

上述语句定义了一个特殊的类型叫做 Species，而这个类型有三个可能值：A、B 及 X。然后，我们可以使用这个枚举作为数组的一个维度，如下：

```
Real C[Species];
```

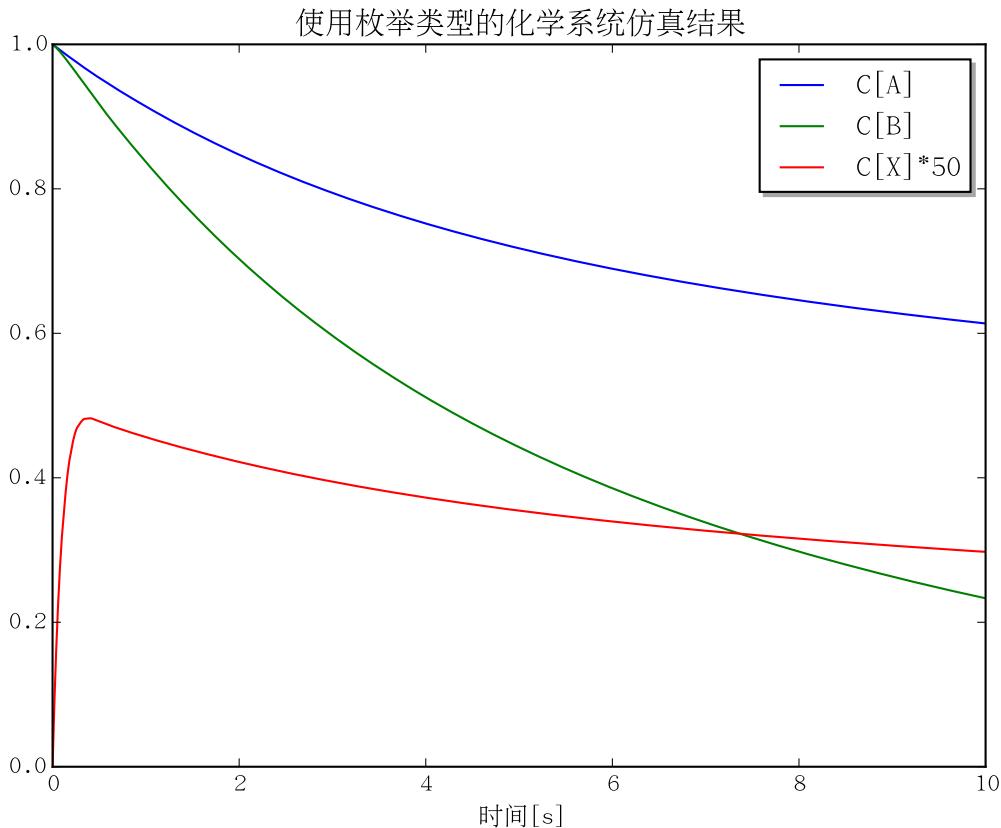
由于 Species 类型只有三种可能的值，这意味着矢量 C 恰好有三个分量。然后，我们可以分别使用 C[Species.A]、C[Species.B]、C[Species.X] 去指代 C 的分量。

由于每次在物质名称前加入 Species 颇为不便，为方便起见，我们可以定义以下常量:

```
constant Species A = Species.A;
constant Species B = Species.B;
constant Species X = Species.X;
```

这样一来，我们现在可以用 C[A] 来指代物质 A 的浓度。综合以上的结果，我们可以使用如下方式用枚举描述我们的化学系统:

```
model Reactions_Enum "Modeling a chemical reaction with enums"
  type Species = enumeration(
    A,
    B,
    X);
  Real C[Species] "Species concentrations";
  parameter Real k[3] = {0.1, 0.1, 10};
  constant Species A = Species.A;
  constant Species B = Species.B;
  constant Species X = Species.X;
  initial equation
    C[A] = 1.0;
    C[B] = 1.0;
    C[X] = 0.0;
  equation
    der(C[A]) = -k[1]*C[A]*C[B] + k[2]*C[X];
    der(C[B]) = -k[1]*C[A]*C[B] + k[2]*C[X] - k[3]*C[B]*C[X];
    der(C[X]) = k[1]*C[A]*C[B] - k[2]*C[X] - k[3]*C[B]*C[X];
  end Reactions_Enum;
```



## 结论

在这一章中，我们介绍了如何在使用以及不使用数组的前提下描述一组化学方程式。我们还演示了如何在数组中使用 enumeration 类型。这样用名称代替数字索引，可以让产生的方程式更具可读性。此外，本节还表明了 enumeration 类型可以不仅用于索引数组，也能在变量声明中定义一个或多个维度。

## 第 3.2 节 回顾

### 第 3.2.1 节 数组声明

#### 语法

数组声明的语法很简单。除了在变量名之后应标以指定数组各维度的大小外，语法与正常变量声明是相同的。数组声明的一般形式为：

```
VariableType varName[dim1, dim2, ..., dimN];
```

其中 VariableType 是一个 Modelica 类型，如：Real 或 Integer。varName 为变量名称。

#### 整数大小

通常情况下，维度定义仅仅是表明该维度的大小的整数。例如：

```
Real x[5];
```

在这种情况下，`x` 是实数型数组。而这个数组仅有大小为 5 的一个维度。使用参数或常量去指定数组大小也是可以的，如：

```
parameter Integer d1=5;
constant Integer d2=2;
Real x[d1, d2];
```

## 相关维度

在后面讨论 Modelica 内的数组函数（97）时，我们就会看到，我们甚至可以使用 `size` 函数去用一个数组的大小定义另外一个的大小。请考虑以下声明：

```
Real x[5];
Real y[size(x,1)];
```

在这种情况下，`y` 将拥有大小为 5 的一个维度。函数 `size(x,1)` 会返回数组 `x` 第 1 维度的大小。在许多应用里，以这种方式表达不同数组维度间的关系是很有用的（例如，为确保数组总有正确的大小以进行矩阵乘法等操作）。

## 未定义维度

一些情况下，我们可以不定义数组的大小，让其大小可以在后来的某些情景里确定。例如，我们会在讨论一些有向量参数的函数（107）时看到这样的例子。

为了表明数组某个特定维数的大小（还）仍为未知，我们可以使用：符号作为维度的大小。所以，在如下的声明里：

```
Real A[:,2];
```

我们定义了一个二维数组。第一维的大小没有被定义。而第二维的大小明确被指定为 2。实际上，我们已经声明了 `A` 为一个行数不详而有两列的矩阵。

## 非整数维度

### 枚举类

正如我们在化学系统（89）例子内看到的，另一种指定维度的方法是用枚举。如果用枚举指定一个维度，则该维度的大小将等于该枚举可能值的总数目。在即将到来的关于数组索引（104）的讨论里，我们将看到正确地索引一个使用枚举作为维数的数组。

### 布尔值

我们还可以将其中一个维度声明为 Boolean，例如：

```
Real x[Boolean];
```

## 第 3.2.2 节 数组的建构

现在我们知道了如何将变量声明为向量（93）。下一步则是填充在这些数组中的元素。有许多不同的方式可以构造的 Modelica 数组。

## 常值符

### 向量

用于构建数组最简单的方法是枚举每一个单独的元素。例如，下面的参数声明表明变量 x 为向量：

```
parameter Real x[3];
```

我们在此使用“向量”这个术语时，其指代的是仅有一个维度下标的数组。如果想要为向量赋值，我们可以这样做：

```
parameter Real x[3] = {1.0, 0.0, -1.0};
```

显然，变量 x 是（被声明为）有三个实值分量的向量。为了保持一致性，等式的右侧也必须是具有三个实值分量的向量。幸运的是，它确实是。表达式 {1.0, 0.0, -1.0} 是 Modelica 里用于创建向量的一种特殊语法。用这种由包含着以逗号分隔的一系列表达式的一对 {}，我们可以用这种语法来构任何大小的向量。如：

```
parameter Real x[5] = {1.0, 0.0, -1.0, 2.0, 0.0};
```

尽管可以使用 {} 表示法来构建任何维度的数组，如

```
parameter Real B[2,3] = {{1.0, 2.0, 3.0}, {5.0, 6.0, 7.0}};
```

### 范围表示法

Modelica 语言包含速记表示法，用以构建有连续或者等差的数列组成的向量。例如，为了构建一个由 1 到 5 之间整数组成的向量，可以使用如下的语法：

```
1:5 // {1, 2, 3, 4, 5}
```

相同的语法可以用以构建一个由 1 到 5 之间整数组成的向量：

```
1.0:5.0 // {1.0, 2.0, 3.0, 4.0, 5.0}
```

应当注意，在用此形式表示实数向量时，浮点精度问题可能会导致终值不包括在向量内的情况。也可以使用另外一种方法（而且可能更安全）：

```
1.0*(1:5) // {1.0, 2.0, 3.0, 4.0, 5.0}
{1.0*i for i in 1:5} // {1.0, 2.0, 3.0, 4.0, 5.0}
```

此外，也可以通过在初始值以及终值之间加入“步长”值，以构建一个元素间间隔不为 1 的范围值。例如，3 和 9 之间的所有奇数可以表示为：

```
3:2:9 // {3, 5, 7, 9}
```

另外，在处理浮点数字时也插入一个步长值。范围表示法还可以应用在 enumeration 类里（但在此情况下不能使用步长值）。

### 矩阵的构造

但值得注意的是，有另一种用于构成矩阵（正好有两个下标维度的数组）的特殊语法。请考虑以下带有初始化语句的参数声明：

```
parameter Real B[2,3] = [1.0, 2.0, 3.0; 5.0, 6.0, 7.0];
```

在这种情况下，参数 B 相当于如下的数学记号：

$$B = \begin{bmatrix} 1.0 & 2.0 & 3.0 \\ 5.0 & 6.0 & 7.0 \end{bmatrix}$$

正如我们可以从 Modelica 代码里或其更数学的表示形式里得知，矩阵 B 有两行三列。使用这种方式构建矩阵的语法比仅仅构建向量时更为复杂。从表面上我们看到，向量的两旁是 {}，而矩阵两旁则为 []。但更重要的是，逗号和分号混在一起用作分隔符。分号用来分隔行，逗号则用来分隔列。

这个矩阵构建方法的一个很好的特性在于，在其中可以嵌入向量或子矩阵。

### 向量

当嵌入向量时，值得注意的是向量均为列向量。还句话说，在建构矩阵时，大小为  $n$  的向量被看作  $n$  行 1 列的矩阵。

为了演示嵌入的流程，可以考虑在这里我们希望构建以下矩阵的情况：

$$C = \begin{bmatrix} | & 2 & 1 | & | & 0 & 0 | & | & 0 & 0 | \\ | & 1 & 2 | & | & 0 & 0 | & | & 0 & 0 | \\ | & 0 & 0 | & | & 2 & 1 | & | & 0 & 0 | \\ | & 0 & 0 | & | & 1 & 2 | & | & 0 & 0 | \\ | & 0 & 0 | & | & 0 & 0 | & | & 2 & 1 | \\ | & 0 & 0 | & | & 0 & 0 | & | & 1 & 2 | \end{bmatrix}$$

我们可以用 Modelica 简洁地构建这个矩阵：先创建每个子矩阵，继而如下将子矩阵填入  $C$ ：

```
parameter D[2,2] = [2, 1; 1, 2];
parameter Z[2,2] = [0, 0; 0, 0];
parameter C[6,6] = [D, Z, Z;
                   Z, D, Z;
                   Z, Z, D];
```

换句话说，，和；分隔符可以用于标量或子矩阵。

正如我们将看到的，对于上述矩阵构建有数个不同的数据建构函数（97）是很有用的。

### 任意大小的数组

到目前为止，我们已经讨论了向量和矩阵。但是，利用一系列嵌套向量，你可以构造任意维数的数组（包括向量和矩阵）。例如，为了构建一个三维矩阵，我们可以简单地嵌套向量，如下：

```
parameter Real A[2,3,4] = { { {1, 2, 3, 4},
                           {5, 6, 7, 8},
                           {9, 8, 7, 6} },
                           { {4, 3, 2, 1},
                           {8, 7, 6, 5},
                           {4, 3, 2, 1} } };
```

如在此示例中可以看出，在此嵌套结构的最内层元素对应于声明内最右的维度。换句话说，此数组为包含两个元素的向量。而这两个元素则均为含有三个元素的向量。进而，上述的三个元素则各为含有 4 个标量的向量。

### 数组解析

到目前为止，我们已经展示了如何通过枚举数组中的元素来构造向量、矩阵以及高维数组。正如我们在高维数组的情况下看到，构造语句会变得非常复杂。幸运的是，Modelica 语言包括数组解析这个方便的语法，让我们可以编程构建数组。这种方法有两个主要优点。首先，这是一个更为紧凑的表示法。其次，这种方法可以让我们轻松地表达数组中的值是如何与指标联系在一起。

为了演示数组解析，请考虑数组内元素与其索引的关系。

$$a_{ijk} = i \ x_j \ y_k$$

其中  $x$ 、 $y$  均为向量。我们已经看到了如何递归地使用一系列嵌套向量去定义这样的数组。但是，我们也看到了表达式可能会变得多长，而读写这样的表达式会变得如何的费力。使用数组解析我们可以容易地建构数组  $a$ :

```
parameter Real a[10,12,15] = {i*x[j]*y[k] for k in 1:15,
                                j in 1:12,
                                i in 1:10};
```

这个代码以区区数行 Modelica 代码生成了有 1800 个元素的数组。

### 第 3.2.3 节 数组函数

Modelica 自带了大量的数组有关的函数。在本节，我们会浏览不同类型的函数，并介绍这些函数的使用方法。

#### 数据建构函数

我们已经讨论过数组的建构 (94)。我们看到用以建立向量和矩阵的不同语法结构。此外，我们看到了如何从其他矩阵来创建新矩阵。Modelica 的另外有几个函数可以用来构造向量、矩阵和更高维数组。这些函数可以替代或补充先前介绍的方法。

`fill`

函数 `fill` 用于创建拥有唯一元素值的数组。`fill` 的参数有:

```
fill(v, d1, d2, ..., dN)
```

其中  $v$  是数组中每个元素的值，而剩下的参数则是每一维的尺寸。所得到的数组中的元素将具有和  $v$  相同的类型。因此，若要用 1.7 这个值填充一个  $5 \times 7$  实数数组，我们可以使用以下语句:

```
parameter Real x[5,7] = fill(1.7, 5, 7);
```

这会让矩阵用以下方式被填充:

$$\begin{bmatrix} 1.7 & 1.7 & 1.7 & 1.7 & 1.7 & 1.7 & 1.7 \\ 1.7 & 1.7 & 1.7 & 1.7 & 1.7 & 1.7 & 1.7 \\ 1.7 & 1.7 & 1.7 & 1.7 & 1.7 & 1.7 & 1.7 \\ 1.7 & 1.7 & 1.7 & 1.7 & 1.7 & 1.7 & 1.7 \\ 1.7 & 1.7 & 1.7 & 1.7 & 1.7 & 1.7 & 1.7 \end{bmatrix}$$

`zeros`

在使用数组时，一个常见的用例是创建一个只包含零元素的数组。实际上，这实现了和 `fill` 函数相同的功能。不过，由于需要填充的值为已知的，用户只需要指定大小。我们可以使用 `zeros` 以如下形式初始化数组:

```
parameter Real y[2,3,5] = zeros(2, 3, 5);
```

`ones`

`ones` 的函数与 `zeros` 函数几乎相同。唯一的不同是，生成的数组中的每个元素取值为 1。因此，例如:

```
parameter Real z[3,5] = ones(3, 5);
```

这会让矩阵用以下方式被填充:

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

identity

另一个常见的需求是方便地建立单位矩阵。单位矩阵的对角线元素都是 1，而所有其他元素均为 0。This can be done very easily with the identity. 通过 identity 函数，这可以很容易完成的。单位矩阵函数需要一个整数参数。这个参数确定在生成矩阵的行数和列数。So, invoking identity as: 因此，调用 identity 如下:

```
identity(5);
```

会得到下列矩阵:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

diagonal

diagonal 函数用来创建一个所有的非对角元素是 0 的矩阵。对角函数的唯一参数是包含一个对角线元素值的数组。所以，要创建以下对角矩阵

$$\begin{bmatrix} 2.0 & 0 & 0 & 0 \\ 0 & 3.0 & 0 & 0 \\ 0 & 0 & 4.0 & 0 \\ 0 & 0 & 0 & 5.0 \end{bmatrix}$$

用户可以使用下列 Modelica 代码:

```
diagonal({2.0, 3.0, 4.0, 5.0});
```

linspace

linspace 函数可以创建一个其元素的取值在一定间隔内线性分布的向量。linspace 函数的调用方法如下:

```
linspace(v0, v1, n);
```

其中 v0 是向量的首个元素，v1 是向量的最后一个元素，而 n 向量的元素个数。因此，例如以如下方式调用 linspace:

```
linspace(1.0, 5.0, 9);
```

会得到向量:

```
{1.0, 1.5, 2.0, 3.5, 3.0, 3.5, 4.0, 4.5, 5.0}
```

## 转换函数

下面的函数提供了变换数组的方法。

scalar

scalar 函数以如下方式被调用:

```
scalar(A)
```

其中 A 为有任意维数的数组，前提是每个维度的大小均为 1。scalar 函数（仅）返回包含在数组中的标量值。例如，

```
scalar([5]) // Argument is a two-dimensional array (matrix)
```

以及

```
scalar({5}) // Argument is a one-dimensional array (vector)
```

均会得到标量值 5。

vector

vector 函数以如下方式被调用:

```
vector(A)
```

其中 A 为有任意维数的数组，前提是仅有有一个维度的大小大于 1。在 vector 函数将数组内容返回为一个矢量（即仅有单个维度的数组）。因此，若我们传入列矩阵或行矩阵，如：

```
vector([1;2;3;4]) // Argument is a column matrix
```

或

```
vector([1,2,3,4]) // Argument is a row matrix
```

我们会得到：

```
{1,2,3,4}
```

matrix

matrix 函数以如下方式被调用:

```
matrix(A)
```

其中 A 为一任意维数的数组，且只有两个大小大于 1 的维度。matrix 函数将数组的内容返回为一个矩阵（即只有两个维度的数组）。

## 数学操作

在线性代数里，有许多不同类型的数学运算通常可以应用在向量和矩阵上。Modelica 语言提供了函数来执行大多数的上述操作。以这种方式，Modelica 的等式可以看起来非常像其线性代数里数学对应版本。

让我们从加法，减法，乘法，除法和求幂等运算开始。在大多数情况下，这些操作符应用在标量，矢量和矩阵的各种组合时，和其在数学上的定义是一样的。但是，为了介绍的完整性和以及提供读者以作参考，下表总结了这些操作的定义。

**符号说明**

以下所述的每个操作涉及两个参数  $a$  和  $b$ , 以及一个结果  $c$ 。如果参数表示一个标量就不会有下标。如果参数是一个向量, 就会有一个下标。如果参数是一个矩阵, 则会有两个下标。如果该操作定义在任意的数组上, 则参数会包含三个下标。如果某个特定的组合没有出现, 那么这种组合是无效的。

**加法 (+)**

表达式	结果
$a + b$	$c = a + b$
$a_i + b_i$	$c_i = a_i + b_i$
$a_{ij} + b_{ij}$	$c_{ij} = a_{ij} + b_{ij}$
$a_{ijk} + b_{ijk}$	$c_{ijk} = a_{ijk} + b_{ijk}$

**Subtraction (-)**

表达式	结果
$a - b$	$c = a - b$
$a_i - b_i$	$c_i = a_i - b_i$
$a_{ij} - b_{ij}$	$c_{ij} = a_{ij} - b_{ij}$
$a_{ijk} - b_{ijk}$	$c_{ijk} = a_{ijk} - b_{ijk}$

**乘法 (\* 和.\* )**

乘法运算符分为两种类型。第一种是正常的乘法运算符 \*。此运算符遵从线性代数里的一般数学定义, 如矩阵向量积等等。\* 操作符的行为如下表所示:

表达式	结果
$a * b$	$c = a * b$
$a * b_i$	$c_i = a * b_i$
$a * b_{ij}$	$c_{ij} = a * b_{ij}$
$a * b_{ijk}$	$c_{ijk} = a * b_{ijk}$
$a_i * b$	$c_i = a_i * b$
$a_{ij} * b$	$c_{ij} = a_{ij} * b$
$a_{ijk} * b$	$c_{ijk} = a_{ijk} * b$
$a_i * b_i$	$c = \sum_i a_i * b_i$
$a_i * b_{ij}$	$c_j = \sum_i a_i * b_{ij}$
$a_{ij} * b_j$	$c_i = \sum_j a_{ij} * b_j$
$a_{ik} * b_{kj}$	$c_{ij} = \sum_k a_{ik} * b_{kj}$

第二类乘法运算符是一种特殊的逐元素运算版本.\*。此版本不进行任何求和运算而简单地将运算逐元素施加在的所有数组元素上。

表达式	结果
$a .* b$	$c = a * b$
$a_i .* b_i$	$c_i = a_i * b_i$
$a_{ij} .* b_{ij}$	$c_{ij} = a_{ij} * b_{ij}$
$a_{ijk} .* b_{ijk}$	$c_{ijk} = a_{ijk} * b_{ijk}$

**除法 (/及./)**

正如乘法 (\* 和.\* ) (100), 除法运算符也分为两种。第一种是通常的除法运算符 /, 用于求数组除以一个标量值的商。下表总结了其行为:

表达式	结果
$a/b$	$c = a/b$
$a_i/b$	$c_i = a_i/b$
$a_{ij}/b$	$c_{ij} = a_{ij}/b$
$a_{ijk}/b$	$c_{ijk} = a_{ijk}/b$

此外，除法运算符也有一个逐元素的版本 $.$ 。其行为如下表所示：

表达式	结果
$a ./ b$	$c = a/b$
$a_i ./ b_i$	$c_i = a_i/b_i$
$a_{ij} ./ b_{ij}$	$c_{ij} = a_{ij}/b_{ij}$
$a_{ijk} ./ b_{ijk}$	$c_{ijk} = a_{ijk}/b_{ijk}$

### 幂指数 ( $\wedge$ 及 $\cdot\wedge$ )

和乘法 (\* 和  $\cdot*$ ) (100) 以及除法 (/ 及  $\cdot/$ ) (100) 一样，指数操作符也有两种形式。第一种是标准的指数操作符  $\wedge$ 。此标准版有两种不同的使用方式。第一种使用方法是两个标量的乘方运算（即  $a \wedge b$ ）。另外一种则是求方阵的标量次乘方（即  $a_{ij} \wedge b$ ）。

另外一种求幂的形式是由  $\cdot\wedge$  操作符代表的逐元素形式。其行为如下表所示：

表达式	结果
$a \cdot\wedge b$	$c = a^b$
$a_i \cdot\wedge b_i$	$c_i = a_i^{b_i}$
$a_{ij} \cdot\wedge b_{ij}$	$c_{ij} = a_{ij}^{b_{ij}}$
$a_{ijk} \cdot\wedge b_{ijk}$	$c_{ijk} = a_{ijk}^{b_{ijk}}$

### 相等 (=)

相等操作符 = 用于构造标量和数组方程，前提是只要两边的维数相等且每个维度的大小也一样。假设满足这一要求，那么操作符会逐元素地作用在数组上。这意味着操作符是应用在左边和右边的每个对应元素上。

### 赋值 (:=)

$:=$  (赋值) 操作符与相等 (=) (101) 操作符一样，均是逐元素地作用在数组上的。

### 关系操作符

所有关系操作符 (and、or、not、>、 $\geq$ 、 $\leq$ 、 $<$ ) 与相等 (=) (101) 操作符一样，均是逐元素地作用在数组上的。

transpose

transpose 函数接受一个矩阵作为参数，并返回该矩阵的转置版本。

outerProduct

outerProduct 函数有两个参数。每个参数均必须是矢量，且须具有相同的尺寸。函数将返回其表示两向量外积的矩阵。从数学上来说，假设  $a$  与  $b$  是相同大小的矢量。调用 outerProduct( $a, b$ ) 会返回有如下元素的矩阵  $c$ :

$$c_{ij} = a_i * b_j$$

`symmetric`

`symmetric` 函数接受一个方阵作为参数。函数返回一个相同大小的矩阵，其对角线下的元素被替换为原矩阵对角线上元素组成矩阵的转置。换句话说，

$$b_{ij} = \text{transpose}(a) = \begin{cases} a_{ij} & \text{if } i \leq j \\ a_{ji} & \text{otherwise} \end{cases}$$

`skew`

`skew` 函数用具有三个分量的矢量作为输入，并返回以下斜对称矩阵：

$$\text{skew}(x) = \begin{bmatrix} 0 & -x_3 & x_2 \\ x_3 & 0 & -x_1 \\ -x_2 & x_1 & 0 \end{bmatrix}$$

`cross`

`cross` 函数用两个矢量（各有 3 个分量）作为输入，并返回以下矢量（有 3 个分量）：

$$\text{cross}(x, y) = \begin{Bmatrix} x_2y_3 - x_3y_2 \\ x_3y_1 - x_1y_3 \\ x_1y_2 - x_2y_1 \end{Bmatrix}$$

## 缩减运算符

缩减运算符把数组缩减为标量值。

`min`

`min` 函数接受一个数组为参数，并返回数组中的最小值。例如：

```
min({10, 7, 2, 11}) // 2
min([1, 2; 3, -4]) // -4
```

`max`

`max` 函数接受一个数组为参数，并返回数组中的最大值。例如：

```
max({10, 7, 2, 11}) // 11
max([1, 2; 3, -4]) // 3
```

`sum`

`sum` 函数接受一个数组为参数，并返回该数组中所有元素的总和。例如：

```
sum({10, 7, 2, 11}) // 30
sum([1, 2; 3, -4]) // 2
```

product

product 函数接受一个数组作为参数，并返回数组中所有元素的乘积。例如：

```
product({10, 7, 2, 11}) // 1540
product([1, 2; 3, -4]) // -24
```

## 其它函数

ndims

ndims 函数接受一个数组作为参数，并返回该数组的维数。例如：

```
ndims({10, 7, 2, 11}) // 1
ndims([1, 2; 3, -4]) // 2
```

size

size 函数可以通过两种不同方式调用。第一种方法是，使用一个数组作为唯一参数。在这种情况下，size 返回一个矢量。这个矢量中各分量分别对应于数组中相应维度的大小。例如：

```
size({10, 7, 2, 11}) // {4}
size([1, 2, 3; 3, -4, 5]) // {2, 3}
```

另外，也可以通过附加可选参数的方法调用 size，从而指定所需的特定维度标号。这种情况下，函数会将该特定维度的大小作为一个标量整数返回。例如：

```
size({10, 7, 2, 11}, 1) // 4
size([1, 2, 3; 3, -4, 5], 1) // 2
size([1, 2, 3; 3, -4, 5], 2) // 3
```

## 向量化

本节中，我们已经讨论了 Modelica 语言里许多为了数组参数而设计的函数。不过，一个很常见的用法是将一个函数逐元素应用到向量的每个元素上。Modelica 的“向量化（vectorization）”语言特性支持这种用例。如果一个函数设计为输入一个标量，但传值却为一个数组，那么 Modelica 语言编译器会自动将函数应用在向量的每个元素上。

为了了解这个功能，首先考虑 abs 函数的正常求值：

```
abs(-1.5) // 1.5
```

显然，abs 通常是接受标量参数，然后返回一个标量。但在 Modelica 里，我们也可以这样做：

```
abs({0, -1, 1, -2, 2}) // {0, 1, 1, 2, 2}
```

由于该函数是为变量设计的，Modelica 语言编译器会将其从：

```
abs({0, -1, 1, -2, 2})
```

变换为

```
{abs(0), abs(-1), abs(1), abs(-2), abs(2)}
```

换句话说，Modelica 把函数应用于由标量组成的向量这种情况，变换为输入标量后函数的取值所组成的向量。

**此特性也适用多个参数的函数**前提是仅仅 1 个预期为标量的参数为向量。要理解这个稍微复杂的功能，可以考虑取模函数 mod。如果将函数应用于标量输入，我们会得到以下行为：

```
mod(5, 2) // 1
```

倘若将首个参数改为向量，我们会得到：

```
mod({5, 6, 7, 8}, 2) // {1, 0, 1, 0}
```

换言之，这个特性会将

```
mod({5, 6, 7, 8}, 2)
```

变换为

```
{mod(5,2), mod(6,2), mod(7,2), mod(8,2)}
```

不过，向量化特性不能用在有多于一个标量参数被改为矢量的情况。例如，下面的表达式将是错的：

```
mod({5, 6, 7, 8}, {2, 3}) // Illegal
```

因为 mod 需要两个标量参数，但得到的却是两个矢量参数。

### 第 3.2.4 节 数组索引

我们已经在本章看到很多说明如何索引数组的例子。因此，似乎没有必要专门用一节来讨论如何索引数组。的确，通常你只会使用在使用整数值下标以索引数组中的元素。但也有其它不少索引数组的方式，从而值得花一些时间来讨论。

索引

整数

#### 数组索引从 1 开始

在用整数指定数组维度时，Modelica 使用的索引是从 1 开始的。某些语言中选择使用零作为起始索引，但要指出 Modelica 是使用从 1 开始的索引。

很明显，最直接的方法来是使用整数去索引数组。数组声明如下：

```
Real x[5,4];
```

我们可以在首个下表使用 1 到 5 之间的整数、在第二个下标使用 1 到 4 之间的整数去索引数组的元素。但值得指出的是，Modelica 允许用向量作为下标。要理解向量指标是如何工作的，首先要考虑以下矩阵：

$$B = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

在 Modelica 语言，这样的数组将被声明如下：

```
parameter Real B[3,3] = [1, 2, 3; 4, 5, 6; 7, 8, 9];
```

想象，我们要从 B 提取一个如下的子矩阵：

```
parameter Real C[2,2] = [B[1,1], B[1,2]; B[2,1], B[2,2]]; // [1, 2; 4, 5];
```

用如下方法，我们可以更容易地用向量下标提取相同的子矩阵：

```
parameter Real C[2,2] = B[{1,2},{1,2}]; // [1, 2; 4, 5];
```

通过使用向量下标，我们可以提取或构造任意子数组。这就是范围表示法（95）派上用场的时候了。同一个提取方法也可以用以下方法表示：

```
parameter Real C[2,2] = B[1:2,1:2]; // [1, 2; 4, 5];
```

## 枚举类

在前面的化学系统（89）例子里，我们看到了如何用枚举类来指定数组的维度。然后，我们看到了如何通过一个 enumeration 类型的值来索引数组。一般来说，对于一个如下的 enumeration 值：

```
type Species = enumeration(A, B, X);
```

然后声明一个用 enumeration 指代维数的数组，即：

```
Real C[Species];
```

那么我们可以使用枚举值 Species.A、Species.B 以及 Species.X 作为索引。例如：

```
equation
  der(C[Species.A]) = ...;
```

## 布尔值

我们可以用 enumeration 类型相同的当时使用 Boolean 类型。考虑声明有某一维定义为 Boolean 类型的数组：

```
Real C[5,Boolean];
```

然后我们可以使用布尔值索引该维度，如：

```
equation
  der(C[1,true]) = ...;
  der(C[1,false]) = ...;
```

end

当指定数组下标时，在下标表达式里使用 end 是合法使用的。在这种情况下，end 对应数组维度的最大可能值。在表达式中，end 允许用最后一个而不是第一个元素为基准进行索引。例如，要指代向量中的倒数第二个元素，那么可以用 end-1 表达式作为下标。

请记住，end 取相应的数组维度的最大可能索引值。因此，对于下面的数组：

```
Integer B[2,4] = [1, 2, 3, 4; 5, 6, 7, 8];
```

下面的表达式计算结果如下：

```
B[1,end] // 4
B[end,1] // 5
B[end,end] // 8
B[2,end-1] // 7
```

## 切片

Modelica 语言中有另一种复杂的数组索引方式。但现在没有讨论的必要。在我们稍后讨论组件数组（280）的时候，便会看到这个功能。

### 第 3.2.5 节 循环

for

数组的主要用途之一是允许代码通过使用循环得到简化。因此，在结束关于数组的本章前，我们将介绍一些基本的循环结构，以及循环是如何与数组功能相结合的。

一般而言，`for` 关键字用以表示循环。但 `for` 也可以用在很多不同的情景里。本章中内的数个例子使用了 `for` 来生成方程组。`for` 用在等式区域内时，每个在 `for` 循环内的方程都会对应每一个循环变量的取值再生成相应的方程。用这种方式，我们可以很容易地生成具有相同总体结构、而仅仅是循环索引变量值不同的一组方程。`for` 循环在等式区域内的一般语法是：

```
equation
  for i in 1:n loop
    // equations
  end for;
```

注意，循环索引变量（如这里的 `i`）**并不需要被声明**。还要注意这些变量仅仅在 `for` 循环的范围内存在（而在循环的前后均不存在）。

`for` 循环当然可以嵌套在自身之中。例如：

```
equation
  for i in 1:n loop
    for j in 1:n loop
      // equation
    end for;
  end for;
```

`for` 循环还可以出现在其它区域里。例如，`for` 循环可以出现在 `initial equation` 区域里，或者 `algorithm` 区域（79）。

另一种可以看到 `for` 关键字的情形是在我们关于[数组解析](#)（96）的讨论里。在这种情况下，`for` 结构没有用于生成方程或语句，而是在填充在数组中的各个元素。

while

Modelica 语言还有另一种类型的循环，那就是 `while` 循环。`while` 循环在 Modelica 里并不常用。其原因在于，不同于通用的语言，Modelica 是一种面向等式的语言。此外，Modelica 规定了一个模型应该包括相等数量的方程和未知数。这样的模型被认为是“平衡的模型”。

之所以说 `while` 结构不常用，是因为平衡的模型要求等式数目（对于编译器）是可预测的。因为 `for` 循环有界而且下标变量值的数目总是已知的（因为下标变量总是来自于可能值组成的向量），所以产生方程的数量也总是已知的。对于 `while` 循环就不能下同样的结论了。因此，`while` 循环仅仅在 `algorithm` 区域（79）里有用处（一般在[函数](#)（107）定义里）。

## 函数

到现在为止，我们已经在前面的例子中使用了 Modelica 语言的许多内置函数。尤其是在我们对数组函数 (97) 的讨论里。虽然 Modelica 对于很多常见运算包含了广泛一系列函数，但用户仍然需要有办法去自定义函数。本章的中心议题是定义函数所需的流程。

和往常一样，我们开始时将讨论几个例子。其目的是让用户明白，我们为何需要自定义函数。然后，我们会回顾 Modelica 用户自定义函数的重要组成部分。

### 第 4.1 节 示例

#### 第 4.1.1 节 多项式计算

我们的第一个例子将以使用函数计算多项式为中心。这将帮助我们理解函数定义和使用的基础知识。

##### 计算直线

##### 数学背景

在能处理任意阶多项式之前，让我们首先思考一下我们如何使用函数计算一条直线上的点。在数学上，我们想定义一个如下所示的函数：

$$y(x, x_0, y_0, x_1, y_1)$$

其中  $x$  是自变量， $(x_0, y_0)$  是定义直线的一个点， $(x_1, y_1)$  是定义直线的另一个点。数学上，这种函数可以被定义为如下形式：

$$y(x, x_0, y_0, x_1, y_1) = x \frac{y_1 - y_0}{x_1 - x_0} - \frac{x_1 + x_0}{2(x_1 - x_0)} \frac{y_1 - y_0}{2} + \frac{y_1 + y_0}{2}$$

为了减少参数数量，让我们假设  $x_0$  和  $y_0$  代表一个单一的点，以矢量  $\vec{p}_0$  表示， $x_1$  和  $y_1$  代表另一个单一的点，以矢量  $\vec{p}_1$  表示，因此现在函数调用如如下所示：

$$\text{Line}(x, \vec{p}_0, \vec{p}_1)$$

##### Modelica 描述

现在的问题是我们如何将这种数学关系转换为可以在 Modelica 模型中调用的函数。为了做到这一点，我们必须定义一个新的 Modelica 函数。

事实证明函数定义与模型定义模型 定义 (25) 非常相似（至少在语法层面）。这里我们展示的是在 Modelica 中定义的 Line 函数：

```

function Line "Compute coordinates along a line"
  input Real x    "Independent variable";
  input Real p0[2] "Coordinates for one point on the line";
  input Real p1[2] "Coordinates for another point on the line";
  output Real y    "Value of y at the specified x";
algorithm
  y := x*(p1[2]-p0[2])/(p1[1]-p0[1])+
    (p1[2]+p0[2]-(p1[1]+p0[1])*(p1[2]-p0[2]))/
    (p1[1]-p0[1])/2.0;
end Line;

```

函数的所有输入参数均以 `input` 限定词作为前缀。函数的输出结果以 `output` 限定词作为前缀。函数主体为 `algorithm` 区域。返回值（本例中是 `y`）通过 `algorithm` 区域计算得到。

所以在本例中，`output` 值 `y` 是根据 `input` 值 `x`、`p0`、`p1` 计算得到。注意，此函数中没有 `return` 语句。函数将自动返回 `algorithm` 区域计算得到的 `output` 变量的结果。

有几个在前面几章已经讨论过的事情需要注意。首先，注意函数本身和参数的说明字符串。它们在记录函数和参数的用法方面很有意义。同时需要注意如何使用数组来表达一个二维向量和这些数组在本例中是如何索引的。

`Line` 模型唯一不足之处在于计算 `y` 表达式的长度。如果我们可以拆分表达式，则就能改善这种不足。

## 中间变量

为了简化 `y` 的表达式，我们需要引入一些中间变量。我们已经看到 `x`、`p0`、`p1` 是我们在函数中使用的变量。我们想引入额外的变量。但是，同时这些变量也不应是参数。换句话说，引入的变量值必须在函数“内部”进行计算。为了实现这个目标，我们创建了一个受保护（protected）变量的集合。这些变量可认为是在函数内部计算得到的。下面的例子展示了使用 `protected` 来声明和计算两个内部变量：

```

function LineWithProtected "The Line function with protected variables"
  input Real x    "Independent variable";
  input Real p0[2] "Coordinates for one point on the line";
  input Real p1[2] "Coordinates for another point on the line";
  output Real y    "Value of y at the specified x";
protected
  Real m = (p1[2]-p0[2])/(p1[1]-p0[1])      "Slope";
  Real b = (p1[2]+p0[2]-m*(p1[1]+p0[1]))/2.0 "Offset";
algorithm
  y := m*x+b;
end LineWithProtected;

```

此模型引入了两个新变量。其中一个变量 `m` 代表了直线的斜率。另一个变量 `b` 代表了当条件为 `x=0` 时的返回值。通过计算这两个中间变量，使得计算 `y` 的表达式变成了非常容易识别的形式：`y := m*x+b`。

## 计算多项式结果

### 数学定义

当然，本节我们的目标是创建一个函数，以此计算任意的多项式。到目前为止，我们已经看到了一些基本的函数。在此基础上让我们继续朝着最终目标前进。我们会构建一个函数，其调用方式如下：

$$p(x, \vec{c})$$

其中，`x` 依然代表独立变量。 $\vec{c}$  代表系数组成的向量。这样我们的多项式以如下方式计算：

$$p(x, \vec{c}) = \sum_{i=1}^N c_i x^{N-i}$$

其中， $N$  是传递给函数的系数的个数。在这一点上有两件重要的事情需要注意。首先， $\vec{c}$  的第一个元素对应于多项式中的最高阶次项。然后，我们通过使用符号并假设  $\vec{c}$  的元素是从 1 开始编号。以此使函数能更容易地转化为 Modelica 代码（其数组索引从 1 开始）。

注意，上文中  $p$  的定义非常容易阅读和理解。但是在处理有限精度的浮点数时，使用递归的方法来计算多项式会更高效和更精确。对于一个 4 阶多项式，其表达式为：

$$p(x, \vec{c}) = ((c_1 x + c_2) x + c_3) x + c_4$$

这种方法相对高效，因为它仅用到了简单的乘法和加法操作，避免了处理更复杂的求幂运算。同时，这种方法更为精确。因为使用有限精度浮点表示形式时，求幂运算更容易引发舍入误差和截尾误差。

### Modelica 定义

到目前为止，我们已经精确定义了我们希望函数执行的计算方法。剩下的工作只是在 Modelica 中定义函数。本例中，我们的多项式计算函数在 Modelica 中描述如下：

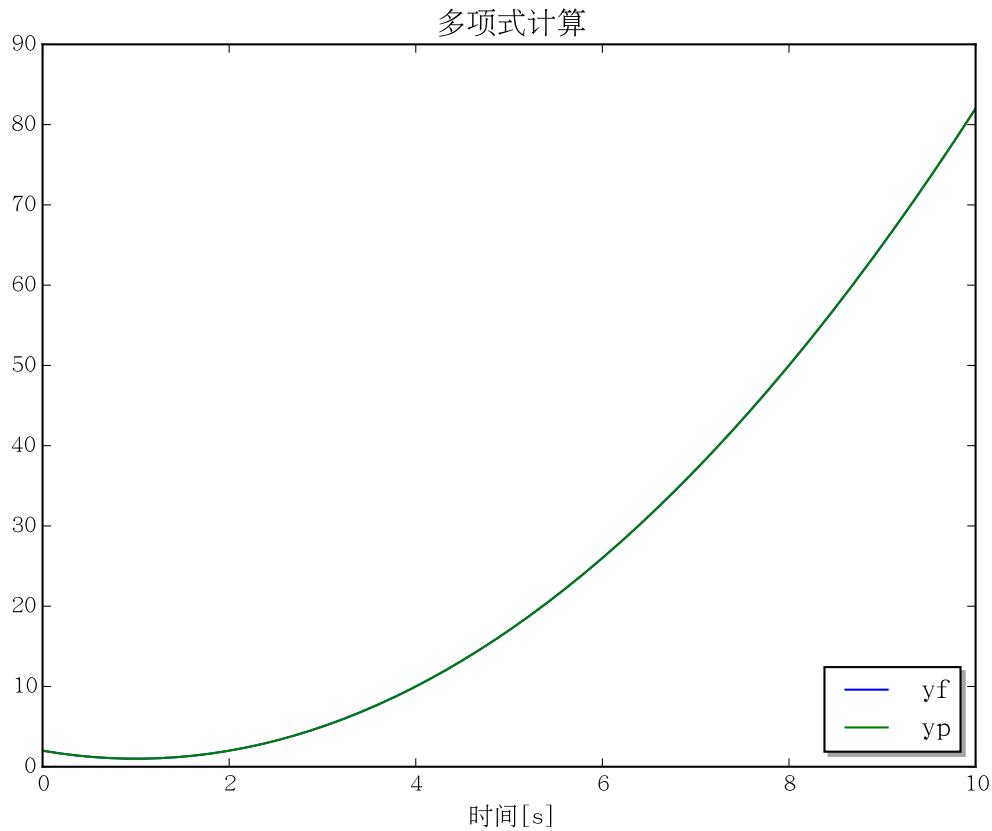
```
function Polynomial "Create a generic polynomial from coefficients"
    input Real x    "Independent variable";
    input Real c[:] "Polynomial coefficients";
    output Real y   "Computed polynomial value";
protected
    Integer n = size(c,1);
algorithm
    y := c[1];
    for i in 2:n loop
        y := y*x + c[i];
    end for;
end Polynomial;
```

回顾一下，函数所有的（输入）参数都有 `input` 限定词，返回值都有 `output` 限定词。与前面的例子一样，我们定义了一个中间变量  $n$ ，以此提供一种简便的方法去查阅系数向量的长度。本例中，我们也展示了如何使用一个 `for` 循环来描述任意阶数的多项式的递归计算。

让我们通过在模型中使用这个函数，以此验证其正确性。使用的 Modelica 模型如下：

```
model EvaluationTest1 "Model that evaluates a polynomial"
    Real yf;
    Real yp;
equation
    yf = Polynomial(time, {1, -2, 2});
    yp = time^2-2*time+2;
end EvaluationTest1;
```

记住， $c$  的第一个元素对应最高阶项。如果我们把多项式的直接计算结果  $yp$  和使用我们函数计算的结果  $yf$  作对比，我们发现计算结果相等：



## 微分

这个多项式所计算的量完全可能最终被 Modelica 编译器求微分。下面的例子虽然有些牵强，但是它展示了一个多项式在什么情况下会在一个模型中被微分：

```
model Differentiation1 "Model that differentiates a function"
  Real yf;
  Real yp;
  Real d_yf;
  Real d_yp;
equation
  yf = Polynomial(time, {1, -2, 2});
  yp = time^2-2*time+2;
  d_yf = der(yf); // How to compute?
  d_yp = der(yp);
end Differentiation1;
```

本例中我们使用了与上述例子相同的 `yf` 和 `yp` 方程。`yf` 使用 `Polynomial` 计算，`yp` 直接使用多项式计算。我们还额外添加了两个变量，`d_yf` 和 `d_yp`。它们分别代表 `yf` 和 `yp` 的导数。如果我们尝试编译此模型，编译器很有可能会引发一个关于 `d_yf` 方程的错误。其原因是编译器无法计算 `yf` 的导数。这是因为与通过简单表达式计算的 `yp` 不同，我们在 `Polynomial` 函数的背后隐藏了计算 `yf` 的细节。通常来说，Modelica 工具不会从函数实现推导其导数。即使编译器做到了这点，如何确定任意算法的导数依然很不容易。

因此，接下来的问题就是我们应如何处理这种情况？这不会使得在函数在模型中变得难以使用吗？幸运的是，Modelica 给我们提供了一个方法来指定函数导数的计算。这就是通过在函数定义时添加一种叫做 annotation 的语句来实现的。

## 标注

标注 (annotation) 是一段元数据。它不直接描述函数的行为（也就是说它不影响函数的返回值）。然而，Modelica 编译器会使用标注以得到“提示”，以此知道如何处理某些特定的情况。标注语句通常 是“可选”的信息。这就意味着即使提供了标注，工具也不会被强制要求使用这些信息。Modelica 语义定义了许多标准的标注语句，以便 Modelica 工具能直接解析。

在本例中，我们需要 derivative 标注。因为此标注可以让我们告诉 Modelica 编译器如何计算函数的导数。为了实现这一点，我们定义了一个新函数，PolynomialWithDerivative 用于计算，如下所示：

```
function PolynomialWithDerivative
  "Create a generic polynomial from coefficients (with derivative information)"
  input Real x    "Independent variable";
  input Real c[:] "Polynomial coefficients";
  output Real y   "Computed polynomial value";
protected
  Integer n = size(c,1);
algorithm
  y := c[1];
  for i in 2:n loop
    y := y*x + c[i];
  end for;
  annotation(derivative=PolynomialFirstDerivative);
end PolynomialWithDerivative;
```

注意，这个函数除了高亮显示的行以外，与之前的函数是一致的。换句话说，我们只需要把这行代码添加到我们的函数中：

```
annotation(derivative=PolynomialFirstDerivative);
```

我们通过在函数内添加这行代码以此向 Modelica 编译器解释如何计算这个函数的导数。这行代码的含义是函数 PolynomialFirstDerivative 应该用于计算 PolynomialWithDerivative 的导数。

在讨论函数 PolynomialFirstDerivative 的执行之前，我们首先要从数学角度明白什么是必需的。回忆我们之前定义的多项式插值函数：

$$p(x, \vec{c}) = \sum_{i=1}^N c_i x^{N-i}$$

注意  $p$  有两个参数。如果我们希望  $p$  对任意的变量  $z$  求导，我们可以使用链式法则来表示  $p$  对  $z$  的全微分，如下所示：

$$\frac{dp(x, \vec{c})}{dz} = \frac{\partial p}{\partial x} \frac{dx}{dz} + \frac{\partial p}{\partial \vec{c}} \frac{d\vec{c}}{dz}$$

我们可以从最初  $p$  的定义中导出如下关系。首先，对于  $p$  对  $x$  的偏导数，我们可以得到：

$$\frac{\partial p}{\partial x} = p(x, c')$$

其中  $c'$  定义如下：

$$c'_i = (N - i)c_i$$

其次，对于  $p$  对  $\vec{c}$  的偏导数，我们可以得到：

$$\frac{\partial p}{\partial c_i} = p(x, \vec{d}_i)$$

其中向量  $\vec{d}_i$  是  $N \times N$  的单位矩阵的第  $i$  列。

事实表明，出于效率的原因，Modelica 编译器提供的  $\frac{dx}{dz}$  和  $\frac{d\vec{c}}{dz}$  要好于提供函数去计算  $\frac{\partial p}{\partial x}$  和  $\frac{\partial p}{\partial c_i}$  的函数。因此从数学角度来说，Modelica 编译器需要的是一个可以用以下参数进行调用的新函数：

$$df(x, \vec{c}, \frac{dx}{dz}, \frac{d\vec{c}}{dz})$$

因此:

$$df(x, \vec{c}, \frac{dx}{dz}, \frac{d\vec{c}}{dz}) = \frac{df}{dz}$$

为此, derivative 标注必须指定一个与  $df$  参数相同的函数。在我们的例子中, 函数 PolynomialFirstDerivative 将被定义为如下形式:

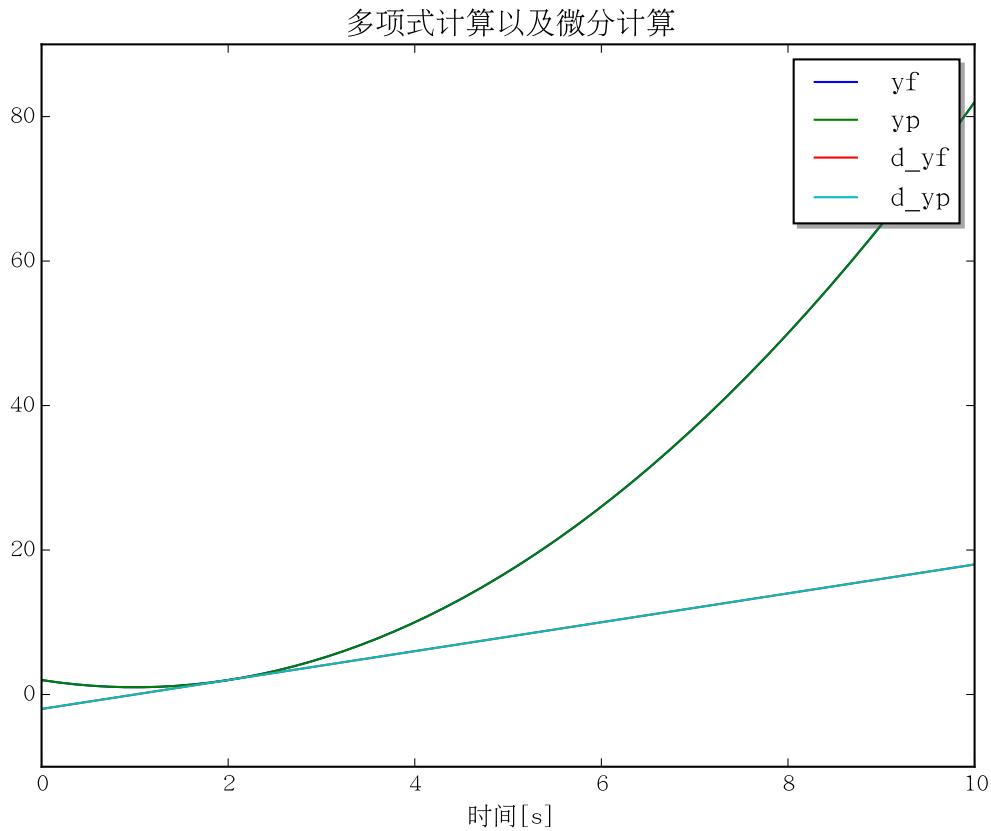
```
function PolynomialFirstDerivative
  "First derivative of the function Polynomial"
  input Real x;
  input Real c[:];
  input Real x_der;
  input Real c_der[size(c,1)];
  output Real y_der;
protected
  Integer n = size(c,1);
  Real c_diff[n-1] = {(n-i)*c[i] for i in 1:n-1};
algorithm
  y_der := PolynomialWithDerivative(x, c_diff)*x_der +
    PolynomialWithDerivative(x, c_der);
end PolynomialFirstDerivative;
```

请注意我们最初函数的参数如何被复制变成 (预期数量的) 两倍那么多。第二组参数分别代表了  $\frac{dx}{dz}$  和  $\frac{d\vec{c}}{dz}$  这些量。注意, 这里假设  $z$  是一个标量。因此输入参数的类型是相同的。利用我们关于多项式的偏导数的知识, 导数的计算就可通过同一个多项式计算函数来完成。

我们可以通过以下的模型来使用所有的这些函数:

```
model Differentiation2 "Model that differentiates a function using derivative annotation"
  Real yf;
  Real yp;
  Real d_yf;
  Real d_yp;
equation
  yf = PolynomialWithDerivative(time, {1, -2, 2});
  yp = time^2-2*time+2;
  d_yf = der(yf);
  d_yp = der(yp);
end Differentiation2;
```

对这个模型进行仿真并且比较结果, 我们看到参数  $yf$  和  $yp$  的对比结果, 以及  $d_yf$  和  $d_yp$  的对比结果, 如下:



### 第 4.1.2 节 插值

本节中，我们将通过示例以不同的方法实现一个简单的一维插值法。我们首先会通过完全使用 Modelica 来实现一维插值法，然后展示一个 Modelica 结合 C 语言的替代方案。最后，将讨论这两种方案的优缺点。

#### Modelica 实现

##### 函数定义

在这个例子中，我们假定插值数据的格式如下：

自变量 $x$	因变量 $y$
$x_1$	$y_1$
$x_2$	$y_2$
$x_3$	$y_3$
...	...
$x_n$	$y_n$

这里我们假设  $x_i < x_{i+1}$ 。

利用以上数据，对于我们某个感兴趣的自变量  $x$  值，我们的函数应对  $y$  赋一个内插值。这样的函数在 Modelica 中实现如下：

```
function InterpolateVector "Interpolate a function defined by a vector"
  input Real x      "Independent variable";
  input Real ybar[;,2] "Interpolation data";
  output Real y     "Dependent variable";
```

```

protected
  Integer i;
  Integer n = size(ybar,1) "Number of interpolation points";
  Real p;
algorithm
  assert(x>=ybar[1,1], "Independent variable must be greater than "+String(ybar[1,1]));
  assert(x<=ybar[n,1], "Independent variable must be less than "+String(ybar[n,1]));
  i := 1;
  while x>=ybar[i+1,1] loop
    i := i + 1;
  end while;
  p := (x-ybar[i,1])/(ybar[i+1,1]-ybar[i,1]);
  y := p*ybar[i+1,2]+(1-p)*ybar[i,2];
end InterpolateVector;

```

接下来，我们会一步步地对上述函数功能进行讲解。首先，从变量声明部分开始：

```

input Real x      "Independent variable";
input Real ybar[:,2] "Interpolation data";
output Real y     "Dependent variable";

```

input 变量 x 表示插值函数求解的独立变量值。变量 ybar 表示插值数据。output 变量 y 表示利用插值函数求得的内插值。函数包含的下一部分内容如下所示：

```

protected
  Integer i;
  Integer n = size(ybar,1) "Number of interpolation points";
  Real p;

```

上述函数定义中也包含了各种 protected 变量的声明。在多项式计算 (107) 示例中可以看到，函数内部使用了很多有效的中间变量。上述例子中，变量 i 表示索引变量，变量 n 表示插值数据点的个数，变量 p 表示插值函数使用的权重系数。

当完成所有的变量声明后，接下来需要定义的是函数的 algorithm 区域，如下所示：

```

algorithm
  assert(x>=ybar[1,1], "Independent variable must be greater than "+String(ybar[1,1]));
  assert(x<=ybar[n,1], "Independent variable must be less than "+String(ybar[n,1]));
  i := 1;
  while x>=ybar[i+1,1] loop
    i := i + 1;
  end while;
  p := (x-ybar[i,1])/(ybar[i+1,1]-ybar[i,1]);
  y := p*ybar[i+1,2]+(1-p)*ybar[i,2];

```

上述部分的前两个语句为 assert 声明，用于验证变量 x 的值是否在插值区间  $[x_1, x_n]$  的范围内。如果变量不在上述区间，语句则会生成相应的错误信息。

函数的其余部分则用于查找满足表达式  $x_i \leq x < x_{i+1}$  的 i 值。当变量 i 的值确定后，变量 x 对应的数值可以通过下述插值表达式进行计算：

$$y = p \bar{y}_{i+1,2} + (1 - p) \bar{y}_{i,2}$$

其中

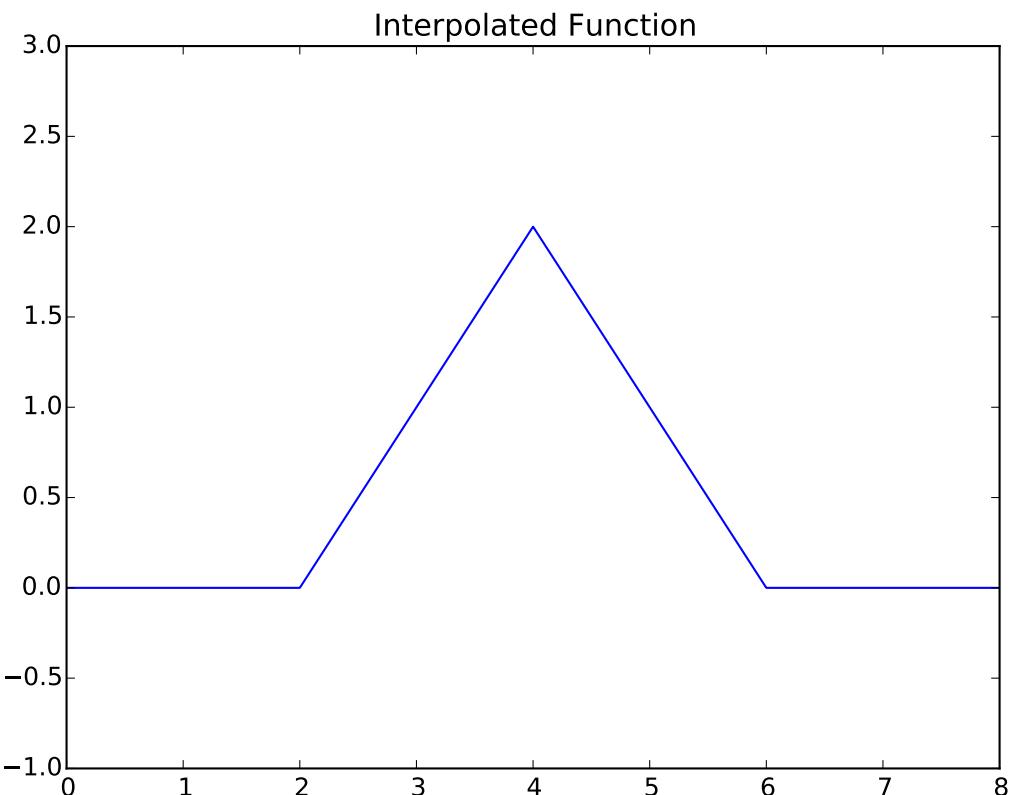
$$p = \frac{x - \bar{y}_{i,1}}{\bar{y}_{i+1,1} - \bar{y}_{i,1}}$$

### 测试用例

现在，我们通过编写一个模型来测试上述函数。作为一个简单的测试用例，我们将对插值函数的返回值进行积分。我们将使用下列数据作为插值函数的基础数据：

$x$	$y$
0	0
2	0
4	2
6	0
8	0

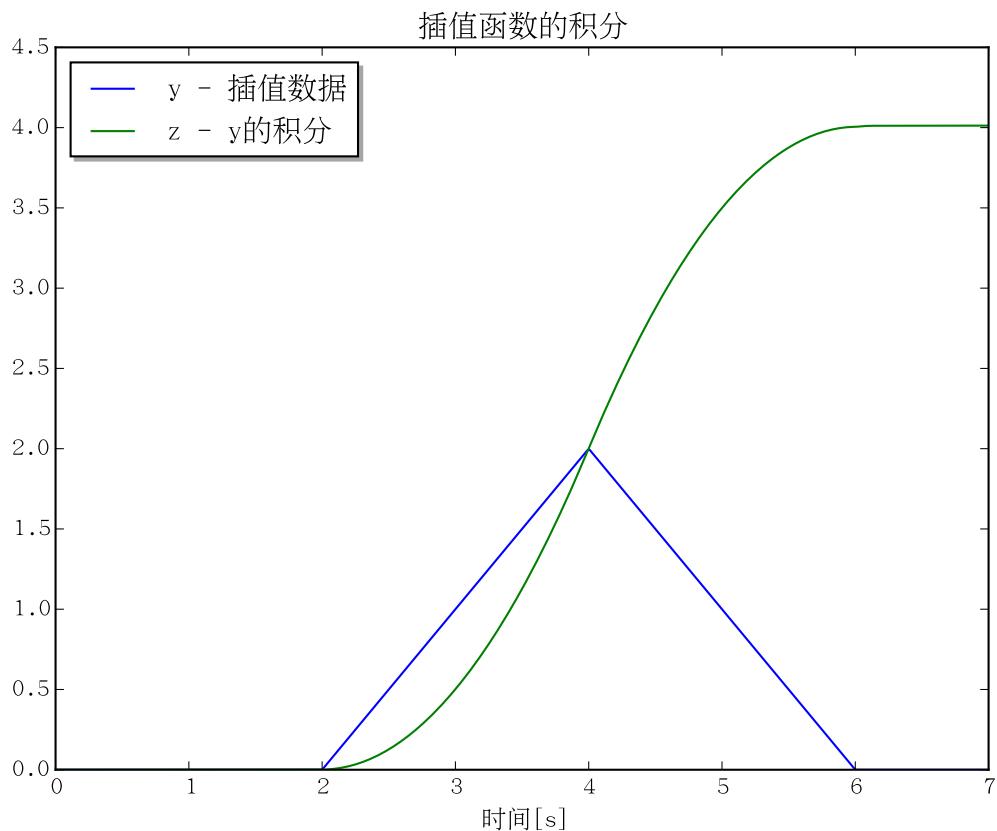
如果绘制这些数据，所使用的插值函数如下图所示：



在下述测试模型中，我们令独立变量  $x$  等于变量 time。插值函数根据样本数据计算输出变量  $y$  的值。然后通过积分  $y$  值，计算得到变量  $z$  的值。

```
model IntegrateInterpolatedVector "Exercises the InterpolateVector"
  Real x;
  Real y;
  Real z;
equation
  x = time;
  y = InterpolateVector(x, [0.0, 0.0; 2.0, 0.0; 4.0, 2.0; 6.0, 0.0; 8.0, 0.0]);
  der(z) = y;
  annotation (experiment(StopTime=6));
end IntegrateInterpolatedVector;
```

上述模型的仿真结果如下图所示：



上述插值方法有几个缺点。首先，需要将数据传递到函数所有需求的地方。另外，对于多维插值函数，所需的数据可能会非常复杂（例如不规则网格）和庞大。因此，以 Modelica 源代码的方式进行数据存储将会非常不方便，可以采用其他方式进行数据存储，例如使用外部文件。但是，如果使用外部文件进行插值数据的填充，需使用 ExternalObject 类型。

### 使用 ExternalObject 类型

ExternalObject 类型主要是指定未(或不必)通过 Modelica 源代码体现的信息。这主要用于指定 Modelica 源代码以外的数据或状态。这些信息可以是插值函数所用的数据，也可以是其他软件的相关状态。

### 测试用例

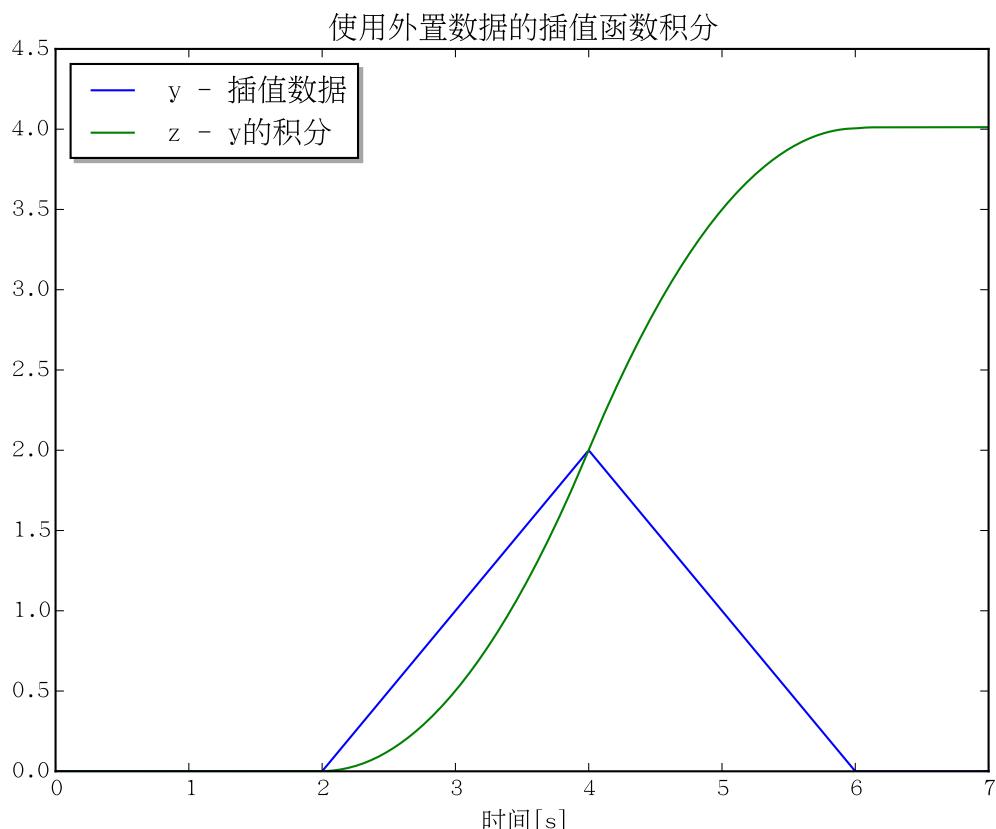
在下述例子中，将围绕上面的测试用例进行讲述。在上述应用背景下，可以更清晰的展示如何使用 ExternalObject 类型。测试用例的 Modelica 源代码如下所示：

```
model IntegrateInterpolatedExternalVector
  "Exercises the InterpolateExternalVector"
  parameter VectorTable vector = VectorTable(ybar=[0.0, 0.0;
                                                 2.0, 0.0;
                                                 4.0, 2.0;
                                                 6.0, 0.0;
                                                 8.0, 0.0]);
  Real x;
  Real y;
  Real z;
equation
  x = time;
  y = InterpolateExternalVector(x, vector);
  der(z) = y;
```

```
annotation (experiment(StopTime=6));
end IntegrateInterpolatedExternalVector;
```

上述测试用例与之前测试用例的主要区别在于：插值数据并不是直接传递给插值函数，而是在模型中创建了一个 VectorTable 类型的变量 vector，然后进行数据传递的。对于 VectorTable 类型，我们可以暂且把它们看作插值数据。在后面的章节中，我们将对其进行详细讨论。除了使用 InterpolateExternalVector 函数调用插值函数，以及创建 vector 变量，以替代之前的原始插值数据进行参数传递以外，上述模型的其他部分与之前的测试模型基本是相同的。

对上述测试模型进行仿真，其结果与之前模型的仿真结果是一致的，如下图所示：



### ExternalObject 类型定义

要明白上述模型的实现，我们首先需要了解如何定义 VectorTable 类型。如前所述，VectorTable 是一种 ExternalObject 类型。在 Modelica 语言中，它用以代表一种特殊的类型。这通常被称为“隐形”指针。这也就意味着，ExternalObject 类型主要表示那些不能直接（通过 Modelica）访问的数据。

在本例里，我们通过下列方式定义 VectorTable 类型：

```
type VectorTable "A vector table implemented as an ExternalObject"
  extends ExternalObject;
  function constructor
    input Real ybar[:,2];
    output VectorTable table;
    external "C" table=createVectorTable(ybar, size(ybar,1))
    annotation(IncludeDirectory="modelica://ModelicaByExample.Functions.Interpolation/source",
               Include="#include \"VectorTable.c\"");
  end constructor;

  function destructor "Release storage"
    input VectorTable table;
  end destructor;
```

```

external "C" destroyVectorTable(table)
annotation(IncludeDirectory="modelica://ModelicaByExample.Functions.Interpolation/source",
           Include="#include \"VectorTable.c\"");
end destructor;
end VectorTable;

```

注意，VectorTable 类型与 ExternalObject 类型是继承关系。在 ExternalObject 类型定义时，可以同时包含两个特殊的函数：constructor 函数（构造函数）和 destructor 函数（析构函数）。下面将对其进行详细讲述。

**构造函数** 当创建 VectorTable 类型时，将调用 constructor 函数（例如在上述测试用例中，对 vector 变量的声明）。此时，constructor 函数主要用于初始化模型中定义的隐形指针。初始化过程中的所有数据都应以参数的形式传递给 constructor 函数。在实例化过程中，相同的数据也必须存在（例如 vector 变量声明时定义的 data 参数）。

不同于之前的示例，constructor 函数的定义有些特殊。它并不包括 algorithm 区域。通常，algorithm 区域是用来计算函数返回值的。相反，constructor 函数包含一个 external 声明，用以表明所定义的函数是通过 Modelica 以外的语言实现的。在上述示例中，实现的语言是 C（在 external 关键字后以”C”标明）。而且 table 变量（函数的 output，也同时是表示定义的隐形指针）其返回值（变量 ybar 的值及其数组长度）是通过 C 语言定义的 createVectorTable 函数实现的。

紧随 createVectorTable 函数其后的是 annotation。此注解的功能是告诉 Modelica 编译器在哪里可以找到外部 C 函数的源代码。

至关重要的一点是，从 Modelica 编译器的角度来看，VectorTable 变量只是 createVectorTable 函数返回的一个隐形指针。仅仅通过 Modelica 并不能访问这些指针指向的数据。在下面的例子中，我们可以看到，这些指针可以传递给其他由 C 语言实现的函数，并且可以以此访问变量 VectorTable 所代表的数据。

**析构函数** 当 ExternalObject 类型不再需要时，将调用 destructor 函数。这使得 Modelica 编译器可以清除 ExternalObject 类型运行所占用的内存。模型内 ExternalObject 类型的实例化通常会一直持续到仿真结束。但是，在函数定义时，如果 ExternalObject 类型被声明为 protected 变量，则可以在单个表达式的求值过程中创建和销毁。出于上述原因，我们必须确保由 ExternalObject 类型分配的任何内存再仿真结束后都被释放。

一般来说，destructor 函数也是作为外部函数来实现的。在这种情况下，Modelica 在调用 destructor 函数时会调用 C 语言定义的 destroyVectorTable。上述 C 函数以 VectorTable 作为参数。当调用 destructor 函数时，所有与 VectorTable 实例相关的内存都将被释放。同样的，函数有着与构造函数相同的注释，以此通知 Modelica 编译器 destoryVectorTable 函数源代码的位置。

**外部 C 代码** 这些外部的 C 函数通过以下方式实现：

```

#ifndef _VECTOR_TABLE_C_
#define _VECTOR_TABLE_C_

#include <stdlib.h>
#include "ModelicaUtilities.h"

/*
 Here we define the structure associated
 with our ExternalObject type 'VectorTable'
 */
typedef struct {
    double *x; /* Independent variable values */
    double *y; /* Dependent variable values */
    size_t npoints; /* Number of points in this data */
    size_t lastIndex; /* Cached value of last index */
} VectorTable;

void *
createVectorTable(double *data, size_t np) {

```

```

VectorTable *table = (VectorTable*) malloc(sizeof(VectorTable));
if (table) {
    /* Allocate memory for data */
    table->x = (double*) malloc(sizeof(double)*np);
    if (table->x) {
        table->y = (double*) malloc(sizeof(double)*np);
        if (table->y) {
            /* Copy data into our local array */
            size_t i;
            for(i=0;i<np;i++) {
                table->x[i] = data[2*i];
                table->y[i] = data[2*i+1];
            }
            /* Initialize the rest of the table object */
            table->npoints = np;
            table->lastIndex = 0;
        }
        else {
            free(table->x);
            free(table);
            table = NULL;
            ModelicaError("Memory allocation error\n");
        }
    }
    else {
        free(table);
        table = NULL;
        ModelicaError("Memory allocation error\n");
    }
}
else {
    ModelicaError("Memory allocation error\n");
}
return table;
}

void
destroyVectorTable(void *object) {
    VectorTable *table = (VectorTable *)object;
    if (table==NULL) return;
    free(table->x);
    free(table->y);
    free(table);
}

double
interpolateVectorTable(void *object, double x) {
    VectorTable *table = (VectorTable *)object;
    size_t i = table->lastIndex;
    double p;

    ModelicaFormatMessage("Request to compute value of y at %g\n", x);
    if (x<table->x[0])
        ModelicaFormatError("Requested value of x=%g is below the lower bound of %g\n",
                            x, table->x[0]);
    if (x>table->x[table->npoints-1])
        ModelicaFormatError("Requested value of x=%g is above the upper bound of %g\n",
                            x, table->x[table->npoints-1]);

    while(x>=table->x[i+1]) i = i + 1;
    while(x<table->x[i]) i = i - 1;

    p = (x-table->x[i])/(table->x[i+1]-table->x[i]);
}

```

```

table->lastIndex = i;
return p*table->y[i+1] + (1-p)*table->y[i];
}

#endif

```

这不是一本关于 C 语言编程的书。因此我们将不再详细讨论这段代码的语法及其所实现的功能。但是，我们可以对此文件的内容作出如下总结。

首先，名为 VectorTable 的 struct 变量内的数据对应 Modelica 变量 VectorTable 的内容。这个结构体中不仅仅包括插值数据（以成员变量 x、y 的形式）。它还包括数据点的个数 npoints 以及用于缓存最后使用的索引值 lastIndex。

其次，可以看到 createVectorTable 函数对 VectorTable 的结构进行了分配并对所有的数据进行了初始化。上述实例化过程完成后返回到 Modelica 中运行。createVectorTable 函数定义完成后，紧接着又对 destroyVectorTable 进行了定义，对 createVectorTable 函数定义的功能进行了有效的撤销。

最后，对 interpolateVectorTable 函数进行了定义。此函数是用 C 语言定义的，将 VectorTable 结构体及所需插值的独立变量进行参数传递，函数的返回值为利用插值算法计算的插入值。此函数实现的功能与之前定义的 InterpolateVector 函数所实现的功能几乎完全相同。在 Modelica 运行时，会执行 ModelicaFormatError 函数，以便报告 C 代码运行过程中出现的错误。对于 interpolateVectorTable 函数，这些错误函数主要用于实现前面 InterpolateVector 函数中的断言。在 interpolateVectorTable 函数中对于变量 i 的查找确认和 Modelica 版本基本相同。唯一的不同在于，查找并非每次都从 1 开始。函数会读取上次调用时找到的 i 值而开始查找。

**插值** 我们已经了解了如何定义 interpolateVectorTable 函数。但是目前为止，我们还不知道在什么地方调用了它。前面我们已经提到，除了使用 VectorTable 对象进行插值数据传递以外，其他的运行机理与 InterpolateVector 函数相同。在 Modelica 中调用 interpolateVectorTable 函数需定义一个简单的 Modelica 函数，如下所示：

```

function InterpolateExternalVector
  "Interpolate a function defined by a vector using an ExternalObject"
  input Real x;
  input VectorTable table;
  output Real y;
  external "C" y = interpolateVectorTable(table, x)
  annotation(IncludeDirectory="modelica://ModelicaByExample.Functions.Interpolation/source",
             Include="#include \"VectorTable.c\"");
end InterpolateExternalVector;

```

在前面已经提到了，VectorTable 表示隐形指针，而且 Modelica 代码不能访问 VectorTable 变量包含的数据。但是，在 Modelica 中定义的 InterpolateExternalVector 函数可以调用 C 语言定义的 interpolateVectorTable 函数，从而可以获取插值数据，根据插值算法计算相应的插入值。

## 讨论

如前所述，最初的插值方法需要传递大量复杂的插值数据。但是，通过定义 VectorTable 变量，可以非常方便的用一个变量表示上述插值数据。

关于 ExternalObject 插值方法，有一点在上述示例中并没有充分的展开讨论。但这点非常重要：初始化数据可以完全独立于 Modelica 源代码。简便起见，在上述示例中展示的代码中，通过定义数组的方式对 VectorTable 变量进行了初始化。但是，它也可以简单的通过文件名完成上述初始化工作。createVectorTable 函数通过读取相应的文件，用文件中对应的数据对 VectorTable 结构体进行赋值，完成数据的初始化工作。很多情况下，这种方法不仅可以更方便的管理数据，而且可以利用 C 语言实现（新的或已有的）更加复杂的算法。

下一章（121）里包含了另一个通过 Modelica 调用外部 C 代码的例子。

### 第 4.1.3 节 软件在环 (SiL) 控制器

前面章节插值 (113) 的示例里我们介绍了利用外部 C 函数去对数据进行管理和插值。本章节将继续探索如何在 Modelica 模型中集成用 C 代码编写的嵌入式控制器。

当利用 Modelica 搭建物理系统的数学模型时，有些时候，将（外部）控制策略与物理模型进行集成会非常有用。很多情况下，上述控制策略的存在形式均是用于嵌入式控制器的自动生成 C 代码。下面展示的例子将会回顾前面示例滞回 (68)。在此的基础上，例子里包含了一些有趣的转折。我们将使用外部 C 函数来实现磁滞特性，而非在 Modelica 中通过离散状态来实现。虽然这个示例非常简单，但是它包含了集成外部控制策略所需的所有步骤。

#### 物理模型

首先，我们从此前的“物理模型”开始。在这个示例中，上述物理模型本质上与前面章节滞回 (68) 中所搭建的模型基本相同。修改后的物理模型如下所示：

```
model HysteresisEmbeddedControl "A control strategy that uses embedded C code"
  type HeatCapacitance=Real(unit="J/K");
  type Temperature=Real(unit="K");
  type Heat=Real(unit="W");
  type Mass=Real(unit="kg");
  type HeatTransferCoefficient=Real(unit="W/K");
  parameter HeatCapacitance C=1.0;
  parameter HeatTransferCoefficient h=2.0;
  parameter Heat Qcapacity=25.0;
  parameter Temperature Tamb=285;
  parameter Temperature Tbar=295;
  Temperature T;
  Heat Q;
initial equation
  T = Tbar+5;
equation
  when sample(0, 0.01) then
    Q = computeHeat(T, Tbar, Qcapacity);
  end when;
  C*der(T) = Q-h*(T-Tamb);
end HysteresisEmbeddedControl;
```

对于上述模型的 equation 区域，如下所示：

```
equation
  when sample(0, 0.01) then
    Q = computeHeat(T, Tbar, Qcapacity);
  end when;
  C*der(T) = Q-h*(T-Tamb);
end HysteresisEmbeddedControl;
```

computeHeat 函数每隔 10 毫秒计算一次所需的热量。我们马上将可以看到在控制器中实现了开关控制策略。即系统在无热量产生和全功率热量产生之间进行切换。正如在前面章节滞回 (68) 中所述，上述方法可能会导致系统“抖振”。出于上述原因，我们在 when 语句中每 10 毫秒计算一次 Q 变量的值。这 10 毫秒间隔实质上是为了实现所谓的“调度”功能，用以决定执行哪种控制策略。

#### 嵌入式控制策略

在 Modelica 中定义的 computeHeat 函数用于计算任意给定时间内传递给物理系统的热量。函数定义如下所示：

```
impure function computeHeat "Modelica wrapper for an embedded C controller"
  input Real T;
  input Real Tbar;
```

```

input Real Q;
output Real heat;
external "C" annotation (Include="#include \"ComputeHeat.c\"",
    IncludeDirectory="modelica://ModelicaByExample.Functions.ImpureFunctions/source");
end computeHeat;

```

注意，在上述代码中同样也出现了 `external` 关键字。而与前面示例不同的是，`external` 关键字后面没有相应 C 函数的名称。这就意味着，外部 C 函数与 Modelica 定义的函数有着完全相同的名字和参数。阅读上述 C 函数的源代码，我们会发现确实如此：

```

#ifndef _COMPUTE_HEAT_C_
#define _COMPUTE_HEAT_C_

#define UNINITIALIZED -1
#define ON 1
#define OFF 0

double
computeHeat(double T, double Tbar, double Q) {
    static int state = UNINITIALIZED;
    if (state==UNINITIALIZED) {
        if (T>Tbar) state = OFF;
        else state = ON;
    }
    if (state==OFF && T<Tbar-2) state = ON;
    if (state==ON && T>Tbar+2) state = OFF;

    if (state==ON) return Q;
    else return 0;
}

#endif

```

换言之，如果按上述方式定义外部 C 函数，在 Modelica 中定义的函数与 C 语言定义的函数之间就无需建立参数的映射关系，这样就可以省掉很多编程麻烦。

由上述代码可以看到，`computeHeat` 函数中定义了 `static` 类型的变量 `state`。`static` 关键字的使用表明变量 `state` 的值可以通过 `computeHeat` 函数在别的地方引用。这种类型的变量在嵌入式控制策略里非常普遍，因为需要将相应信息在不同调度函数内进行传递（例如在上述磁滞控制示例中）。

`static` 类型变量的出现会引起潜在的问题，因为它意味着 `computeHeat` 函数对于相同的输入参数具有不同的返回值。从数学上来讲，这不是一个纯粹的数学函数。因为数学函数只依赖于其输入参数。在计算机科学领域，一般称上述函数为“非纯”函数。这也就意味着，每次调用上述函数其内存或变量都会产生相应的变化，从而影响函数的返回值。

在嵌入式控制策略设计阶段会遇到上述问题，在面向数学的环境中，例如 Modelica 环境，调用上述函数时需特别小心。因为，Modelica 编译器默认所有的函数都是纯函数或者说无副作用的。如果出现了非纯函数或者说副作用，在仿真中就会出现问题。最好的结果是仿真效率非常低，最坏的结果是仿真结果完全不正确。

这种问题的出现，原因在于底层求解器在找到“正确”解前必须计算很多的“备选”解。如果生成的备选解需要求解器调用具有副作用的函数时，求解器将无法预测变量变更所造成的影响。

出于上述原因，`computeHeat` 函数在定义时添加了 `impure` 限定词，如下所示：

```

impure function computeHeat "Modelica wrapper for an embedded C controller"
    input Real T;

```

以上述方式通知 Modelica 编译器这个函数具有副作用或返回值不仅仅依赖输入参数，而且当生成备选解时不能调用此函数。这样看起来会完全禁止其他函数调用上述函数。但其实并非如此。回顾上述要集成的控制策略：

```

equation
when sample(0, 0.01) then

```

```

Q = computeHeat(T, Tbar, Qcapacity);
end when;
C*der(T) = Q-h*(T-Tamb);
end HysteresisEmbeddedControl;

```

特别的，需要注意的是 `computeHeat` 函数只在 `when` 语句中调用了。该函数并不是作为某个“连续”方程的一部分。因此，我们可以肯定的是 `computeHeat` 函数只会在响应事件时被调用，而在不会计算连续变量备选解时被调用。

## 仿真结果

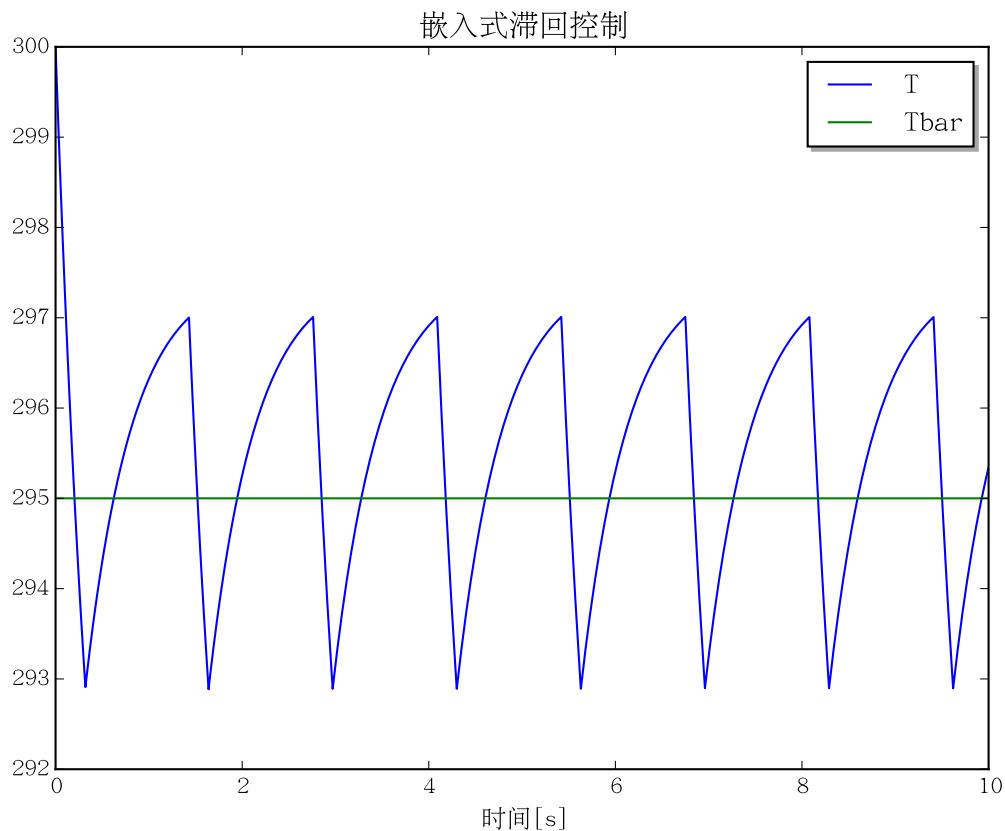
在 C 语言定义的 `computeHeat` 函数中，实现了在设定点附近浮动  $+/-2$  度的算法，如下所示：

```

if (state==OFF && T<Tbar-2) state = ON;
if (state==ON && T>Tbar+2) state = OFF;

```

上述功能正能有效地消除系统抖振。仿真结果中可以清晰的看到这点，如下图所示：



### 第 4.1.4 节 非线性

下一个示例主要展示了如何解决非线性系统方程调用函数的问题。

首先我们以一个简单的模型开始介绍。该模型包含了非常简单的数学关系，如下所示：

$$y = 2t^2 + 3t + 1$$

其中， $t$  表示时间。上述模型在 Modelica 中的实现如下：

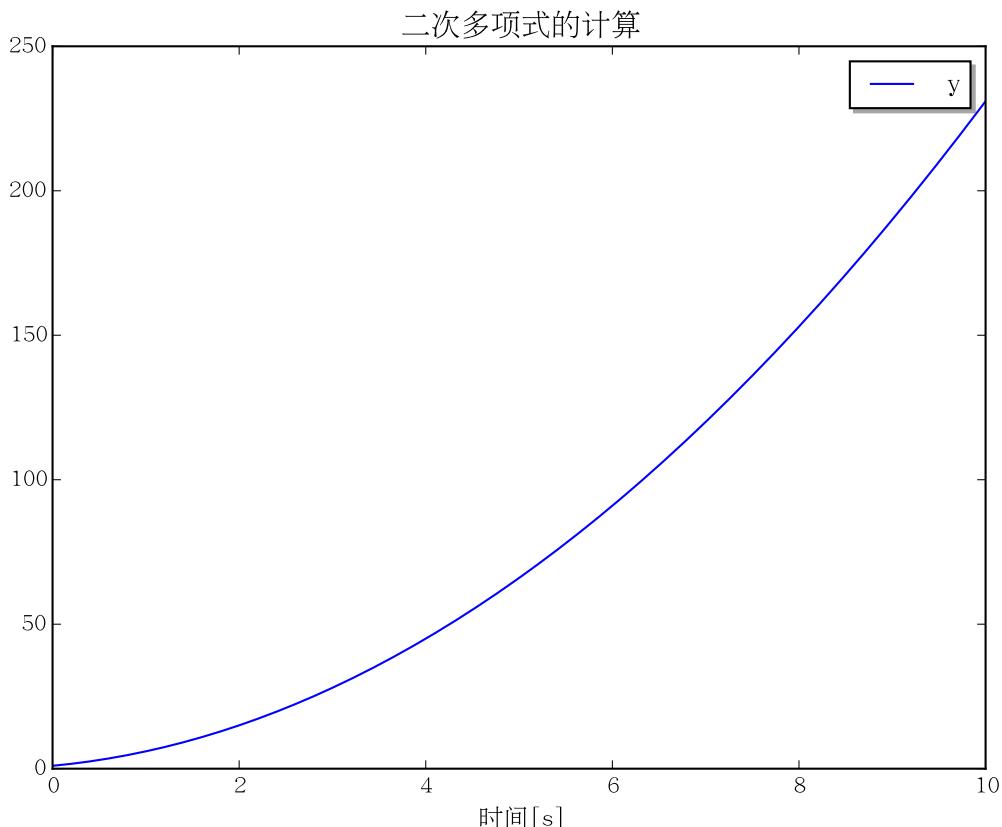
```

model ExplicitEvaluation
  "Model that evaluates the quadratic function explicitly"
  Real y;
equation
  y = Quadratic(2.0, 3.0, 1.0, time);
end ExplicitEvaluation;

```

其中，Quadratic 函数用于二次多项式的计算。我们在后面会对其进行简单的讨论。

对上述模型进行仿真，变量 y 的仿真结果如下图所示：



目前为止，所有的一切似乎都很合理。基于前面章节多项式计算（107）中的讨论，Quadratic 函数的使用似乎也在意料之中。但是，对于如下所示的更加复杂的模型：

```

model ImplicitEvaluation
  "Model that requires the inverse of the quadratic function"
  parameter Real y_guess=2;
  Real y(start=y_guess);
equation
  time+1 = Quadratic(2.0, 3.0, 1.0, y);
end ImplicitEvaluation;

```

上述模型相当于求解下列方程：

$$t + 5 = 2y^2 + 3y + 1$$

明显的区别在于，上述模型中等式左边是已知的。我们需要计算满足方程的 y 值。换言之，不同于前面示例中计算二次多项式的值，上述模型需对二次多项式进行求解。

需要求解非线性系统方程的模型并不太好。Modelica 编译器具有非凡识别和求解非线性系统方程的能力（尽管为了收敛，这些方法通常需依赖于合理的初始猜测）。

但事实上，上述情况下 Modelica 编译器并不要求解一个非线性系统。若我们了解了 Quadratic 函数是如何实现的，就能清楚的了解上述情况：

```

function Quadratic "A quadratic function"
  input Real a "2nd order coefficient";
  input Real b "1st order coefficient";
  input Real c "constant term";
  input Real x "independent variable";
  output Real y "dependent variable";
algorithm
  y := a*x*x + b*x + c;
  annotation(inverse(x = InverseQuadratic(a,b,c,y)));
end Quadratic;

```

特别的，需重点注意上述代码中的 inverse 注释。对于上述函数的定义，不仅告诉 Modelica 编译器如何求解 Quadratic 方程，而且通过 inverse 注释利用 InverseQuadratic 函数可以计算得到变量 x 和变量 y 的关系。

InverseQuadratic 函数的定义如下所示：

```

function InverseQuadratic
  "An inverse of the quadratic function returning the positive root"
  input Real a;
  input Real b;
  input Real c;
  input Real y;
  output Real x;
algorithm
  x := sqrt(b*b - 4*a*(c - y))/(2*a);
end InverseQuadratic;

```

Note: 通过上述代码，可以看到 InverseQuadratic 函数只能计算二次方程的正根。对我们来说，这个结果可好可坏。只计算一个根的话，在转换二次函数关系时就可以避免多个解。但是，如果你恰巧需要负根，就可能会产生相应的问题。

在上述 ImplicitEvaluation 模型中，Modelica 编译器可以替代上述反函数方程。因此，忽略当前方程的参数，需要求解的方程如下所示：

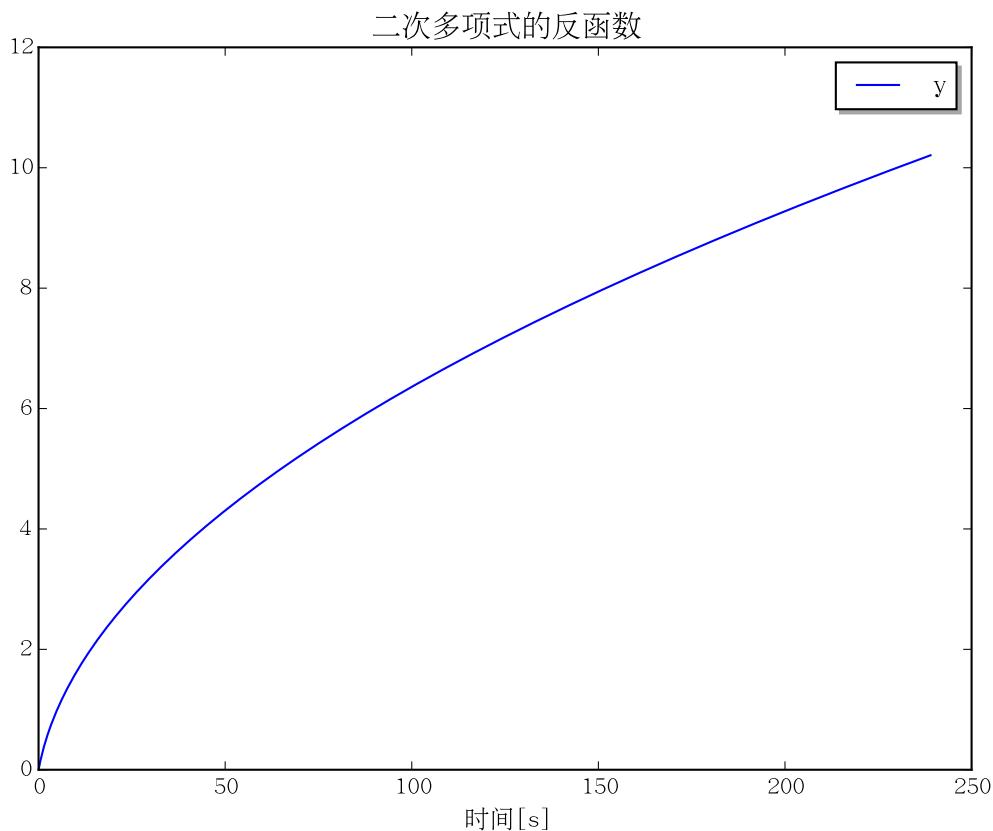
$$t + 5 = f(y)$$

其中，必须求解变量 y 的隐式方程，可以得到如下所示显式方程：

$$y = f^{-1}(t + 5)$$

而通过使用 InverseQuadratic 函数作为原函数的反函数。

对上述 ImplicitEvaluation 模型进行，可以得到变量 y 仿真结果，如下图所示：



通过观察上图，可以发现，模型计算得到了正确的结果。而且，一般情况下并不要求解非线性系统。否则，如果没有注解，那 ImplicitEvaluation 模型就会引入非线性的方程组。

## 第 4.2 节 回顾

### 第 4.2.1 节 函数定义

如前所述，Modelica 语言已经包含了很多有用的函数，可以方便地描述物体的数学特性。但不可避免地，有时候有必要创建用于特殊目的的新函数。定义这些新函数的方式及语法与 [模型定义](#) (25) 相似。

#### 基本语法

基本的 Modelica 函数都包含一个或多个参数、一个函数返回值以及根据参数计算函数返回值的 algorithm。函数的输入参数以限定词 input 进行声明。函数的返回值以限定词 output 进行声明。例如，以下述简单函数为例，函数计算输入参数的平方值：

```
function Square
  input Real x;
  output Real y;
algorithm
  y := x*x;
end Square;
```

在上述例子中，输入变量 x 的类型为 Real。输出变量 y 同样也是 Real 类型。函数的参数和返回值可以是标量或数组（甚至是记录类型，[later](#) (173) 章节才介绍记录类型）。

## 中间变量

对于复杂的计算过程，通过定义变量保存中间结果将会非常有帮助。这些变量必须与函数的参数和返回值明确的区分。为了声明这些中间变量，我们以限定词 `protected` 进行声明。以 `protected` 形式声明这些变量是为了告诉 Modelica 编译器这些变量既不是函数的参数也不是函数的返回值，而是函数计算过程中内部使用的变量。例如，如果我们希望编写一个函数来计算圆的周长，那就可能需要一个中间变量来储存圆的直径，即：

```
function Circumference
    input Real radius;
    output Real circumference;
protected
    Real diameter := radius*2;
algorithm
    circumference := 3.1415*diameter;
end Circumference;
```

在上述例子中，我们可以看到中间结果或公用子表达式可以与某个中间变量相关联。

## 默认的输入参数

在某些情况下，有必要为某些输入参数设置默认值。这种情况下，在声明输入参数时就可以包含了默认值。考虑如下计算物体势能的函数：

```
function PotentialEnergy
    input Real m "mass";
    input Real h "height";
    input Real g=9.81 "gravity";
    output Real pe "potential energy";
algorithm
    pe := m*g*h;
end PotentialEnergy;
```

为参数 `g` 设定默认值后，用户调用此函数时就不必每次都为 `g` 赋值了。当然，这种情况只适用于给定参数具有合理默认值的情况。如果你期望用户为参数提供数值的话，此种方法不适用。

在调用 [函数 \( 127 \)](#) 时，这些具有默认值的参数将产生重要的影响。我们将稍后讨论这点。

## 多个返回值

其实，一个函数可以有多个返回值（即多个带有 `output` 限定词的变量声明）。例如，我们可以利用一个函数同时计算圆的周长和面积：

```
function CircleProperties
    input Real radius;
    output Real circumference;
    output Real area;
protected
    Real diameter := radius*2;
algorithm
    circumference := 3.1415*diameter;
    area := 3.1415*radius^2;
end CircleProperties;
```

我们接下来将在[调用函数 \( 127 \)](#)部分讨论如何设置函数的多个返回值。

## 调用函数

目前为止，我们已经介绍了如何去定义一个新的函数。接下来，我们将讨论各种调用函数的方法。一般的，函数调用的方式应该同时满足数学工作者和程序员的需求，例如：

```
f(z, t);
```

如上，我们可以看到典型语法函数调用语法：函数名称后带有括号，而里面的参数列表则由逗号进行分隔。但是，也有一些有趣的个例需要讨论。

上述语法需注意的是“参数位置”。这意味着，在调用此函数进行参数赋值时，要注意参数顺序。由于 Modelica 语言定义了函数的参数名称，因此也可以通过参数名调用函数。例如，参考下面计算立方体体积的函数：

```
function CylinderVolume
    input Real radius;
    input Real length;
    output Real volume;
algorithm
    volume = 3.1415*radius^2*length;
end CylinderVolume;
```

当调用此函数时，一定要注意不要混淆参数 radius 和 length 的顺序。为了避免由于参数顺序引起的混淆，我们可以使用参数名称来调用此函数。以这种方式实现函数调用的写法如下所示：

```
CylinderVolume(radius=0.5, length=12.0);
```

当调用具有默认参数值的函数时，使用参数名称调用将尤其方便。对于前面介绍的 PotentialEnergy 函数，可以有以下多种调用方式：

```
PotentialEnergy(1.0, 0.5, 9.79)      // m=1.0, h=0.5, g=9.79
PotentialEnergy(m=1.0, h=0.5, g=9.79) // m=1.0, h=0.5, g=9.79
PotentialEnergy(h=0.5, m=1.0, g=9.79) // m=1.0, h=0.5, g=9.79
PotentialEnergy(h=0.5, m=1.0)        // m=1.0, h=0.5, g=9.81
PotentialEnergy(0.5, 1.0)           // m=1.0, h=0.5, g=9.81
```

之所以对于具有默认参数值的函数，应使用参数名称实现函数调用，最重要的原因是：对于具有多个默认参数值的函数，你可以通过指定这些参数的名称以选择性地覆盖其默认值。

现在，我们讨论一下前面提到的一个函数可能具有多个返回值的情况。但问题是，我们该如何引用函数的多个返回值？让我们回顾一下本节前面部分定义的函数 CircleProperties，下面的表达式展现了在实际使用时，我们应如何引用函数的多个返回值：

```
(c, a) := CircleProperties(radius);
```

换句话说，等号左边括号内用逗号分隔开的参数值是由函数的相应返回值进行赋值的（或在函数内部使用了 equation 区域时，等于相应值）。

根据上述讨论的结果，在 Modelica 语言中，有多种方式可以实现函数调用。

## 重要的限制条件

一般来说，我们可以利用函数或模型实现相同的计算功能。但是，两者之间存在一些重要的限制条件。

1. 输入变量都是只读的——不可以对函数的输入变量进行赋值。
2. 在函数中，不允许引用全局变量 time。
3. 不允许存在多个方程或 when 语句——函数中只允许包含一个 algorithm 区域，并且不能包含 when 语句。
4. 以下功能不能从函数中进行调用：der、initial、terminal、sample、pre、edge、change、reinit、delay、cardinality、inStream、actualStream。
5. 参数、结果以及中间变量（protected）不能是模型或块。
6. 数组的大小受限——参数是数组类型的可不指定其维数（请参考未定义维度（94）部分）并且其大小由引用它的函数隐性确定。函数返回值是数组类型的，其大小必须由常数或与其相关联的输入参数的大小来定义。

需要非常注意的一点是，函数能不受限制地实现递归运算（即函数允许调用自身）。

## 副作用

在软件在环 (SiL) 控制器 (121) 例子中，我们对外部函数的副作用进行了介绍。也就是说，函数的返回值并不是其参数在严格意义上的函数。像这类函数就具有所谓的“副作用”。具有副作用的函数应该以关键字 `impure` 进行声明。这就告诉 Modelica 编译器，这些函数不能被视为纯粹的数学函数。

这些以 `impure` 函数在使用时有其限制。它们只能被其他 `impure` 函数调用或在 `when` 语句中使用。

## 函数模板

考虑上述所有情况，下面的例子可以认为是一个广义函数定义的模板：

```
function FunctionName "A description of the function"
  input InputType1 argName1 "description of argument1";
  ...
  input InputTypeN argNameN := defaultValueN "description of argumentN";
  output OutputType1 returnName1 "description of return value 1";
  ...
  output OutputTypeN returnNameN "description of return value N";
protected
  InterType1 intermedVarName1 "description of intermediate variable 1";
  ...
  InterTypeN intermedVarNameN "description of intermediate variable N";
  annotation(key1=value1,key2=value2);
algorithm
  // Statements that use the values of argName1..argNameN
  // to compute intermedVarName1..intermedVarNameN
  // and ultimately returnName1..returnNameN
end FunctionName;
```

## 第 4.2.2 节 控制流程

某些情况下，函数只是按照固定的流程一步一步地进行计算。但是在其他一些情况下，函数也需要某种循环或迭代。在本节里，我们将讨论函数定义内所支持的不同控制结构。

### 分支结构

在前面章节，我们已经看到了涉及 `if` 语句和 `if` 表达式的例子。函数内部自然也可以包含这些元素。实际上，`equation` 区域对 `if` 语句有一定的限制。那就是，`if` 语句的每个分支（即所有可能条件）均必须生成相同数量的方程。但是，上述限制对 `algorithm` 区域并不适用（例如在函数定义时）。

### 循环

`equation` 区域要确保在任意系统状态下生成的方程数量相等。所以，循环（就像分支一样）在使用时受到了严格限制。出于这个原因，`equation` 区域（目前为止，我们唯一讨论过的区域）唯一允许使用的循环结构是 `for` 循环。

在函数定义中，`for` 循环的语法与其在其他情况下完全一样。使用 `for` 循环首先要确定循环变量。然后循环将数组中的数值分配给循环变量，即：

```
algorithm
  for i in 1:10 loop
    // Statements
  end for;
```

equation 区域与 algorithm 区域有两个主要的区别。这分别是：algorithm 区域使用显式赋值语句而不使用等式；因此在使用 if 语句或 for 语句时就不需要考虑生成方程的数量。

此外，algorithm 区域允许 while 循环的使用。这让我们能更灵活地进行建模。equation 区域不允许使用 while 循环的原因在于，其在本质上生成的方程数量（即 equation 区域生成的方程数量）是不可预测的。但是，这种不可预见性对于 algorithm 区域并没有什么影响。

在前面章节 [插值 \(113\)](#) 中，我们已经讨论了 InterpolateVector 函数。其包含的 while 循环的语法如下所示：

```
while x>=ybar[i+1,1] loop
  i := i + 1;
end while;
```

while 循环的主要元素是一系列条件表达式。其作用在于决定是否继续执行 while 循环内的语句。

#### break 以及 return

在进行循环时，某些时候需要提前结束循环过程。例如，在 for 循环中，循环重复的次数通常是由循环数组的大小来决定的。但在某些情况下，后续的循环过程是没有必要的。同样的，在 while 循环中，它可以方便的检查 while 循环何时终止。在这类情况下，可以使用 break 语句终止最内层循环。

控制流程的另一个问题是何时终止并从 algorithm 区域本身退出。在很多情况下，所有的 output 变量都被分配了它们的最终值。纵然，我们总可以使用 if 和 else 语句禁止变量的进一步计算和赋值。但若能直接表明无须进行进一步计算，那就简明和清晰了。这时，我们可以在函数的 algorithm 区域使用 return 语句终止所有方程的进一步运算。在执行 return 语句时，output 变量将返回当前时刻与之相关联的变量值。

### 第 4.2.3 节 外部函数

#### external

在 [插值 \(113\)](#) 章节，我们讨论了 InterpolateExternalVector 函数相关的例子。我们了解到可以通过 Modelica 语言来调用由其他语言定义的函数。通常，这些函数都是用 C 或 Fortran 语言来定义的。使用 Modelica 之外的语言所定义的函数不包含 algorithm 区域。相反的，函数应该包括 external 语句，以提供外部函数的信息以及如何实现与外部函数的信息传递。

对于外部函数的最低要求是需包含关键字 external，即：

```
external;
```

在这里，我们假设外部函数是由 C 语言定义的。而且，函数的名称需与 Modelica 语言的“包装”函数相匹配。最后，函数的参数必须以相同的顺序传递给 Modelica 函数的 input 参数。

让我们考虑一个稍微复杂的情况，例如在 [插值 \(113\)](#) 例子中展示的 VectorTable 类型：

```
function destructor "Release storage"
  input VectorTable table;
  external "C" destroyVectorTable(table)
  annotation(IncludeDirectory="modelica://ModelicaByExample.Functions.Interpolation/source",
             Include="#include \"VectorTable.c\"");
end destructor;
```

我们可以看到，此函数定义的语言被明确的指定为“C”。还有另外两种可以实现上述功能的语言：“FORTRAN 77”和“builtin”。对“builtin”使用感兴趣的主要是一些 Modelica 工具厂商。

我们看到，上述语句已经明确指定函数的名称。external 语句内的 destroyVectorTable(table) 这部分明确指定了与外部函数传递的参数及其顺序。

有些情况下，还需明确定义传递给外部函数的参数值及输出变量和函数调用结果之间的映射关系。我们可以在下述 function 定义中，看到上述信息：

```

function constructor
  input Real ybar[:,2];
  output VectorTable table;
  external "C" table=createVectorTable(ybar, size(ybar,1))
  annotation(IncludeDirectory="modelica://ModelicaByExample.Functions.Interpolation/source",
             Include="#include \"VectorTable.c\"");
end constructor;

```

在这个例子中，外部函数需要获取 `ybar` 数组的大小。因为这项信息没有直接地传递给外部函数。此外，上述函数还声明了 `createVectorTable` 的计算结果应该分配给 `output` 变量 `table`。很明显，我们可以看到 C 函数的返回值就是 Modelica 语言定义的函数返回值。但是，有些情况下，`output` 变量应该作为参数传递给函数。接下来我们就会看到，在这种情况下，外部函数可以使用指针完成相应变量的赋值。

## 数据映射

### C 语言

下面这张表展示了将参数传递给外部函数时，Modelica 语言与 C 语言数据类型之间的映射关系。

Modelica 语言	C 语言（输入变量）	C 语言（输出变量）
Real	double	double *
Integer	int	int *
Boolean	int	int *
String	const char *	const char **
T[d1]	T' *, size_t d1	T' *, size_t d1
T[d1,d2]	T' *, size_t d1, size_t d2	T' *, size_t d1, size_t d2
T[d1,...,dn]	T' *, size_t d1, ..., size_t dn	T' *, size_t d1, ..., size_t dn
size(...)	size_t	N/A
enumeration	int	int *
record	struct *	struct *

我们简单解释一下上述表格。首先，它假设所有的字符串都是以空字符 (\0) 终止的。另外，数组类型 T 在 C 语言与 Modelica 语言间可以相互映射（使用相同的表）。最后，Modelica 语言的 record 类型对应 C 语言的 struct 类型。而且 struct 类型与 record 类型的数据成员顺序一致。record 成员的数据类型使用该表格第二列的映射关系（即把它们当做输入参数）。

对于 C 函数的返回值类型，采用下表的类型映射关系：

Modelica 语言	C 语言
Real	double
Integer	int
Boolean	int
String	const char *
T[d1]	T' *, size_t
T[d1,d2]	T' *, size_t d1, size_t d2
T[d1,...,dn]	T' *, size_t d1, ..., size_t dn
size(...)	size_t
enumeration	int
record	struct *

### Fortran 语言

如果你需要调用 Fortran 语言定义的函数或子程序，采用下表定义的类型映射关系：

Modelica 语言	Fortran 语言
Real	DOUBLE PRECISION
Integer	INTEGER
Boolean	LOGICAL
T[d1]	T, INTEGER
T[d1,d2]	T, INTEGER d1, INTEGER d2
T[d1,...,dn]	T, INTEGER d1, ..., INTEGER dn
size(...)	INTEGER
enumeration	INTEGER

对于上述表格，有两点需特别注意。首先，字符串和记录类型没有相应的映射关系。其次，Fortran 语法是传递引用，所有输入、输出这些函数的变量都被假定为指针。出于这个原因，Fortran 函数不会区分获取的变量是 Modelica 函数的输入还是输出。

## 特殊函数

在与 Modelica 交互运行时，有许多特殊函数可以从外部函数进行调用。下面对这些函数的名称、原型以及开发的目的进行详细的介绍。

### ModelicaVFormatMessage

```
void ModelicaVFormatMessage(const char*string, va_list);
```

此函数的输出与 C 函数 vprintf 相同格式的消息。

### ModelicaError

```
void ModelicaError(const char* string);
```

此函数主要处理类似 Modelica 代码的声明错误，输出错误信息的字符串（无输出格式控制），对于调用函数无返回值。

### ModelicaFormatError

```
void ModelicaFormatError(const char* string, ...);
```

此函数主要处理类似 Modelica 代码的声明错误，输出与 C 函数 printf 相同格式的错误信息，对于调用函数无返回值。

### ModelicaVFormatError

```
void ModelicaVFormatError(const char* string, va_list);
```

此函数主要处理类似 Modelica 代码的声明错误，输出与 C 函数 vprintf 相同格式的错误信息，对于调用函数无返回值。

### ModelicaAllocateString

```
char* ModelicaAllocateString(size_t len);
```

此函数主要用于为外部 Modelica 函数的返回参数分配内存。需注意的是，该字符串数组的存储空间（即指向字符串数组的指针）仍然由被调用函数提供。当出现错误时，该函数不返回主函数，而是调用 ModelicaError ( 132 ) 函数输出错误信息。

ModelicaAllocateStringWithErrorReturn

```
char* ModelicaAllocateStringWithErrorReturn(size_t len);
```

该函数功能和ModelicaAllocateString ( 132) 函数几乎一样。不同点只是在出现错误时, 函数会返回 0 值。当出现错误时, 此功能允许外部函数关闭文件并清空其他的资源。完成资源清理以后, 调用 ModelicaError ( 132) 函数和ModelicaVFormatError ( 132) 函数输出错误信息。

#### 第 4.2.4 节 函数标注

在标注 ( 34) 章节中, 我们已经对标注进行了讨论。Modelica 语言包括一些专门用于函数的标准标注。这些标注的含义正式定义在 Modelica 规范中。在本节中, 我们将讨论与函数相关的三大类标注, 并讨论这些标注的必要性以及如何使用这些标注。

##### 数学函数标注

第一大类标注的主要功能是为数学函数提供一些额外的信息。因为这些数学函数的功能都是在 algorithm 区域实现的, 因此通常情况下, 编译器不能根据 Modelica 语言内的符号操作得出上述函数方程的行为。但是, 本节中的标注可以用来为函数定义补充相关数学信息。

derivative

正如我们在多项式计算 ( 107) 示例中讨论的那样, 有些情况下, 我们需告诉 Modelica 编译器如何计算给定函数的导数。上述功能主要是通过在函数定义时添加 derivative 标注来实现的。

**简单的一阶导数** derivative 标注的基本用途是为了指定另一个 Modelica 函数的名称, 用来计算被标注函数的一阶导数。例如:

```
function f
  input Real x;
  input Real y;
  output Real z;
  annotation(derivative=df);
algorithm
  z := // some expression involving x and y
end f;

function df
  input Real x;
  input Real y;
  input Real dx;
  input Real dy;
  output Real dz;
algorithm
  dz := // some expression involving x, y, dx and dy
end df;
```

通过上述例子可以看到, 在这种情况下, 导数函数的第一个参数 df 与原始函数 f 的第一个参数相同。然后, 紧随此参数后面的是原函数输入变量的微分形式。最后, 导数函数的输出变量等同于原始函数输出变量的微分。听上去上述过程非常复杂, 但读者应该能从上述代码中发现, 要构造这样的函数其实是非常简单的。

给定如下的 Modelica 函数, Modelica 编译器就可以使用下列函数来计算函数的一系列导数, 例如:

$$\frac{df}{dv}(x, y) = df(x, y, \frac{\partial x}{\partial v}, \frac{\partial y}{\partial v})$$

**不敏感参数** 我们考虑这样一种情况，表达式  $\frac{\partial y}{\partial v}$  等于 0。导数函数将会输出这个零值（或零值数组，若参数为数组）。上述零值会在导数函数内部被数次调用，其中大多数（如果不是全部）将调用零值进行乘法运算。此时计算的结果也将为零。最终，这些计算结果将会与其他结果相加，但是对函数的最终输出没有任何影响。换句话说，这种情况下，有很多计算其实可以跳过，因为它们对结果不会有任何影响。

在这种情况下，Modelica 语言提供了一种避免上述冗余计算的方法。如果 Modelica 编译器能预先知道参数的微分为 0，它就可以（在 derivative 标注内）检查是否有函数计算导数。这些情况是通过在 derivative 标注使用 zeroDerivative 参数进行指定的。因此，对于上述示例中的函数 f，我们增加了如下标注内容：

```
function f
  input Real x;
  input Real y;
  output Real z;
  annotation(derivative=df, derivative(zeroDerivative=y)=df_onlyx);
algorithm
  z := // some expression involving x and y
end f;
```

其中，df\_onlyx 函数被定义为如下形式：

```
function df_onlyx
  input Real x;
  input Real y;
  input Real dx;
  output Real dz;
algorithm
  dz := // some expression involving x, y, dx
end df_onlyx;
```

通过比较可以看到，此函数不包括 dy 项。因此，函数只适用于 dy 为 0 的情况。而且，因为参数不包括 dy 项，该函数只包括那些涉及参数 dx 的计算方程。

**二阶导数** 对于二阶导数，这里有一些变化需要说明一下。首先要知道应如何指定函数的二阶导数。这主要是通过添加一个 order 变量来实现的。需要注意的是，函数可以包括多个 derivative 标注，例如：

```
function f
  input Real x;
  input Real y;
  output Real z;
  annotation(derivative=df, derivative(order=2)=ddf);
algorithm
  z := // some expression involving x and y
end f;

function df
  ...
end df;

function ddf
  input Real x;
  input Real y;
  input Real dx;
  input Real dy;
  input Real ddx;
  input Real ddy;
  output Real ddz;
algorithm
  ddz := // some expression involving x, y, dx, dy,
        // ddx and ddy
end ddf;
```

希望你对上述内容并不陌生。为了实现二阶导数的计算，需要在原函数内添加额外的 derivative 标注，即：

```
annotation(derivative=df, derivative(order=2)=ddf);
```

这些额外的标注增加了一个 order 变量用于表明函数求解的是第几阶的导数。

**非实数参数** 此外，对于其他一些复杂情况还需讨论。例如，函数内有时会包含一些非实数的参数，即：

```
function g
  input Real x;
  input Integer y;
  output Real z;
algorithm
  z := // some expression involving x and y
end g;
```

要计算上述函数对参数 y 的导数就没有什么意义了，因为参数 y 的数据类型是整型。在求解函数导数时，我们可以忽略任何非实数的参数。因此，如果我们希望计算上述函数的导数，可以采用下述方式：

```
function g
  input Real x;
  input Integer y;
  output Real z;
  annotation(derivative=dg);
algorithm
  z := // some expression involving x and y
end g;

function dg
  input Real x;
  input Integer y;
  input Real dx;
  output Real dz;
algorithm
  dz := // some expression involving x, y and dx
end dg;
```

换句话说，我们只对那些类型为实数的参数进行微分计算。

### inverse (反函数)

在**非线性** (123) 章节中，我们讨论了使用 inverse 标注。该标注为 Modelica 编译器提供了如何计算相应函数反函数的信息。逆函数作用在于它能显式地求解函数的某个输入变量。因此，inverse 标注包含一个涉及当前函数输入、输出变量的显式方程。这样，标注可利用另一个函数去直接求解其中一个输入变量。例如，对于一个定义如下的 Modelica 函数：

```
function h
  input Real a;
  input Real b;
  output Real c;
  annotation(inverse(b = h_inv_b(a, c)));
algorithm
  c := // some calculation involving a and b
end h;
```

我们可以在根据上述代码里看到，通过向函数 h\_inv\_b 中输入变量 a 和 c，就可以计算得到参数 b 的表达式，如下所示：

```

function h_inv_b
  input Real a;
  input Real c;
  output Real b;
algorithm
  b := // some calculation involving a and c
end h_inv_b;

```

## 代码生成

另一大类标注是关于如何将定义的函数转换为仿真用的代码。这些标注允许模型开发者在 Modelica 编译器生成代码的过程提供一定的帮助。

### Inline

Inline 标注用于提示 Modelica 编译器，函数中的语句是否应被“内联”。此标注的取值用于决定了执行内联与否。标注的默认值为 false（如果函数定义中没有出现 Inline 标注）。下面是使用 Inline 标注定义的函数示例：

```

function SimpleCalculation
  input Real x;
  input Real y;
  output Real z;
  annotation(Inline=true);
algorithm
  z := 2*x-y;
end SimpleCalculation;

```

在上述例子中，我们可以看到，Inline 标注表明 Modelica 编译器应内嵌上述定义的 SimpleCalculation 函数。函数的内嵌是通过调用计算输出变量的函数进行替换的。这对于执行非常简单计算功能的函数是非常有用的。这些情况下，调用该函数的“成本”（计算机 CPU 的计算时间）与函数执行的成本基本是一个数量级的。通过内嵌函数，调用函数的成本可以大大消除，同时依然能实现函数的功能。

Inline 标注的功能只是用于提示 Modelica 编译器，编译器不会自动关联内嵌函数。此外，编译器内嵌函数的能力取决于函数的复杂程度。通常情况下，编译器没有必要（或者期望）去内嵌函数。

### LateInline

与 Inline (136) 标注一样，LateInline 标注的功能是告诉 Modelica 编译器采用内嵌函数的方式将更有效。LateInline 标注也会分配一个 Boolean 值用以指定是否应采用内嵌函数的方式。Inline 和 LateInline 标注的区别是，LateInline 标注表明应在符号运算完成后执行内嵌功能。关于函数内嵌以及符号运算之间的交互关系超出了本书的讨论范围。

还有一点需要注意，当一个函数内同时应用了 Inline 和 LateInline 标注时，LateInline 标注的优先级要高于 Inline 标注，即：

Inline	LateInline	说明
false	false	Inline=false
true	false	Inline=true
false	true	LateInline=true
true	true	LateInline=true

## 外部函数

最后一大类标注是与定义的 external 函数相关的。这些外部函数的定义通常都依靠外部文件或库。使用这类标注就是为了通知 Modelica 编译器这些依赖关系以及如何定位这些外部函数。

## Include

通常情况下，Modelica 编译器生成代码过程中如果使用了某个外部库，需添加相应的 `Include` 标注，以此标示包含语句。`Include` 标注的值是字符串，并且会嵌入到生成的代码文件中，例如：

```
annotation(Include="#include \"mydefs.h\"");
```

---

Note: `Include` 标注的值是一个字符串，如果它包含嵌入的字符串，需进行相应的转义。

---

## IncludeDirectory

在 `Include` ( 137) 标注，我们已经讨论了该标注允许在生成的代码内插入路径。`IncludeDirectory` 标注指定了 `Include` 标注声明的内容应在哪个目录下搜索。

该标注的值是一个字符串。字符串可以是一个目录或者 URL。例如，`IncludeDirectory` 标注的默认值如下所示：

```
IncludeDirectory=modelica://LibraryName/Resources/Include
```

稍后，我们将对 `modelica:// URL 地址` ( 149) 的含义进行解释。

## Library

`Library` 标注主要用于指定函数可能依赖的所有编译库，该标注的值可以是一个表示库名字的字符串，也可以是上述字符串组成的数组，即：

```
annotation(Library="somelib");
```

或者

```
annotation(Library={"onelib","anotherlib"});
```

Modelica 编译器在“链接”生成代码时将使用这些信息。

## LibraryDirectory

对于 `Library` 和 `Include` 标注，我们有同样的问题。`Library` 标注告诉我们需要添加那些库文件，但却没有告诉我们去哪里查找。另外，`LibraryDirectory` 与 `IncludeDirectory` ( 137) 标注具有相同的作用，如同 `IncludeDirectory` 标注，`LibraryDirectory` 标注的值也可以是一个 URL，其默认值如下所示：

```
LibraryDirectory=modelica://LibraryName/Resources/Library
```



## 第二部分

# 面向对象建模



## 包

到目前为止，我们已经介绍了所有类型的模型，而没有任何真正讨论过如何正确地对模型进行组织。在某些情况下，如在 NewtonCoolingWithTypes 例子讨论[添加物理量类型信息](#) (10) 时，把所有信息放在单一的模型内实在有些奇怪。有许多情况，相同的信息在多个模型里被重复，这使得对这些模型的维护变得非常困难。

好消息是，许多这样的前述例子能够通过借助 Modelica 的 package 系统得到极大改善。package 在概念上类似于目录。它包含一系列 Modelica 实体的集合。通过引用或者导入这些实体，我们就可以避免重复。

本章提供了几个例子，去演示如何使用 Modelica 的包这个功能。本章结束时将讨论[Modelica 标准库](#) ( 151)，这个所有 Modelica 工具都附带的包含了大量可重用内容的模型库。

## 第 5.1 节 示例

### 第 5.1.1 节 组织内容

让我们先来简单地演示如何将内容组织成包。要做到这一点，我们将重温[经典猎食者猎物系统](#) ( 18) 模型。在我们前述的模型中，所有的变量类型均是 Real。让我们来优化模型中一系列不同量的类型。

我们可以将这些类型组织成如下的包：

```
package Types
  type Rabbits = Real(quantity="Rabbits", min=0);
  type Wolves = Real(quantity="Wolves", min=0);
  type RabbitReproduction = Real(quantity="Rabbit Reproduction", min=0);
  type RabbitFatalities = Real(quantity="Rabbit Fatalities", min=0);
  type WolfReproduction = Real(quantity="Wolf Reproduction", min=0);
  type Wolffatalities = Real(quantity="Wolf Fatalities", min=0);
end Types;
```

对于这段 Modelica 代码，需要注意的第一件事情是，它使用了 package 关键字。定义 package 的语法与定义 model 或 function 的语法非常相似。而它们的主要区别在于 package 内仅包含定义或常量。package 定义除了 constant 量外不能包含任何变量声明。在本例，我们看到这个 package 只包含 type 的定义。

现在，让我们把注意力转向 Lotka-Volterra 模型本身。假设此模型自身不需要定义类型，而去依赖于我们刚才定义的类型，模型可以被重构如下：

```
model LotkaVolterra "Lotka-Volterra with types"
  parameter Types.RabbitReproduction alpha=0.1;
  parameter Types.RabbitFatalities beta=0.02;
  parameter Types.WolfReproduction gamma=0.4;
  parameter Types.WolfFatalities delta=0.02;
  parameter Types.Rabbits x0=10;
  parameter Types.Wolves y0=10;
```

```

Types.Rabbits x(start=x0);
Types.Wolves y(start=y0);
equation
  der(x) = x*(alpha-beta*y);
  der(y) = -y*(gamma-delta*x);
end LotkaVolterra;

```

请注意，现在所有的参数和变量如何得到一个特定类型（而不仅仅是普通的 Real 类型）。相反，我们能够将变量和附加信息相关联，而不局限于其为连续变量这一事实。例如，通过在类型定义加入 min=0 这一修饰符，我们可以规定变量非负。

仅通过观察 Lotka-Volterra 模型，我们很难发现模型在哪里找到上述的类型定义。Modelica 编译器会使用一系列查找规则（150）来查找上述定义。在后面我们将会介绍查找规则。就目前而言重要的一点是，我们有引用模型外数据的能力。

让我们稍稍离开目前的主题，去看看关于组织的一些额外细节。前述的 Types 包以及对其进行引用的 LotkaVolterra 模型均是一个称为 NestedPackages 的包的一部分。NestedPackages 的定义如下：

```

within ModelicaByExample.PackageExamples;
package NestedPackages
  "An example of how packages can be used to organize things"
  package Types
    type Rabbits = Real(quantity="Rabbits", min=0);
    type Wolves = Real(quantity="Wolves", min=0);
    type RabbitReproduction = Real(quantity="Rabbit Reproduction", min=0);
    type RabbitFatalities = Real(quantity="Rabbit Fatalities", min=0);
    type WolfReproduction = Real(quantity="Wolf Reproduction", min=0);
    type WolfFatalities = Real(quantity="Wolf Fatalities", min=0);
  end Types;

  model LotkaVolterra "Lotka-Volterra with types"
    parameter Types.RabbitReproduction alpha=0.1;
    parameter Types.RabbitFatalities beta=0.02;
    parameter Types.WolfReproduction gamma=0.4;
    parameter Types.WolfFatalities delta=0.02;
    parameter Types.Rabbits x0=10;
    parameter Types.Wolves y0=10;
    Types.Rabbits x(start=x0);
    Types.Wolves y(start=y0);
  equation
    der(x) = x*(alpha-beta*y);
    der(y) = -y*(gamma-delta*x);
  end LotkaVolterra;
end NestedPackages;

```

需要注意，关于 NestedPackages 包一个非常重要的点是，它是包含在另一个名为 PackageExamples 的包内。而 PackageExamples 则又是一个名为 ModelicaByExample 的包的一部分。由顶部的 within 子句我们便可以发现这点：

```
within ModelicaByExample.PackageExamples;
```

在这本书至今模拟过的每一个模型均是包含在包内的。而在展示那些例子的源代码时，我们都将首行隐藏了起来。原因是当时我们还没有准备好讨论 within 子句的作用。但 within 子句在那些例子中均是存在的。

注意，Types 包以及 LotkaVolterra 模型并没有任何的 within 子句。原因是由于它们均是直接定义在 NestedPackages 包内，因而我们知道它们具体在什么包内。那么，为什么这个子句会直接出现在 NestedPackages 的定义之前呢？这是因为 NestedPackages 是个独立的文件。换言之，Modelica 定义映射到文件和目录时，我们需要明确指定定义之间的关系。我们将在后面讨论的文件、目录和包的定义（147）之间的关系。暂时而言，重要的是了解 within 子句仅用于指定文件所从属的父包。

## 第 5.1.2 节 引用包内内容

现在，我们已经介绍了组织内容（141）。下面我们将开始讨论如何访问不同包里的内容。让我们观察以下例子：

```
within ModelicaByExample.PackageExamples;
model RLC "An RLC circuit referencing types from the Modelica Standard Library"
  parameter Modelica.SIunits.Voltage Vb=24 "Battery voltage";
  parameter Modelica.SIunits.Inductance L = 1;
  parameter Modelica.SIunits.Resistance R = 100;
  parameter Modelica.SIunits.Capacitance C = 1e-3;
  Modelica.SIunits.Voltage V;
  Modelica.SIunits.Current i_L;
  Modelica.SIunits.Current i_R;
  Modelica.SIunits.Current i_C;
equation
  i_R = V/R;
  i_C = C*der(V);
  i_L=i_R+i_C;
  L*der(i_L) = (Vb-V);
end RLC;
```

正如我们在上节里知道的，第一行：

```
within ModelicaByExample.PackageExamples;
```

告诉我们 RLC 模型是 ModelicaByExample.PackageExamples 包的一部分。正如前面的例子一样，我们将利用 Modelica 的 package 系统，以避免直接在模型中定义类型。这样一来，我们一旦在包内定义了类型，就可以简单地通过引用而在不同的地方而对类型进行重用。

不像本章之前例子，我们并没有在本例内定义任何类型。相反，我们依赖于在 Modelica 标准库（151）内定义的类型。Modelica 标准库（151）包含了许多有用的类型、模型、常量等。在本例里，我们只会利用其中的几个。这些类型可以很容易识别，因为其名字均以 Modelica. 开头。

在本章后面，我们会进一步研究查找规则（150）。就目前而言，可以暂时认为所有以 Modelica. 开头的类型均存在于 Modelica 包内。在此例里，所有的类型均以 Modelica.SIunits 开始。SIunits 为 Modelica 包内的一个包。SIunits 包的目的是存储符合 ISO 标准的物理量和量度单位定义。

在示例代码中可以看出，这些类型是以其“全限定名”引用的。这意味着，类型名称的开头是顶层包（不包含在其他包内的包）的名称。每个在名称里的. 代表一个新的子包。序列中的最后一个名称标记所引用的实际类型。

在本例里，我们从 Modelica.SIunits 包内使用了 5 种不同的类型：Voltage、Inductance、Resistance、Capacitance 以及 Current。这些类型提供了每种类型的单位，以及对这些类型值所在范围的限制（例如，电容不能小于零）等信息。这些类型在 Modelica 标准库（151）内定义如下：

```
// Base Definitions
type ElectricPotential = Real(final quantity="ElectricPotential",
                                 final unit="V");
type ElectricCurrent = Real(final quantity="ElectricCurrent",
                            final unit="A");

// The types referenced in our example
type Voltage = ElectricPotential;
type Inductance = Real(final quantity="Inductance",
                       final unit="H");
type Resistance = Real(final quantity="Resistance",
                       final unit="Ohm");
type Capacitance = Real(final quantity="Capacitance",
                        final unit="F", min=0);
type Current = ElectricCurrent;
```

除了能提供更好的文档外，将变量和类型关联起来有一个直接的好处。那就是支持方程单位的一致性检查。例如，注意本例中的以下等式：

```
i_R = V/R;
```

显然，这是对欧姆定律的陈述。但是，假若我们犯了一个错误，而将其意外写成了：

```
i_R = V*R;
```

在语法上来讲，这个公式是完全合法的。再者，若变量 `i_R`、`V`、`R` 均声明为 `Real` 类型，那么上述方程便没有任何问题。然而，因为（从类型定义）我们知道，这些变量分别表示电流、电压和电阻。Modelica 语言编译器由此能够确定（完全自动地使用上述定义），这个等式的左侧和右侧的物理单位不一致。换句话说，将变量和物理类型相关联可以自动地检查建模错误。

### 第 5.1.3 节 导入物理类型

在前面的小节里，我们学会了如何引用其他包中定义的类型。这令开发者不必不断地在自己的局部模型进行重复定义。相反，他们可以将定义放在包内，然后去引用这些软件包。

不过，一遍又一遍地键入冗长的全限定名进行引用不一定十分有趣。因此，Modelica 语言包括 `import` 语句，令我们可以方便地使用这些定义，仿佛这些定义是在本模型进行的一样。

再次回忆前面讨论物理类型 (10) 时的这个例子：

```
within ModelicaByExample.BasicEquations.CoolingExample;
model NewtonCoolingWithTypes "Cooling example with physical types"
  // Types
  type Temperature=Real(unit="K", min=0);
  type ConvectionCoefficient=Real(unit="W/(m2.K)", min=0);
  type Area=Real(unit="m2", min=0);
  type Mass=Real(unit="kg", min=0);
  type SpecificHeat=Real(unit="J/(K.kg)", min=0);

  // Parameters
  parameter Temperature T_inf=298.15 "Ambient temperature";
  parameter Temperature T0=363.15 "Initial temperature";
  parameter ConvectionCoefficient h=0.7 "Convective cooling coefficient";
  parameter Area A=1.0 "Surface area";
  parameter Mass m=0.1 "Mass of thermal capacitance";
  parameter SpecificHeat c_p=1.2 "Specific heat";

  // Variables
  Temperature T "Temperature";
initial equation
  T = T0 "Specify initial value for T";
equation
  m*c_p*der(T) = h*A*(T_inf-T) "Newton's law of cooling";
end NewtonCoolingWithTypes;
```

前面已经介绍了，我们要如何通过使用 Modelica 标准库 (151) 内的类型以避免进行本地定义。但是，我们也可以使用 `import` 命令一次性地从 Modelica 语言标准库中导入这些类型，然后不需输入指定其全限定名而去使用它们。这样代码变为如下：

```
within ModelicaByExample.PackageExamples;
model NewtonCooling
  "Cooling example importing physical types from the Modelica Standard Library"
  import Modelica.SIunits.Temperature;
  import Modelica.SIunits.Mass;
  import Modelica.SIunits.Area;
  import ConvectionCoefficient = Modelica.SIunits.CoefficientOfHeatTransfer;
  import SpecificHeat = Modelica.SIunits.SpecificHeatCapacity;

  // Parameters
  parameter Temperature T_inf=300.0 "Ambient temperature";
  parameter Temperature T0=280.0 "Initial temperature";
```

```

parameter ConvectionCoefficient h=0.7 "Convective cooling coefficient";
parameter Area A=1.0 "Surface area";
parameter Mass m=0.1 "Mass of thermal capacitance";
parameter SpecificHeat c_p=1.2 "Specific heat";

// Variables
Temperature T "Temperature";
initial equation
T = T0 "Specify initial value for T";
equation
m*c_p*der(T) = h*A*(T_inf-T) "Newton's law of cooling";
end NewtonCooling;

```

在这里，我们将类型定义换成 import 语句。注意被高亮行如何等价于以前的代码。让我们仔细观察其中的两个导入语句以了解其对模型的影响。首先观察下面的导入语句：

```
import Modelica.SIunits.Temperature;
```

这将类型 Modelica.SIunits.Temperature 导入到当前的模型。默认情况下，导入类型的名称会是全限定名最后一项的名称，即 Temperature。这意味着，只要有上述 import 语句，我们可以简单地使用类型名 Temperature，而引用将自动转到 Modelica.SIunits.Temperature 处。

现在，让我们来看看另一种 import 语句：

```
import ConvectionCoefficient = Modelica.SIunits.CoefficientOfHeatTransfer;
```

这里的语法有点不同。在这种情况下，我们正在导入的类型是 Modelica.SIunits.CoefficientOfHeatTransfer。但是，相对于直接用全限定名最后一项的名称，即 CoefficientOfHeatTransfer，我们将此类型的本地名称指定为 ConvectionCoefficient。在这里，改变名称允许我们使用在最初几个例子中定义的名称。通过这种方式，我们能够避免重构任何使用了旧名称的代码。另一个指定替代名称（而非 Modelica 编译器通常指派的默认名称）的原因是为了避免命名冲突。试想，我们希望从两个不同的包导入两种类型，如：

```
import Modelica.SIunits.Temperature; // Celsius
import ImperialUnits.Temperature; // Fahrenheit
```

这将使两种名为 Temperature 的类型。通过为本地别名定义替代名称，我们进行以下行为：

```
import DegK = Modelica.SIunits.Temperature; // Kelvin
import DegR = ImperialUnits.Temperature; // Rankine
```

## 国际单位

请注意，此示例导入英制单位只是为了演示潜在的命名冲突可能如何发生。但这样做在实践中是非常不好的做法。使用 Modelica 语言时，用户应该完全使用国际单位，切勿使用任何其他单位系统。如果你想用其他单位来输入数据或显示效果，请使用在属性 (27) 小节里讨论的 displayUnit 属性。

import 语句最后一个值得讨论的形式是通配符导入语句。一个一个地导入单位可能有些乏味。通配符导入允许我们从给定的包里一次性导入所有类型。回想一下早前的如下例子：

```

within ModelicaByExample.BasicEquations.RotationalSMD;
model SecondOrderSystem "A second order rotational system"
type Angle=Real(unit="rad");
type AngularVelocity=Real(unit="rad/s");
type Inertia=Real(unit="kg.m2");
type Stiffness=Real(unit="N.m/rad");
type Damping=Real(unit="N.m.s/rad");
parameter Inertia J1=0.4 "Moment of inertia for inertia 1";
parameter Inertia J2=1.0 "Moment of inertia for inertia 2";
parameter Stiffness k1=11 "Spring constant for spring 1";
parameter Stiffness k2=5 "Spring constant for spring 2";

```

```

parameter Damping d1=0.2 "Damping for damper 1";
parameter Damping d2=1.0 "Damping for damper 2";
Angle phi1 "Angle for inertia 1";
Angle phi2 "Angle for inertia 2";
AngularVelocity omega1 "Velocity of inertia 1";
AngularVelocity omega2 "Velocity of inertia 2";
initial equation
  phi1 = 0;
  phi2 = 1;
  omega1 = 0;
  omega2 = 0;
equation
  // Equations for inertia 1
  omega1 = der(phi1);
  J1*der(omega1) = k1*(phi2-phi1)+d1*der(phi2-phi1);
  // Equations for inertia 2
  omega2 = der(phi2);
  J2*der(omega2) = k1*(phi1-phi2)+d1*der(phi1-phi2)-k2*phi2-d2*der(phi2);
end SecondOrderSystem;

```

我们可以用导入语句替换这些类型定义，如：

```

import Modelica.SIunits.Angle;
import Modelica.SIunits.AngularVelocity;
import Modelica.SIunits.Inertia;
import Stiffness = Modelica.SIunits.RotationalSpringConstant;
import Damping = Modelica.SIunits.RotationalDampingConstant;

```

不过，导入的类型越多，我们需要添加的导入语句就越多。相对地，我们可以将模型写成如下形式：

```

within ModelicaByExample.PackageExamples;
model SecondOrderSystem
  "A second order rotational system importing types from Modelica Standard Library"
  import Modelica.SIunits.*;
  parameter Angle phi1_init = 0;
  parameter Angle phi2_init = 1;
  parameter AngularVelocity omega1_init = 0;
  parameter AngularVelocity omega2_init = 0;
  parameter Inertia J1=0.4;
  parameter Inertia J2=1.0;
  parameter RotationalSpringConstant k1=11;
  parameter RotationalSpringConstant k2=5;
  parameter RotationalDampingConstant d1=0.2;
  parameter RotationalDampingConstant d2=1.0;
  Angle phi1;
  Angle phi2;
  AngularVelocity omega1;
  AngularVelocity omega2;
initial equation
  phi1 = phi1_init;
  phi2 = phi2_init;
  omega1 = omega1_init;
  omega2 = omega2_init;
equation
  omega1 = der(phi1);
  omega2 = der(phi2);
  J1*der(omega1) = k1*(phi2-phi1)+d1*der(phi2-phi1);
  J2*der(omega2) = k1*(phi1-phi2)+d1*der(phi1-phi2)-k2*phi2-d2*der(phi2);
end SecondOrderSystem;

```

注意高亮的 import 语句。这条（通配符）导入语句从 Modelica.SIunits 导入其中的所有定义到当前的模型里。通配符导入语句不能“重命名”类型。导入类型的本地名称将和其在包内的名称一样。

在使用通配符进口之前，请务必阅读[这条警告 \( 151 \)](#)。

在本章里，我们已经看到 import 语句如何用于从其他包内导入类型。事实证明，import 语句并非总是那么有用的。当使用图形化建模环境开发模型时，工具一般采用歧义最少、最明确的方法去引用类型：使用全限定名。毕竟，使用图形工具时，名称的长度不会是问题了，用户不再需要输入类型的名称。这也避免了名称查找，命名冲突等问题。

## 第 5.2 节 回顾

### 第 5.2.1 节 包的定义

#### 基本语法

正如我们在本章所了解的一样，package 是一种允许我们对定义进行组织（包含其它包的定义）的 Modelica 实体。package 的语法定义与其它 Modelica 定义有很多共同点。定义包的一般语法是：

```
package PackageName "Description of package"
  // A package can contain other definitions or variables with the
  // constant qualifier.
end PackageName;
```

包定义可用 encapsulated 限定词作为前缀。我们将在考察的 Modelica 的 [查找规则](#) (150) 时进行进一步讨论。

如果必要包也可以进行嵌套例如：

```
package OuterPackage "A package that wraps a nested package"
  // Anything contained in OuterPackage
  package NestedPackage "A nested package"
    // Things defined inside NestedPackage
  end NestedPackage;
end OuterPackage;
```

实际上，包嵌套十分常见。这使我们能够表达复杂的分类。

#### 文件夹储存

虽然我们将整个 Modelica 定义库存作为一系列嵌套的包储存在单一的文件内，但至少有两个原因令这种做法不可取。首先，由于文件长度和缩进层次，所得到的文件将相当没有可读性。然后，从版本控制的角度来看，将库分成更小的文件可以帮助避免合并冲突。

##### 储存在单一文件内

Modelica 语言源代码可以用数种方法映射到文件系统中。最简单的方法将所有信息存储为一个文件。此类文件会带有.mo 后缀。这个文件可能只包含一个模型定义，也可能包含一个内有深层嵌套的包或任何两者之间的内容。

##### 储存为文件夹

正如上述讨论，将一切存储在单个文件内通常不是一个好主意。另一种方法是 Modelica 语言定义映射到一个目录结构里。要将包储存为一个文件夹，可以通过创建一个与这个包名称相同的文件夹以达成目的。然后，该目录内必须有一个名为 package.mo 的文件存储此包内的定义，不过不包括任何嵌套定义。嵌套定义可以存储为（如上文所述的）单个文件，或者（如在本段中描述般）作为目录形式。下图试图以可视化形式描述一个文件夹布局的例子：

```
/RootPackage      # Top-level package stored as a directory
  package.mo      # Indicates this directory is a package
  NestedPackageAsFile.mo  # Definitions stored in one file
  /NestedPackageAsDir    # Nested package stored as a directory
    package.mo      # Indicates this directory is a package
```

与名为 RootPackage 的包关联的 package.mo 文件将有类似如下的形式:

```
within;
package RootPackage
  // only annotations can be stored in a package.mo
end RootPackage;
```

这里要注意两点。首先，这里没有本应存在的 within 子句。这表明该包不在任何其它的包内。再者，NestedPackageAsFile 和 NestedPackageAsDir 的定义并没有(也不能)出现。两者必须存储在 package.mo 文件之外。

同样，与名为 NestedPackageAsDir 的包关联的 package.mo 文件如下:

```
within RootPackage;
package NestedPackageAsDir
  // only annotations can be stored in a package.mo
end NestedPackageAsDir;
```

再一次，此包没有包含任何定义，只有标注。within 子句则略有不同，以反映 NestedPackageAsDir 属于 RootPackage 包这一事实。

最后，NestedPackageAsFile.mo 文件看起来会是这样的:

```
within RootPackage;
package NestedPackageAsFile
  // The following can be stored here including:
  // * constants
  // * nested definitions
  // * annotations
end NestedPackageAsFile;
```

这里的 within 子句和 NestedPackageAsDir 的包内的一致。不过，由于我们将本包存为单独的文件，常量、模型、包、函数等的嵌套定义均是允许的。

### 文件夹内的排序

若所有定义被存储在单个文件内，那么定义出现在文件中出现的顺序就是其在可视化环境（例如包浏览器）内的顺序。但是，若定义被存储在文件系统中，上述隐含的顺序便不存在。出于这个原因，开发者可以在 package.mo 旁边加入一个可选的 package.order 文件以指定定义的顺序。该文件不过是包内嵌套的实体名称一行一个地组成的列表。因此，例如我们要为示例包结构加上顺序，那么文件系统将有如下内容:

```
/RootPackage      # Top-level package stored as a directory
  package.mo      # Indicates this directory is a package
  package.order    # Specifies an ordering for this package
  NestedPackageAsFile.mo  # Definitions stored in one file
  /NestedPackageAsDir    # Nested package stored as a directory
    package.mo      # Indicates this directory is a package
    package.order    # Specifies an ordering for this package
```

若没有 package.order 文件，Modelica 语言工具很可能会简单地按字母顺序对包的内容进行排序。但如果我们将 RootPackage 的内容按字母顺序逆向排序，那么 RootPackage 文件夹内的 package.order 文件将如下所示:

```
NestedPackageAsFile
NestedPackageAsDir
```

这将告诉 Modelica 工具 NestedPackageAsFile 应该排在 NestedPackageAsDir 之前。

## 版本

### MODELICAPATH

多数 Modelica 工具允许用户通过指定文件的完整路径，或通过使用文件选择对话框打开文件。但每次查找打开大量文件并不十分有趣。出于这个原因，Modelica 规范定义了名为 MODELICAPATH 的特殊环境变量。用户可以用此变量来帮助工具自动地定位源代码的位置。

MODELICAPATH 环境变量包含了应搜索目录的列表。在 Windows 中，该列表的分隔符为;，而在 Unix 中则为:。当 Modelica 编译器遇到一个尚未读取的包时，编译器会从 MODELICAPATH 环境变量的给定目录里搜索寻找匹配的文件或文件夹。例如，若 MODELICAPATH 定义如下（假设我们使用 Unix 惯例）：

```
/home/mtiller/Dir1:/home/mtiller/Dir2
```

而编译器在寻找一个名为 MyLib 的包。那么编译器会首先在/home/mtiller/Dir1 内寻找名为 MyLib.mo 的（作为单个文件存储的）包，或者一个名为 MyLib 且内含 package.mo 文件的文件夹。如果两者都没有出现，那么编译器会继而在/home/mtiller/Dir1 内（对相同的内容）进行查找。

### modelica:// URL 地址

在许多情况下，我们需要在 Modelica 包内包含非 Modelica 文件。这些非 Modelica 的文件可能包括数据、脚本、图像等。我们将这些非 Modelica 的文件称为“资源（resource）”。现在，我们已经介绍了 Modelica 定义是如何映射到文件系统中的。因此，我们可以讨论 Modelica 的一个非常有用的功能，即利用 URL 地址来指代资源的位置。

例如，当我们讨论 [外部函数 \(136\)](#) 时，我们引入了数个标注来指定资源的位置。具体而言，IncludeDirectory 和 LibraryDirectory 标注分别指定了 Modelica 编译器寻找文件和库文件的位置。正如之前简要地提到的一样，上述标注的默认值以 modelica://LibraryName/Resources 开始。这样的 URL 地址允许我们相对于一个给定 Modelica 定义所在文件系统上的位置定义资源的位置。让我们重新考虑前述的目录结构，不过这次我们增加了一些资源文件：

```
/RootPackage          # Top-level package stored as a directory
  package.mo         # Indicates this directory is a package
  package.order      # Specifies an ordering for this package
  NestedPackageAsFile.mo # Definitions stored in one file
/NestedPackageAsDir    # Nested package stored as a directory
  package.mo         # Indicates this directory is a package
  package.order      # Specifies an ordering for this package
  datafile.mat       # Data specific to this package
/Resources            # Resources are stored here by convention
  logo.jpg          # An image file
```

如果我们有模型需要包含在 NestedPackageAsDir 内的数据文件，我们可以使用下面的 URL 来引用这个文件：

```
modelica://RootPackage/NestedPackageAsDir/datafile.mat
```

这样的 URL 地址以 modelica://开始。这表示我们所引用的资源相对于一个 Modelica 模型所在的位置。请不要将其错误理解为，比如说，一个通过网络获取的文件。在//后面接着是 Modelica 定义的完全限定名，唯一的不同只是组件之间是由一个. 分隔开的，而是用/。Modelica 编译器会将其解释为包含该定义的文件夹名称。最后，URL 的最末元素确定要使用文件的名称。

在另一个例子中，如果要引用 Resources 包中的 logo.jpg 文件，我们会使用如下 URL：

```
modelica://RootPackage/Resources/logo.jpg
```

对于模型库的相关文件，常用惯例是将其存储在名为 Resources 的子包内（因此 IncludeDirectory 和 LibraryDirectory 有上述默认值）。

## 第 5.2.2 节 查找规则

回想我们在讨论组织内容 (141) 时的以下例子:

```
within ModelicaByExample.PackageExamples;
package NestedPackages
    "An example of how packages can be used to organize things"
    package Types
        type Rabbits = Real(quantity="Rabbits", min=0);
        type Wolves = Real(quantity="Wolves", min=0);
        type RabbitReproduction = Real(quantity="Rabbit Reproduction", min=0);
        type RabbitFatalities = Real(quantity="Rabbit Fatalities", min=0);
        type WolfReproduction = Real(quantity="Wolf Reproduction", min=0);
        type Wolffatalities = Real(quantity="Wolf Fatalities", min=0);
    end Types;

    model LotkaVolterra "Lotka-Volterra with types"
        parameter Types.RabbitReproduction alpha=0.1;
        parameter Types.RabbitFatalities beta=0.02;
        parameter Types.WolfReproduction gamma=0.4;
        parameter Types.WolfFatalities delta=0.02;
        parameter Types.Rabbits x0=10;
        parameter Types.Wolves y0=10;
        Types.Rabbits x(start=x0);
        Types.Wolves y(start=y0);
    equation
        der(x) = x*(alpha-beta*y);
        der(y) = -y*(gamma-delta*x);
    end LotkaVolterra;
end NestedPackages;
```

在我们讨论引用包内内容 (143) 时, 演示例子中所有引用了的类型均使用了全限定名。但上述例子则不然。我们从 LotkaVolterra 模型中可以看到, Wolves 类型被引用为:

```
parameter Types.Wolves y0=10;
```

而不是:

```
parameter ModelicaByExample.PackageExamples.NestedPackages.Types.Wolves y0=10;
```

换言之, 我们没有使用全限定名。但 LotkaVolterra 模型的编译毫无问题。那么 Modelica 编译器是如何知道要使用哪个 Wolves 的定义呢?

答案涉及到 Modelica 的“名称查找”。Modelica 的名称查找涉及指定定义的搜索。Modelica 的类型名称一般为限定(虽然不一定为完全限定)名称。这意味着名称中可能包含了..。例如 Modelica.SIunits.Voltage。为了找到与一个名称相匹配的定义, Modelica 的编译器开始由限定名称中寻找第一项名称, 如 Modelica。编译器按以下顺序搜索匹配的定义:

1. 在内建类型中寻找匹配的名称
2. 在当前定义 (包括了其继承的定义) 内寻找 j 的具有匹配名称的内嵌定义
3. 查找在当前定义内名称匹配的导入定义 (不包括继承的导入定义)
4. 在当前定义的父包查找与名称匹配的内嵌定义 (包括继承的定义)
5. 查找在当前定义的父包内名称匹配的导入定义 (不包括继承的导入定义)
6. (用上述算法) 上溯每个父包直到下列条件至少有一个成立:
  - 某个父包有 encapsulated 限定词, 在这种情况下搜索终止。
  - 没有更多的父包, 在这种情况下会在根级别的包内搜索匹配。

如果编译器在搜索所有这些地点后不能发现给定名称, 那么搜索失败, 类型无法找到。如果搜索成功, 那么我们便找到在限定名定义里的首个名称。如果名称是非限定名称 (即名称内不包含.), 那么我们就完成任务了。然而, 如果名称中有其他组件, 那么该组件必须包含在由前述搜索返回的定义范围内, 亦

即前述定义内的内嵌定义。如果编译器在相应的内嵌定义里不能找到限定名称内的任一组件，那么搜索失败，类型无法找到。

第一眼看上去这很复杂。然而大多数情况下，这些规则其实不太重要。原因是，正如我们前面所讨论的一样，大多数的图形化 Modelica 环境会使用全限定名称。Modelica 代码里大部分类型的名称要么引用本地定义或是用全限定名称来指定。

### 重复命名

读者一定要避免在嵌套包里使用和顶级包相同的名称。原因是，基于查找规则，搜索会上溯包的层次结构，从而会造成一个问题。其结果是，编译器会先发现嵌套定义，而不是根级别的定义。这意味着基于全限定名称（相对于包根目录引用的名称）的查找将会失败，因为编译器将首先找到嵌套定义。

## 第 5.2.3 节 导入

正如我们前面所看到的，导入基本上有三种形式。在所有情况下，import 语句创建了一个“别名”去引用模型外定义的类型。

第一种形式采用类型的全限定名引入定义，例如：

```
import Modelica.SIunits.Temperature;
```

导入的结果是，对名称 Temperature 的引用会映射到全限定名 Modelica.SIunits.Temperature 处。换言之，由 import 声明引入别名 Temperature，并将其映射到在 Modelica.SIunits.Temperature 出现的定义。这种形式的导入会让别名总与被导入名称的最后一个元素相同。

在某些情况下，我们希望别名与被导入名称的最后一个元素不同。这种情况下，我们可以明确地引入一个替代名称。例如：

```
import DegK = Modelica.SIunits.Temperature; // Kelvin
```

在上述的导入后，我们就可以使用别名 DegK 指代 Modelica.SIunits.Temperature。提供替代名称可避免命名冲突，或者只是为了让模型更具可读性。

最后，也可以在一个包中导入所有定义到当前作用域。这可以通过通配符导入实现。例如，要导入 Modelica.SIunits 包内的所有定义，我们可以使用下面的 import 语句：

```
import Modelica.SIunits.*;
```

上述导入会创建和 Modelica.SIunits 内总定义数目相等的别名。别名名称的唯一可能性就是和所导入包内定义的原名相同（即不可能为别名分配替代名称）。

### 通配符有害

这种通配符导入很危险。其原因如上所述。用户不能对类型进行重命名。这样的结果是，在一个模型内进行两次或多次通配符导入可能产生命名冲突。再者，明确的导入（或直接使用全限定的类型）使得对于类型所关联定义的回溯与定位变得更容易。而使用通配符则使这变得非常困难。原因是使用通配符后，导入类型和所在包的关系便不再清晰。

## 第 5.2.4 节 Modelica 标准库

Modelica 包的功用是将常用的类型、模型、函数等组织成包，并直接通过对其的引用来实现重用（而不是重新建模）。但倘若每个用户都需要建立对常用定义制作模型包，那么重复建模的问题仍将存在。因此，Modelica 协会维护着一个 Modelica 标准库的模型包。这个库包括了对科学家和工程师有用的规定。

在本节中，我们概述 Modelica 标准库，让读者对标准库内可用的定义有一个大概的概念。由于这不会是一个彻底的介绍，而且 Modelica 标准库持续也在持续更新以及改进，这里的介绍并不一定会反映标准库最新版本的情况。但介绍涵盖了基础部分，作者希望由此能够让读者了解如何寻找需要的定义。

## 常数

Modelica 的标准库包含了一些常见的物理、计算机常量的定义。此部分由于篇幅较少，其源码可以直接引用如下。以下内容来自 Modelica 标准库 3.2.1 版本内的 Modelica.Constants 包（带有一些排版上的美化）：

```
within Modelica;
package Constants
    "Library of mathematical constants and constants of nature (e.g., pi, eps, R, sigma)"

    import SI = Modelica.SIunits;
    import NonSI = Modelica.SIunits.Conversions.NonSIunits;

    // Mathematical constants
    final constant Real e=Modelica.Math.exp(1.0);
    final constant Real pi=2*Modelica.Math.asin(1.0); // 3.14159265358979;
    final constant Real D2R=pi/180 "Degree to Radian";
    final constant Real R2D=180/pi "Radian to Degree";
    final constant Real gamma=0.57721566490153286060
        "see http://en.wikipedia.org/wiki/Euler\_constant";

    // Machine dependent constants
    final constant Real eps=ModelicaServices.Machine.eps
        "Biggest number such that 1.0 + eps = 1.0";
    final constant Real small=ModelicaServices.Machine.small
        "Smallest number such that small and -small are representable on the machine";
    final constant Real inf=ModelicaServices.Machine.inf
        "Biggest Real number such that inf and -inf are representable on the machine";
    final constant Integer Integer_inf=ModelicaServices.Machine.Integer_inf
        "Biggest Integer number such that Integer_inf and -Integer_inf are representable on the machine";

    // Constants of nature
    // (name, value, description from http://physics.nist.gov/cuu/Constants/)
    final constant SI.Velocity c=299792458 "Speed of light in vacuum";
    final constant SI.Acceleration g_n=9.80665
        "Standard acceleration of gravity on earth";
    final constant Real G(final unit="m3/(kg.s2)") = 6.6742e-11
        "Newtonian constant of gravitation";
    final constant SI.FaradayConstant F = 9.64853399e4 "Faraday constant, C/mol";
    final constant Real h(final unit="J.s") = 6.6260693e-34 "Planck constant";
    final constant Real k(final unit="J/K") = 1.3806505e-23 "Boltzmann constant";
    final constant Real R(final unit="J/(mol.K)") = 8.314472 "Molar gas constant";
    final constant Real sigma(final unit="W/(m2.K4)") = 5.670400e-8
        "Stefan-Boltzmann constant";
    final constant Real N_A(final unit="1/mol") = 6.0221415e23
        "Avogadro constant";
    final constant Real mue_0(final unit="N/A2") = 4*pi*1.e-7 "Magnetic constant";
    final constant Real epsilon_0(final unit="F/m") = 1/(mue_0*c*c)
        "Electric constant";
    final constant NonSI.Temperature_degC T_zero=-273.15
        "Absolute zero temperature";
```

值得注意的是 `pi`、`e`、`g_n` 以及 `eps` 的定义。

前两个定义 `pi` 和 `e` 分别表示数学常数 `pi` 以及 `e`。这两个常量定义不仅免去了用户自己提供这些（无理数）常量的麻烦，而且使用这些定义可以让用户得到一个在精度范围限制内的最优值。

紧接的常数，`g_n` 是代表地球的重力加速度（可用于计算势能如:math:m g h）。

最后，`eps` 是一个计算机相关的常数，代表一个为任何正在使用计算平台 “极小数”。

## 国际单位

正如上述讨论的一样，为参数与变量标示单位不仅可以让提高代码可读性，而且 Modelica 编译器可以由此测试变量量纲是否匹配。出于这个原因，尽量在参数和变量里使用物理类型是相当有益的。

Modelica.SIunits 包是内容非常多，而且充满了很少使用的物理单位。之所以包含这些非常用单位是为了完整地遵守 ISO 31-1992 标准。以下是例子说明了 SIunits 包中常用物理单位是如何定义的：

```
type Length = Real (final quantity="Length", final unit="m");
type Radius = Length(min=0);
...
type Velocity = Real (final quantity="Velocity", final unit="m/s");
type AngularVelocity = Real(final quantity="AngularVelocity",
    final unit="rad/s");
...
type Mass = Real(quantity="Mass", final unit="kg", min=0);
type Density = Real(final quantity="Density", final unit="kg/m3",
    displayUnit="g/cm3", min=0.0);
type MomentOfInertia = Real(final quantity="MomentOfInertia",
    final unit="kg.m2");
...
type Pressure = Real(final quantity="Pressure", final unit="Pa",
    displayUnit="bar");
...
type ThermodynamicTemperature = Real(
    final quantity="ThermodynamicTemperature",
    final unit="K",
    min = 0.0,
    start = 288.15,
    nominal = 300,
    displayUnit="degC")
"Absolute temperature (use type TemperatureDifference for relative temperatures)";
type Temperature = ThermodynamicTemperature;
type TemperatureDifference = Real(final quantity="ThermodynamicTemperature",
    final unit="K");
```

## 模型

Modelica 标准库包含了很多描述不同特定领域的模型库。本节概述这些领域的内容，并讨论如何在各个领域模型的组织。

### 框图

Modelica 标准库包含了一系列模型用以进行因果性的框图模型的建模。这些模型的定义均在 Modelica.Blocks 包内。本库内的模型包括了以下例子：

- 输入端口（Real、Integer 以及 Boolean）
- 输出端口（Real、Integer 以及 Boolean）
- 增益模块、加法器模块、乘法模块
- 积分与微分模块
- 死区以及滞回模块
- 逻辑及关系运算模块
- 多路复用器和多路分配器模块

Blocks 包里包含了相当多种对信号进行操作的模块。这样的模块常用于描述控制系统和控制策略内的功能模块。

## 电气

Modelica.Electrical 包内有数个子包，分别用于描述模拟、数字以及多相的电子系统。相当多种对信号进行操作的模块。此包内还包括一个描述基本电机的模型。在这个库中，你会发现以下形式的部件：

- 电阻、电容、电感
- 电压源和电流源
- 电压和电流传感器
- 晶体管以及其他半导体相关模型
- 二极管与开关
- 逻辑门
- 星形与三角连接（多相位）
- 同步电机与异步电机
- 电机模型（直流电机、永磁电机等）
- Spice3 模型

## 机械

Modelica.Mechanics 库包含三个主要的库。

**Translational (平移)** 平移库包含用于模拟的一维平移运动的组件模型。这个库包含了以下组件：

- 弹簧、减震器以及间隙
- 质点
- 传感器和执行器
- 摩擦

**Rotational (转动)** 旋转库包含用于模拟一维旋转运动的组件模型。这个库包含了以下组件：

- 弹簧、减震器以及间隙
- 惯量
- 离合器和制动器
- 齿轮机构
- 传感器和执行器

**MultiBody (多体)** 多体库包含用于模拟三维机械系统的组件模型。这个库包含了以下组件：

- 物体（包括相关的惯性张量和 3D CAD 几何结构）
- 关节（如棱形关节、回转关节、万向关节）
- 传感器和执行器

## 流体与介质

Modelica 标准库中有两个包与流体系统建模有关。首先是 Modelica.Media。此库包括各种介质的属性模型，如：

- 理想气体（基于 NASA 格伦系数）
- 空气（干燥空气、参考状态空气、潮湿空气）
- 水（简单，含盐，两相）
- 一般不可压缩流体
- R134a（四氟乙烷）制冷剂

这些属性模型为计算各种纯液体和混合物的流体性质如焓，密度和比热比提供了函数。

除了这些介质模型，Modelica 标准库还包括 Modelica.Fluid 库，流体库提供了一系列组件用以描述流体装置，如：

- 容积、水箱与合流点
- 管道
- 泵
- 阀
- 压力损失
- 热交换器
- 源以及环境条件

## 磁

Modelica.Magnetic 库包含两个子包。其一是用以构造的磁性元件集总网络模型的 FluxTubes 包。这包括了一系列组件用以表示基本圆柱形和棱柱形元件的磁特性形状、相关的传感器以及执行器。其二则是被用于模拟在旋转电机电场的 FundamentalWave 库。

## 热

Modelica.Thermal 库有两个子包：

**HeatTransfer (热传递)** HeatTransfer 用以模拟集总模型的热传递。这个库中的模型可以用于建立集总热网络模型，如：

- 集总热容
- 热传导
- 热对流
- 热辐射
- 环境条件
- 传感器

**FluidHeatFlow (流体传热)** 一般而言，模拟热流体系统开发者应当使用 Modelica.Fluid 和 Modelica.Media，因为它们能够处理各种各样的涉及复杂媒介和多个相的问题。然而，对于某些特定的简单问题，FluidHeatFlow 库可以用来构建简单的热流体网络。

## 工具

Utilities 库为其他库以及模型开发者提供了支持功能。此库包括几个子包以处理建模里和数学无关的问题。

### Files (文件)

Modelica.Utilities.Files 库提供了函数用于访问和操作计算机的文件系统。以下函数提供了 Files 包:

- list - 列举一个文件或目录的内容
- copy - 复制一个文件或目录
- move - 移动一个文件或目录
- remove - 删除文件或目录
- createDirectory - 创建一个目录
- exist - 确定给定文件或目录是否已存在
- fullPathName - 得到给定文件或目录的完整路径
- splitPathName - 按路径分割的文件名
- temporaryFileName - 返回一个不存在的临时文件的名字。
- loadResource - 将 Modelica URL 地址 (149) 转换成一个文件系统的绝对路径(用于不接受 Modelica 的 URL 的函数)。

### Streams (流)

Streams 包用于从终端或文件读取和写入数据。此包括以下函数:

- print - 向终端或文件写入数据。
- readFile - 从文件中读取数据，并返回一个字符串向量以表示文件中各行的内容。
- readLine - 从文件中读取一行文字。
- countLines - 返回的文件的总行数。
- error - 用于打印错误信息。
- close - 关闭文件。

### Strings (字符串)

在 Strings 包中包含对字符串进行操作的函数。这个库的基本功能包括:

- 确定字符串的长度
- 构建和提取字符串
- 比较字符串
- 解析和搜索字符串

### System (系统)

System 包用于与底层操作系统的交互。它包括以下功能:

- getWorkingDirectory - 获取当前工作目录。
- setWorkingDirectory - 设置当前工作目录。

- `getEnvironmentVariable` - 获取特定环境变量的值。
- `setEnvironmentVariable` - 设置特定环境变量的值。
- `command` - 让操作系统来执行一个命令。
- `exit` - 终止执行。



## 连接器

### 第 6.1 节 简介

#### 第 6.1.1 节 基于组件的建模

在深入研究实例之前，读者需要了解一些必要的背景知识。这样的目的是去了解，什么是 Modelica 的 connector，为什么要使用 connector 以及其组成的数学基础。我们将在继续前首先介绍上述要点。

到目前为止，我们的讨论主要关于对行为的建模。我们至此看到的大多是由方程代码所组成的模型。但从现在开始，我们将探讨如何创建可重用的组件模型。因此，我们将不再在每次模拟电阻都写出欧姆定律。相反，我们将把电阻组件模型的实例添加到系统里。

到现在为止，我们看到的模型全部都是完整的。整个系统的所有行为均囊括在单个模型内，且通过方程代码来表示。但这种方法不能很好地扩展开。我们真正想要的，是创建可重用组件模型的能力。

但在研究我们如何可以创建这些部件之前，我们需要先讨论如何将部件连接在一起。我们在本书的其余部分也会继续讨论组件模型。这些组件模型仍将表示为 model 定义。但与迄今所见模型的不同点是，这些模型均带有连接器。

connector 是一种让模型与模型交换信息的方法。我们将看到其他来交换这些信息的方法。这一章将重点讲解在 Modelica 中用来描述连接器的各种不同语义。

#### 第 6.1.2 节 非因果连接

为了明白一类特定接口的语义，首先要了解物理系统的非因果表示方式。非因果物理建模方法区分了两类不同的变量。

我们将讨论的第一类变量是“横跨”变量（也称为势或功用变量）。横跨变量跨越部件时的增减值就是让部件运动的原因。横跨变量的典型例子有温度、电压和压力。我们将马上对这些变量进行讨论。这些量的差异值通常会导致在系统的动态行为。

我们将讨论的第二类变量是“穿越”变量（也称为流变量）。流变量通常代表了一些保守量，如质量、动量、能量、电荷等等。这些流通常是横跨变量在跨越部件时有差值的结果。例如，通过电阻的电流流动是由于电阻两侧的电压差。正如我们将在许多例子中看到的一样，穿越变量与横跨变量间有很多种不同的关系（欧姆定律仅是冰山一角）。

#### 正负号规则

一定一定要知道 Modelica 遵从如下的正负号规则：穿越变量的正值表示守恒量在流入某个组件。我们将在本书后面多次重复这个规则（特别是在我们开始讨论如何构建组件（173）模型的时候）。

下一节将为一些基本工程领域分别定义穿越变量和横跨变量。

## 第 6.2 节 示例

### 第 6.2.1 节 简单领域

本节中，我们将讨论一些比较简单的工程领域。这里指的是其 connector 仅带有一个穿越变量以及一个横跨变量的工程领域。概念上，这意味着该连接器仅涉及一个保守量。

下表涵盖了四种不同的工程领域。在每个领域中会分别看到所用到的穿越变量、横跨变量以及这些量相对应的国际单位。

领域	穿越变量	横跨变量
电气	电流 [A]	电压 [V]
热学	热 [W]	温度 [K]
平移	力 [N]	位置 [m]
旋转	力矩 [N.m]	角 [rad]

你可能在之前就看过了类似的表格，不过其选择的变量却有所不同。举个例子，你会有时会看到速度（单位为  $m/s$ ）作为平移运动的横跨变量以上选择受制于两个约束条件。

第一个约束条件要求穿越变量应为某个守恒量的时间导数。这样做的原因是穿越变量会被用于建立系统的广义守恒方程。因此，穿越变量为守恒量的导数便至关重要。

第二个约束条件则要求横跨变量应为领域内所有本构方程和经验公式内的最低阶导数。因此，举个例子，在平移运动时我们选择位置作为横跨变量，因为位置会用于描述弹簧的行为（即胡克定律）。如果我们选择了速度（位置的时间导数），那么我们就会一直处于一种尴尬局面。对比起位置，我们必须试图用速度去描述弹簧行为。这里有一点很重要的：微分是有损的。如果知道位置，我们可以很容易地表达速度。但倘若只知道速度，而没有额外的积分常数，我们就不能计算位置。这就是为什么我们希望横跨变量不要经过太多次微分。

现在让我们分别来看看每个领域。

#### 电气

我们可以定义电气领域的 connector，如下：

```
connector Electrical
  Modelica.SIunits.Voltage v;
  flow Modelica.SIunits.Current i;
end Electrical;
```

在这里我们看到，连接器内的变量  $v$  代表电压，而变量  $i$  则代表电流。

请注意电流  $i$  声明里的 flow 限定词。flow 限定词告诉了 Modelica 编译器， $i$  是穿越变量。回忆我们对非因果连接（159）的讨论：flow 变量应该是守恒量的时间导数。我们可以看到，该连接器遵循了上述规则。Current 的确为电荷的导数（而电荷为守恒量）。

注意电压  $v$  的声明里并无限定词。没有任何限定词的变量可以认为是 across 变量。在本章后面，你会见到对变量（167）（包括对其可用的各种限定词）的进一步讨论。

有兴趣的读者不妨跳到关于 [电气部件](#)（181）的章节，去看看我们是如何使用连接器定义来创建电路元件的。

#### 热学

用于建模集总传热的连接器和电气领域的连接器没有太大不同：

```
connector Thermal
  Modelica.SIunits.Temperature T;
  flow Modelica.SIunits.HeatFlowRate q;
end Thermal;
```

该连接器包括了 Temperature 和 HeatFlowRate，分别对应 Voltage 和 Current。虽然变量名称不同，两个连接器的整体结构基本相同。connector 包括一个穿越变量（带有 flow 限定词的 q）和一个横跨变量（没有限定词的 T）。我们再一次看到，带有 flow 限定词的变量 HeatFlowRate 是守恒量能量的时间导数。

在建模集总传热网路时，上述连接器可以用于建立部件模型。具体例子可以在下面关于[传热组件 \(173\)](#)的讨论里找到。如果你觉得对此 connector 的定义没有疑问，请尽管跳到该章节。但我仍然建议之后回过头来读读[连接器 \(159\)](#)章的[回顾 \(167\)](#)一节。

## 平移

为了对平移运动进行建模，我们将定义如下的连接器：

```
connector Translational
  Modelica.SIunits.Position x;
  flow Modelica.SIunits.Force f;
end Translational;
```

再一次，我们看到和之前相同的基本结构。connector 包括一个穿越变量 f 和一个横跨变量 x。需要注意的是，虽然这是个一维的力学连接器，但其物理类型特定于平移运动，不同于下面提到的[旋转 \(161\)](#)的物理类型。

对于力学连接器大家经常忽视了一个重要的事实，flow 变量确系某个守恒量的时间导数。例如，在平移运动的情况下，flow 变量 f 是力。力则是（线）动量的时间导数。而动量就正正是个守恒量。

## 旋转

只能进行转动运动的系统应使用下列的 Modelica connector 定义：

```
connector Rotational
  Modelica.SIunits.Angle phi;
  flow Modelica.SIunits.Torque tau;
end Rotational;
```

这里我们看到，横跨变量是 phi（代表角位移），而穿越变量则为 torque。与所有前述例子一样，flow 变量是守恒量的时间导数。在本例里，守恒量为角动量。

## SimpleDomains

为便于参考，所有在本节中定义的接口组成一个单一的包：

```
within ModelicaByExample.Connectors;
package SimpleDomains "Examples of connectors for simple domains"
  connector Electrical
    Modelica.SIunits.Voltage v;
    flow Modelica.SIunits.Current i;
  end Electrical;

  connector Thermal
    Modelica.SIunits.Temperature T;
    flow Modelica.SIunits.HeatFlowRate q;
  end Thermal;

  connector Translational
    Modelica.SIunits.Position x;
    flow Modelica.SIunits.Force f;
  end Translational;

  connector Rotational
    Modelica.SIunits.Angle phi;
```

```

flow Modelica.SIunits.Torque tau;
end Rotational;
end SimpleDomains;
```

## 第 6.2.2 节 流体连接器

其中一个广泛使用 Modelica 的领域是各种类型流体系统的建模。我们在上一节看到了如何为各种简单领域 (160) 创建连接器。但是, Modelica 在流体系统建模里如此引人注目是事出有因的。那就是其创建同时涉及多个守恒量的复杂连接器的能力。这样复杂连接器在建模流体系统时必不可少。在这样的一个连接器内可能涉及到质量、动量、能量和/或介质类型的流动。这样的情况下连接器定义需要支持很多的功能。

本节开始时, 我们会介绍一个非常类似此前讨论简单领域 (160) 时的基本连接器。但是, 我们在结束时得到的连接器会与前面的例子有着根本的不同。因为那个连接器会同时涉及质量守恒以及能量守恒。

### 不可压缩流体

不可压缩流体的建模在许多工程应用相当常用的。其中最值得注意的是液压致动系统。我们首先会介绍可用于模拟不可压缩系统的简单的连接器, 以及一些重要的注意事项。

请考虑以下连接器定义

```

connector Incompressible
  Modelica.SIunits.Pressure p;
  flow Modelica.SIunits.VolumeFlowRate q;
end Incompressible;
```

正如我们在讨论简单领域 (160) 时看到的一样, 这个连接器也符合由一个穿越变量和一个横跨变量组成这一定义。在这种情况下, 横空变量是  $p$  (压力), 而穿越变量则为  $q$  (体积流率)。但是, 该连接器和所有前述例子不同。flow 变量并不是一个保守量的时间导数, 因为体积不是一个保守量。

只要流体被建模是不可压缩, 该连接器就有效。要理解为什么, 请考虑以下的公式:

$$q_1 + q_2 + q_3 + q_4 = 0$$

其中  $q_1$ 、 $q_2$ 、 $q_3$ 、 $q_4$  代表体积流率量 (即其单位均为  $m^3/s$ )。一般而言, 这个方程没有资格作为一个守恒方程。因为体积 (再次强调, 是一般情况下) 是不保守的。但是, 如果我们知道这些流内是不可压缩的流体, 那么我们就可以在方程的左右两边乘以此不可压缩流体的密度, 即:

$$\rho q_1 + \rho q_2 + \rho q_3 + \rho q_4 = 0$$

现在这些量单位均为  $kg/s$ 。这是一个守恒方程, 因为质量是守恒量。不过, 如果将此连接器定义用在任何能显著压缩的流体上, 你就会得到错误的答案。

这样的连接器定义对比较简单的不可压缩流体网络颇为有用。因为此定义总可以在不必指定 (或知道) 工作流体密度的前提下描述系统的行为。不过, 这种方法本质上是有限制的。所以, 在其所解决问题比创建的更多这一前提下, 我们才应该使用此定义。

### 可压缩流体

前述接口定义仅可应用于不可压缩流体 (162), 而下面的连接器更为普遍:

```

connector GenericFluid
  Modelica.SIunits.Pressure p;
  flow Modelica.SIunits.MassFlowRate m_dot;
end GenericFluid;
```

此连接器同时可用于不可压缩流体及可压缩流体。这是因为其对流体的可压缩性没有任何固有的假设。请注意，横跨变量仍然是压力  $p$ ，但是穿越变量变为质量流率  $m_{dot}$ 。这样，该穿越变量便符合之前的惯例，即穿越变量应该是一个保守量（在这里是质量）的时间导数。因此，该连接器定义中没有隐含假设。这也就是为什么它可以同时用来模拟可压缩和不可压缩流体组成的流。

实际上，此连接器并非与简单领域（160）内的连接器有着根本上的不同。此连接器之所以出现在这一节，不过是因为它是下个例子的铺垫。

### 热流体建模

本节到目前为止，我们已经提出了一个用于不可压缩流体系统的连接器 `Incompressible`，以及一个更普遍的连接器 `GenericFluid`。但在这两种情况下，我们唯一考虑过的守恒量是质量。前述的连接器并没有在任何时候提到或支持对液体温度建模。

在许多应用里，工作流体的温度是非常重要的。某些情况下，温度变化会改变工作流体的密度。而在其他情况下，温度可以触发相变（例如从液体到气体）。温度也可以影响像流体粘性等其它关键性质，这对例如润滑系统等的性能有显著的影响。所以，要去建模任何对工作流体温度敏感的系统，前述的连接器定义将不足够。

为了预测工作流体的温度，有必要跟踪流体流过网络时的能量。要做到这一点，连接器定义除了质量外还要必须包括能量这个流经的守恒量。下面的连接器正定义了这一点：

```
connector ThermoFluid
  Modelica.SIunits.Pressure p;
  flow Modelica.SIunits.MassFlowRate m_dot;
  Modelica.SIunits.Temperature T;
  flow Modelica.SIunits.HeatFlowRate q;
end ThermoFluid;
```

注意，此连接器包括两个有 `flow` 限定词的变量  $m_{dot}$  和  $q$ 。这分别代表了质量流和能量流。每个变量都分别搭配一个横跨变量。正如我们在本节前面的连接器看到的一样，其中一个横跨变量是压力  $p$ 。另一个横跨变量  $T$  是工作流体的温度。

### 第 6.2.3 节 框图连接器

到目前为止，所有前述的连接器均为非因果性的。这意味着它们均是由穿越变量以及横跨变量组成的。这样的连接器是建模物理相互作用（组件之间进行守恒量交换）的基础。Modelica 也可以建模其他类型的相互作用，以及使用其他的建模方式。

框图连接器用于建模系统的信息流。在这里，我们不关心例如电流这种流向有时正向、有时反向的物理量。相反，我们只会考虑如何对信号进行建模。信号就是系统中的一些组件会产生的信息。然后，另外一些部件则会消费这些信息（并反过来产生其它信息）。在这种情况下，我们一般分别吧这样的信号称为“输入信号”以及“输出信号”。

为了模拟这种相互作用，我们可以使用如下的接口定义：

```
within ModelicaByExample.Connectors;
package BlockConnectors "Connectors for block diagrams"
  connector RealInput = input Real;
  connector RealOutput = output Real;
  connector IntegerInput = input Integer;
  connector IntegerOutput = output Integer;
  connector BooleanInput = input Boolean;
  connector BooleanOutput = output Boolean;
end BlockConnectors;
```

很显然，`BooleanInput` 连接器表明输入信号为 `Boolean`。`RealOutput` 则标示 `Real` 输出信号。

我们会在框图组件（223）的讨论里看到这些接口定义有何用处。

## 第 6.2.4 节 图形连接器

### 代码 vs. 图形

到目前为止，我们讨论过的 Modelica 是一个纯粹的文本语言。不过实际情况是，Modelica 建模一般是图形化的。从这个角度上，图形界面会在显示更复杂的 Modelica 模型时发挥更大的作用。

首先我们从连接器的可视化开始。Modelica 模型的文本组件将始终存在。变量和方程式将总会像之前一样，用代码形式来表示。不过，我们不断在后面看到 Modelica 的 annotation 特性是如何将图形化的外观不同的 Modelica 实体关联在一起。

第一种介绍的图形关联是和 connector 相关的图形。具体而言，我们会介绍如何将图形与 connector 的定义联系起来。这些图形将出现在连接器实例化了的每个图中（在讨论组件（173）时我们将进一步详细阐述）。

### 图标标注

要让一个标注和定义关联起来，我们要将其放置在该定义内。但标注不能和该定义内的任何声明或其他实体相关联。这是，这个标注是定义其中的一个元素。为了证明这一点，考虑下面的电气针脚接口的定义：

```
within ModelicaByExample.Connectors;
package Graphics
  connector PositivePin
    Modelica.SIunits.Voltage v;
    flow Modelica.SIunits.Current i;
    annotation (
      Icon(graphics={
        Ellipse(
          extent={{-100,100},{100,-100}},
          lineColor={0,0,255},
          fillColor={85,170,255},
          fillPattern=FillPattern.Solid),
        Rectangle(
          extent={{-10,58},{10,-62}},
          fillColor={0,128,255},
          fillPattern=FillPattern.Solid,
          pattern=LinePattern.None),
        Rectangle(
          extent={{-60,10},{60,-10}},
          fillColor={0,128,255},
          fillPattern=FillPattern.Solid,
          pattern=LinePattern.None,
          lineColor={0,0,0}),
        Text(
          extent={{-100,-100},{100,-140}},
          lineColor={0,0,255},
          fillColor={85,170,255},
          fillPattern=FillPattern.Solid,
          textString="%name")),
      Documentation(info=<html>
<p>This connector is used to represent the &quot;positive&quot; pins on electrical components. This does not imply that the voltage at this pin needs to be positive or even greater than voltages on <a href=\"modelica://ModelicaByExample.Connectors.Graphics.NegativePin\">&quot;negative&quot; pins</a>. It is simply a convention used to distinguish different connectors on components (particularly those with only two pins).</p>
</html>));
    end PositivePin;

  connector NegativePin
```

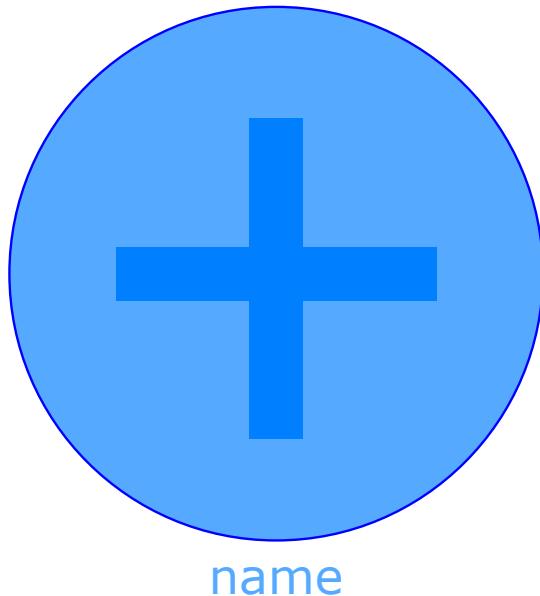
```

Modelica.SIunits.Voltage v;
flow Modelica.SIunits.Current i;
annotation (
  Icon(graphics={
    Ellipse(
      extent={{-100,100},{100,-100}},
      lineColor={0,0,255},
      fillColor={85,170,255},
      fillPattern=FillPattern.Solid),
    Rectangle(
      extent={{-60,10},{60,-10}},
      fillColor={0,128,255},
      fillPattern=FillPattern.Solid,
      pattern=LinePattern.None,
      lineColor={0,0,0}),
    Text(
      extent={{-100,-100},{100,-140}},
      lineColor={0,0,255},
      fillColor={85,170,255},
      fillPattern=FillPattern.Solid,
      textString="%name")),
  Documentation(info=<html>
<p>This pin and
<a href=\"modelica://ModelicaByExample.Connectors.Graphics.PositivePin\">
its counterpart</a> are documented in
<a href=\"modelica://ModelicaByExample.Connectors.Graphics.PositivePin\">
PositivePin</a>.</p>
</html>));
  end NegativePin;
end Graphics;

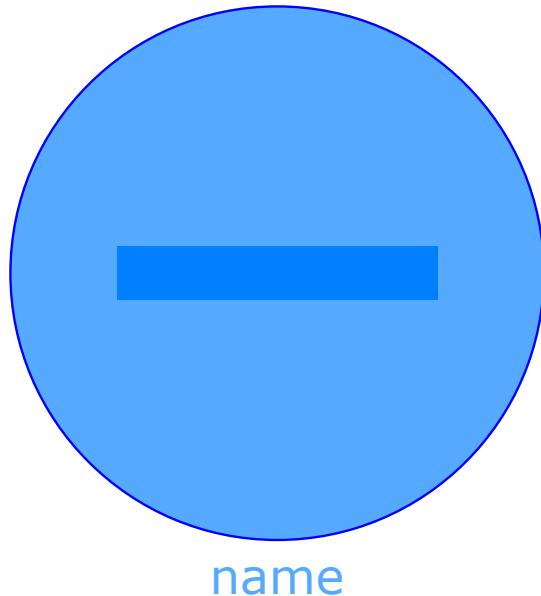
```

请注意这些定义各自的长度。这些定义里的标注几乎占了代码的全部。除去标注外，PositivePin 和 NegativePin 定义与简单领域 (160) 里讨论的 Electrical 接口定义相同。

我们之所以定义两个电插针连接器的原因是，这样可以其制成不同的图形。PositivePin 连接器的实例如下：



而 NegativePin 的实例如下：



让我们更详细地看看在 PositivePin 定义的 Icon 标注:

```
Icon(graphics={  
    Ellipse(  
        extent={{-100,100},{100,-100}},  
        lineColor={0,0,255},  
        fillColor={85,170,255},  
        fillPattern=FillPattern.Solid),  
    Rectangle(  
        extent={{-10,58},{10,-62}},  
        fillColor={0,128,255},  
        fillPattern=FillPattern.Solid,  
        pattern=LinePattern.None),  
    Rectangle(  
        extent={{-60,10},{60,-10}},  
        fillColor={0,128,255},  
        fillPattern=FillPattern.Solid,  
        pattern=LinePattern.None,  
        lineColor={0,0,0}),  
    Text(  
        extent={{-100,-100},{100,-140}},  
        lineColor={0,0,255},  
        fillColor={85,170,255},  
        fillPattern=FillPattern.Solid,  
        textString="%name"))},
```

我们很快将要讨论 [图形标注](#) (167)。但是, 让我们先快速浏览下这些定义有什么用。我们可以看到, Icon 标注包含另一个变量 graphics。graphics 变量的赋值是一系列图形元素组成的向量。我们看到, 这些矢量图形元素包括 Ellipse (用于呈现图标内圆圈)、两个 Rectangle 元素 (用于渲染“+”号) 和 Text 元素。注意, Text 内的元素 textString 包含了 "%name" 代码。图形标注中能填入不同的替换字符串。上述的替换字符串则指代声明为 PositivePin 类型的变量实例名称。因此, 例如在下面的声明有:

```
PositivePin p;
```

在图形界面里 %name 会表示为 p。这样, 在图表层中连接器显示的文本名称总是匹配于模型里相应连接器声明的名称。

本章后面我们会回顾关于 [图形标注](#) (167) 的内容。而我们也会更多地看到使用图形标注的模型。因为我们会从纯代码模型过渡为包含图形渲染的实现。

## 第 6.3 节 回顾

### 第 6.3.1 节 连接器定义

#### 语法

正如我们已经多次看到的一样，Modelica 定义之间有着相当的句法相似性。这也对 connector 定义成立。连接器定义的一般语法是：

```
connector ConnectorName "Description of the connector"
  // Declarations for connector variables
end ConnectorName;
```

不同于 model 或 function，connector 不得包含任何行为。所以在 connector 内永远不会出现 equation 或 algorithm 区域。

#### 变量

##### 因果性变量

在之前对框图连接器（163）的讨论里，我们介绍了 Modelica 的 connector 定义内的变量可以和因果关系联系起来。如果信号应在部件外计算出来，则变量就应该加上 input 限定词。另一方面，若信号应在部件内计算出来（然后再传输到其他部件），则变量就应该加上 output 限定词。

##### 非因果变量

在我们对简单领域（160）以及流体连接器（162）的讨论里，我们看到包括许多穿越变量和横跨变量的 connector 定义。这些变量总是成对出现的。穿越变量带有 flow 限定词。横跨变量则没有限定词。

我们将在接下来的章节中看到，这样的连接器在定义物理系统建模时非常方便。因为这些连接器帮助 Modelica 语言编译器自动生成组件网络的守恒方程。此外，连接器允许数量等、质量、动量、能量、电荷、介质种类等双向地流过网络。

#### 参数

connector 定义的变量也可以加入 parameter 限定词。这个限定词和我们第一次讨论参数（26）时意思一样。也就是说，该变量的值在模拟过程中不能改变。parameter 变量在连接器定义里经常用于指示连接器内数组的大小。

#### 结语

应该指出的是，单个 connector 定义可以都在同一个连接器内混合因果、非因果和参数变量。事实上，连接器的变量本身可以也是一个连接器。Modelica 的丰富表现力允许用户模拟一系列不同类型的相互作用，以及为每个变量选择对潜在相互作用最有意义的语义。

### 第 6.3.2 节 图形标注

虽然本节出现在连接器（159）章内，本话题适用于与一般的模型定义相关的图形标注。因此，这里提供的信息可以对 Modelica 的许多方面提供有益的参考。

## 图形层

有两种不同的表现形式可以去描述 Modelica 的实体的外观。一个是所谓“图标”表示。另一个则是所谓“简图”表示。在 Modelica 里，图标表示展现的是从“外面”观察模型时看到的内容。一般来说，图标包括一些独特的视觉表现形式。此外，通过替换 (171)（稍后我们会进行介绍），它也包含了关于该实体的进一步信息。

在另一方面，“简图”表示是用来展现组件的“内部”视图。简图表示通常用于 Modelica 组件图形化的详细文档。特别是当这些内容对于“图标”视图来说过于仔细的时候。

一个定义在“图标”层所展现的外观是由 Icon 标注指定的（在之前对图形连接器 (164) 讨论里有简要介绍）。毫无疑问，定义在“简图”层所展现的外观则是由 Diagram 标注指定的。两者均是直接出现在定义里，不和声明或者 extends 条款等现有元素相关联。

一般来说，大部分的定义都包括一个“图标”的表示，但只有少数有空去包括“简图”表示。不过，事实上尽管是在不同的情况下进行的渲染，图形表示的规范却是相同的。

### 使用 Icon 的例子

在这本书的其余部分，我们将利用 Icon 标注举例介绍图形标注。这些例子对 Diagram 标注同样有效。但由于 Icon 标注相对常见，关于图形标注的进一步例子仅会出现 Icon 标注里。

## 通用图形定义

下面的定义将在本节中引用：

```
type DrawingUnit = Real(final unit="mm");
type Point = DrawingUnit[2] "{x, y}";
type Extent = Point[2]
  "Defines a rectangular area {{x1, y1}, {x2, y2}}";
type Color = Integer[3](min=0, max=255) "RGB representation";
constant Color Black = zeros(3);
type LinePattern = enumeration(None, Solid, Dash, Dot, DashDot, DashDotDot);
type FillPattern = enumeration(None, Solid, Horizontal, Vertical,
  Cross, Forward, Backward,
  CrossDiag, HorizontalCylinder,
  VerticalCylinder, Sphere);
type BorderPattern = enumeration(None, Raised, Sunken, Engraved);
type Smooth = enumeration(None, Bezier);
type Arrow = enumeration(None, Open, Filled, Half);
type TextStyle = enumeration(Bold, Italic, UnderLine);
type TextAlignment = enumeration(Left, Center, Right);

record FilledShape "Style attributes for filled shapes"
  Color lineColor = Black "Color of border line";
  Color fillColor = Black "Interior fill color";
  LinePattern pattern = LinePattern.Solid "Border line pattern";
  FillPattern fillPattern = FillPattern.None "Interior fill pattern";
  DrawingUnit lineThickness = 0.25 "Line thickness";
end FilledShape;
```

另外，很多我们马上会讨论到的标注包括一组在以下 record 定义里的共同元素：

```
partial record GraphicItem
  Boolean visible = true;
  Point origin = {0, 0};
  Real rotation(quantity="angle", unit="deg")=0;
end GraphicItem;
```

为了明晰地介绍表示图形元素的常用标注，我们会以 GraphicItem 为基础扩展出其他模型。

## Icon 和 Diagram 标注

以下数据描述了应该出现在模型图标层的元素：

```
record Icon "Representation of the icon layer"
  CoordinateSystem coordinateSystem(extent = {{-100, -100}, {100, 100}});
  GraphicItem[] graphics;
end Icon;
```

其中坐标系统中的数据被定义为：

```
record CoordinateSystem
  Extent extent;
  Boolean preserveAspectRatio=true;
  Real initialScale = 0.1;
  DrawingUnit grid[2];
end CoordinateSystem;
```

换言之，该 Icon 标注包括了 coordinateSystem 定义内的坐标系信息。而且，标注还包括了 graphics 内的图形物件数组。Diagram 标注的定义也是一样：

```
record Diagram "Representation of the diagram layer"
  CoordinateSystem coordinateSystem(extent = {{-100, -100}, {100, 100}});
  GraphicItem[] graphics;
end Diagram;
```

## 图形物件

Modelica 标准里有许多不同的图形物件可以放入 Icon 或 Diagram 标注内的 graphics 矢量。其定义在这里列出以供参考。

### Line

```
record Line
  extends GraphicItem;
  Point points[:];
  Color color = Black;
  LinePattern pattern = LinePattern.Solid;
  DrawingUnit thickness = 0.25;
  Arrow arrow[2] = {Arrow.None, Arrow.None} "{start arrow, end arrow}";
  DrawingUnit arrowSize=3;
  Smooth smooth = Smooth.None "Spline";
end Line;
```

### Polygon

```
record Polygon
  extends GraphicItem;
  extends FilledShape;
  Point points[:];
  Smooth smooth = Smooth.None "Spline outline";
end Polygon;
```

### Rectangle

```
record Rectangle
  extends GraphicItem;
  extends FilledShape;
  BorderPattern borderPattern = BorderPattern.None;
  Extent extent;
  DrawingUnit radius = 0 "Corner radius";
end Rectangle;
```

Ellipse

```
record Ellipse
  extends GraphicItem;
  extends FilledShape;
  Extent extent;
  Real startAngle(quantity="angle", unit="deg")=0;
  Real endAngle(quantity="angle", unit="deg")=360;
end Ellipse;
```

Text

```
record Text
  extends GraphicItem;
  extends FilledShape;
  Extent extent;
  String textString;
  Real fontSize = 0 "unit pt";
  String fontName;
  TextStyle textStyle[:];
  Color textColor=lineColor;
  TextAlignment horizontalAlignment = TextAlignment.Center;
end Text;
```

Bitmap

```
record Bitmap
  extends GraphicItem;
  Extent extent;
  String fileName "Name of bitmap file";
  String imageSource "Base64 representation of bitmap";
end Bitmap;
```

## 继承图形标注

当一个模型定义从另一个定义继承时，图形标注在默认情况下也会继承。不过，这种行为可以通过对 `extends` 子句添加以下标注来控制（分别对应图标层和简图层）：

```
record IconMap
  Extent extent = {{0, 0}, {0, 0}};
  Boolean primitivesVisible = true;
end IconMap;

record DiagramMap
  Extent extent = {{0, 0}, {0, 0}};
  Boolean primitivesVisible = true;
end DiagramMap;
```

在这两种情况下，extent 数据允许继承图形元素的位置进行调整。将 primitivesVisible 设置为 false 则会隐藏所继承的图形元素。

## 替换

[Text](#) ( 170) 标注内的 textString 字段可以包含替换模式。标注支持以下的替换模式：

- %name – 这个字符串模式会被给定定义的实例名称所取代。
- %class – 这个字符串模式会被这个定义的名称所取代。
- %<name>, 其中 <name> 为参数名称 - 这个模式会被替换为相应参数的值。
- %% – 这个模式会被替换为%。

## 组合以上内容

在讨论了图形标注的这些方面后，让我们回顾在对图形连接器 ( 164) 的讨论中提出的图标定义。

```
Icon(graphics={
    Ellipse(
        extent={{-100,100},{100,-100}},
        lineColor={0,0,255},
        fillColor={85,170,255},
        fillPattern=FillPattern.Solid),
    Rectangle(
        extent={{-10,58},{10,-62}},
        fillColor={0,128,255},
        fillPattern=FillPattern.Solid,
        pattern=LinePattern.None),
    Rectangle(
        extent={{-60,10},{60,-10}},
        fillColor={0,128,255},
        fillPattern=FillPattern.Solid,
        pattern=LinePattern.None,
        lineColor={0,0,0}),
    Text(
        extent={{-100,-100},{100,-140}},
        lineColor={0,0,255},
        fillColor={85,170,255},
        fillPattern=FillPattern.Solid,
        textString="%name"))},
```

这里我们看到了与 PositivePin 定义相关联的 annotation 是个模型标注。此外，我们可以看到这个标注内 Icon 元素包含一系列的图形物体。第一个图形物体是 [Ellipse](#) ( 170) 标注。紧接着的是两个 [Rectangle](#) ( 169) 标注。最后是 [Text](#) ( 170) (这里使用了前面所讨论的 [替换](#) ( 171))。

注意数据在 annotation 里的呈现方式与我们在前面讨论过的记录 (record) 定义中描述的数据一致。



## 组件

大多数人眼里的 Modelica 是一种可以使用面向组件方法的语言。因此，如何去建立组件模型就大概是本书所涵盖的最重要课题了。

到现在为止，我们主要专注于如何描述（连续和离散的）数学行为。现在是时候了解这些行为怎样可以包装成可重用的组件模型。事实上，这些组件模型可重用意味着，同样的代码在编写和测试后就可以反复使用。这种重用可节省开发时间，避免编程错误，同时也简化了维护。

### 第 7.1 节 示例

#### 第 7.1.1 节 传热组件

我们会以建立传热领域组件模型来开始对组件模型的讨论。这些模型能让我们重构之前（7）看到的模型。这次可以使用组件模型去表示各种不同的效应。投入时间建立组件模型将允许我们轻易地结合底层物理行为，去为各种热系统创建模型。

##### 热连接器

在此前简单领域（160）的讨论里，我们描述了如何编写热连接器。对于本节中的组件模型，我们将使用 Modelica 标准库内的热连接器模型。这些连接器定义如下：

```
within Modelica.Thermal.HeatTransfer;
package Interfaces "Connectors and partial models"
    partial connector HeatPort "Thermal port for 1-dim. heat transfer"
        Modelica.SIunits.Temperature T "Port temperature";
        flow Modelica.SIunits.HeatFlowRate Q_flow
            "Heat flow rate (positive if flowing from outside into the component)";
    end HeatPort;

    connector HeatPort_a "Thermal port for 1-dim. heat transfer (filled rectangular icon)"
        extends HeatPort;
        annotation(...,
            Icon(coordinateSystem(preserveAspectRatio=true,
                extent={{-100,-100},{100,100}}),
            graphics={Rectangle(
                extent={{-100,100},{100,-100}},
                lineColor={191,0,0},
                fillColor={191,0,0},
                fillPattern=FillPattern.Solid)}));
    end HeatPort_a;

    connector HeatPort_b "Thermal port for 1-dim. heat transfer (unfilled rectangular icon)"
        extends HeatPort;
```

```

annotation(...,
Icon(coordinateSystem(preserveAspectRatio=true,
extent={{-100,-100},{100,100}}),
graphics={Rectangle(
extent={{-100,100},{100,-100}},
lineColor={191,0,0},
fillColor={255,255,255},
fillPattern=FillPattern.Solid)}));
end HeatPort_b;
end Interfaces;

```

仔细检查这些接口的定义表明，HeatPort\_a 和 HeatPort\_b 在其内容上与 HeatPort 模型是相同的。唯一的区别在于，HeatPort\_a 和 HeatPort\_b 的图标有所不同。

本节其余部分的组件模型都将会利用这些接口定义。

## 组件模型

在构建组件模型时，创建组件模型的目标是（仅）实现一个物理效应（例如热容、对流）。通过以此方式中建立组件模型，我们将看到，模型就可以在无数不同的配置中任意组合，而不需要添加任何更多的方程。这种方程的重用使得模型的开发更有效率，并避免引入错误的可能。

### 热容

我们的首个组件模型是温度分布均匀的集总热容模型。我们希望与此组件模型相关联的公式是：

$$C\dot{T} = Q_{flow}$$

表示这个公式 Modelica 模型（去掉了 Icon 标注）很简单：

```

within ModelicaByExample.Components.HeatTransfer;
model ThermalCapacitance "A model of thermal capacitance"
parameter Modelica.SIunits.HeatCapacity C "Thermal capacitance";
parameter Modelica.SIunits.Temperature T0 "Initial temperature";
Modelica.Thermal.HeatTransfer.Interfaces.HeatPort_a node
annotation (Placement(transformation(extent={{-10,-10},{10,10}})));
initial equation
node.T = T0;
equation
C*der(node.T) = node.Q_flow;
end ThermalCapacitance;

```

其中 C 为热容，T0 是初始温度。

请注意在这个模型中带有 node 连接器。这是 ThermalCapacitance 组件模型与“外面的世界”进行交互的地方。我们将使用 node 的温度 node.T 去代表热容的温度。flow 变量 node.Q\_flow 表示热在流入热容。我们可以在热容的公式看到这点：

$$C\dot{der}(node.T) = node.Q\_flow;$$

注意，当 node.Q\_flow 为正时，热容的温度 node.T 会增加。这证实了我们遵循了 Modelica 的约定。连接器 flow 变量代表到组件的守恒量的流（在稍候我们会对核算（238）进行更详细的讨论）。这里的守恒量为热。

仅仅使用这个模型，我们就已经可以构建一个如下的简单“系统”模型：

```

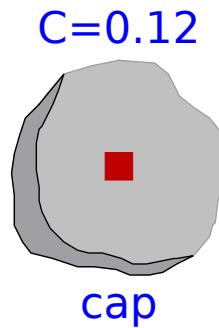
within ModelicaByExample.Components.HeatTransfer.Examples;
model Adiabatic "A model without any heat transfer"
ThermalCapacitance cap(C=0.12, T0(displayUnit="K") = 363.15)
"Thermal capacitance component"

```

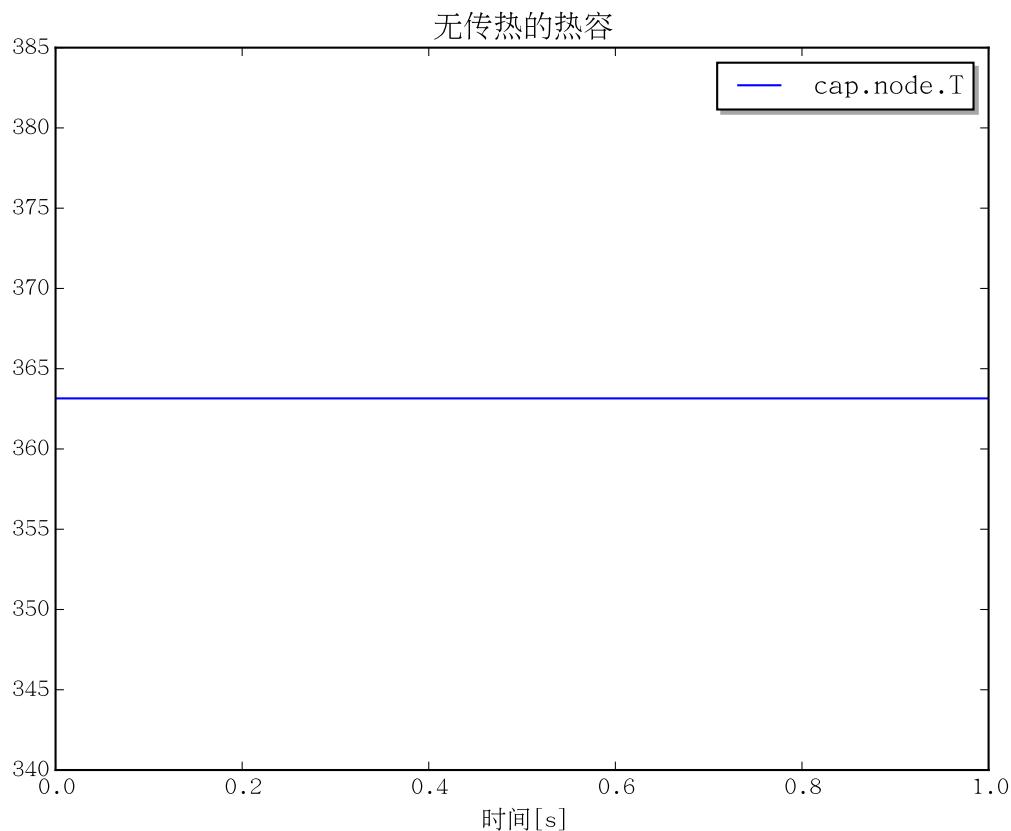
```
annotation (Placement(transformation(extent={{-30,-10},{-10,10}})));
end Adiabatic;
```

这种模式只包含热容元件（正如 ThermalCapacitance 类型的变量 cap 其声明所示），而没有其它传热元件（例如传导、对流、辐射）。请暂时忽略 Placement 标注，我们会在后面关于组件模型标注（245）的章节中对其提供完整的解释。

模型中使用的图形标注（其中一部分没有在上面列举出来）可以显示为：



由于没有热量进入或离开热容元件 cap。热容的温度保持恒定，如下图所示：



#### ConvectionToAmbient

要快速添加热传递，我们可以定义另一个组件模型，去表示到与环境温度的传热。这样的模型可以在 Modelica 表示（再次，去掉了 Icon 标注）如下：

```

within ModelicaByExample.Components.HeatTransfer;
model ConvectionToAmbient "An overly specialized model of convection"
  parameter Modelica.SIunits.CoefficientOfHeatTransfer h;
  parameter Modelica.SIunits.Area A;
  parameter Modelica.SIunits.Temperature T_amb "Ambient temperature";
  Modelica.Thermal.HeatTransfer.Interfaces.HeatPort_a port_a
    annotation (Placement(transformation(extent={{-110,-10},{-90,10}})));
equation
  port_a.Q_flow = h*A*(port_a.T-T_amb) "Heat transfer equation";
end ConvectionToAmbient;

```

该模型包括了传热系数  $h$  作为参数, 表示表面积的参数  $A$  以及环境温度  $T_{amb}$ 。模式通过连接器  $port\_a$  连接到其它传热元件。

同样, 我们必须密切注意的正负号约定。请回想我们此前对热容 (174) 的讨论。Modelica 遵从如下的正负号约定:  $flow$  变量的正值表示正在流入部件。特别是, 让我们来仔细看一下 `ConvectionToAmbient` 模型内的公式:

```
port_a.Q_flow = h*A*(port_a.T-T_amb) "Heat transfer equation";
```

注意, 当  $port\_a.T$  大于  $T_{amb}$  时,  $port\_a.Q\_flow$  的符号为正。这意味着热量正流入这个组件。换句话说, 当  $port\_a.T$  大于  $T_{amb}$  时, 这个组件将从  $port\_a$  抽走热量 (相反, 当  $T_{amb}$  大于  $port\_a.T$  时, 此组件会向  $port\_a$  注入热能)。

有了这样的组件模型, 可让我们将其与 ThermalCapacitance 模型相结合, 并像我们的一些早期的热传递的例子 (7) 一样, 用以下 Modelica 代码进行系统模拟:

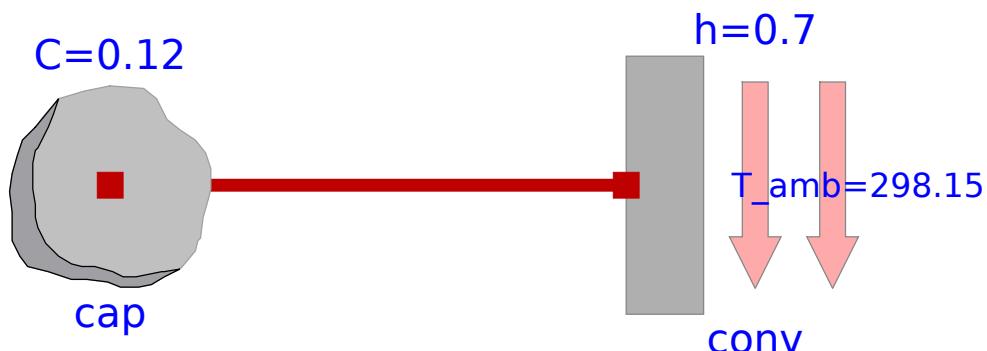
```

within ModelicaByExample.Components.HeatTransfer.Examples;
model CoolingToAmbient "A model using convection to an ambient condition"

ThermalCapacitance cap(C=0.12, T0(displayUnit="K") = 363.15)
  "Thermal capacitance component"
  annotation (Placement(transformation(extent={{-30,-10},{-10,10}})));
ConvectionToAmbient conv(h=0.7, A=1.0, T_amb=298.15)
  "Convection to an ambient temprature"
  annotation (Placement(transformation(extent={{20,-10},{40,10}})));
equation
  connect(cap.node, conv.port_a) annotation (Line(
    points={{-20,0},{20,0}},
    color={191,0,0},
    smooth=Smooth.None));
end CoolingToAmbient;

```

在模型中, 我们看到了两个组件的声明: `cap` 和 `conv`。每个组件的参数也在声明时指定了。下面为 `CoolingToAmbient` 模型的示意图:



但是, 本模型特别的地方在于其方程区域:

```
equation
  connect(cap.node, conv.port_a) annotation (Line(
    points={{-20,0},{20,0}},
    color={191,0,0},
    smooth=Smooth.None));
```

此语句引入 Modelica 语言最重要的其中一个特性。请注意语句出现在 equation 区域。虽然 connect 操作符看起来像一个函数，其实其功能不止如此。操作符代表了表述两个指定连接器 cap.node 和 conv.port\_a 间相互作用的公式。

在这种情况下，连接完成了两件重要的事情。第一件事就是产生让两连接器上“横跨”变量相等的等式。在这种情况下，这意味着下面的等式：

```
cap.node.T = conv.port_a.T "Equating across variables";
```

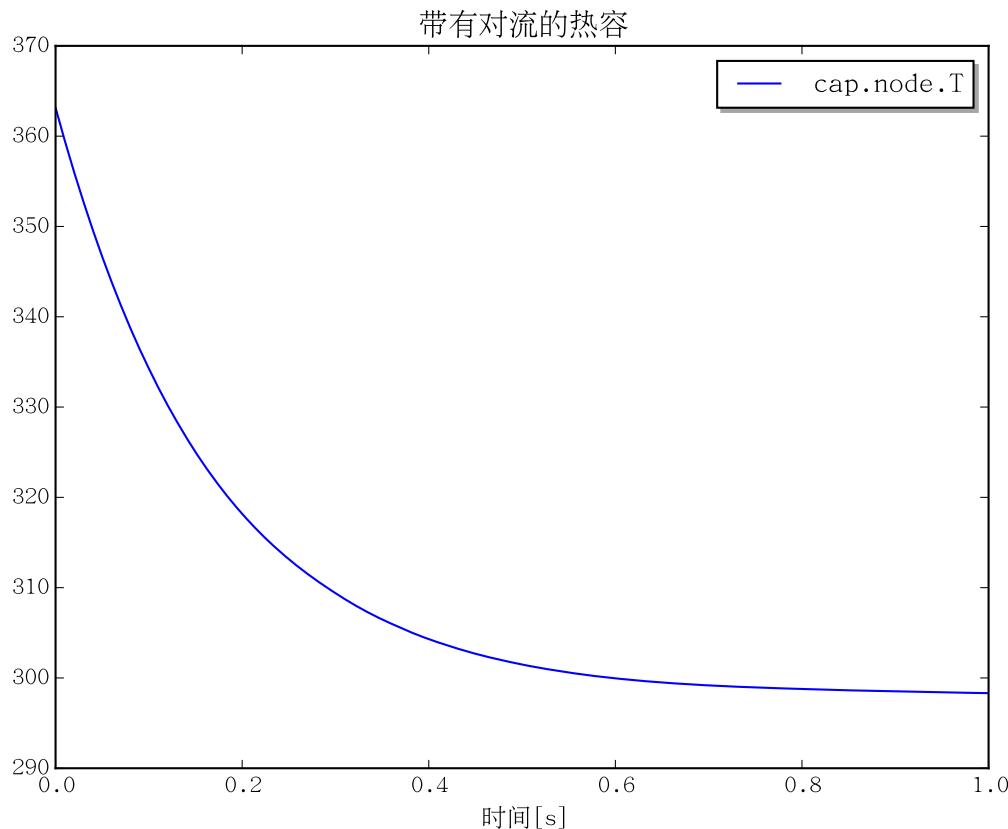
此外，连接也会为所有的贯通变量生成一个等式。生成的方程为守恒方程。你可以认为此守恒方程是基尔霍夫电流定律对所有守恒量的推广。基本上，它代表该连接本身事实上没有“存储”能力。无论有多少守恒量（这里是热量）从某部件里流出，那么这些量必然会进入另一个（些）部件。因此在这种情况下，连接语句会为 flow 变量生成以下方程：

```
cap.node.Q_flow + conv.port_a.Q_flow = 0 "Sum of heat flows must be zero";
```

注意这里的正负号约定。所有的 flow 变量相加在一起。我们将在稍候研究有多个组件进行交互的更复杂的情况。但因为这个简单的例子只带有两个部件。所以，我们可以清楚看到，若  $Q_{flow}$  有一个值为正，则另一个必须为负。换句话说，如果热从一个部件里流出，就必定会流入另一个不见。这些守恒方程确保我们对网络内守恒量进行恰当的追踪，避免任何守恒量的“丢失”。

在热学问题背景下，有一个很简单的方法可以概括连接行为。我们可以认为连接是一个没有热容完美的导热元件。

如果我们对上述 CoolingToAmbient 模型进行仿真，我们会得到以下的温度轨迹：



## 进一步研究

CoolingToAmbient 模型中有个小问题。我们前面提到了，在建立组件模型时，最好是每个物理效应单独地隔离到一个组件中。但我们这里实际上在一个组件里集成了两种不同的效应。我们会马上看到，这限制了组件模型的可重用性。但首先，让我们通过重构代码将两个效应分开。然后，我们会使用这些新组件再次建立系统级模型。

### 对流

第一个新组件是 Convection 模型。在这种情况下，我们将不对左右两端的温度作任何假设。相反，我们只假设连接到每个模型都拥有适当的热连接器。其结果就是如下的一个模型：

```
within ModelicaByExample.Components.HeatTransfer;
model Convection "Modeling convection between port_a and port_b"
  parameter Modelica.SIunits.CoefficientOfHeatTransfer h;
  parameter Modelica.SIunits.Area A;
  Modelica.Thermal.HeatTransfer.Interfaces.HeatPort_a port_a
    annotation (Placement(transformation(extent={{-110,-10},{-90,10}})));
  Modelica.Thermal.HeatTransfer.Interfaces.HeatPort_b port_b
    annotation (Placement(transformation(extent={{90,-10},{110,10}})));
equation
  port_a.Q_flow + port_b.Q_flow = 0 "Conservation of energy";
  port_a.Q_flow = h*A*(port_a.T-port_b.T) "Heat transfer equation";
end Convection;
```

这个模型包含两个等式。第一个等式：

```
port_a.Q_flow + port_b.Q_flow = 0 "Conservation of energy";
```

表明该组件并不存储热量。方程实施了所有从某个连接器流入热量必须从其它连接器流出这一约束（这也正是我们在本节前面的 connect 语句所看到的相同行为）。下一个公式：

```
port_a.Q_flow = h*A*(port_a.T-port_b.T) "Heat transfer equation";
```

通过表达穿过该组件的热流和两端的温度之间的关系，描述了对流的热传递关系。

### 等式数目

所有之前的模型都带有一个连接器和一个方程式。Convection 模型则有两个连接器。其结果是，模型有两个方程。一个简单的经验规则是，有多少个连接器就需要多少个方程。但是，请记住这个经验法则假设你正在使用的连接器只有一个穿越变量。而且模型内没有“内部变量”（如 protected 变量）。下一节组件模型（237）会对组件需要的方程数提供一个更全面的讨论。具体来说，该节将对如何建立所谓的平衡的组件（241）提供指导。

### 环境条件

现在，我们有对流模型，我们需要另一个模型来代表环境条件。我们需要一个类似热容的模型。但这个模型需要保持恒定的温度。试想一下，如果我们使用 ThermalCapacitance 模型，并为热容 C 赋一个非常大的值。然后我们就会有一个温度变化非常缓慢的模型。但我们需要的模型完全不改变温度。模型如同有一个无限大的 C 值。

这种模型的出现得相当频繁。它通常被称为“无限大容器”模型。一般而言，这种模型的特征在于：无论多少的守恒量（这里为热）流入或流出组件，横跨变量都会保持恒定。在电气领域，这样的模型代表电气接地。在机械方面，这种模型代表一个机械地面（不管施加多大的力都不改变位置的物体）。

我们将使用 AmbientConditions 模型去描述环境条件：

```

within ModelicaByExample.Components.HeatTransfer;
model AmbientCondition
  "Model of an \\"infinite reservoir\\" at some ambient temperature"
  parameter Modelica.SIunits.Temperature T_amb "Ambient temperature";
  Modelica.Thermal.HeatTransfer.Interfaces.HeatPort_a node annotation (
    Placement(transformation(extent={{-10,-10},{10,10}}), iconTransformation(
      extent={{-10,-10},{10,10}})));
equation
  node.T = T_amb;
end AmbientCondition;

```

由于我们正在谈论传热领域，这个模型是一个无限储热容器。不管有多少热量流入或流出该组件，其温度均保持不变。

### 灵活性

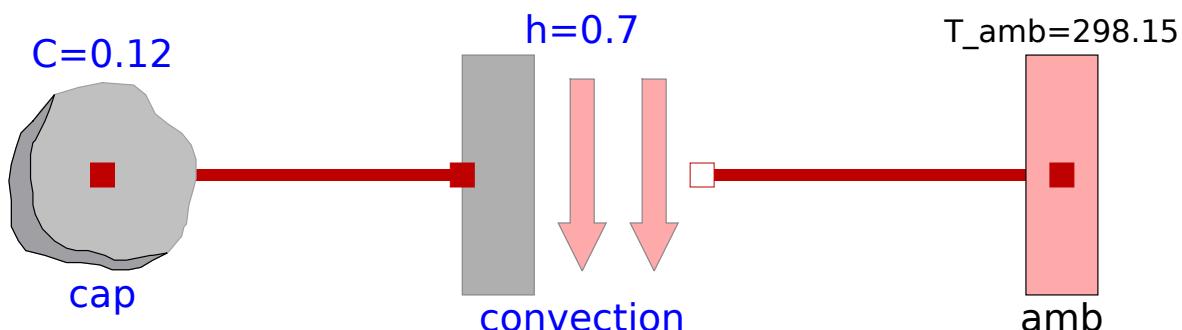
使用这些新的 Convection 和 AmbientCondition 模型，我们可以使用重建此前的简单传热系统模型，如下：

```

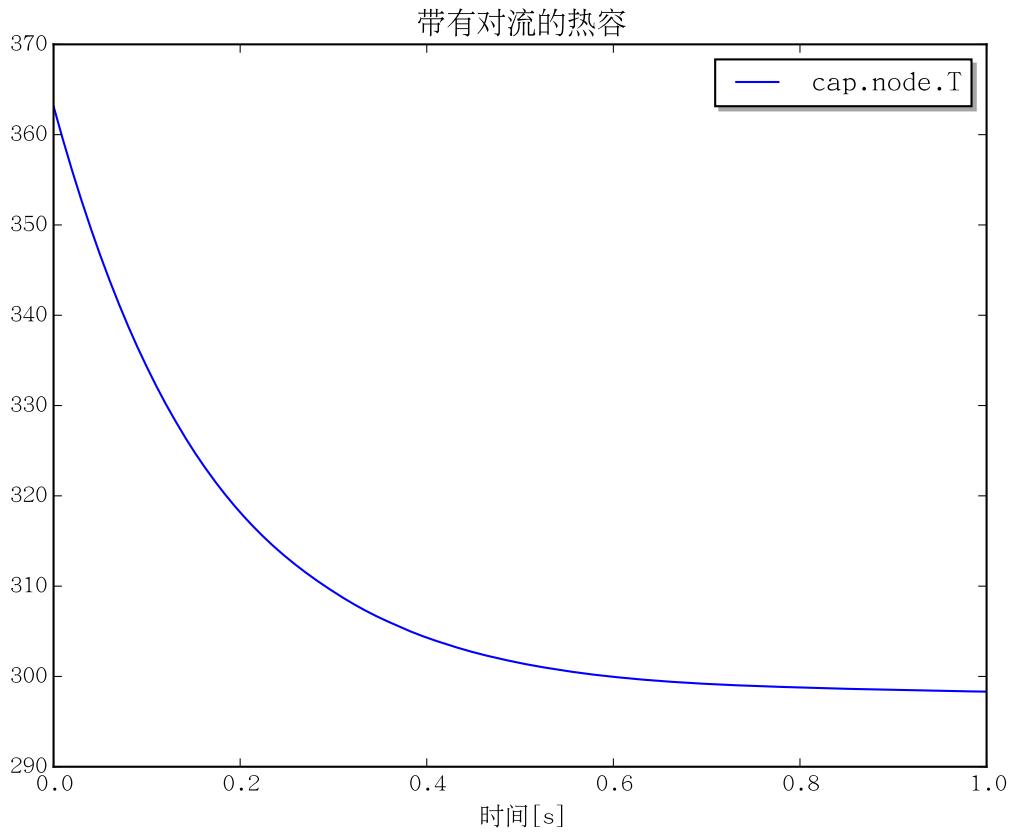
within ModelicaByExample.Components.HeatTransfer.Examples;
model Cooling "A model using generic convection to ambient conditions"
  ThermalCapacitance cap(C=0.12, T0(displayUnit="K") = 363.15)
    "Thermal capacitance component"
    annotation (Placement(transformation(extent={{-30,-10},{-10,10}})));
  Convection convection(h=0.7, A=1.0)
    annotation (Placement(transformation(extent={{10,-10},{30,10}}));
  AmbientCondition amb(T_amb(displayUnit="K") = 298.15)
    annotation (Placement(transformation(extent={{50,-10},{70,10}})));
equation
  connect(convection.port_a, cap.node) annotation (Line(
    points={{10,0},{-20,0}},
    color={191,0,0},
    smooth=Smooth.None));
  connect(amb.node, convection.port_b) annotation (Line(
    points={{60,0},{30,0}},
    color={191,0,0},
    smooth=Smooth.None));
end Cooling;

```

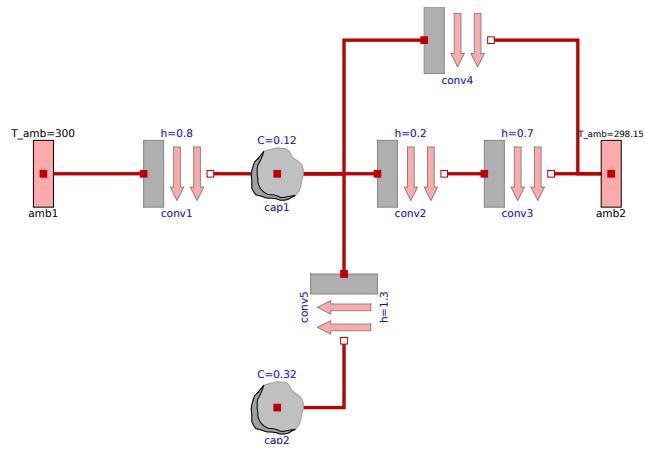
该模型显示如下：

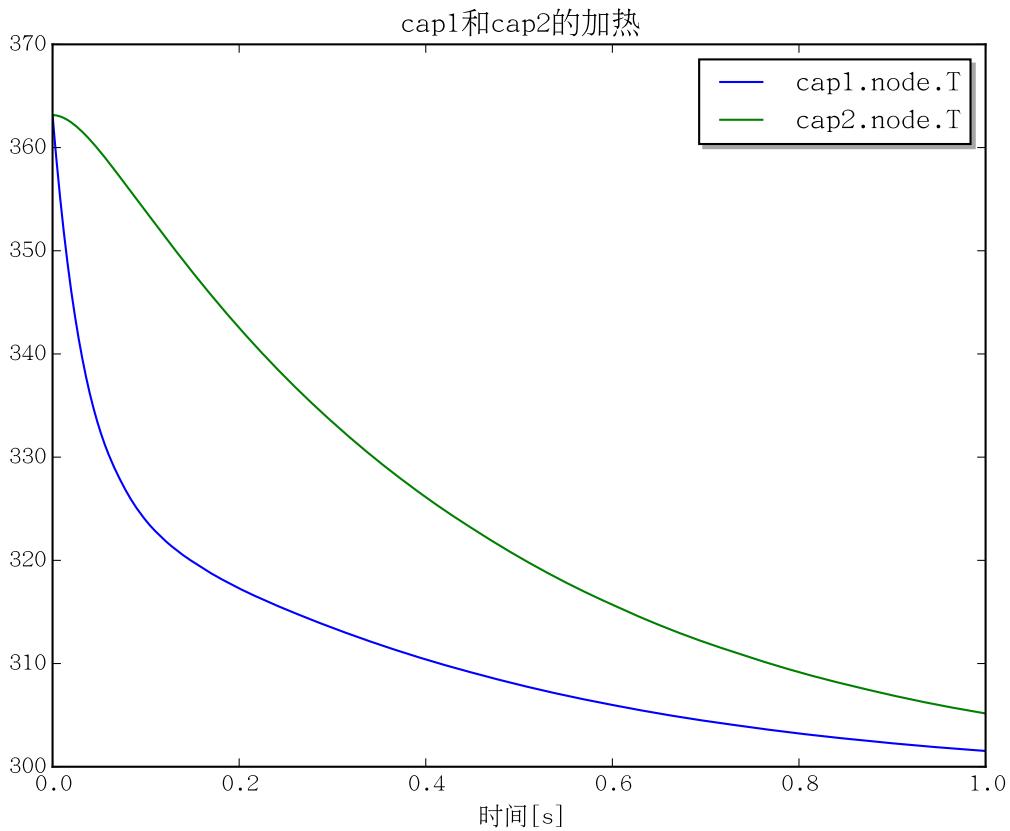


这看起来可能没有什么大不了的改善。虽然我们费心去将 ConvectionToAmbient 分拆成单独的 Convection 和 AmbientTemperature 模型，我们最终还是得到了相同的基本行为，即：



将 ConvectionToAmbient 分拆成 Convection 和 AmbientTemperature 模型的一大好处在于，现在我们可以将它们以不同的方式重新组合。下面的示意图展示了一个例子，说明我们使用到目前为止建立的数个基本组件，就可以重新组合成一个全新（和更复杂）的模型：





其实，我们现在可以使用这些组件去建立任意复杂的组件网，而不需要费心写出相关联的动力学方程。所需要做的一切已包括在组件模型里。这让我们能够专注于创建和设计系统的过程，而忽略背后繁琐、耗时且容易出错的方程变换工作。

### 第 7.1.2 节 电气部件

前一节中讨论了如何在传热领域中创建组件模型。现在让我们将注意力转移到如何构建一些基本的电气元件。然后，我们会用这些模型来建立所看到的电气系统 (11)。

在这一节中，我们将两次实现基本电气组件的模型。第一次时，我们将独立地实现每个组件，不考虑组件间的关系。但在第二次过程中，我们将看到怎样通过使用 Modelica 的继承机制去减轻我们的工作量。

但在两种情况下，我们都将使用相同的接口定义。在对简单领域 (160) 的讨论里，我们看到了如何构建一个电连接器。与上一节传热一样，本节中的例子将依赖于 Modelica 标准库的连接器定义。这些接口的定义如下：

```

connector PositivePin "Positive pin of an electric component"
  Modelica.SIunits.Voltage v "Potential at the pin";
  flow Modelica.SIunits.Current i "Current flowing into the pin";
end PositivePin;

connector NegativePin "Negative pin of an electric component"
  Modelica.SIunits.Voltage v "Potential at the pin";
  flow Modelica.SIunits.Current i "Current flowing into the pin";
end NegativePin;

```

## 基本组件模型

有了这些 connector 定义，构建电阻模型就相对简单了。电阻模型的目标是使用欧姆定律去封装电阻两端电压与电阻器内电流之间的关系。下面的模型表示上述电阻模型的一种可能形式：

```
within ModelicaByExample.Components.Electrical.VerboseApproach;
model Resistor "A resistor model"
  parameter Modelica.SIunits.Resistance R;
  Modelica.Electrical.Analog.Interfaces.PositivePin p
    annotation (Placement(transformation(extent={{-110,-10},{-90,10}})));
  Modelica.Electrical.Analog.Interfaces.NegativePin n
    annotation (Placement(transformation(extent={{90,-10},{110,10}})));
protected
  Modelica.SIunits.Voltage v = p.v-n.v;
equation
  p.i + n.i = 0 "Conservation of charge";
  v = p.i*R "Ohm's law";
end Resistor;
```

以同样的方式，我们可以创建电感和电容的模型，如下：

```
within ModelicaByExample.Components.Electrical.VerboseApproach;
model Inductor "An inductor model"
  parameter Modelica.SIunits.Inductance L;
  Modelica.Electrical.Analog.Interfaces.PositivePin p
    annotation (Placement(transformation(extent={{-110,-10},{-90,10}})));
  Modelica.Electrical.Analog.Interfaces.NegativePin n
    annotation (Placement(transformation(extent={{90,-10},{110,10}})));
protected
  Modelica.SIunits.Voltage v = p.v-n.v;
equation
  p.i + n.i = 0 "Conservation of charge";
  L*der(p.i) = p.v;
end Inductor;
```

```
within ModelicaByExample.Components.Electrical.VerboseApproach;
model Capacitor "A capacitor model"
  parameter Modelica.SIunits.Capacitance C;
  Modelica.Electrical.Analog.Interfaces.PositivePin p
    annotation (Placement(transformation(extent={{-110,-10},{-90,10}})));
  Modelica.Electrical.Analog.Interfaces.NegativePin n
    annotation (Placement(transformation(extent={{90,-10},{110,10}})));
protected
  Modelica.SIunits.Voltage v = p.v-n.v;
equation
  p.i + n.i = 0 "Conservation of charge";
  C*der(v) = p.i;
end Capacitor;
```

要注意的这些模型之间共同代码。在软件开发中，这种程度的冗余时常会被鄙视。事实上，有条常用的软件工程格言说：“冗余是万恶根源”。这种冗余之所以是一个问题，一方面是由于你将同样的工作重复了多次，另一方面也因为代码也是需要维护的。若你在重复代码内找到错误，你就要在每一处进行修复。

## DRY 原则

冗余是一个重要的问题。因此，为减少重复的代码，让我们再观察了电阻、电感和电容的模型。在软件工程里有所谓的 DRY 原则。其中 DRY 表示“不要重复”。因此，我们的下一步任务就是使电阻、电容和电感模型让变得 DRY。

要消除冗余代码的关键在于，要确定这些模型内所有的公共代码，并创建一个我们可以从之继承的 partial 模型。我们在前面已经高亮了公共代码行。现在，我们可以将这些代码放到一个如下的单独模型

内:

```
within ModelicaByExample.Components.Electrical.DryApproach;
partial model TwoPin "Common elements of two pin electrical components"
  Modelica.Electrical.Analog.Interfaces.PositivePin p
    annotation (Placement(transformation(extent={{-110,-10},{-90,10}})));
  Modelica.Electrical.Analog.Interfaces.NegativePin n
    annotation (Placement(transformation(extent={{90,-10},{110,10}})));
protected
  Modelica.SIunits.Voltage v = p.v-n.v;
  Modelica.SIunits.Current i = p.i;
equation
  p.i + n.i = 0 "Conservation of charge";
end TwoPin;
```

简而言之，我们将 p、n 和 v 的声明从旧模型提取这个模型里。我们还引入了一个变量 i，以代表从引脚 p 到引脚 n 的电流。最后，电荷守恒方程也包括在内。

创建这样的模型后，我们就可以创建一个更简洁的电阻模型，如下：

```
within ModelicaByExample.Components.Electrical.DryApproach;
model Resistor "A DRY resistor model"
  parameter Modelica.SIunits.Resistance R;
  extends TwoPin;
equation
  v = i*R "Ohm's law";
end Resistor;
```

这个 Resistor 模型有几点事情需要注意。首先是模型变得有多短。这是因为我们从 TwoPin 模型继承了电引脚、电荷守恒方程和变量 v 以及 i。另一个要注意的点是，通过使用 v 和 i 的定义，欧姆定律看起来一本教科书里的一模一样。

我们可以用相同方法处理电感和电容的模型：

```
within ModelicaByExample.Components.Electrical.DryApproach;
model Capacitor "A DRY capacitor model"
  parameter Modelica.SIunits.Capacitance C;
  extends TwoPin;
equation
  C*der(v) = i;
end Capacitor;
```

```
within ModelicaByExample.Components.Electrical.DryApproach;
model Inductor "A DRY inductor model"
  parameter Modelica.SIunits.Inductance L;
  extends TwoPin;
equation
  L*der(i) = v;
end Inductor;
```

我们再一次看到的模型编得更为简洁。基本上，以这种方式抽出公共代码意味着，组件模型编得更容易编写以及更方便维护。

## 电路模型

到目前为止，我们只创建了组件模型。为了创建电路模型，我们首先还需要多定义几个组件模型。具体来说，我们需要创建一个阶跃电压源模型：

```
within ModelicaByExample.Components.Electrical.DryApproach;
model StepVoltage "A DRY step voltage source"
  parameter Modelica.SIunits.Voltage V0;
  parameter Modelica.SIunits.Voltage Vf;
  parameter Modelica.SIunits.Time stepTime;
```

```
extends TwoPin;
color={0,0,255},
```

请注意 StepVoltage 模型也利用了 TwoPin 模型。我们还需要如下的接地模型:

```
within ModelicaByExample.Components.Electrical.DryApproach;
model Ground "Electrical ground"
  Modelica.Electrical.Analog.Interfaces.PositivePin ground "Ground pin"
    annotation (Placement(transformation(extent={{-10,70},{10,90}})));
equation
  ground.v = 0;
end Ground;
```

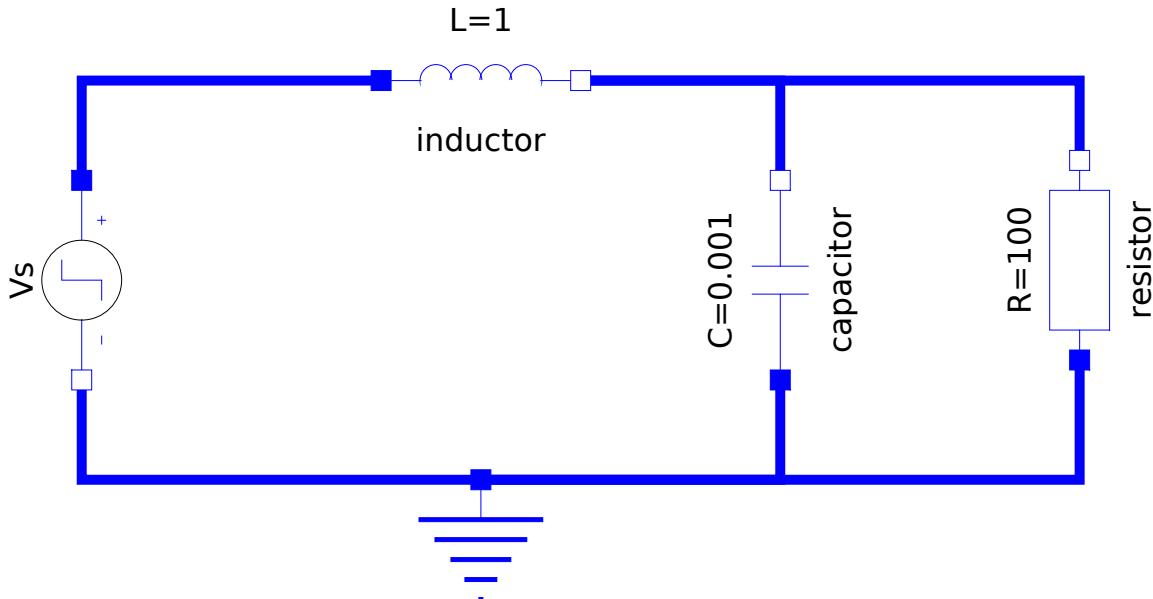
Ground 模型只有一个引脚, 因此不能继承 TwoPin。回想一下在讨论传热组件 (173) 的环境条件 (178) 模型时, 我们是如何描述无限大容器的。Ground 模型提供了非常类似的功用。不管多少电流流入或流出电接地, 电压保持为零。

定义了所有这些组件后, 现在我们可以创建电路模型如下:

```
within ModelicaByExample.Components.Electrical.Examples;
model SwitchedRLC "Recreation of the switched RLC circuit"
  DryApproach.StepVoltage Vs(V0=0, Vf=24, stepTime=0.5)
    annotation (Placement(transformation(
      extent={{-10,-10},{10,10}},
      rotation=270,
      origin={-40,0})));
  DryApproach.Inductor inductor(L=1)
    annotation (Placement(transformation(extent={{-10,10},{10,30}}));
  DryApproach.Capacitor capacitor(C=1e-3) annotation (Placement(transformation(
    extent={{-10,-10},{10,10}},
    rotation=90,
    origin={30,0})));
  DryApproach.Resistor resistor(R=100) annotation (Placement(transformation(
    extent={{-10,-10},{10,10}},
    rotation=90,
    origin={60,2})));
  DryApproach.Ground ground
    annotation (Placement(transformation(extent={{-10,-38},{10,-18}})));
equation
  connect(inductor.n, resistor.n) annotation (Line(
    points={{10,20},{60,20},{60,12}},
    color={0,0,255},
    smooth=Smooth.None));
  connect(capacitor.n, inductor.n) annotation (Line(
    points={{30,10},{30,20},{10,20}},
    color={0,0,255},
    smooth=Smooth.None));
  connect(inductor.p, Vs.p) annotation (Line(
    points={{-10,20},{-40,20},{-40,10}},
    color={0,0,255},
    smooth=Smooth.None));
  connect(capacitor.p, ground.ground) annotation (Line(
    points={{30,-10},{30,-20},{0,-20}},
    color={0,0,255},
    smooth=Smooth.None));
  connect(resistor.p, ground.ground) annotation (Line(
    points={{60,-8},{60,-20},{0,-20}},
    color={0,0,255},
    smooth=Smooth.None));
  connect(Vs.n, ground.ground) annotation (Line(
    points={{-40,-10},{-40,-20},{0,-20}},
    color={0,0,255},
    smooth=Smooth.None));
```

```
end SwitchedRLC;
```

该模型的示意图显示如下：



### 第 7.1.3 节 基本旋转组件

在这一节中，我们将介绍如何为一维转动系统创建基本组件。我们将以此前对旋转连接器的讨论为基础，展示如何将连接器应用于定义基本旋转组件的接口。最后，我们将展示如何将这些旋转部件可以被组装成系统模型。这里的系统模型将与第一章内的以方程为基础的旋转系统模型有着相同的行为。

#### 组件模型

在第一章中，我们严格地利用方程式（即无组件模型）去建模了[机械示例 \(13\)](#)。在本节中，我们将通过使用组件重新开始系统模型。要做到这一点，我们首先要定义需要的基本组件模型。这些将包括惯性、弹簧、阻尼器和机械地面的模型。

正如[上一节 \(181\)](#)，我们将首先通过累赘的方法定义组件模型。然后，我们将重新审视这些定义，并尝试将公共代码分解出来，以避免在组件模型里的重复。

#### 坐标系

创建这些模型的方法与我们在传热和电气域上创建组件模型时的方法非常类似。但是，在开始建立组件模型之前，我们应该首先讨论与机械系统相关的一个复杂性：坐标系。

在机械领域，我们研究是守恒量是动量。与此前介绍了的守恒量热量和电荷不同，动量有方向性。我们在这里仅考虑一维的情况。因此，这个方向性的后果是，动量是一个带符号的量（即动量可正可负）。

考虑一个转动惯量为  $J$  的转动质量元件。如果在惯量元件的角位置由  $\varphi$  表示，则惯量的角速度  $\omega$  就定义为：

$$\omega = \dot{\varphi}$$

显然，正的  $\omega$  值将导致  $\varphi$  随着时间的推移增加。此外，惯量元件的角加速度  $\alpha$  定义为：

$$\alpha = \ddot{\varphi}$$

和角速度一样，我们可以看到，正的  $\alpha$  将导致角速度增加。最后，惯量元件的角动量被定义为  $J\omega$ 。我们从欧拉运动定律可以知道（假设  $J$  为常数）：

$$J \frac{d\omega}{dt} = \tau$$

显然由此关系可得，正的转矩  $\tau$  将增加存储在转动质量元件里的动量。

之所以展示所有这些关系，是为了强调与  $\varphi$ 、 $\omega$ 、 $\alpha$  以及  $\tau$  相关的符号规则。这些规则都与角位移正负的基本定义有关。让  $\varphi$  增加的方向就是正速度、正加速度、正转矩的方向。

### 转动惯量元件

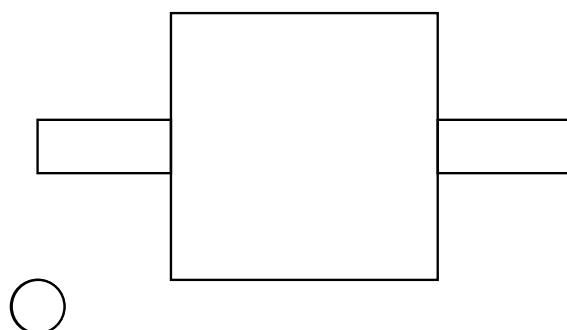
我们已经讨论正负号规则以及坐标系。现在，我们就可以开始创建组件模型。首先，我们创建惯量模型：

```
within ModelicaByExample.Components.Rotational.VerboseApproach;
model Inertia "Rotational inertia without inheritance"
  parameter Modelica.SIunits.Inertia J;
  Modelica.Mechanics.Rotational.Interfaces.Flange_a flange_a
    annotation (Placement(transformation(extent={{-110,-10},{-90,10}})));
  Modelica.Mechanics.Rotational.Interfaces.Flange_b flange_b
    annotation (Placement(transformation(extent={{90,-10},{110,10}})));
protected
  Modelica.SIunits.Angle phi_rel;
  Modelica.SIunits.Torque tau;
  Modelica.SIunits.AngularVelocity w;
equation
  // Variables
  phi_rel = flange_a.phi-flange_b.phi;
  tau = flange_a.tau;
  w = der(flange_a.phi);

  // Conservation of angular momentum (includes storage)
  J*der(w) = flange_a.tau + flange_b.tau;

  // Kinematic constraint (inertia is rigid)
  phi_rel = 0;
  annotation (Icon(graphics={
    extent={{-100,90},{100,50}},
```

Inertia 模型包括两个“法兰”，两端各一个。这些法兰的意义是可以更清楚地从 Inertia 模型的图标看出：



换言之，Inertia 模型在每一端都有一个法兰。你可以把这个模型看作两端都有接口的轴。

现在，Inertia 模型所需要表示的基本公式为：

```
J*der(w) = flange_a.tau + flange_b.tau;
```

这表达的基本事实为，惯量元素内储存的动量增加率等于施加其上的转矩之和。回想一下，在之前对非因果连接（159）的讨论里，我们所规定的连接器流变量（这里为 flange\_a.tau 和 flange\_b.tau）的符号规则：正值表示保守量的流在进入到组件模型。flange\_a 和 flange\_b 有相同的符号约定这一点，意味着 Inertia 模型是对称的（即可以在翻转后保持行为不变）。

不过，这个等式引用了内部变量 w（代表  $\omega$ ）和 tau。所以我们也需要包含这些变量的声明和定义。

### 弹簧模型

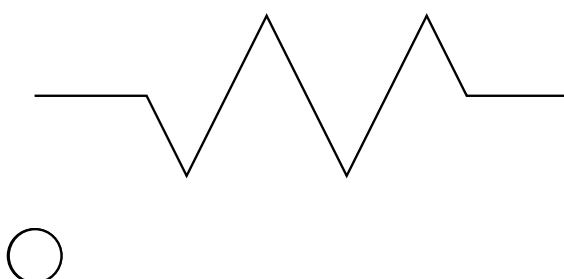
接下来，让我们考虑一个弹簧模型的定义：

```
within ModelicaByExample.Components.Rotational.VerboseApproach;
model Spring "Rotational spring without inheritance"
parameter Modelica.SIunits.RotationalSpringConstant c;
Modelica.Mechanics.Rotational.Interfaces.Flange_a flange_a
annotation (Placement(transformation(extent={{-110,-10},{-90,10}})));
Modelica.Mechanics.Rotational.Interfaces.Flange_b flange_b
annotation (Placement(transformation(extent={{90,-10},{110,10}})));
protected
Modelica.SIunits.Angle phi_rel;
Modelica.SIunits.Torque tau;
equation
// Variables
phi_rel = flange_a.phi-flange_b.phi;
tau = flange_a.tau;

// No storage of angular momentum
flange_a.tau + flange_b.tau = 0;

// Hooke's law
tau = c*phi_rel;
end Spring;
```

弹簧模型的图标如下所示：



和 Inertia 模型一样，Spring 模型有两个连接器，一端各一个。模型还定义了许多相同的内部变量。最终，弹簧的行为可以总结为如下公式：

```
// Hooke's law
tau = c*phi_rel;
end Spring;
```

事实上，除了上述方程和参数  $c$  以外，Spring 模型的内容与 Inertia 模型内容大部分是一样的。

### 阻尼器模型

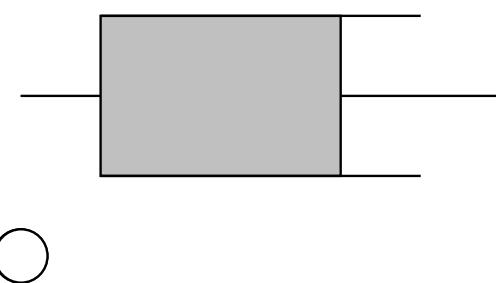
Damper 模型也和 Spring 模型非常相似。同样，主要区别是参数（这里是  $d$ ），以及一个方程：

```
within ModelicaByExample.Components.Rotational.VerboseApproach;
model Damper "Rotational damper without inheritance"
  parameter Modelica.SIunits.RotationalDampingConstant d;
  Modelica.Mechanics.Rotational.Interfaces.Flange_a flange_a
    annotation (Placement(transformation(extent={{-110,-10},{-90,10}})));
  Modelica.Mechanics.Rotational.Interfaces.Flange_b flange_b
    annotation (Placement(transformation(extent={{90,-10},{110,10}})));
protected
  Modelica.SIunits.Angle phi_rel;
  Modelica.SIunits.Torque tau;
equation
  // Variables
  phi_rel = flange_a.phi-flange_b.phi;
  tau = flange_a.tau;

  // No storage of angular momentum
  flange_a.tau + flange_b.tau = 0;

  // Damping relationship
  tau = d*der(phi_rel);
end Damper;
```

Damper 模型的图标如下所示：



### DRY 组件模型

我们已经有了惯性元件、弹簧和阻尼器的模型。要完成此前的双弹簧质量阻尼系统（13）唯一缺少的模型就是机械地面的模型了。但在此之前，让我们花一点时间来重新审视这些模型。我们的目标是抽取出这些模型之间的大量共同代码。和上节（181）一样，让我们慢慢来去实践 DRY（不要重复）原则。

## 共同代码

值得一提的是，由于 Modelica 的标准库拥有大量的旋转部件，几乎从一开始，标准库就要处理冗余代码的问题。不过，我们不会在这里使用 Modelica 标准库的 partial 模型。原因简单，这些模型都在设计时都考虑了许多与本节讨论不相关的其他情况。因此，这些（虽然必要的）复杂性使得上述模型不适用于教学。

但我们将从 Modelica 的标准库学习一点。这就是多个 partial 模型的必要性。这是因为，不像此前讨论的电气部件 (181)，我们的转动组件模型彼此间共享不同的代码。

一个相同点在于，所有模型均带有两个法兰连接器 flange\_a 和 flange\_b。然而，虽然 Inertia 模型能够存储角动量，但 Spring 和 Damper 模型则不能。其结果是，这些组件之间有不同的守恒方程。

让我们从所有三种模型共有的元素开始。共有元素由以下的 TwoFlange 模型表示：

```
within ModelicaByExample.Components.Rotational.Interfaces;
partial model TwoFlange
  "Definition of a partial rotational component with two flanges"

  Modelica.Mechanics.Rotational.Interfaces.Flange_a flange_a
    annotation (Placement(transformation(extent={{-110,-10},{-90,10}})));
  Modelica.Mechanics.Rotational.Interfaces.Flange_b flange_b
    annotation (Placement(transformation(extent={{90,-10},{110,10}})));
protected
  Modelica.SIunits.Angle phi_rel;
equation
  phi_rel = flange_a.phi-flange_b.phi;
end TwoFlange;
```

除了定义的两个法兰 flange\_a 和 flange\_b 外，这个模型还定义了两法兰间的相对角度，即 phi\_rel。当然，这种模型也被标记为 partial。因为模型缺少对组件行为的任何描述。

所有的三种模型都可以继承上述模型。但是，Spring 和 Damper 模型之间仍会有一些冗余的方程式。因此，我们将转而创建 TwoFlange 模型的一个稍为特别的版本。这个版本表示不存储动量的可压缩模型：

```
within ModelicaByExample.Components.Rotational.Interfaces;
partial model Compliant "A compliant rotational component"
  extends ModelicaByExample.Components.Rotational.Interfaces.TwoFlange;
protected
  Modelica.SIunits.Torque tau;
equation
  tau = flange_a.tau;
  flange_a.tau + flange_b.tau = 0
  "Conservation of angular momentum (no storage)";
end Compliant;
```

Compliant 模型加上额外的内部变量（以代表从 flange\_a 进入该组件然后传递到 flange\_b 的转矩）。另外，模型也加入了方程，以表示组件不会存储任何角动量。

有了这些定义的基类，让我们快速重构各个组件模型的定义。这就可以看出继承能减少多少冗余。

## 转动惯量元件

凭借 TwoFlanges 模型，我们的 Inertia 模型可以简化为：

```
within ModelicaByExample.Components.Rotational.Components;
model Inertia "A rotational inertia model"
  parameter Modelica.SIunits.Inertia J;
  extends ModelicaByExample.Components.Rotational.Interfaces.TwoFlange;
  Modelica.SIunits.AngularVelocity w "Angular Velocity"
    annotation(Dialog(group="Initialization", showStartAttribute=true));
  Modelica.SIunits.Angle phi "Angle"
    annotation(Dialog(group="Initialization", showStartAttribute=true));
```

```

equation
  phi = flange_a.phi;
  w = der(flange_a.phi) "velocity of inertia";
  phi_rel = 0 "inertia is rigid";
  J*der(w) = flange_a.tau + flange_b.tau
    "Conservation of angular momentum with storage";
end Inertia;

```

### 弹簧模型

用相同的方式继承了 Compliant 模型后, Spring 模型可以更紧凑地表示为:

```

within ModelicaByExample.Components.Rotational.Components;
model Spring "A rotational spring component"
  parameter Modelica.SIunits.RotationalSpringConstant c;
  extends ModelicaByExample.Components.Rotational.Interfaces.Compliant;
equation
  tau = c*phi_rel "Hooke's Law";
end Spring;

```

### 阻尼器模型

同样, Damper 模型也类似地简化为:

```

within ModelicaByExample.Components.Rotational.Components;
model Damper "A rotational damper"
  parameter Modelica.SIunits.RotationalDampingConstant d;
  extends ModelicaByExample.Components.Rotational.Interfaces.Compliant;
equation
  tau = d*der(phi_rel) "Damping relationship";
end Damper;

```

### 机械地面

最后, 我们可以完成双弹簧质量阻尼系统 (13) 里唯一剩余的模型了。机械地面模型定义如下:

```

within ModelicaByExample.Components.Rotational.Components;
model Damper "A rotational damper"
  parameter Modelica.SIunits.RotationalDampingConstant d;
  extends ModelicaByExample.Components.Rotational.Interfaces.Compliant;
equation
  tau = d*der(phi_rel) "Damping relationship";
  annotation (Icon(graphics={

```

### 双弹簧质量阻尼系统

最后, 我们已经得到了重建第一章内例子所需的所有部件。使用本节中定义的各个组件, 我们基于组件的系统模型其 Modelica 代码如下:

```

within ModelicaByExample.Components.Rotational.Examples;
model SMD
  Components.Ground ground annotation (Placement(transformation(
    extent={{-10,-10},{10,10}},
    rotation=90,
    origin={76,0})));
  Components.Damper damper2(d=1)
  annotation (Placement(transformation(extent={{30,10},{50,30}})));

```

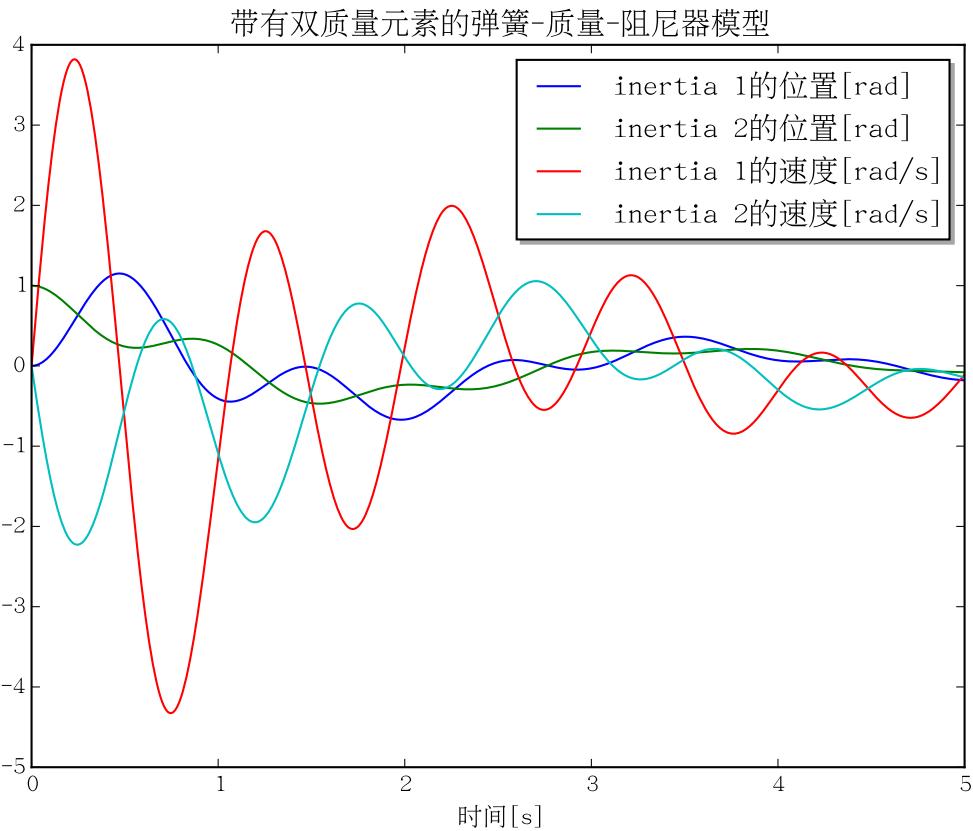
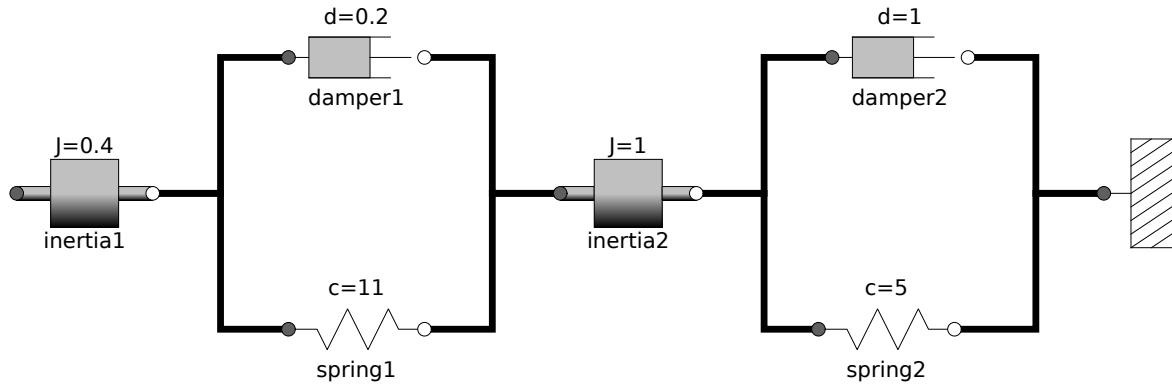
```

Components.Spring spring2(c=5)
annotation (Placement(transformation(extent={{28,-30},{48,-10}})));
Components.Inertia inertia2(
J=1,
phi(fixed=true, start=1),
w(fixed=true, start=0))
annotation (Placement(transformation(extent={{-10,-10},{10,10}})));
Components.Damper damper1(d=0.2)
annotation (Placement(transformation(extent={{-50,10},{-30,30}})));
Components.Spring spring1(c=11)
annotation (Placement(transformation(extent={{-50,-30},{-30,-10}})));
Components.Inertia inertia1(
J=0.4,
phi(fixed=true, start=0),
w(fixed=true, start=0))
annotation (Placement(transformation(extent={{-90,-10},{-70,10}})));
equation
connect(ground.flange_a, damper2.flange_b) annotation (Line(
points={{70,0},{66,0},{66,0},{60,0},{60,20},{50,20}},
color={0,0,0},
smooth=Smooth.None));

connect(ground.flange_a, spring2.flange_b) annotation (Line(
points={{70,0},{60,0},{60,-20},{48,-20}},
color={0,0,0},
smooth=Smooth.None));
connect(damper2.flange_a, inertia2.flange_b) annotation (Line(
points={{30,20},{20,20},{20,0},{10,0}},
color={0,0,0},
smooth=Smooth.None));
connect(spring2.flange_a, inertia2.flange_b) annotation (Line(
points={{28,-20},{20,-20},{20,0},{10,0}},
color={0,0,0},
smooth=Smooth.None));
connect(inertia2.flange_a, damper1.flange_b) annotation (Line(
points={{-10,0},{-20,0},{-20,20},{-30,20}},
color={0,0,0},
smooth=Smooth.None));
connect(inertia2.flange_a, spring1.flange_b) annotation (Line(
points={{-10,0},{-20,0},{-20,-20},{-30,-20}},
color={0,0,0},
smooth=Smooth.None));
connect(damper1.flange_a, inertia1.flange_b) annotation (Line(
points={{-50,20},{-60,20},{-60,0},{-70,0}},
color={0,0,0},
smooth=Smooth.None));
connect(spring1.flange_a, inertia1.flange_b) annotation (Line(
points={{-50,-20},{-60,-20},{-60,0},{-70,0}},
color={0,0,0},
smooth=Smooth.None));
end SMD;

```

模型的简图如下所示



以此完成了我们对基本旋转组件的讨论。但在下一节高级旋转组件 (192)，我们还会进一步深入探讨旋转组件。

#### 第 7.1.4 节 高级旋转组件

在上一节中，我们讨论了基本旋转组件 (185) 并展示了如何构建基本组件的系统模型。在这一节中，我们将演示如何加入事件处理。我们将在模拟齿隙时使用到这个功能。此外，我们还将展示如何使用参数值来实现组件的接口。

## 建模齿隙

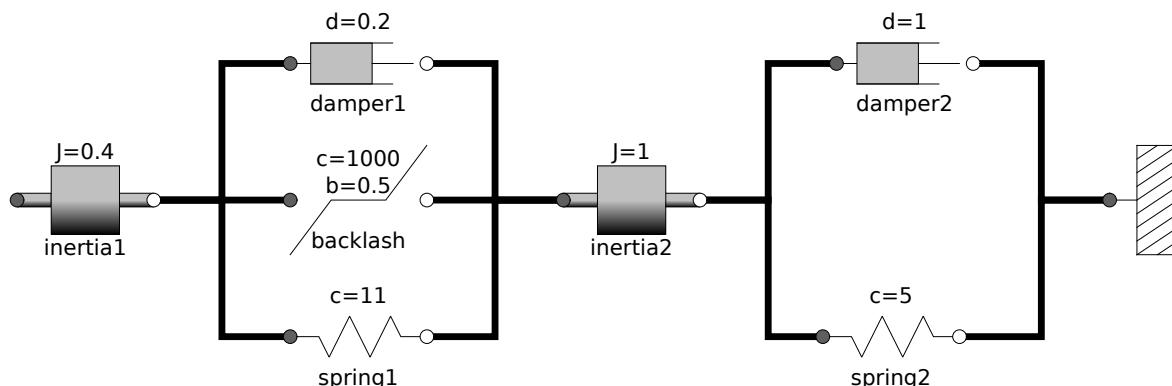
让我们从旋转齿隙元素开始对高级旋转组件模型的探索。齿隙模型的方程非常简单:

$$\tau = \begin{cases} c(\Delta\varphi - \frac{b}{2}) & \text{if } \Delta\varphi > \frac{b}{2} \\ c(\Delta\varphi + \frac{b}{2}) & \text{if } \Delta\varphi < -\frac{b}{2} \\ 0 & \text{otherwise} \end{cases}$$

该组件可以用 Modelica 描述如下:

```
within ModelicaByExample.Components.Rotational.Components;
model Backlash "A rotational backlash model"
parameter Modelica.SIunits.RotationalSpringConstant c;
parameter Modelica.SIunits.Angle b(final min=0) "Total lash";
extends ModelicaByExample.Components.Rotational.Interfaces.Compliant;
equation
if phi_rel>b/2 then
  tau = c*(phi_rel-b/2);
elseif phi_rel<-b/2 then
  tau = c*(phi_rel+b/2);
else
  tau = 0 "In the lash region";
end if;
end Backlash;
```

我们可以将齿隙模型的实例加入此前的模型里。将其与弹簧及阻尼器相并联，即:

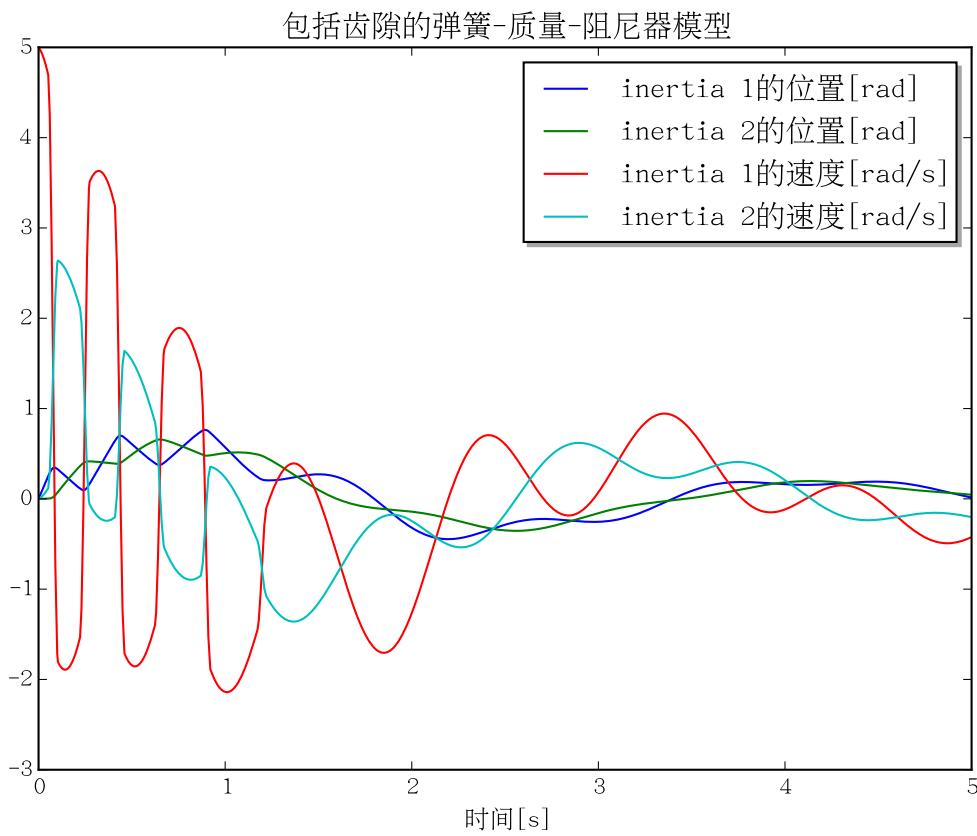


若我们使用的 Modelica 的继承机制，由此产生的 Modelica 模型会很简单:

```
within ModelicaByExample.Components.Rotational.Examples;
model SMD_WithBacklash "The spring-mass-damper system with backlash"
  extends SMD(inertia2(phi(fixed=true, start=0)), inertia1(phi(fixed=true, start=0), w(start=5)));
  Components.Backlash backlash(c=1000, b(displayUnit="rad") = 0.5)
  annotation (Placement(transformation(extent={{-50,-10},{-30,10}})));
equation
  connect(inertia1.flange_b, backlash.flange_a) annotation (Line(
    points={{-70,0}, {-50,0}},
    color={0,0,0},
    smooth=Smooth.None));
  connect(backlash.flange_b, inertia2.flange_a) annotation (Line(
    points={{-30,0}, {-10,0}},
    color={0,0,0},
    smooth=Smooth.None));
end SMD_WithBacklash;
```

在这种情况下，如果 inertia1 和 inertia2 之间的相对角度超过 0.5 弧度（即我们的齿隙实例里 b 的值），那么齿隙元件会带来转矩。

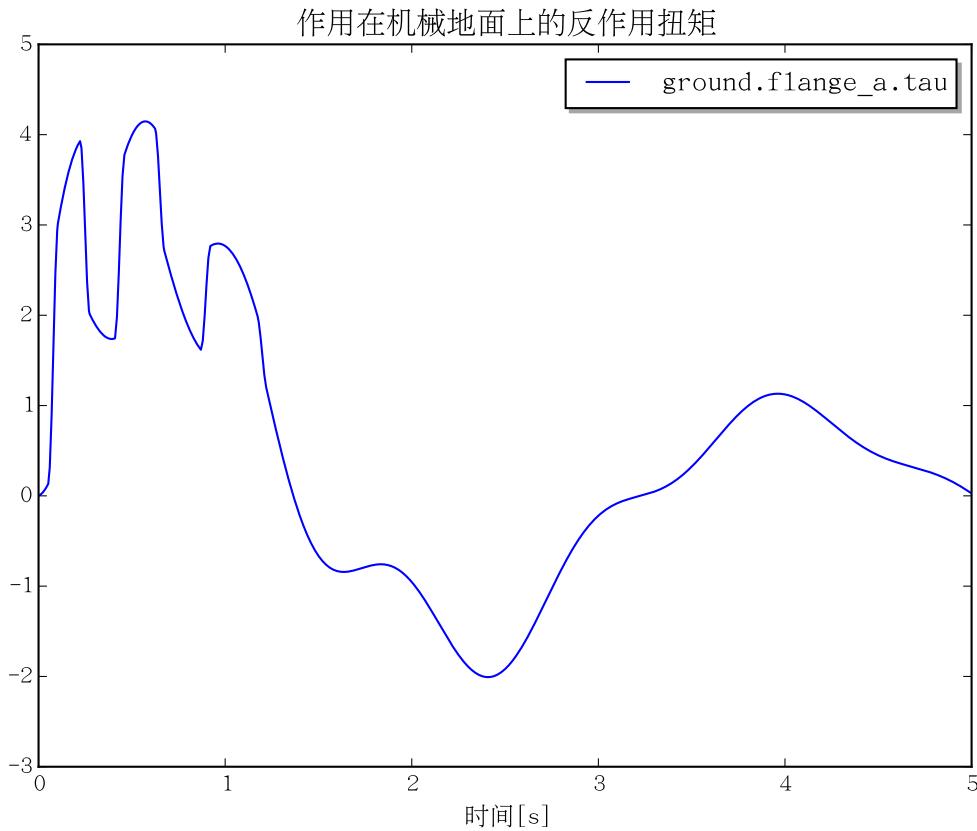
如果对该模型进行仿真，我们就可以看到，齿隙的存在对系统的响应的影响:



另一个值得观察的现象是（我们将在下个主题更深入地探讨）是机械地面元件的受力。从我们的示意图能够明显看出，该机械地面元件的作用是固定系统其中一边的角位移。用以约束系统内某个点运动（或在这里，无运动）的方程被称为运动学约束。

当一个运动约束施加在系统上时，该施加约束的组件必须产生某种作用力或力矩，以此影响系统的运动。这就是所谓的反作用力或反作用转矩。

下面的图显示的反作用转矩，就是该机械地面元件为了固定角位置所必须施加给系统的：



### 地面扭矩与反作用扭矩

正如我们在前面的例子中所看到的，机械地面元件必须对系统施加反作用转矩来约束其运动。在本节中，我们将稍进一步研究这个效应。

为了演示运动学约束的某些复杂性，我们需要创建一个机械齿轮模型。在这个模型中，我们将忽略齿轮元件的惯量、齿轮间的效率损失以及可能在齿轮齿之间存在的任何齿隙。回想一下我们在本章前面对[进一步研究 \(178\)](#)讨论内容，我们提到的组件模型应侧重于单个的物理效应。同样的原则也适用于这里。惯性、摩擦和齿隙都可以建模成单独的效应（正如我们在本章所看到的）。我们没有必要所有效应都放入齿轮模型里。相反，我们将只专注于齿轮输入速度与输出速度间的关系。

在典型的系统动力学类里，描述齿轮行为的方程推导如下。首先，我们必须理解，该齿轮引入了输入速度与输出速度间的关系，即：

$$\omega_a = R\omega_b$$

其中  $R$  是齿轮比。记得我们假设了齿轮效率为 1。这意味着齿轮的输入功率必须等于输出功率。这可以在数学上表示为：

$$\tau_a \omega_a + \tau_b \omega_b = 0$$

请注意，我们在这使用了 Modelica 的正负号约定。因此，守恒量的流为正意味着其正在流入组件内。在这种情况下， $\tau_a \omega_a$  是从 flange\_a 流入齿轮的机械功率。而  $\tau_b \omega_b$  则从 flange\_b 流出齿轮的机械功率。因此，两者的总和必须为零。因为，我们的齿轮模型不包括齿轮元件的惯性。其结果就是齿轮模型内不可能储存能量或动力。

鉴于这两个事实，我们可以将速度关系代入到功率关系。由此可以得到：

$$\tau_a R\omega_b + \tau_b \omega_b = 0$$

这让我们从方程消去了  $\omega_b$ 。重新组织方程我们可以得到:

$$\tau_b = -R\tau_a$$

这样的推导对大多数工程师看起来可能会很熟悉。但是，一定要意识到这里还差了些什么。更具体而言，推理的一些隐含假定并不一定合理。

要理解这个问题，让我们先考虑欧拉第二定律:

$$J\ddot{\varphi} = \sum_i \tau_i$$

换言之，施加在物体上扭矩的总和应等于新累积的角动量。回想我们的齿轮模型并不包括齿轮元件的惯性。因此，模型无法以存储能量或角动量。在这种情况下，先前的方程简化为:

$$\sum_i \tau_i = 0$$

我们的齿轮只有两个外部扭矩  $\tau_a$  和  $\tau_b$ 。应用之前推导出的关系，我们知道其总和为:

$$\tau_a + \tau_b = \tau_a - R\tau_a = \tau_a(1 - R) = 0$$

这个方程意味着对任何不等于 1.0 的齿轮比  $R$ , flange\_a 上转矩（以及由此推出转矩 flange\_b 也）必须为零。但是，若我们的齿轮能提供齿轮功能的话，这就不可能是正确的。

为了显示上述数学关系向我们展示了系统的一种怎样的物理特性，我们可以观察从下列等式。该等式更清楚地表明了系统的属性。

$$\tau_a - R\tau_a = 0$$

第一项  $\tau_a$ ，是从 flange\_a 进入齿轮的扭矩。第二项  $\tau_b$ ，是从 flange\_b 进入齿轮的扭矩。这个等式告诉我们，这两个扭矩的总和绝不会为零（对于  $R \neq 1$ ）。我们看起来在数学上证明了  $\tau_a = 0$ 。但实际上，我们实际上证明了公式里并不平衡。这种不平衡由于我们在列式中忘记一些东西。这里缺少了反作用扭矩。

如果你还不熟悉这个问题，你可能会感到困惑。这个反作用扭矩从何而来？毕竟，我们对齿轮只有两个机械连接。而我们也把这两个点的扭矩表示了出来。但一直以来，这里有一个隐含的假设，即齿轮的外壳与地面相连。在现实中，一个齿轮有三个机械连接。第三个连接是在齿轮的外壳与齿轮安装的安装位置之间。如果壳体被连接到机械地面，那么目前为止我们的方程是正确的。因为我们可以用如下方式描述（接地）齿轮的行为：

```
within ModelicaByExample.Components.Rotational.Components;
model GroundedGear "An ideal non-reversing gear with grounded housing"
  parameter Real ratio "Ratio of phi_a/phi_b";
  extends Interfaces.TwoFlange;
equation
  ratio*flange_a.tau + flange_b.tau = 0 "No storage";
  flange_a.phi = ratio*flange_b.phi "Kinematic constraint";
annotation (Icon(graphics={
  Rectangle(
    extent={{-100,10},{-40,-10}},
    lineColor={0,0,0},
    fillPattern=FillPattern.HorizontalCylinder,
    fillColor={192,192,192}),
  Rectangle(
    extent={{-40,20},{-20,-20}},
    lineColor={0,0,0},
    fillPattern=FillPattern.HorizontalCylinder,
    fillColor={192,192,192}),
  Rectangle(
    extent={{-40,100},{-20,20}},
    lineColor={0,0,0},
    fillPattern=FillPattern.HorizontalCylinder,
    fillColor={192,192,192}),
  Rectangle(
    extent={{-40,100},{-20,20}},
    lineColor={0,0,0},
    fillPattern=FillPattern.HorizontalCylinder,
    fillColor={192,192,192}),
}))
```

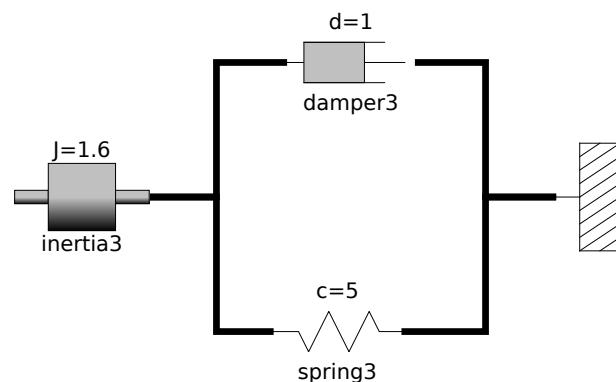
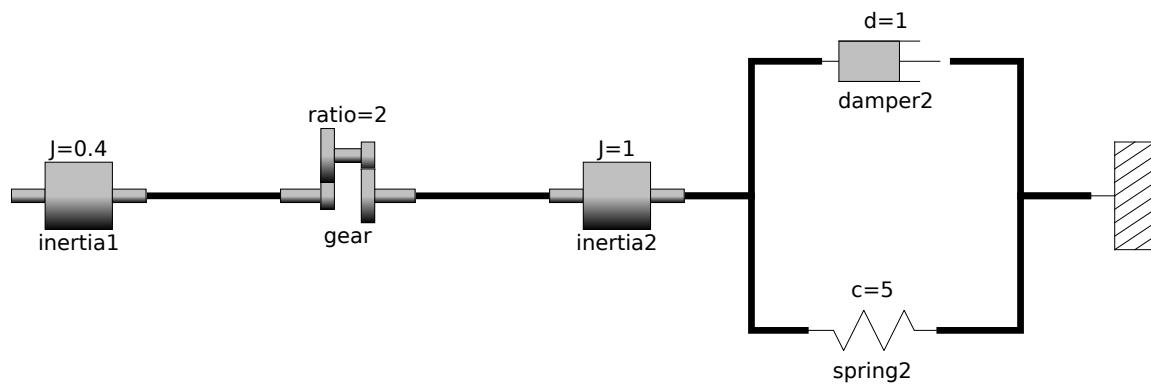
```

extent={{-20,70},{20,50}},
lineColor={0,0,0},
fillPattern=FillPattern.HorizontalCylinder,
fillColor={192,192,192}),
Rectangle(
extent={{20,80},{40,39}},
lineColor={0,0,0},
fillPattern=FillPattern.HorizontalCylinder,
fillColor={192,192,192}),
Rectangle(
extent={{20,40},{40,-40}},
lineColor={0,0,0},
fillPattern=FillPattern.HorizontalCylinder,
fillColor={192,192,192}),
Rectangle(
extent={{40,10},{100,-10}},
lineColor={0,0,0},
fillPattern=FillPattern.HorizontalCylinder,
fillColor={192,192,192}),
Text(
extent={{-100,140},{100,100}},
lineColor={0,0,0},
fillColor={255,255,255},
fillPattern=FillPattern.Solid,
textString="ratio=%ratio"),
Text(
extent={{-100,-40},{100,-80}},
lineColor={0,0,0},
fillColor={255,255,255},
fillPattern=FillPattern.Solid,
textString="%name")));
end GroundedGear;

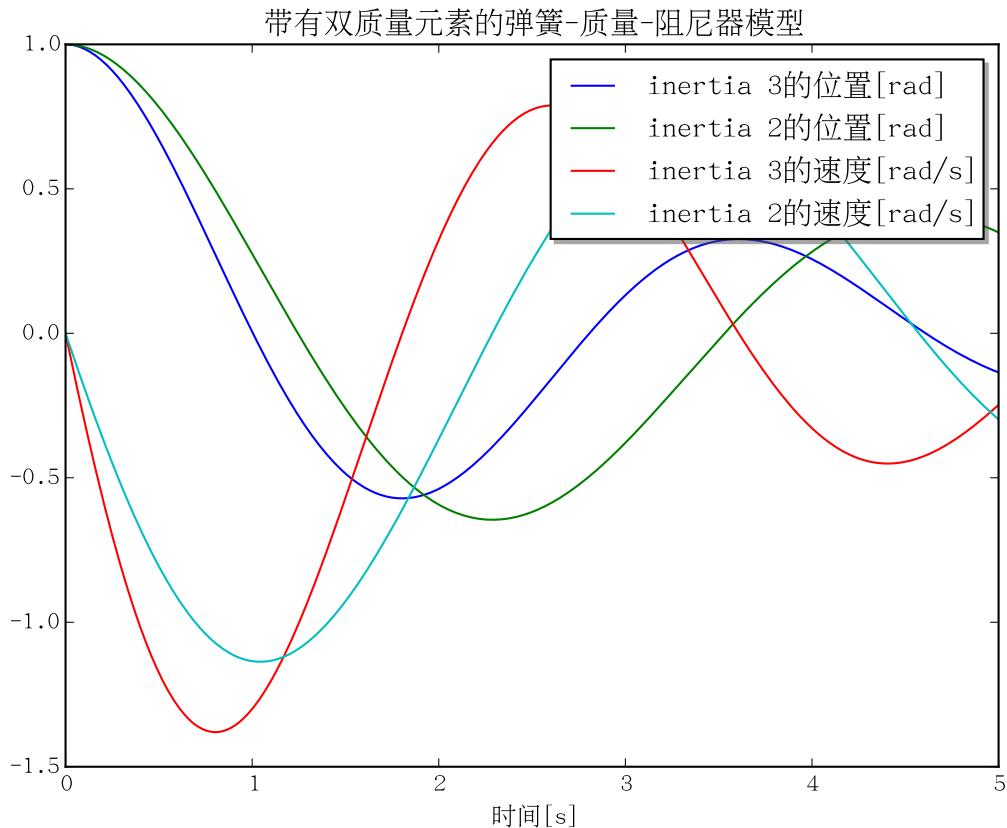
```

请注意在 GroundedGear 模型使用的关系不再是  $\omega_a = R\omega_b$ , 而是  $\varphi_a = R\varphi_b$ 。这实际上更为准确。因为, 在组装后, 齿轮的齿确实限制了两轴的角位置。此外, 在一些应用里 (例如步进电机) 保留位置关系而非仅保留速度关系可能会非常重要。

使用 GroundedGear 模型, 我们可以创建如下系统模型:



请注意，此系统有两种实现。第一种使用我们刚刚开发的齿轮模型。第二个将齿轮、惯量元件总成替换为单个惯量元件。这个惯量元件的值特别设定为上述总成的“有效惯量”。因此，在我们模拟这个系统时，我们看到 **inertia2** 和 **inertia3** 有相同的相应：



## 比较

正如前面所提到的，GroundedGear 模型的问题在于和地面相连这一隐含假定。这种假设可能不总是合理的（例如，汽车的变速器中的齿轮通常连接到可形变的支架上）。我们希望了解连接地面与否会对系统的响应有多大的影响。因此，我们将首先创建一个不隐含连接地面的更完整的齿轮模型。然后，将其性能与连接地面的齿轮一并比较。

没有了齿轮外壳连接地面这一隐含的假设，两个轴和壳体之间的运动学关系能更完整地表示为：

$$(1 - R)\varphi_h = \varphi_a - R\varphi_b$$

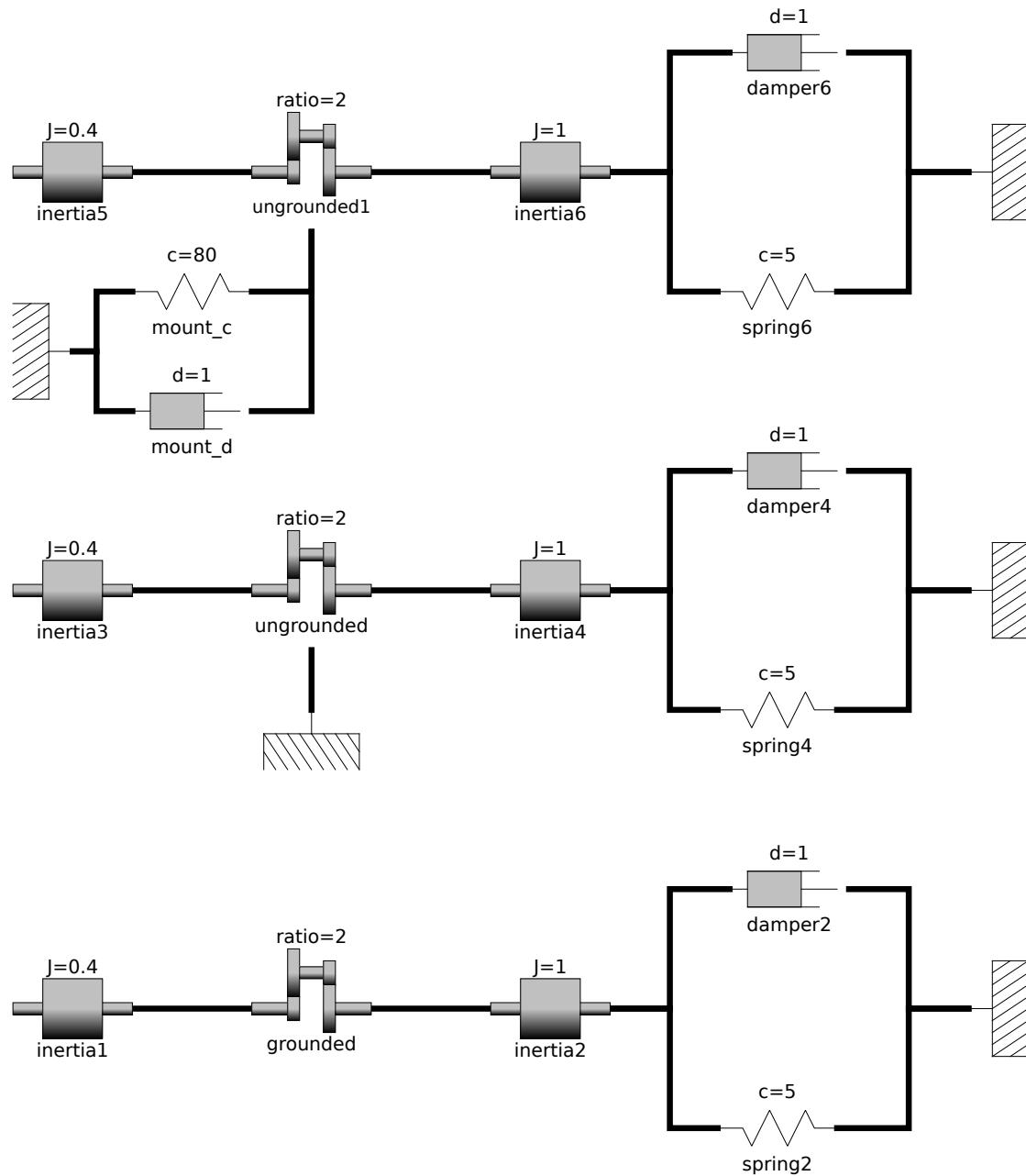
虽然这已经超出了本文的讨论范围，我们可以用能量守恒和动量守恒得出以下两个等式：

$$\tau_b = -R\tau_a\tau_h = -(1 - R)\tau_a$$

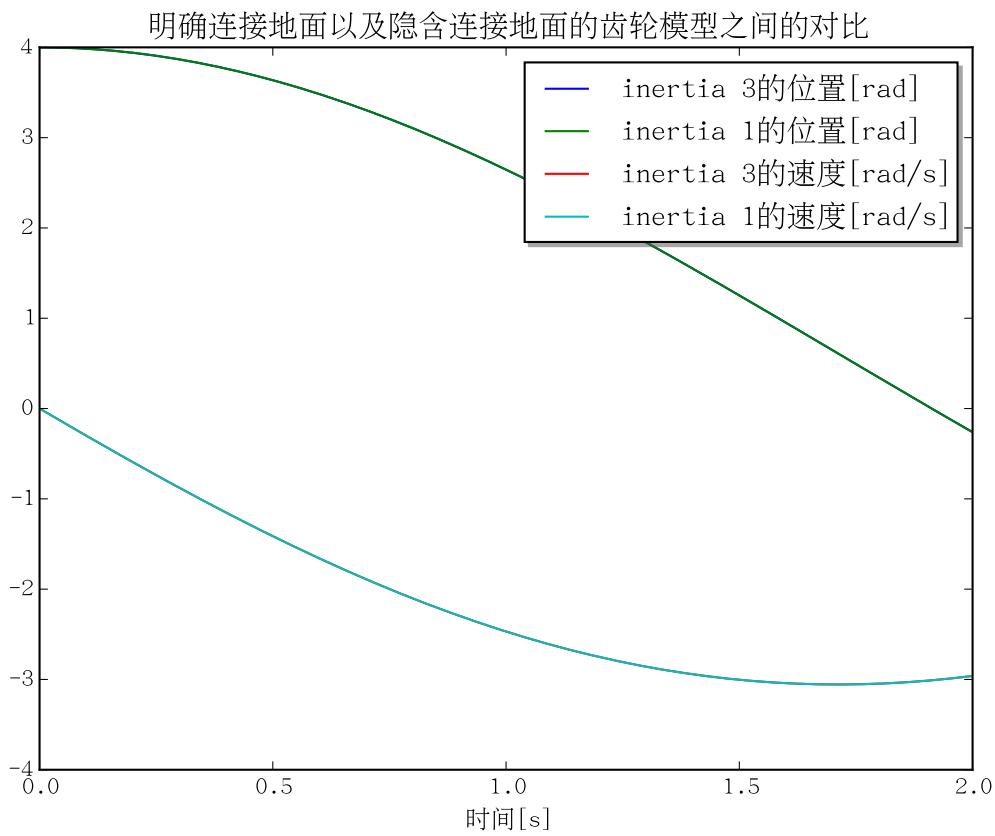
将这些关系结合在一起，并增加机械连接器去表示壳体后，我们可以得到以下理想齿轮的 Modelica 模型：

```
within ModelicaByExample.Components.Rotational.Components;
model UngroundedGear "An ideal non-reversing gear with a free housing"
  parameter Real ratio "Ratio of phi_a/phi_b";
  extends Interfaces.TwoFlange;
  Modelica.Mechanics.Rotational.Interfaces.Flange_b housing
    "Connection for housing"
    annotation (Placement(transformation(extent={{-10,-110},{10,-90}})));
equation
  (1-ratio)*housing.phi = flange_a.phi - ratio*flange_b.phi;
  flange_b.tau = -ratio*flange_a.tau;
  housing.tau = -(1-ratio)*flange_a.tau;
end UngroundedGear;
```

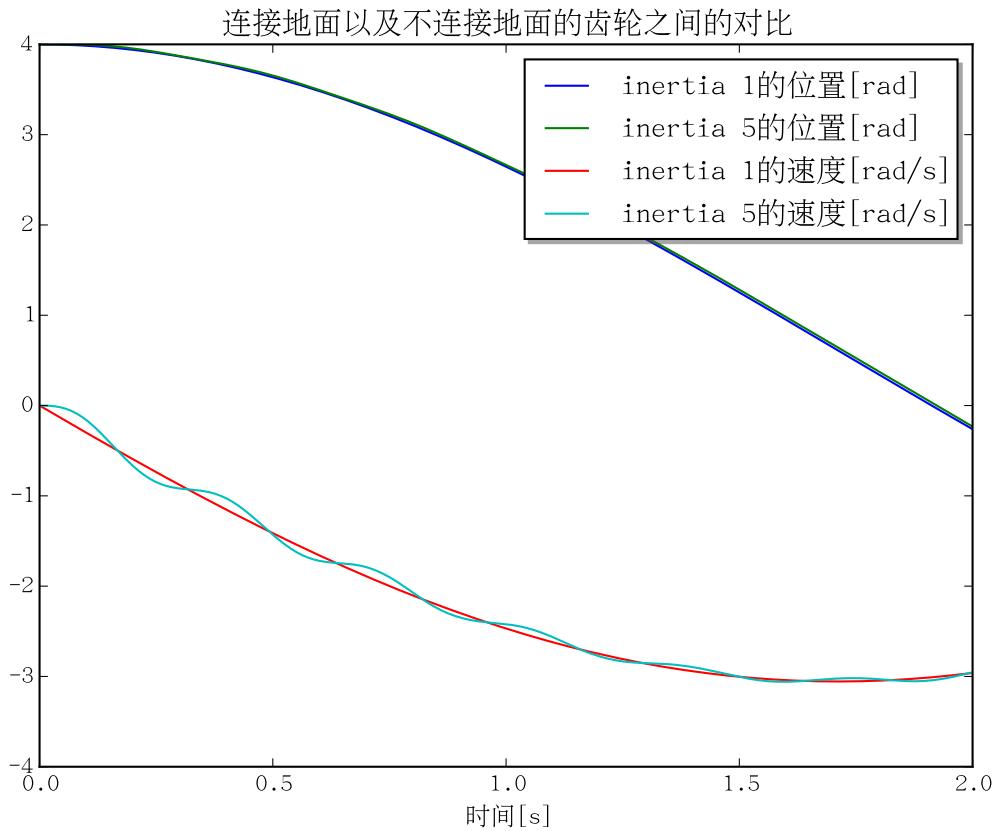
现在，让我们一起建立是用上述三种不同机制的系统模型。在各机构齿轮、惯性、弹簧和阻尼器的参数都是相同的。唯一的区别是，我们是使用隐式连接地面齿轮、显式连接地面齿轮或不直接连接到地面上而是连接在硬支架上的齿轮。我们的系统示意图如下所示：



我们可以预期的第一件事是，隐式和显示连接地面的齿轮机构造成的响应应该是相同的。这在以下的图中得到证实：



但此前的问题仍然存在。如果我们假定齿轮和地面隐式连接，而实际上不是的时候，这会造成多大的差异呢？问题在下图中得到清晰的解答：



### 可选地面连接器

到目前为止，在对转动系统的讨论里，我们已经创建了两个不同的齿轮模型。GroundedGear 为隐式连接地面。而 UngroundedGear 则包括了壳体的机械连接器。其实，两个部件的方程式非常相似。其代码中有相当一部分是共同代码。正如我们之前谈到这样，这样的冗余应该避免。

在这种情况下，我们避免冗余的一种方法是组合这两个模型。这似乎是这是不切实际的。既然两模型有很不同的基本假设。更重要的是，它们的接口不同（即带有不同的连接器）。不过，我们也可以通过利用所谓的有条件声明去统一两个模型。

请考虑以下的 ConfigurableGear 模型：

```
within ModelicaByExample.Components.Rotational.Components;
model ConfigurableGear
  "An ideal non-reversing gear which can be free or grounded"
  parameter Real ratio "Ratio of phi_a/phi_b";
  parameter Boolean grounded(start=false) "Set if housing should be grounded";
  extends Interfaces.TwoFlange;
  Modelica.Mechanics.Rotational.Interfaces.Flange_b housing(phi=housing_phi,
    tau = -flange_a.tau-flange_b.tau) if not grounded "Connection for housing"
    annotation (Placement(transformation(extent={{-10,-110},{10,-90}})));
protected
  Modelica.SIunits.Angle housing_phi;
equation
  if grounded then
    housing_phi = 0;
  end if;
  (1-ratio)*housing_phi = flange_a.phi - ratio*flange_b.phi;
  flange_b.tau = -ratio*flange_a.tau;
end ConfigurableGear;
```

特别要注意 housing 声明结尾带有 if no grounded。若 if 出现在声明的结尾，则表明该变量只会在 if 后的条件表达式为真时存在。所以，若 grounded 参数为真，模型中就不存在 housing 连接器。此外，housing 声明包括的方程，如修改语句（即  $\phi=housing\_phi$  以及  $\tau=-flange\_a.\tau-flange\_b.\tau$ ），也会随着声明消失。

同时，equation 区域中，我们看到 if 语句在模型连接地面时提供了一条额外的方程  $housing\_phi=0$ 。这是必要的。原因是变量  $housing\_phi$  总是存在（即在其声明的结尾没有 if）。所以该变量必须对应一条公式。

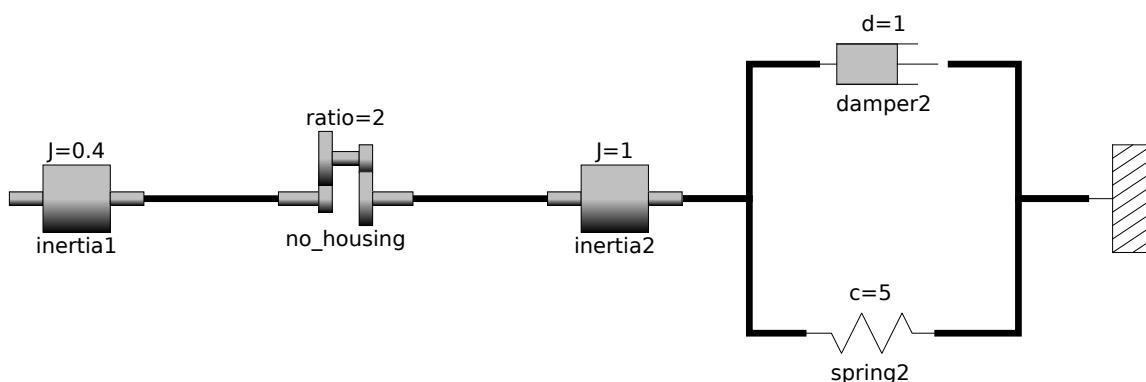
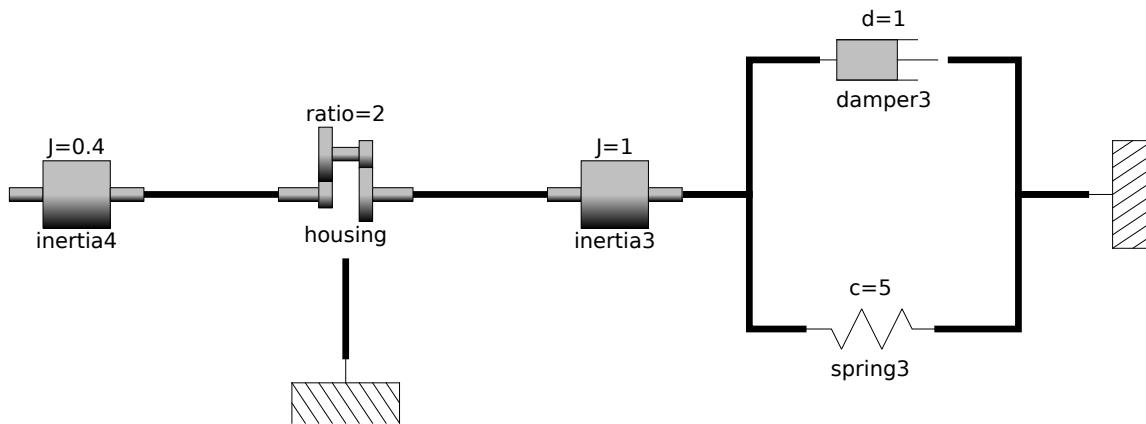
要了解更多的彻底理解是怎么回事，我们要记得一点。组件模型所需的方程数目等于所有组件内连接器的流变量数目 + 在模型中声明的（非参数）变量的数目。

下表总结了分别为 grounded 值的真假两种情况总结了上述量的关系：

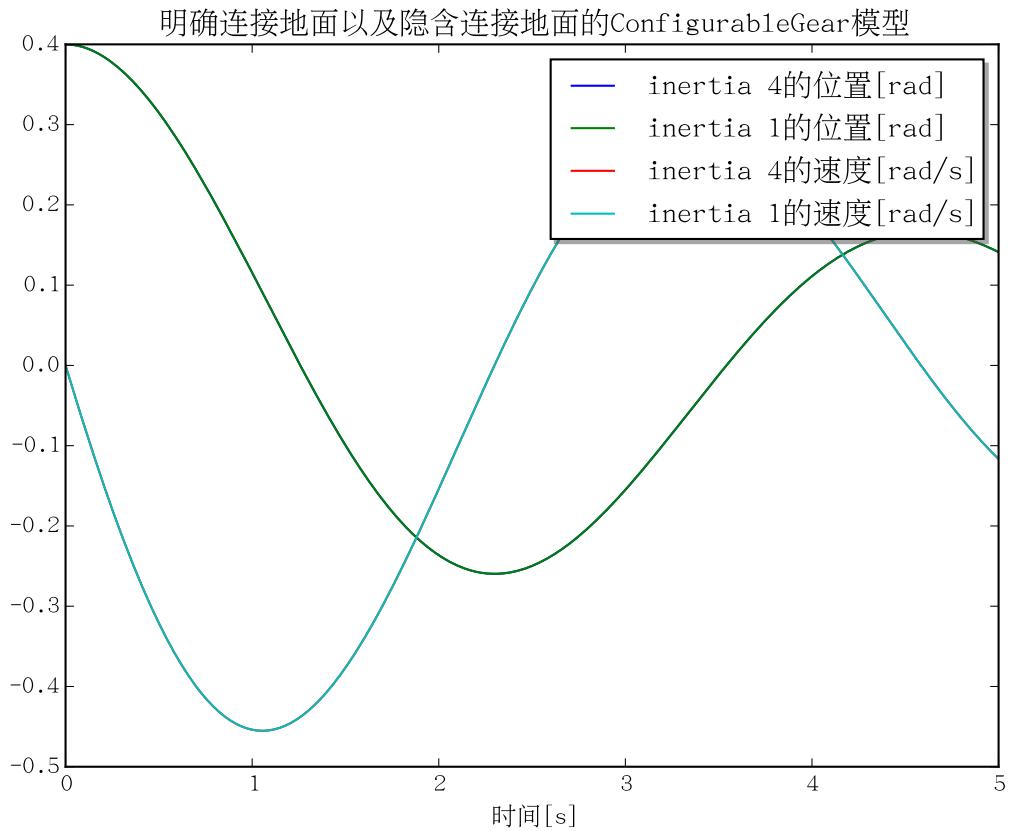
量	grounded==true	grounded==false
flow 变量总数	2	3
变量数	1 ( $housing\_phi$ )	1 ( $housing\_phi$ )
所需方程数	3	4
声明内的方程数	0	2 ( $housing$ 内)
equation 区域的方程数	3	2
提供的方程数	3	4

当使用条件声明时，非常重要的一点是确保在所有情况下，提供的方程数和所需方程数相等。在这里，我们只有两个条件需要考虑。而我们可以清楚地从该表中看到，两种情况下要求均满足。

下面的模型展示了我们如何将 ConfigurableGear 模型用在隐式、显式接地齿轮的情况：



而且，如我们所料，inertia1 和 inertia4 的响应相同：



### 第 7.1.5 节 再探猎食者猎物模型

在本节中，我们将重温第一章的猎食者猎物系统（18）。不过，这一次我们将利用单独的组件创建系统模型。在重复第一章得到的行为后，我们将扩大考虑的范围。我们会重新配置这些组件模型，以创建系统模型去演示不同的动态等。

#### 经典猎食者猎物

我们首先观察经典的猎食者猎物系统。为了使用组件模型来建立这样一个系统，我们将需要不同模型。这包括兔子和狐狸的人口模型，以及代表繁殖、饥饿和捕食的模型。

#### 连接器

但是，我们在对连接器（159）的讨论中了解到了一点。在开始建立组件模型前，我们必须先定义相互作用的部件间所交换的信息。要做到这点，我们要定义连接器。在本节我们会使用的 connector 是 Species 连接器。它的定义如下：

```
within ModelicaByExample.Components.LotkaVolterra.Interfaces;
connector Species "Used to represent the population of a specific species"
  Real population "Animal population";
  flow Real rate "Flows that affect animal population";
end Species;
```

此连接器的定义很有趣。因为这些定义并非来自工程领域。相反，这些定义其实来源于生态学。在这种情况下，我们的横跨变量是 population。这代表特定物种动物的实际数量。flow 限定词所标示的通过变量是 rate。此变量表示，连接器所在部件的新动物“输入”率。

## 区域种群大小

要追踪一个地区内一个特定物种的数目，我们将使用 RegionalPopulation 模型。这个模型有几个值得注意的方面。所以我们会分步介绍模型。首先：

```
within ModelicaByExample.Components.LotkaVolterra.Components;
model RegionalPopulation "Population of animals in a specific region"
  encapsulated type InitializationOptions = enumeration(
    Free "No initial conditions",
    FixedPopulation "Specify initial population",
    SteadyState "Population initially in steady state");
```

The first two lines are as expected. 前两行没有意外。但此后，我们看到模型定义了一个名为 InitializationOptions 的类型。类型定义前添加了 encapsulated 关键字。这是必要的。因为该类型是在模型而不是包内定义的。Modelica 有这样的一条规则。如果我们想从 model 定义外引用此类型，类型定义前必须添加 encapsulated。从定义可以看到，enumeration 定义了三个不同的值：Free、FixedPopulation 以及 SteadyState。我们将很快看到如何使用这些值。

RegionalPopulation 模型的第一个声明为：

```
parameter InitializationOptions init=InitializationOptions.Free
annotation(choicesAllMatching=true);
```

需要注意的是 init，也就是第一个 parameter，利用了 InitializationOptions 枚举值。不但以此指定了其类型（为枚举值），而且也以此指定了起始值 Free。还要注意这里有 choicesAllMatching 标注。我们会在本章的后面回顾概念时以及随后的章节里更多地讨论 annotation。

紧接的声明是：

```
parameter Real initial_population
annotation(Dialog(group="Initialization",
enable=init==InitializationOptions.FixedPopulation));
```

initial\_population 参数是用来表示在该区域在仿真开始时种群大小的初始值。然而，我们在不久后就会在等式区域里看到，只有在 init 值设为 FixedPopulation 时，模型才会使用上述值。出于这个原因，parameter 的 enable 标注设定为 init==InitializationOptions.FixedPopulation。这个标注把上述关系告知了 Modelica 工具。在建立与该模型相关联的图形用户界面（例如参数的对话框）时，工具就可以使用该信息了。

另外值得注意的一点是 initial\_population 定义内的 Dialog 标注。该标注允许模型开发者把参数组织成不同的类别。在这种情况下参数在“初始化”类别里。Modelica 工具通常会使用这些信息去帮助组织参数对话框。

模型的最后一次公有声明为 connector 实例。该连接器可以让组件与外界其他组件进行相互作用：

```
Interfaces.Species species
annotation (Placement(transformation(extent={{-10,90},{10,110}}),
iconTransformation(extent={{-10,90},{10,110}})));
```

这里我们再次看到了组件位置 (245) 标注。不过，我们还是暂时不对其进行讨论。这样就剩下最后一条声明。这条声明恰好为 protected：

```
protected
  Real population(start=10) = species.population "Population in this region";
```

这个变量表示在这个地区的动物总数。变量的 start 值非零。这样做是为了避免前面对猎食者猎物系统稳定状态初始化 (20) 讨论里所看到的平凡解。从这个声明我们也可以看到，该声明令局部变量 population 与 species 连接器内横跨变量 species.population 的值相等。结果，population 变量其实是 species.population 表达式的别名。

以上就是所有声明。现在，让我们来看看 RegionalPopulation 模型的相关公式：

```
initial equation
  if init==InitializationOptions.FixedPopulation then
```

```

population = initial_population;
elseif init==InitializationOptions.SteadyState then
  der(population) = 0;
else
end if;
equation
  der(population) = species.rate;
  assert(population>=0, "Population must be greater than zero");
end RegionalPopulation;

```

initial equation 展示了 init 参数值对组件行为的作用。在一些情况下, init 值等于在 InitializationOptions 枚举内的 FixedPopulation 值。此时模型会引入一个方程, 以指定在仿真开始时的 population 变量的值等于 initial\_population 参数。在另一些情况下, init 值等于 SteadyState 枚举值。这时模型引入等式, 以指定仿真开始时的种群大小变化速率必须为零。很显然, 如果 init 等于 Free (最后剩下的可能选择), 那模型就没有初始方程。

在 equation 区域, 我们看到 population 变化率等于 species 连接器内 flow 变量 species.rate 的值。再一次, 我们要记得变量的正负号约定。flow 变量正值意味着流入组件内。上述公式与正负号约定相一致 (即若 species.rate 为正, 则会让区域内 population 增加)。

值得一提的最后一点是模型是在 equation 区域内的 assert 调用。这是用来定义模型的边界 (即令模型中的方程无效的点)。该语句用于实行区域内人口不能小于零这一约束。若某个解被发现违反约束, 模型就会产生 “人口必须大于零” 的错误信息。

这个模型在定义里也有相关的 Icon 标注。像往常一样, Icon 标注不包含在显示。但是, 当这个组件模型是在系统模型内渲染后, 其图标看起来就像这样:

## 繁殖

我们将研究的第一个真正的效应是繁殖。从前面的讨论我们知道, 由繁殖带来的种群数量增长正比于该区域中动物的数量。因此, 我们可以非常简洁地将繁殖描述为:

```

within ModelicaByExample.Components.LotkaVolterra.Components;
model Reproduction "Model of reproduction"
  extends Interfaces.SinkOrSource;
  parameter Real alpha "Birth rate proportionality constant";
equation
  growth = alpha*species.population "Growth is proportional to population";
end Reproduction;

```

其中 alpha 为比例常数。不过, 这个模型之所以简单明了, 主要要归功于其对 SinkOrSource 模型的继承。此前的 “DRY” 电气部件 (181) 也得益于对 TwoPin 模型的继承。因此在这点上, 两个继承的作用

别无二致。

SinkOrSource 模型任何增加或减少动物数的模型的基点。其定义如下：

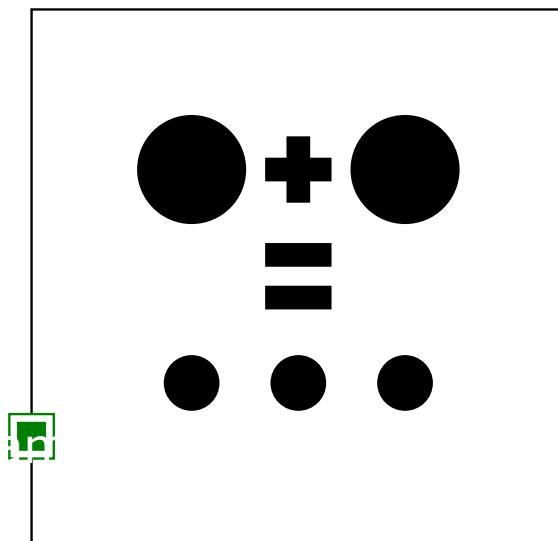
```
within ModelicaByExample.Components.LotkaVolterra.Interfaces;
partial model SinkOrSource "Used to describe single species effects"

Species species
  annotation (Placement(transformation(extent={{-10,90},{10,110}})));
protected
  Real growth "Growth in the population (if positive)";
  Real decline "Decline in the population (if positive)";
equation
  decline = -growth;
  species.rate = decline;
end SinkOrSource;
```

要理解这些方程，首先要理解一点。任何以 SinkOrSource 为基础 extends 的模型，通常都将连接到 RegionalPopulation 实例（但本身不会是一个 RegionalPopulation 模型）。这意味着，如果实例的 flow 变量 species.rate 为正时，模型将动物从 RegionalPopulation 模型内抽出来。通过观察 SinkOrSource 模型，我们可以看到变量 decline 只是 species.rate 的一个别名。换句话说，若 decline 为正那么 species.rate 也会为正。因此，任何与此 SinkOrSource 实例连接的 RegionalPopulation 其种群大小都会受到损失。相反，当 species.rate 为负时，growth 变量就会为正。在这种情况下，所连接的 RegionalPopulation 其种群大小将会增加。

通过定义并继承 SinkOrSource 模型可以隐藏大部分复杂性。这样一来，像 Reproduction 这样的模型可以将方程写得更直观，更好地反映模型的行为。例如：growth = alpha\*species.population。

虽然先前没有显示，Icon 为 Reproduction 模型如下所示：



饥饿

正如先前介绍的 Reproduction 模型一样，Starvation 模型也继承了 SinkOrSource 模型。不过，模型与 decline 变量有关的行为可以描述如下：

```
within ModelicaByExample.Components.LotkaVolterra.Components;
model Starvation "Model of starvation"
  extends Interfaces.SinkOrSource;
  parameter Real gamma "Starvation coefficient";
```

```

equation
  decline = gamma*species.population
  "Decline is proportional to population (competition)";
end Starvation;

```

## 猎食

建立经典猎食者猎物系统模型前，我们需要考虑最后一个效应是猎食模型。

回想此前的 SinkOrSource 模型以及与正负号约定相关的潜在问题等等的讨论。SinkOrSource 模型设计时仅仅考虑了与一个 RegionalPopulation 进行交互（因为模型只有一个 Species 连接器）。为了解决可能的正负号规定混淆问题，对于涉及两个不同地区种群之间相互效应，我们定义了下面的 partial 模型 Interaction:

```

within ModelicaByExample.Components.LotkaVolterra.Interfaces;
partial model Interaction "Used to describe interactions between two species"
  Species a "Species A"
    annotation (Placement(transformation(extent={{-110,-10},{-90,10}})));
  Species b "Species B"
    annotation (Placement(transformation(extent={{90,-10},{110,10}})));
protected
  Real a_growth "Growth in population of species A (if positive)";
  Real a_decline "Decline in population of species A (if positive)";
  Real b_growth "Growth in population of species B (if positive)";
  Real b_decline "Decline in population of species B (if positive)";
equation
  a_decline = -a_growth;
  a.rate = a_decline;
  b_decline = -b_growth;
  b.rate = b_decline;
end Interaction;

```

再一次，我们定义了有增长变量以及减少变量。但是这一次，有每个变量均有两个版本。其中一个与 a 连接器相关联，另一个则与 b 连接器相关联。

使用上述定义，我们很容易可以定义下面的 Predation:

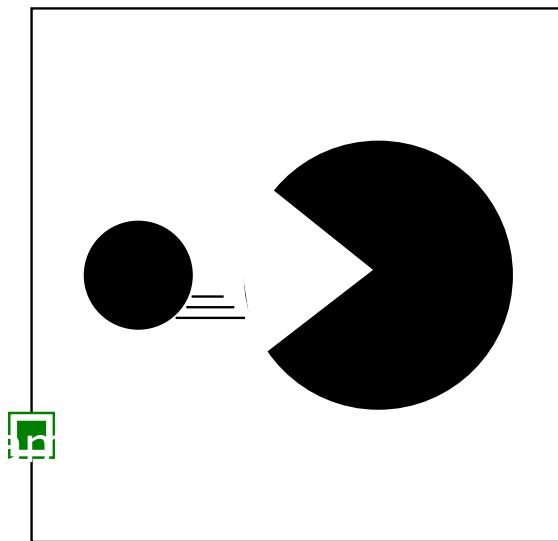
```

within ModelicaByExample.Components.LotkaVolterra.Components;
model Predation "Model of predation"
  extends Interfaces.Interaction;
  parameter Real beta "Prey (Species A) consumed";
  parameter Real delta "Predators (Species B) fed";
equation
  b_growth = delta*a.population*b.population;
  a_decline = beta*a.population*b.population;
end Predation;

```

此模型描述了，“B”（掠食者）种群大小增长与捕食者以及猎物种群大小两者乘积成正比。同样，“A”（猎物）种群大小的减少也与捕食者以及猎物的种群大小这两者乘积成正比（虽然使用的是另一个比例常数）。

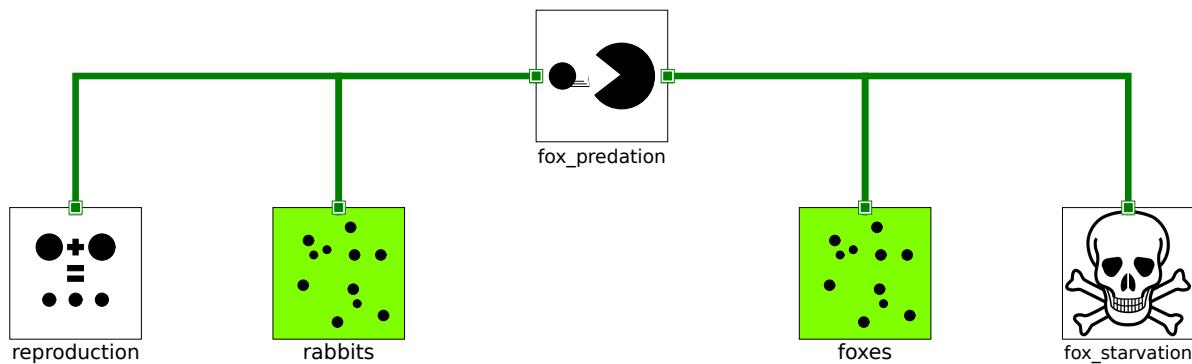
虽然先前没有显示，Predation 模型的 Icon 如下所示:



需要注意的是 Predation 模型并不对称的。b 连接器应连接到猎食者种群。而 a 连接器应连接到猎物种群。模型的图像以及不对称的图标也强调了这一点。

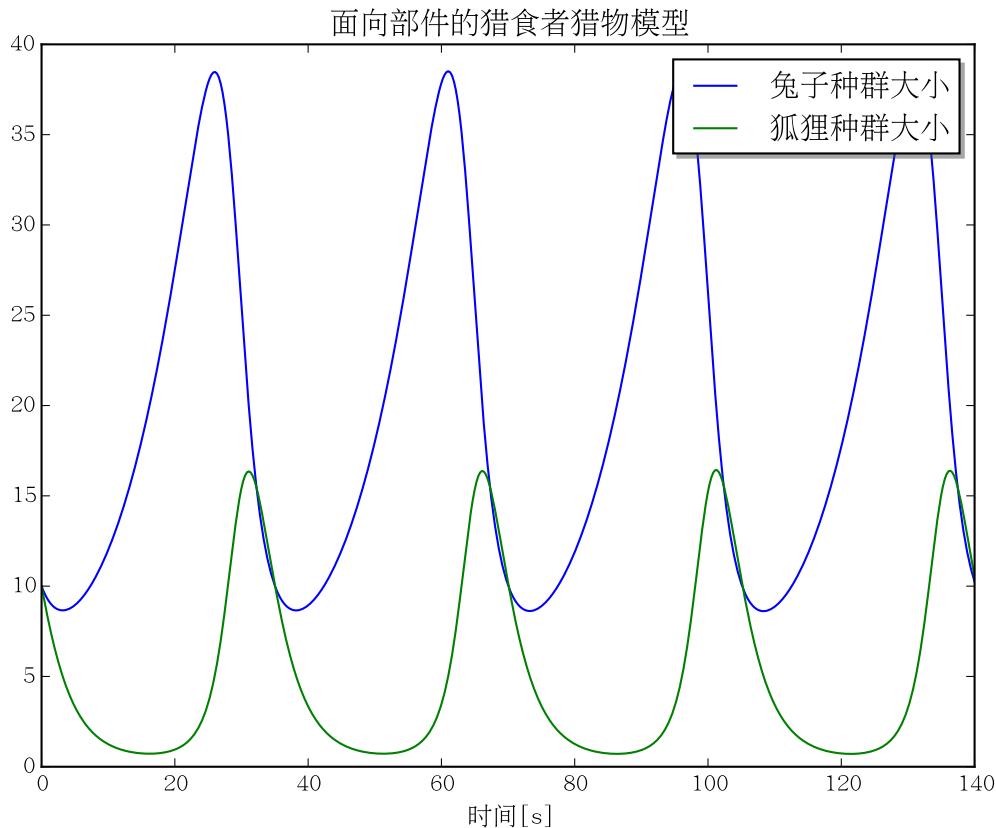
#### 经典系统模型

有了上述组件，我们就可以很容易地构建面向组件版本的经典猎食者猎物系统。通过拖放组件，可以得到以下系统配置：



这里我们看到 Starvation 模型连接到 foxes 种群上。而 Reproduction 模型则连接到了 rabbits 种群。Predation 模型连接则同时连接到两个种群上。a (猎物) 连接器连接到 rabbits 处。而 b (捕食) 连接器则连接到 foxes 处。

我们可以从下面的图中看出，这个系统的行为等同于之前讨论的猎食者猎物系统 (18) 的结果：



### 引入第三个物种

我们将一而再地看到，建立组件模型对比直接写出其封装的公式而言，需要一定的初始投入。但这个过程也有显著的“回报”。因为作为结果，我们就可以基于概要地去组合系统。例如对于猎食者猎物系统，我们可以在经典的猎食者猎物系统添加第三个物种狼。

#### 引入狼群

要建立带有第三个物种模型，并不需要定义任何其他组件模型。相反，我们不但可以重用现有的 Predation、Starvation、RegionalPopulation 模型，我们还可以重用 ClassicLotkaVolterra 模型本身：

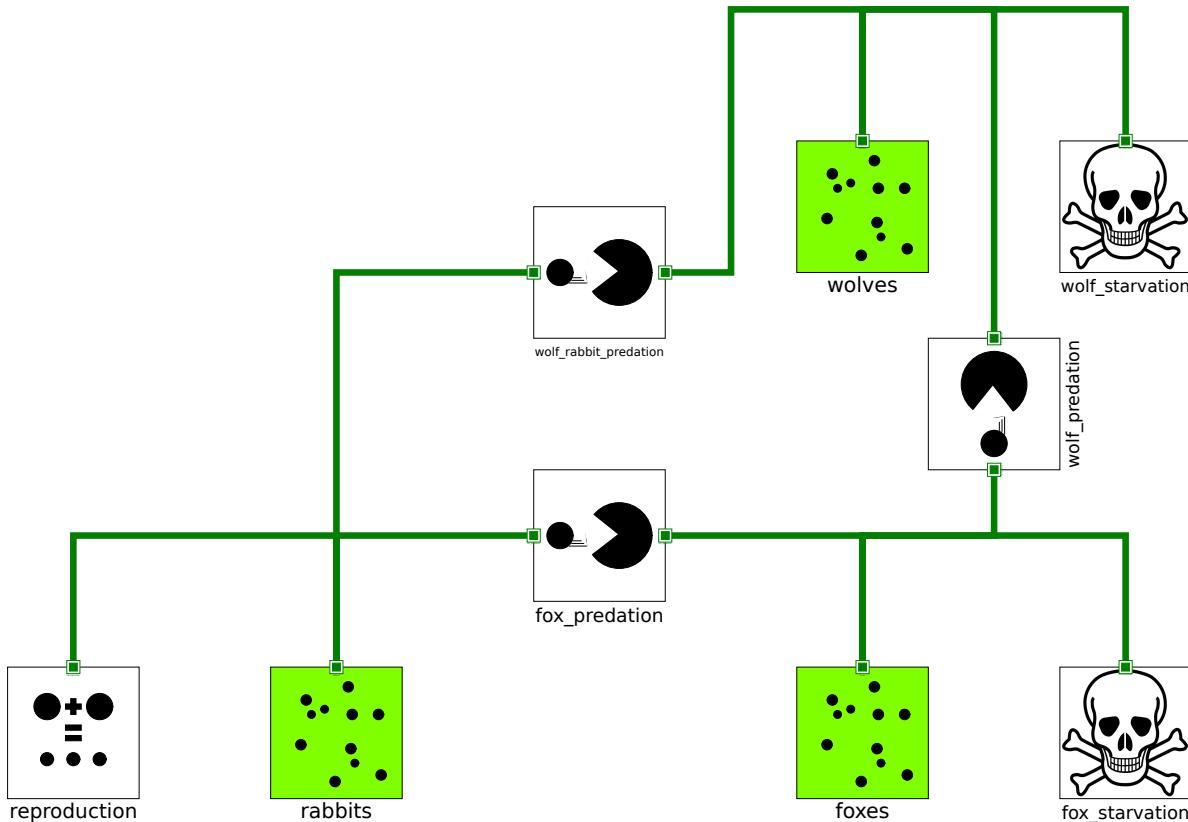
```
within ModelicaByExample.Components.LotkaVolterra.Examples;
model ThirdSpecies "Adding a third species to Lotka-Volterra"
  import ModelicaByExample.Components.LotkaVolterra.Components.RegionalPopulation.InitializationOptions.FixedPopulation;
  extends ClassicLotkaVolterra(rabbits(initial_population=25), foxes(initial_population=2));
  Components.RegionalPopulation wolves(init=FixedPopulation, initial_population=4)
    annotation (Placement(transformation(extent={{30,40},{50,60}})));
  Components.Starvation wolf_starvation(gamma=0.4)
    annotation (Placement(transformation(extent={{70,40},{90,60}})));
  Components.Predation wolf_predation(beta=0.04, delta=0.08) "Wolves eating Foxes"
    annotation (Placement(transformation(extent={{-10,-10},{10,10}}, rotation=90, origin={60,20})));
  Components.Predation wolf_rabbit_predation(beta=0.02, delta=0.01) "Wolves eating rabbits"
    annotation (Placement(transformation(extent={{-10,30},{10,50}}));
equation
  connect(wolf_predation.b, wolves.species) annotation (Line(
    points={{60,30},{60,80},{40,80},{40,60},{40,60}},
    color={0,127,0},
    smooth=Smooth.None));
  connect(wolf_rabbit_predation.a, rabbits.species) annotation (Line(
```

```

points={{-10,40},{-40,40},{-40,-20}},
color={0,127,0},
smooth=Smooth.None));
connect(wolf_predation.a, foxes.species) annotation (Line(
points={{60,10},{60,0},{40,0},{40,-20}},
color={0,127,0},
smooth=Smooth.None));
connect(wolf_starvation.species, wolves.species) annotation (Line(
points={{80,60},{80,80},{40,80},{40,60}},
color={0,127,0},
smooth=Smooth.None));
connect(wolves.species, wolf_rabbit_predation.b) annotation (Line(
points={{40,60},{40,80},{20,80},{20,40},{10,40}},
color={0,127,0},
smooth=Smooth.None));
annotation (experiment(StopTime=100, Tolerance=1e-006));
end ThirdSpecies;

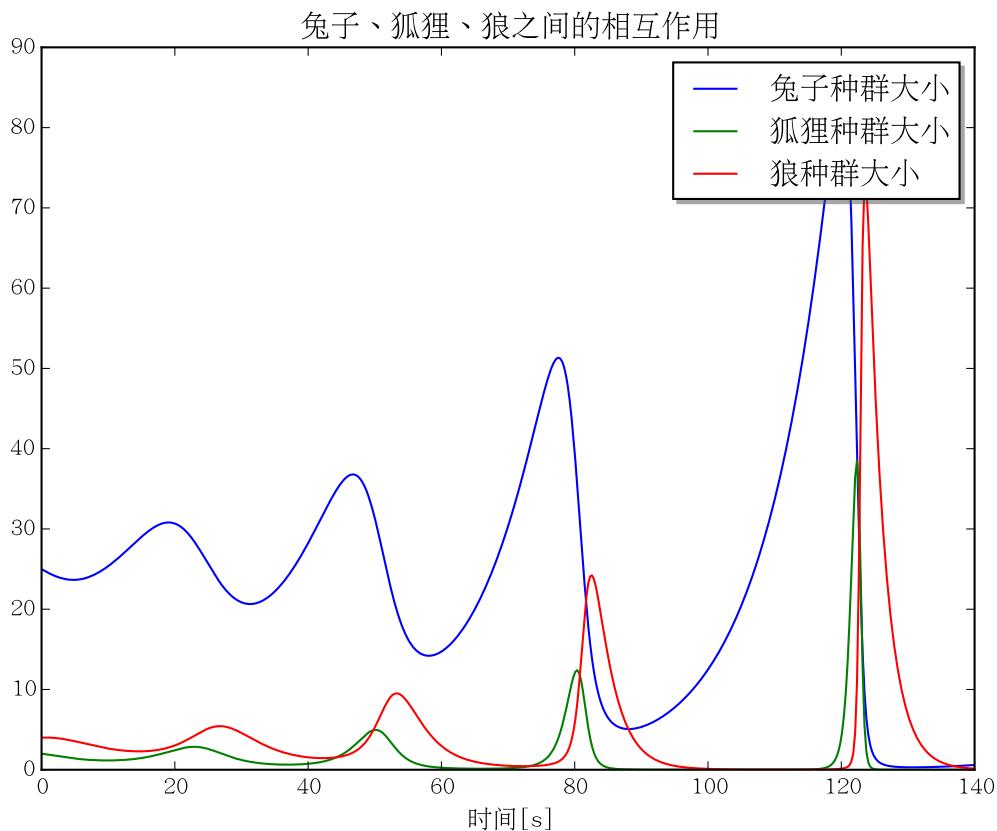
```

你通常不会被通过键入上述源代码去建立这样的模型。相反，在一个图形化开发环境里，通过扩大现有的 ClassicLotkaVolterra 模型去组装这样的系统，将需要不到一分钟。可视化后，生成系统的示意图为如下所示：



### 变更后的动态

通过建立这样一个模型，我们可以很容易观察经典模型与三种群模型在系统动力学之间的差异。下面的图显示了这三个物种是进行如何相互作用的：



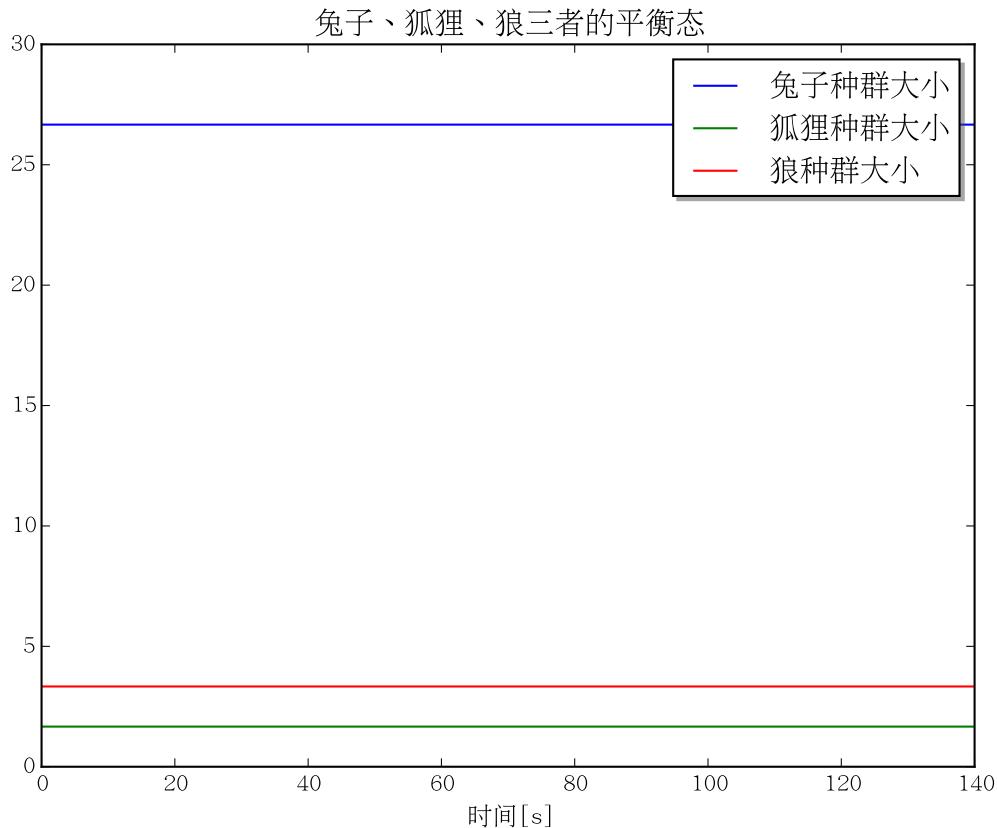
通过在不同的 RegionalPopulation 实例内的 init 参数，我们还可以快速创建一个模型来求解三个种群的在平衡态的大小。

```

within ModelicaByExample.Components.LotkaVolterra.Examples;
model ThreeSpecies_Quiescent "Three species in a quiescent state"
  import ModelicaByExample.Components.LotkaVolterra.Components.RegionalPopulation.InitializationOptions.SteadyState;
  extends ThirdSpecies(
    rabbits(init=SteadyState, population(start=30)),
    foxes(init=SteadyState, population(start=2)),
    wolves(init=SteadyState, population(start=4)));
  annotation (experiment(StopTime=100, Tolerance=1e-006));
end ThreeSpecies_Quiescent;

```

需要做的仅仅是以 ThirdSpecies 模型为基础进行扩展，然后在包含的每个种群模型里修改 init 参数。对模型进行仿真就可以得到每个物种在平衡态的种群大小



从生态学的角度来看，我们已可以对这个系统做出一个有趣的观察。如果我们从一个非平衡条件启动仿真，系统将迅速变为不稳定。换句话说，引入狼群对原本稳定、只涉及兔子狐狸的生态系统内的种群动态产生了显著影响。

### 第 7.1.6 节 再探速度测量

回想一下我们前面对速度的测量 (59) 的讨论。讨论发生在我们引入构建可重用组件方法之前。这样一来，为不同的速度测量技术相关添加方程颇为麻烦。

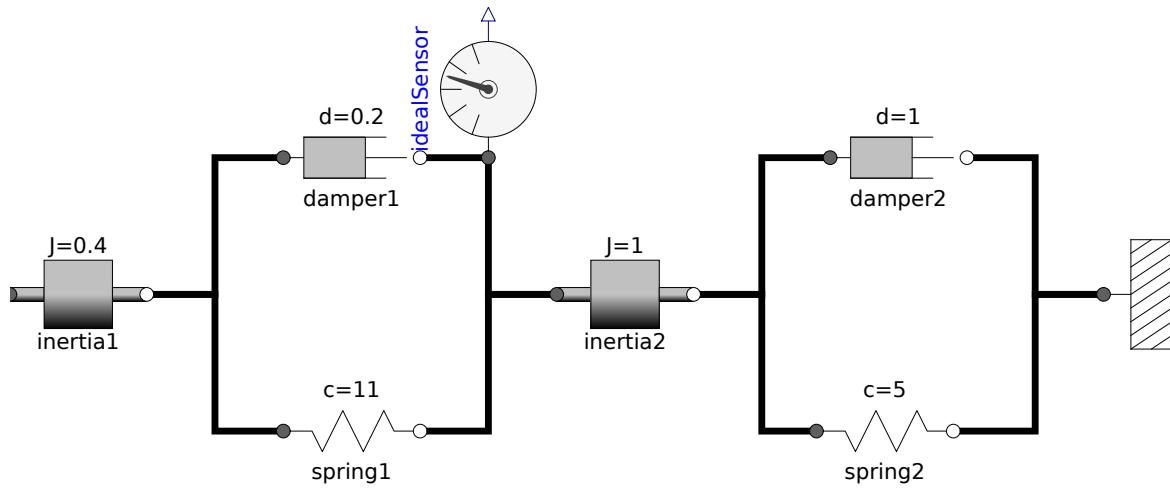
本节中，我们将重新审视该主题，并再次讨论所有不同的速度估测技术。和之前一样，我们假设“受控对象”模型（我们希望进行测速的系统）已经存在。但这一次，我们将为不同的速度估测算法创建可重用的组件模型，并将其以图形组件形式添加到受控模型里。

#### 受控对象模型

我们首先处理“受控对象模型”。模型行为和在此前讨论速度的测量 (59) 时的版本一致。系统示意图如下：

相应的 Modelica 模型源代码为：

```
within ModelicaByExample.Components.SpeedMeasurement.Examples;
model Plant "The basic plant model"
  extends ModelicaByExample.Components.Rotational.Examples.SMD;
  Components.IdealSensor idealSensor
  annotation (Placement(transformation(extent={{-10,-10},{10,10}}),
    rotation=90, origin={-20,30}));
equation
  connect(idealSensor.flange, inertia1.flange_a) annotation (Line(
    points={{-20,20},{-20,0},{-10,0}}, color={0,0,0},
```

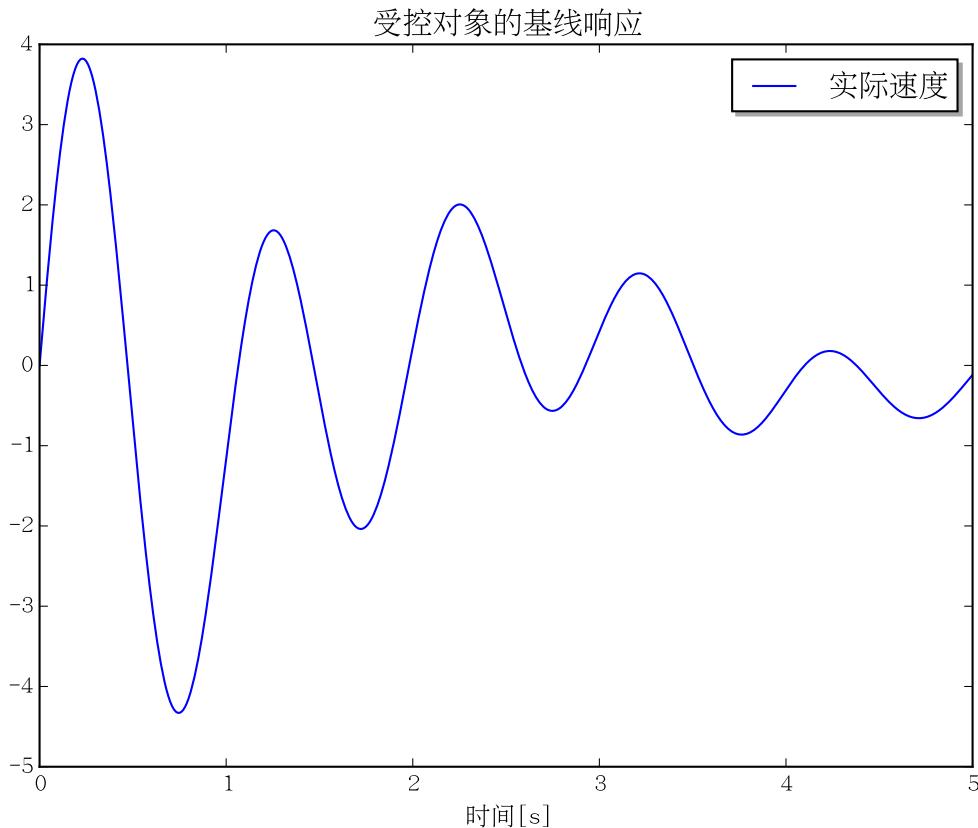


```

smooth=Smooth.None));
annotation(experiment(StopTime=10, Tolerance=1e-006));
end Plant;

```

在本节剩余部分，我们将创建使用不同估算方法的模型，对受控对象模型的 `inertia1` 组件进行测速。我们在观察不同的速度近似方法前，让我们先看看受控对象模型的实际响应速度。



请注意，该响应与我们首次讨论此话题时的一模一样。

### 取样保持传感器

此前，我们讨论了[取样保持 \(60\)](#) 的测速方法。在这里，我们将再次讨论这个话题。不过，我们会把速度估测封装为可重用的传感器模型。下面模型实现了速度测量的“采样保持”近似：

```
within ModelicaByExample.Components.SpeedMeasurement.Components;
model SampleHold "A sample-hold ideal speed sensor"
  extends Interfaces.SpeedSensor;
  parameter Modelica.SIunits.Time sample_rate;
initial equation
  w = der(flange.phi);
equation
  when sample(0, sample_rate) then
    w = der(flange.phi);
  end when;
end SampleHold;
```

在行为上，这里的估测方法和我们此前的[取样保持 \(60\)](#) 实现之间并没有差异。但是，我们这里的方法有所不同。不同在于，我们将估测技术放入可重用的组件模型里。

我们再次用技巧免去一些麻烦。这里，我们通过利用 partial 模型来表示各种传感器模型内的通用代码。我们可以从 SpeedSensor 模型的定义里发现：

```
within ModelicaByExample.Components.SpeedMeasurement.Interfaces;
partial model SpeedSensor "The base class for all of our sensor models"
  extends Modelica.Mechanics.Rotational.Interfaces.PartialAbsoluteSensor;
  Modelica.Blocks.Interfaces.RealOutput w "Sensed speed"
  annotation (Placement(transformation(extent={{100,-10},{120,10}})));
end SpeedSensor;
```

我们可以从 SpeedSensor 模型中看到，输出信号命名为 w。但是，我们也看到 SpeedSensor 是继承自 Modelica 标准库的另一个模型 PartialAbsoluteSensor。PartialAbsoluteSensor 模型定义为：

```
partial model PartialAbsoluteSensor
  "Partial model to measure a single absolute flange variable"
  extends Modelica.Icons.RotationalSensor;

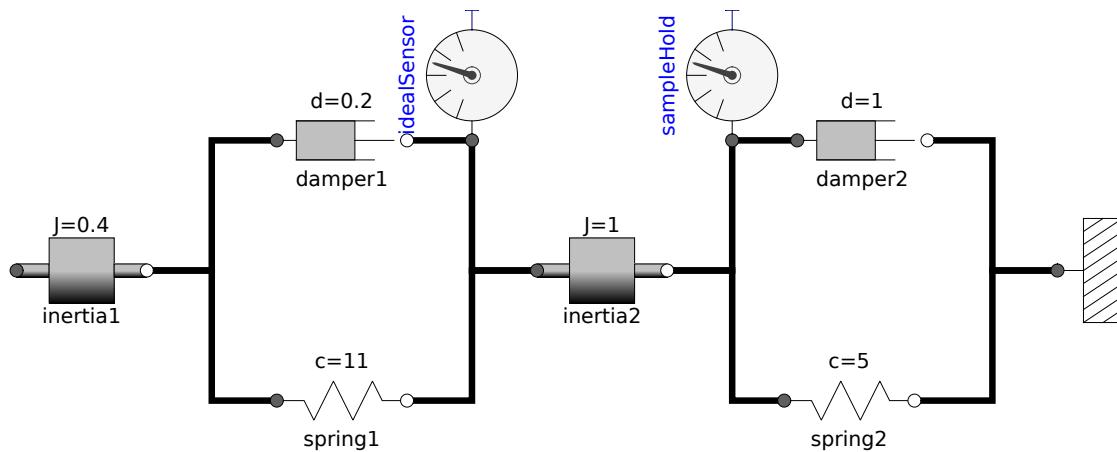
  Flange_a flange "Flange from which speed will be measured"
  annotation(Placement(transformation(extent={{-110,-10},{-90,10}}, rotation=0)));
equation
  0 = flange.tau;
end PartialAbsoluteSensor;
```

除了提供一个很好的图标外，PartialAbsoluteSensor 模型还带有旋转连接器 flange。此外，该模型假定传感器模型完全是被动的（即传感器对其所测系统无影响）。因此，模型对连接点施加零转矩。

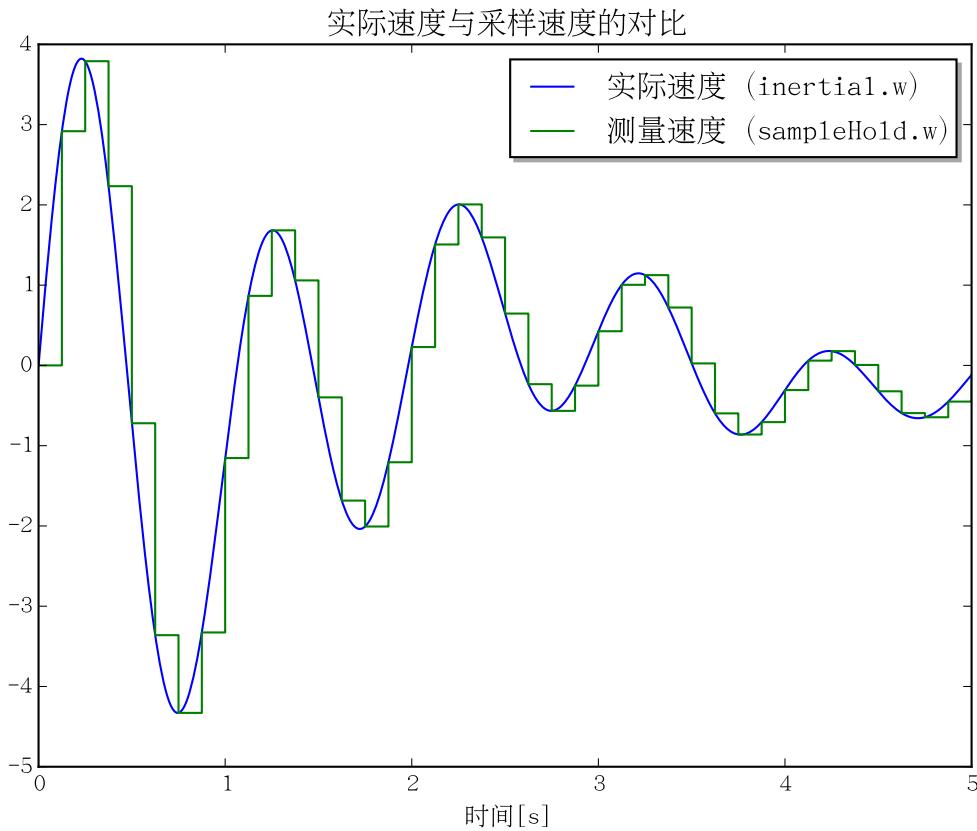
为了测试这个模型，我们只是从 Plant 模型进行扩展，加上 SampleHold 传感器的实例，并将传感器连接到 inertia 上，如：

```
within ModelicaByExample.Components.SpeedMeasurement.Examples;
model PlantWithSampleHold "Comparison between ideal and sample-hold sensor"
  extends Plant;
  Components.SampleHold sampleHold(sample_rate=0.125)
    annotation (Placement(transformation(extent={{-10,-10},{10,10}},
                                         rotation=90, origin={20,30})));
  equation
    connect(sampleHold.flange, inertia1.flange_b) annotation (Line(
      points={{20,20},{20,0},{10,0}}, color={0,0,0},
      smooth=Smooth.None));
    annotation (experiment(StopTime=10, Tolerance=1e-006));
end PlantWithSampleHold;
```

组装后，我们最终的系统看起来如下：



如果我们对此系统仿真 5 秒钟，我们可以比较传感器返回的信号与惯性元件的实际速度：



这些结果与我们此前在对取样保持 (60) 方法的讨论里的相一致。

### 间隔测量

现在让我们转而关注间隔测量 (61) 方法。同样，我们会从 Sensor 模型扩展创建出一个可重用的组件模型。这次，传感器实现将使用齿间的时间来估算速度：

```
within ModelicaByExample.Components.SpeedMeasurement.Components;
model IntervalMeasure
  "Estimate speed by measuring interval between encoder teeth"
  extends Interfaces.SpeedSensor;
  parameter Integer teeth;
  Real next_phi, prev_phi;
  Real last_time;
protected
  parameter Modelica.SIunits.Angle tooth_angle=2*Modelica.Constants.pi/teeth;
initial equation
  next_phi = flange.phi+tooth_angle;
  prev_phi = flange.phi-tooth_angle;
  last_time = time;
equation
  when {flange.phi>=pre(next_phi),flange.phi<=pre(prev_phi)} then
    w = tooth_angle/(time-pre(last_time));
    next_phi = flange.phi+tooth_angle;
    prev_phi = flange.phi-tooth_angle;
    last_time = time;
  end when;
end IntervalMeasure;
```

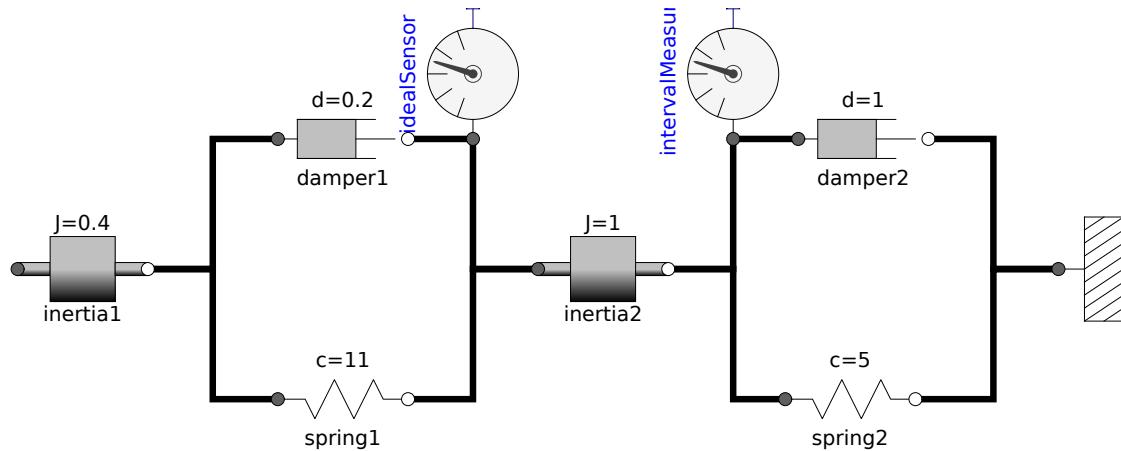
向受控对象模型添加上述传感器很容易。只要创建以下的 Modelica 模型：

```

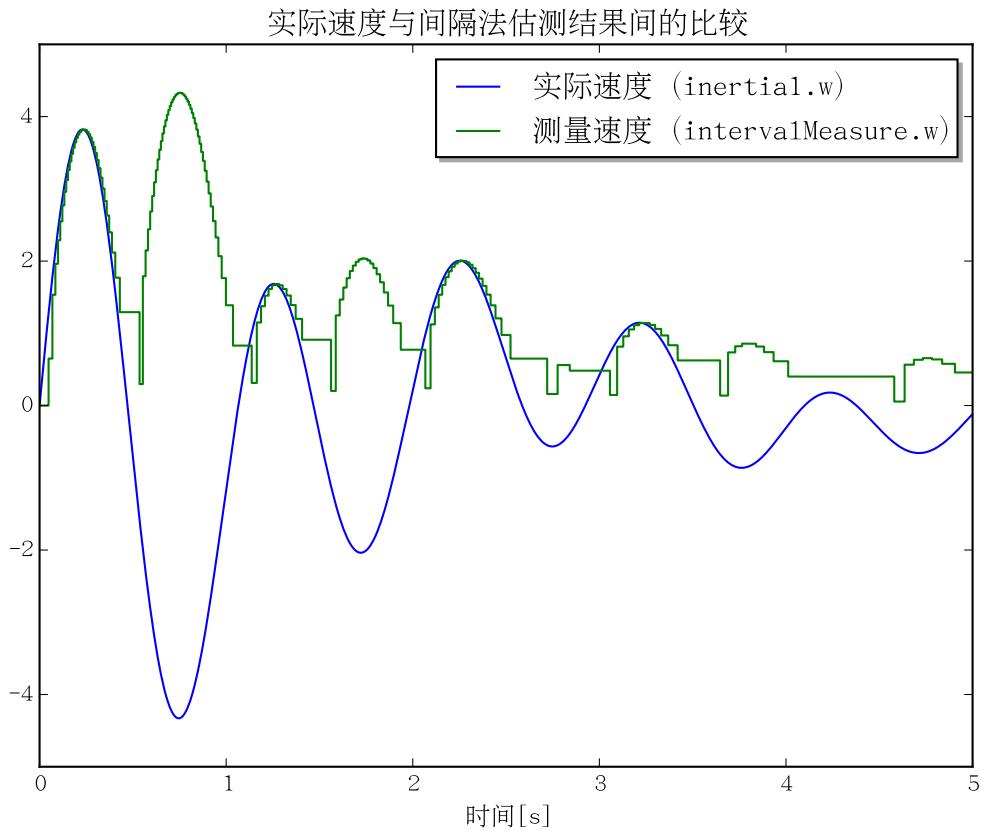
within ModelicaByExample.Components.SpeedMeasurement.Examples;
model PlantWithIntervalMeasure
  "Comparison between ideal and an interval measuring sensor"
  extends Plant;
  Components.IntervalMeasure intervalMeasure(teeth=200)
    annotation (Placement(transformation(
      extent={{-10,-10},{10,10}}, rotation=90,
      origin={20,30)}));
  equation
    connect(intervalMeasure.flange, inertia1.flange_b) annotation (Line(
      points={{20,20},{20,0},{10,0}}, color={0,0,0},
      smooth=Smooth.None));
    annotation (experiment(StopTime=10, Tolerance=1e-006));
  end PlantWithIntervalMeasure;

```

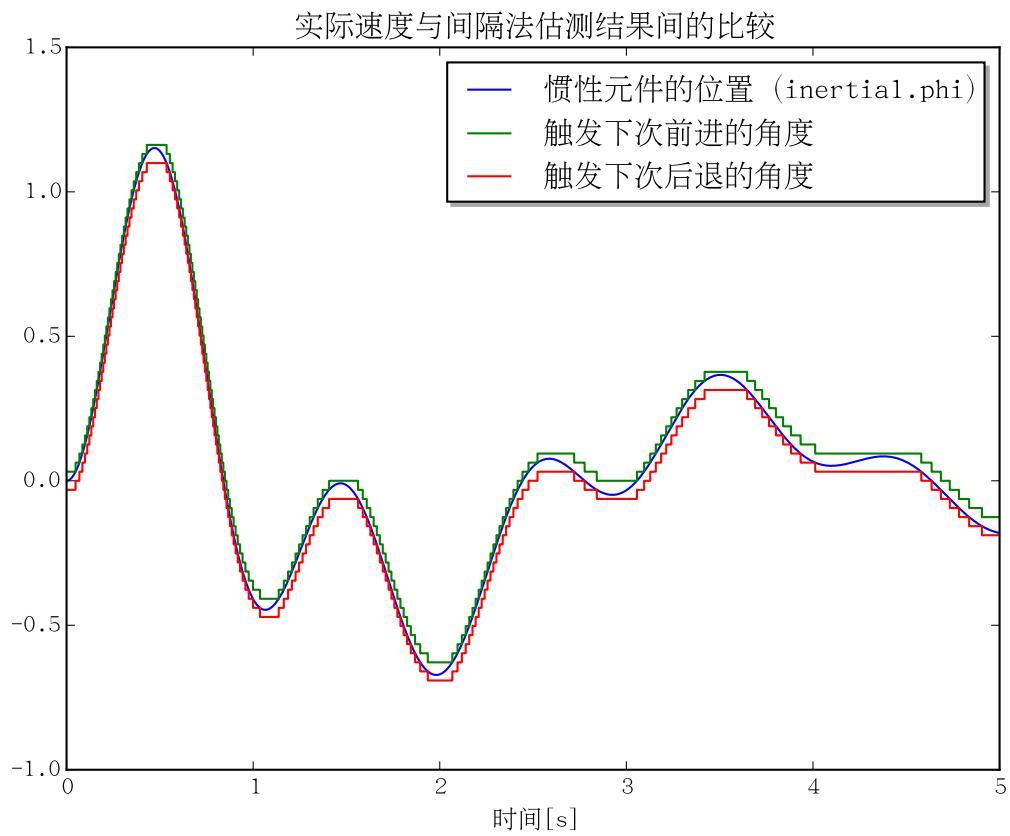
组装后，我们的系统看起来如下：



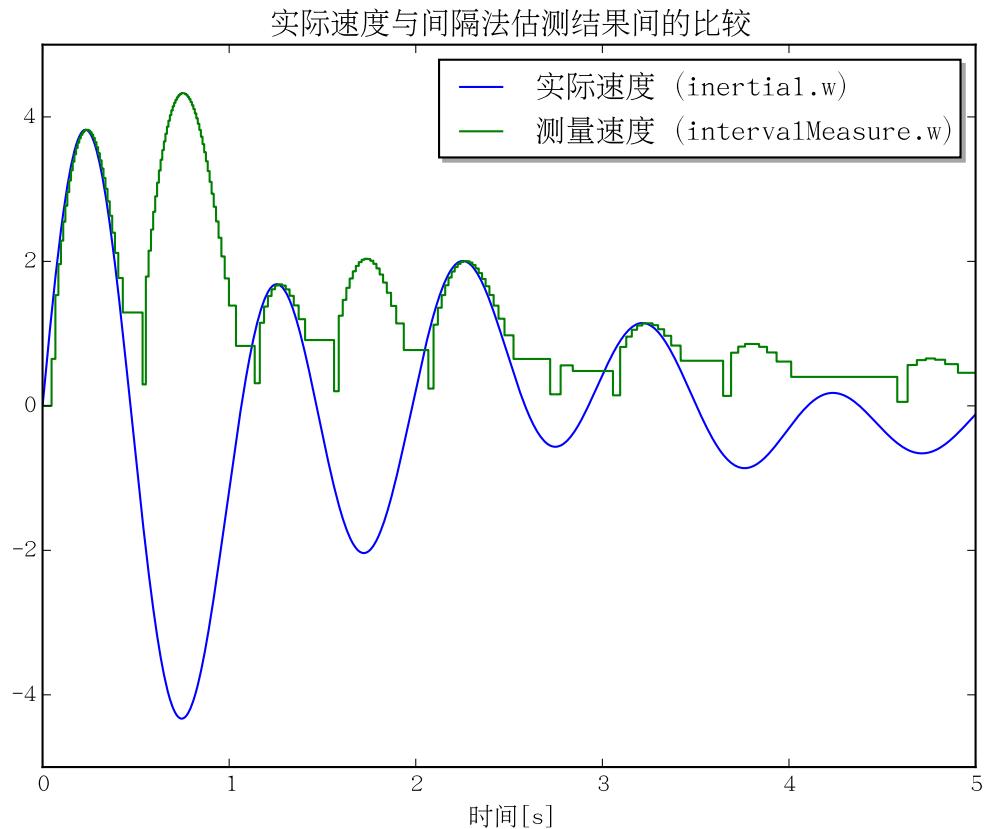
对上述系统进行仿真，我们得到估测速度的如下结果：



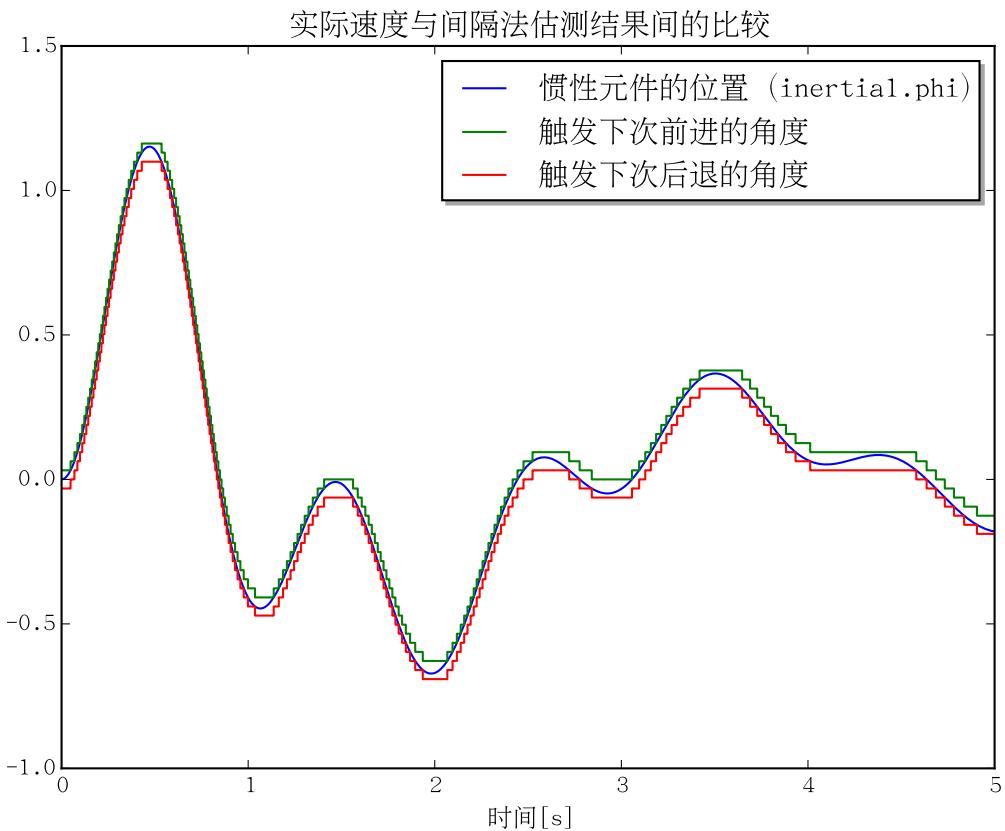
正如我们此前在对[间隔测量 \( 61 \)](#)方法中的讨论所看到的一样，估测信号的质量随着齿数减少会严重降低。上图使用的传感器每圈有 200 齿。如果我们绘制轴的转角以及与其相邻的齿角度，我们会看到，轴在不触发测量的前提下不能移动很远：



另一方面，如果将每圈的齿数降到 20，我们会得到以下的结果：



绘制轴的转角以及与其相邻的齿角度，我们会看到测量触发的次数变得更少了。其结果是测量的准确度大幅度降低：



上述结果与此前对 [间隔测量 \(61\)](#) 的讨论内的结果相一致。因此，我们同样可以验证这些面向组件传感器实现的正确性。

## 脉冲计数器

最后是脉冲计数 (65) 方法。同样，我们可将上述估测方法展现为扩展自 Sensor 模型的可重用组件。

```
within ModelicaByExample.Components.SpeedMeasurement.Components;
model PulseCounter "Compute speed using pulse counting"
  extends Interfaces.SpeedSensor;
  parameter Modelica.SIunits.Time sample_time;
  parameter Integer teeth;
  Modelica.SIunits.Angle next_phi, prev_phi;
  Integer count;
protected
  parameter Modelica.SIunits.Angle tooth_angle=2*Modelica.Constants.pi/teeth;
initial equation
  next_phi = flange.phi+tooth_angle;
  prev_phi = flange.phi-tooth_angle;
  count = 0;
algorithm
  when {flange.phi>=pre(next_phi), flange.phi<=pre(prev_phi)} then
    next_phi := flange.phi + tooth_angle;
    prev_phi := flange.phi - tooth_angle;
    count := pre(count) + 1;
  end when;
  when sample(0,sample_time) then
    w := pre(count)*tooth_angle/sample_time;
```

```

count := 0 "Another equation for count?";
end when;
end PulseCounter;

```

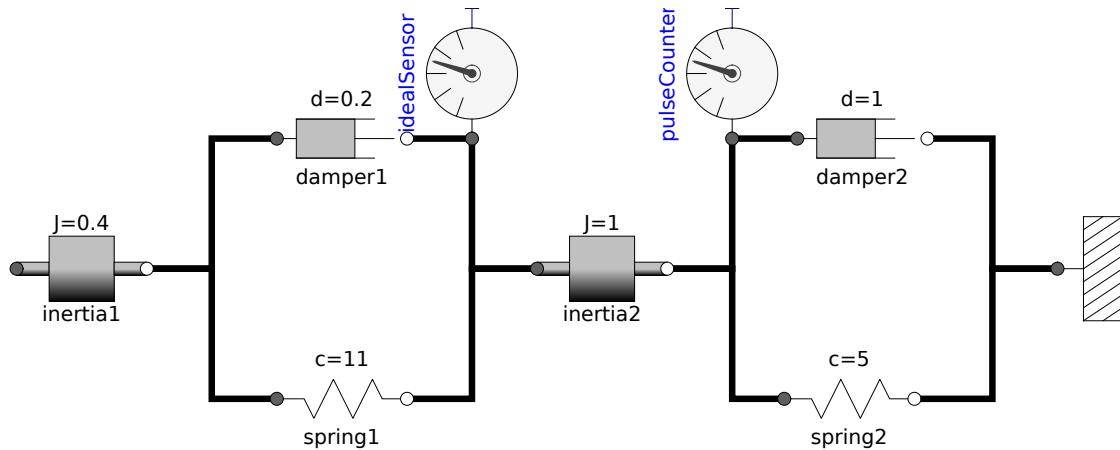
然后将其添加到我们的 Plant 总模型:

```

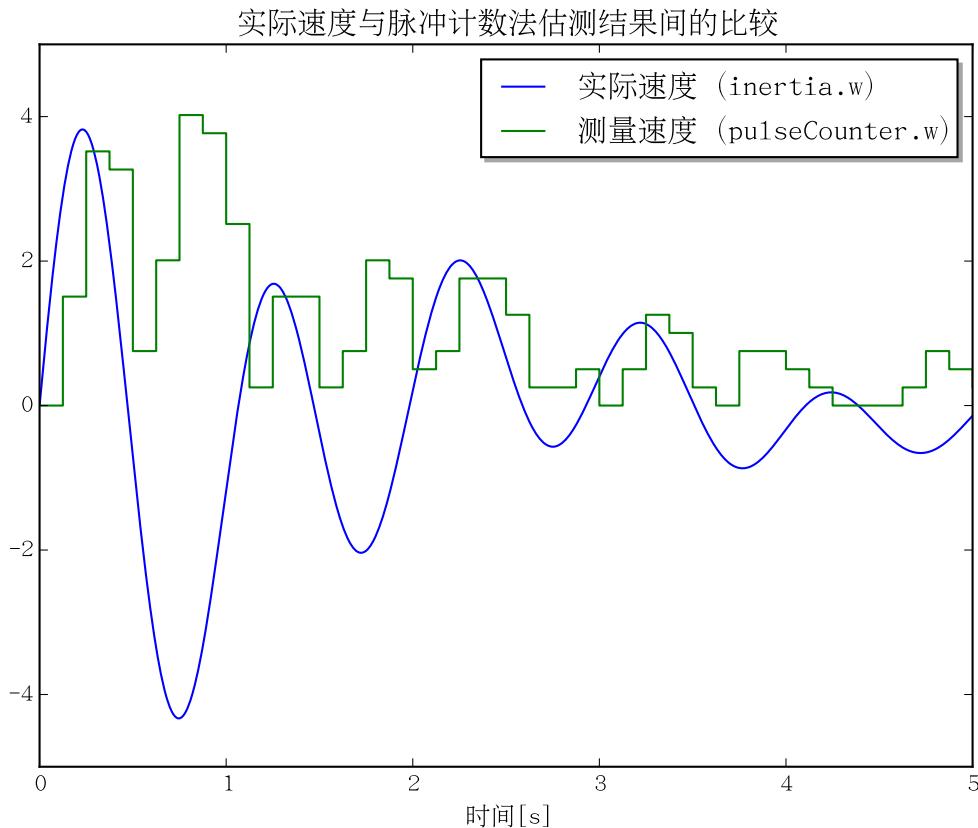
within ModelicaByExample.Components.SpeedMeasurement.Examples;
model PlantWithPulseCounter
  "Comparison between ideal and pulse counting sensor"
  extends Plant;
  Components.PulseCounter pulseCounter(sample_time=0.125, teeth=200)
    annotation (Placement(transformation(
      extent={{-10,-10},{10,10}}, rotation=90,
      origin={20,30})));
  equation
    connect(pulseCounter.flange, inertia1.flange_b) annotation (Line(
      points={{20,20},{20,0},{10,0}}, color={0,0,0},
      smooth=Smooth.None));
    annotation (experiment(StopTime=10, Tolerance=1e-006));
  end PlantWithPulseCounter;

```

生成的系统显示如下:



对上述系统进行仿真，我们看到结果与此前对脉冲计数 (65) 的讨论内的结果相一致:



## 结论

本书在此前对速度的测量 (59) 的讨论里非常详细地描述了用于估测旋转轴速度的各种测量技术。本节的目的不是为了重温这些讨论。实际上，我们要展示 Modelica 语言面向组件特性对此前提出的估测技术有着很大的助益。如我们所见，所有这些方法均可以很好地封装在组件模型内。其结果是，要使用这些不同的估测技术现在变得很容易。只需要将这些传感器模型拖放到系统内，并将其连接到要进行测速的旋转元素即可。

### 第 7.1.7 节 框图组件

到目前为止，本章的重点一直放在非因果建模里。但 Modelica 同时还支持因果形式的建模。我们强调非因果建模的主要原因在于，非因果方法非常适用于物理系统的建模。这使得组装物理系统可以使用示意图进行组合，开发者无须进行数学推导。组合后的系统会自动生成守恒方程，以确保模型运行正确。

框图则是另一种建模方法。框图需要重点创建代表大量数学运算的组件模型。然后，上述数学运算作用在（通常随时间变化）的信号，然后返回而其他信号。事实上，我们将在本节介绍一种特殊的 model。这种模型名为 block。block 仅有 input 和 output 的信号。

本节中，我们会先看看如何构建代表基本数学运算的因果模块。然后，我们会看到如何用两种不同的方式使用这些模块。第一种使用方式是对一个简单的物理系统进行建模。我们将讨论并对比因果方法以及非因果方法。第二种使用方式是将模块用于建模控制系统。如我们所见，使用模块来构建控制系统更符合框图模型的形式。

好在 Modelica 同时允许因果和非因果的方法。而我们会马上看到，两种建模方法也可以混合在一起。这样一来，Modelica 可以让模型开发者选择在特定情况下效果最好的方法。

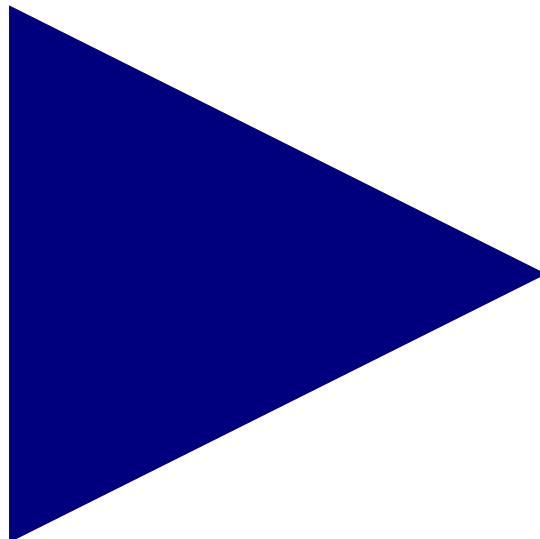
## 模块

### Modelica 标准库

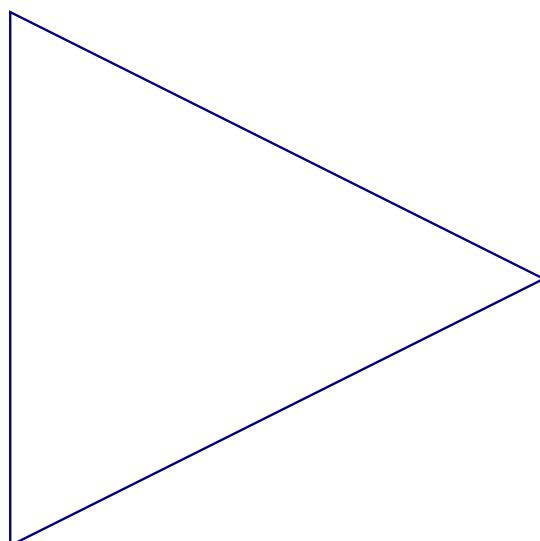
本节中，我们会充分利用 Modelica 标准库里的定义。首先是连接器的几个定义：

```
connector RealInput = input Real ""input Real" as connector";
connector RealOutput = output Real ""output Real" as connector";
```

如其名称所示，RealInput 和 RealOutput 分别代表实值的输入和输出信号连接器。在简图视图里，RealInput 连接器显示为蓝色的实心三角：



RealOutput 连接器是一个蓝色的三角形边框：



我们将充分利用 Modelica 标准库内的几个不同 partial 模块定义。第一个使用的 partial 定义是 SO，或者说“单输出”(single output) 的定义：

```
partial block SO "Single Output continuous control block"
  extends Modelica.Blocks.Icons.Block;

  RealOutput y "Connector of Real output signal" annotation (Placement(
    transformation(extent={{100,-10},{120,10}}, rotation=0)));
end SO;
```

显然，这个定义用于拥有单个输出的模块。按照惯例，该输出信号命名为  $y$ 。我们使用的另一个定义是 SISO，或者说“单输入、单输出”(single input, single output) 的模块：

```
partial block SISO "Single Input Single Output continuous control block"
  extends Modelica.Blocks.Icons.Block;

  RealInput u "Connector of Real input signal" annotation (Placement(
    transformation(extent={{-140,-20},{-100,20}}, rotation=0)));
  RealOutput y "Connector of Real output signal" annotation (Placement(
    transformation(extent={{100,-10},{120,10}}, rotation=0)));
end SISO;
```

此模型除了输出信号  $y$ ，还多了一个输入信号  $u$ 。最后，对于具有多个输入的模块，MISO 模块可以将输入信号  $u$  定义为向量：

```
partial block MISO "Multiple Input Single Output continuous control block"
  extends Modelica.Blocks.Icons.Block;
  parameter Integer nin=1 "Number of inputs";
  RealInput u[nin] "Connector of Real input signals" annotation (Placement(
    transformation(extent={{-140,-20},{-100,20}}, rotation=0)));
  RealOutput y "Connector of Real output signal" annotation (Placement(
    transformation(extent={{100,-10},{120,10}}, rotation=0)));
end MISO;
```

$nin$  参数用于指定模块输入的数目。

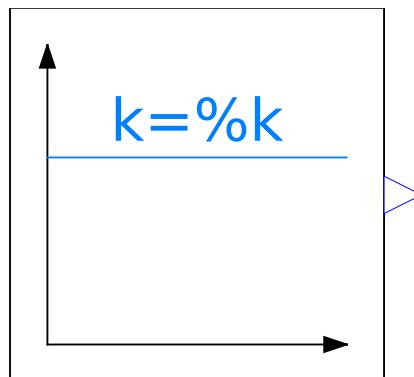
值得指出的是，我们即将定义的所有模块都可以在 Modelica 标准库中找到。但我们会在这里创建自己的版本。这样做的目的是去展示可以如何定义这样的模型。

### Constant

我们能想象的最简单的模块应该是简单的恒定值输出了。因为这种模型只有一个输出，模型扩展自 SO 模块

```
within ModelicaByExample.Components.BlockDiagrams.Components;
block Constant "A constant source"
  parameter Real k "Constant output value";
  extends Icons.Axes;
  extends Interfaces.SO;
equation
  y = k;
end Constant;
```

模块显示如下：

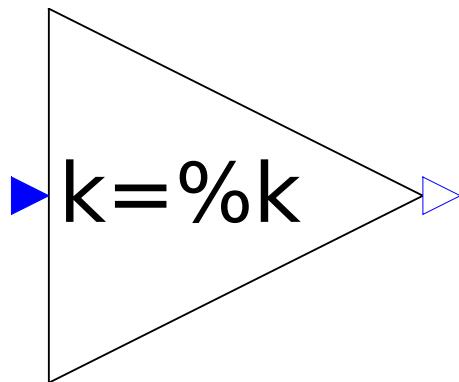


## Gain

另一个简单的 block 是“增益模块”。其输出信号为输入信号乘以一个常数的结果。这样的模块将有一个输入信号和一个输出信号。因此，它可从 SISO 模型扩展如下：

```
within ModelicaByExample.Components.BlockDiagrams.Components;
block Gain "A gain block model"
  extends Interfaces.SISO;
  parameter Real k "Gain coefficient";
equation
  y = k*u;
end Gain;
```

模块显示如下：



## Sum

Sum 模块可以使用任意数量的输入信号。此次，它扩展自 MISO 模块：

```
within ModelicaByExample.Components.BlockDiagrams.Components;
block Sum "Block that sums its inputs"
  extends Interfaces.MISO;
equation
  y = sum(u);
end Sum;
```

Sum 模块使用 `sum` ( 102) 函数来计算输入信号数组 `u` 的总和，以此得到输出信号 `y`。

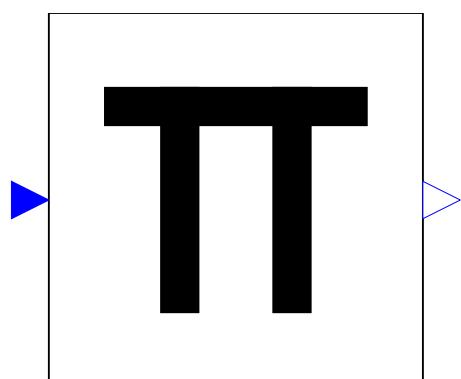
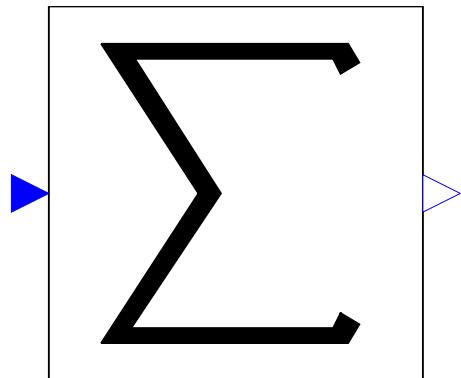
模块显示如下：

## Product

Product 模块与 Sum 模块几乎相同。不同点在于它采用了 `product` ( 103) 函数：

```
within ModelicaByExample.Components.BlockDiagrams.Components;
block Product "Block that outputs the product of its inputs"
  extends Interfaces.MISO;
equation
  y = product(u);
end Product;
```

模块显示如下：



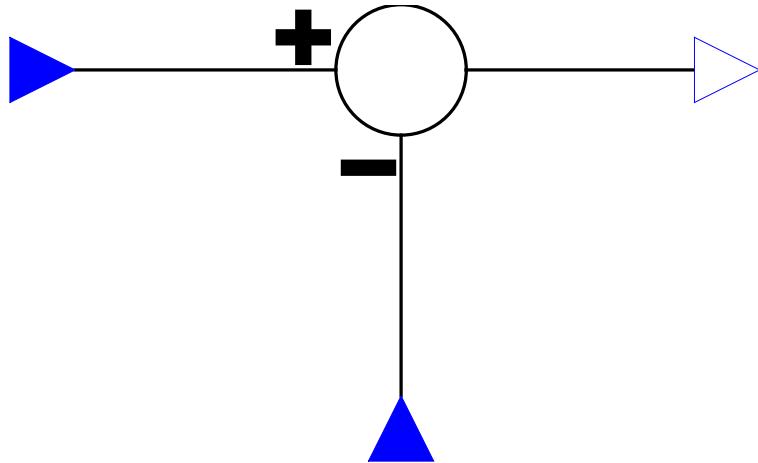
## Feedback

与先前模块不同，Feedback 模块并没有扩展自 Modelica 标准库中的任何定义。相反，该模块明确声明了它使用的所有连接器：

```
within ModelicaByExample.Components.BlockDiagrams.Components;
block Feedback "A block to compute feedback terms"
  Interfaces.RealInput u1
    annotation (Placement(transformation(extent={{-120,-10},{-100,10}})));
  Interfaces.RealInput u2 annotation (Placement(transformation(
    extent={{-10,-10},{10,10}},
    rotation=90,
    origin={0,-110})));
  Interfaces.RealOutput y
    annotation (Placement(transformation(extent={{90,-10},{110,10}})));
  equation
    y = u1-u2;
end Feedback;
```

Feedback 模块的输出是两个 input 信号 u1 和 u2 之间的差。

模块显示如下：

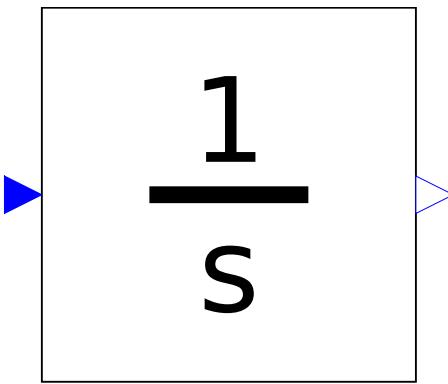


## Integrator

Integrator 模块是又一个 SISO 模块。模块对输入信号 u 进行积分，以此计算输出信号 y。由于此 block 执行积分，它需要一个初始条件。参数 y0 用以指定初始条件：

```
within ModelicaByExample.Components.BlockDiagrams.Components;
block Integrator
  "This block integrates the input signal to compute the output signal"
  parameter Real y0 "Initial condition";
  extends Interfaces.SISO;
  initial equation
    y = y0;
  equation
    der(y) = u;
  end Integrator;
```

模块显示如下：



## 系统

我们已经创建了一系列模块。现在我们将在一些例子里探讨，如何用上述模块来对系统进行建模。我们将看到，因果性的 block 组件的适用性因应用而异。

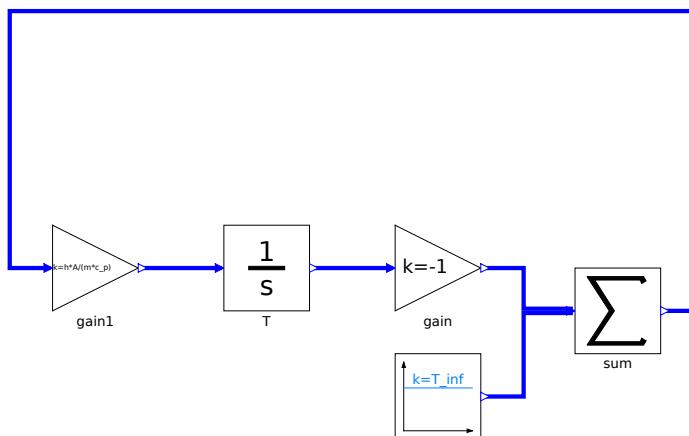
### 冷却示例

我们用 block 定义建模的首个系统是本章前面部分 (173) 提到的传热示例。不过，这次我们不会使用非因果组件来构建模型。相反，我们将使用与 block 定义相关的数学运算方面来进行建模。

由于这些模块均代表了数学运算，那让我们先重温这个例子相关的方程：

$$mc_p \dot{T} = hA(T_{\infty} - T)$$

以下框图可以解出温度曲线  $T$ ：



这个例子的 Modelica 的源代码为：

```
within ModelicaByExample.Components.BlockDiagrams.Examples;
model NewtonCooling "Newton cooling system modeled with blocks"
  import Modelica.SIunits.Conversions.from_degC;
  parameter Real h = 0.7 "Convection coefficient";
  parameter Real A = 1.0 "Area";
  parameter Real m = 0.1 "Thermal mass";
  parameter Real c_p = 1.2 "Specific heat";
  parameter Real T_inf = from_degC(25) "Ambient temperature";
  Components.Integrator T(y0=from_degC(90))
  annotation (Placement(transformation(extent={{-30,-10},{-10,10}})));

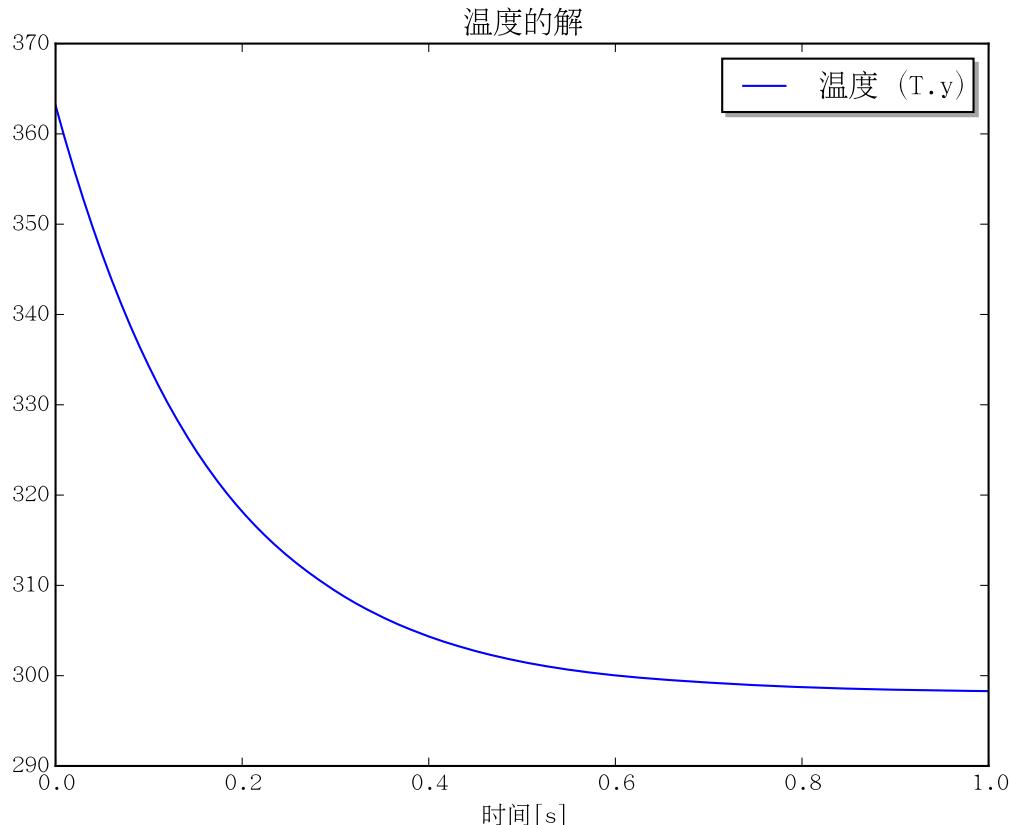
```

```

Components.Gain gain(k=-1)
annotation (Placement(transformation(extent={{10,-10},{30,10}})));
Components.Constant ambient(k=T_inf)
annotation (Placement(transformation(extent={{10,-40},{30,-20}})));
Components.Sum sum(nin=2)
annotation (Placement(transformation(extent={{52,-20},{72,0}})));
Components.Gain gain1(k=h*A/(m*c_p))
annotation (Placement(transformation(extent={{-70,-10},{-50,10}})));
equation
connect(T.y, gain.u) annotation (Line(
    points={{-9,0},{9,0}}, color={0,0,255},
    smooth=Smooth.None));
connect(sum.y, gain1.u) annotation (Line(
    points={{73,-10},{80,-10},{80,60},{-80,60},{-80,0},{-71,0}},
    color={0,0,255}, smooth=Smooth.None));
connect(gain.y, sum.u[2]) annotation (Line(
    points={{31,0},{40,0},{40,-9.5},{51,-9.5}},
    color={0,0,255}, smooth=Smooth.None));
connect(ambient.y, sum.u[1]) annotation (Line(
    points={{31,-30},{40,-30},{40,-10.5},{51,-10.5}},
    color={0,0,255}, smooth=Smooth.None));
connect(gain1.y, T.u) annotation (Line(
    points={{-49,0},{-31,0}},
    color={0,0,255}, smooth=Smooth.None));
end NewtonCooling;

```

温度  $T$  是在该模型中由变量  $T.y$  表示。对上述系统进行仿真，我们会得到温度的以下解：



我们可以看到，该解与该例此前所有形式内所得的解完全一样。

到目前为止，我们用三种方式表述了上述问题。第一种描述是使用了单个公式的数学结构。第二种方式是使用非因果的单个物理效应元件模型来表示相同的动力学。最后，我们有了上述最新的框图表述。但

真正的问题是，哪种方法最适合于这个特定问题？

实际上，需要考虑两种极端的情况。如果，我们仅仅想解决问题的这一个特定的配置。也就是说，我们仅仅考虑单热容与一些环境无限热容进行对流。这样的话，基于等式的版本（7）很可能是最好的选择。因为整个问题的行为，可以通过单一的公式来表示：

$$m * c_p * \text{der}(T) = h * A * (T_{\text{inf}} - T) \text{ "Newton's law of cooling";}$$

这样的公式可以非常迅速地键入。相比之下，基于组件的版本会要求用户拖拽、放入然后连接组件模型。这样总是会需要更长时间。

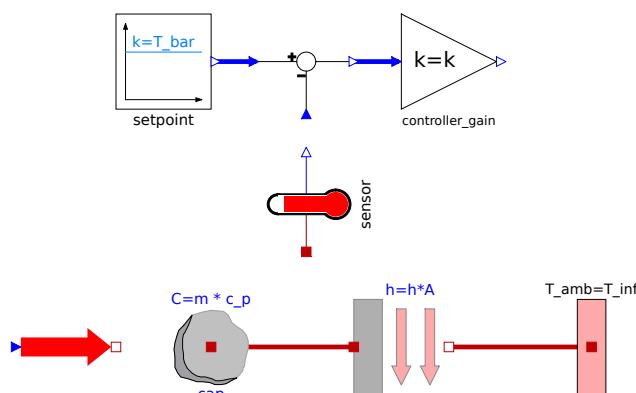
但是，如果你要创建问题的变种，例如结合不同模式的热传递、不同的边界条件等，那么非因果版本就更好了。这是因为虽然创建组件模型需要一些投入，但要重新配置部件模型就几乎毫不费力了。

有些人可能说框图版本的模型也是一样的（即可以毫不费力对其重新配置），但事实并非如此。框图版模型是问题的一个数学表述，而不是基于原理的表述。如果你要设定不同的边界条件、添加热容、加入不同模式的热传递等，那么更改原理图甚为简单。然而，对于框图表述，你将需要彻底重新制定框图。这是因为，所得的状态空间形式数学方程可能非常不同。非因果、以原理为基础的方法其中一大优势在于，Modelica 编译器会自动将教科书里的方程翻译为状态空间形式。这节省了模型开发要做的大量繁琐、耗时且容易出错的工作。这也正是何以非因果方法是首选方法。

### 热控制

在接下来的例子中，我们将两者在本节中开发的因果组件与本章前面开发的传热组件（173）这些非因果成分组合在一起。我们会看到这是一个强大的组合。因为这使我们能够用原理表示物理组件。但这同时可以让我们用数学表述控制策略。

以下是显示这两种方法可以如何组合的示意图：



在与控制系统一起建模的物理系统中，物理组件和效果将使用非因果表述。代表控制策略的组件则通常会使用因果表述。将两种方法连接在一起的是传感器和执行器。该传感器测量系统的某些方面（本例为温度）与执行器对系统施加一定的“影响”（这里为热通量）。

执行器的模型通常具有信号输入以及到物理系统的非因果连接（“影响”，如力或电流，将通过其应用在物理系统上）。传感器模型与之正正相反。这时，因果连接器为输出，而非因果连接器将被用于“感应”物理系统的某些方面（如电压、温度等）。

我们的示例模型可以用 Modelica 的表述为：

```
within ModelicaByExample.Components.BlockDiagrams.Examples;
model MultiDomainControl
  "Mixing thermal components with blocks for sensing, actuation and control"
  import Modelica.SIunits.Conversions.from_degC;
  parameter Real h = 0.7 "Convection coefficient";
```

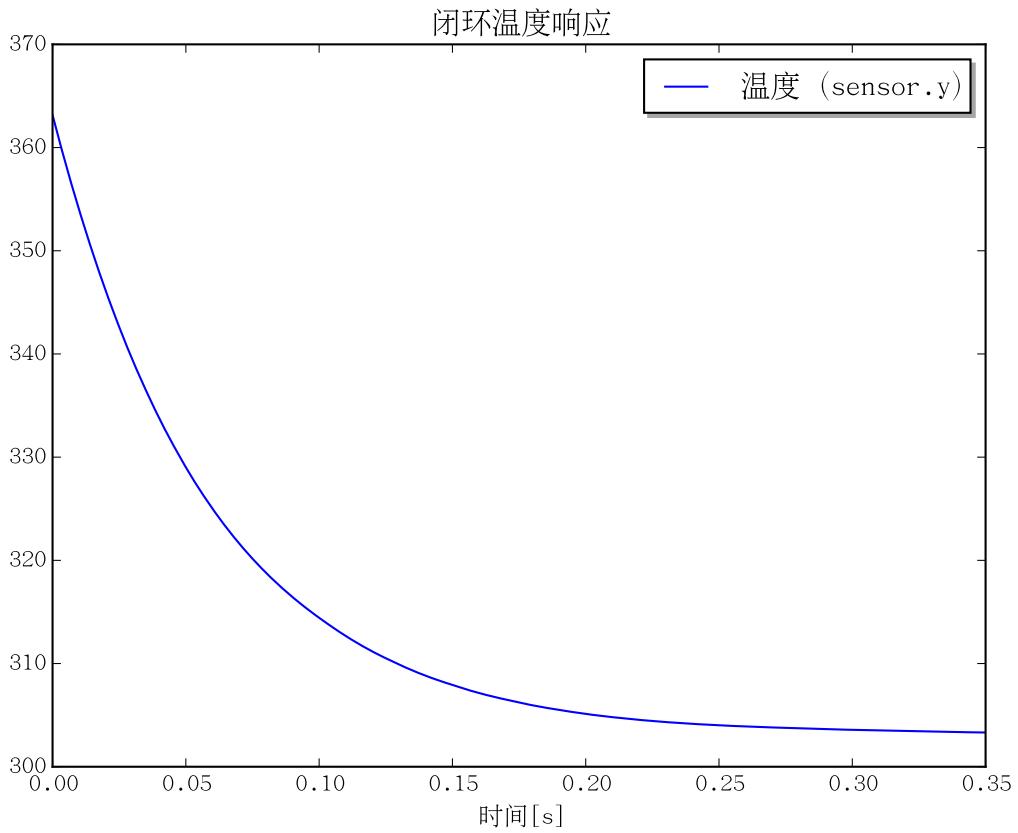
```

parameter Real A = 1.0 "Area";
parameter Real m = 0.1 "Thermal maass";
parameter Real c_p = 1.2 "Specific heat";
parameter Real T_inf = from_degC(25) "Ambient temperature";
parameter Real T_bar = from_degC(30.0) "Desired temperature";
parameter Real k = 2.0 "Controller gain";

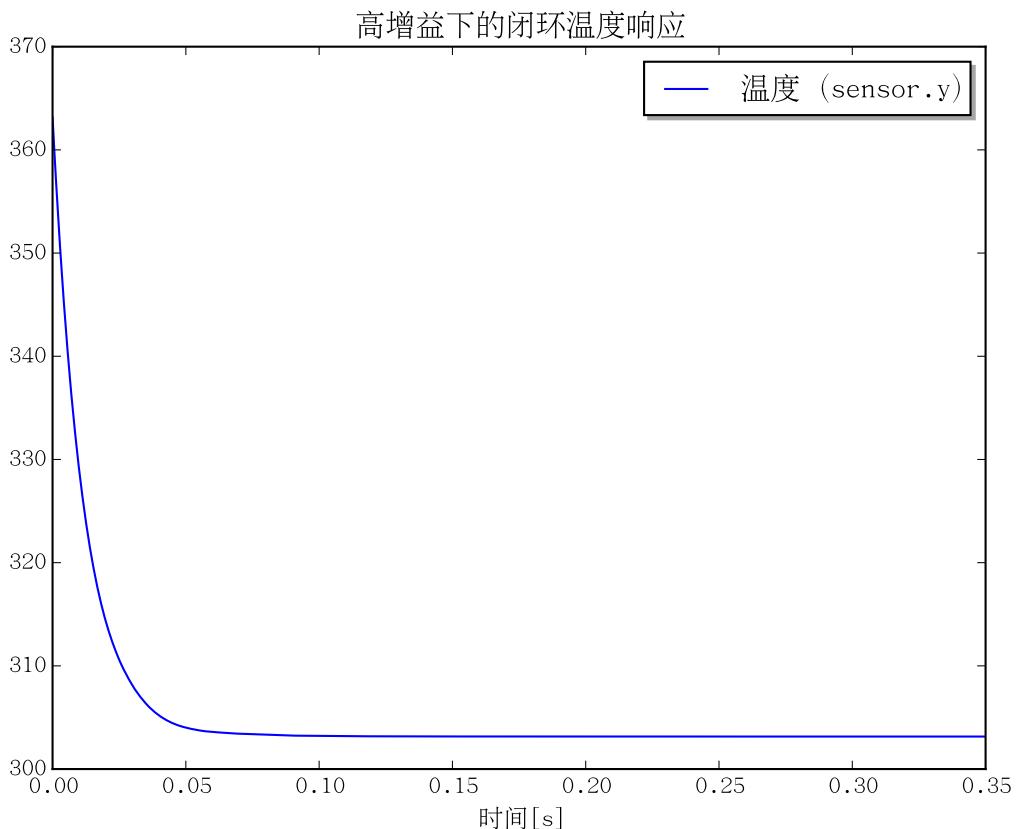
Components.Constant setpoint(k=T_bar)
annotation (Placement(transformation(extent={{-40,30},{-20,50}})));
Components.Feedback feedback
annotation (Placement(transformation(extent={{-10,30},{10,50}})));
Components.Gain controller_gain(k=k) "Gain for the proportional control"
annotation (Placement(transformation(extent={{20,30},{40,50}})));
HeatTransfer.ThermalCapacitance cap(C=m*c_p, T0 = from_degC(90))
"Thermal capacitance component"
annotation (Placement(transformation(extent={{-30,-30},{-10,-10}}));
HeatTransfer.Convection convection2(h=h*A)
annotation (Placement(transformation(extent={{10,-30},{30,-10}}));
HeatTransfer.AmbientCondition
amb(T_amb(displayUnit="K") = T_inf)
annotation (Placement(transformation(extent={{50,-30},{70,-10}}));
Components.IdealTemperatureSensor sensor annotation (Placement(transformation(
extent={{-10,-10},{10,10}},
rotation=90,
origin={0,10})));
Components.HeatSource heatSource
annotation (Placement(transformation(extent={{-60,-30},{-40,-10}}));
equation
connect(setpoint.y, feedback.u1) annotation (Line(
points={{-19,40},{-11,40}}, color={0,0,255},
smooth=Smooth.None));
connect(feedback.y, controller_gain.u) annotation (Line(
points={{10,40},{19,40}}, color={0,0,255},
smooth=Smooth.None));
connect(convection2.port_a, cap.node) annotation (Line(
points={{10,-20},{-20,-20}}, color={191,0,0},
smooth=Smooth.None));
connect(amb.node, convection2.port_b) annotation (Line(
points={{60,-20},{30,-20}}, color={191,0,0},
smooth=Smooth.None));
connect(sensor.y, feedback.u2) annotation (Line(
points={{0,21},{0,24.5},{0,24.5},{0,29}},
color={0,0,255},
smooth=Smooth.None));
connect(heatSource.node, cap.node) annotation (Line(
points={{-40,-20},{-20,-20}}, color={191,0,0},
smooth=Smooth.None));
connect(controller_gain.y, heatSource.u) annotation (Line(
points={{41,40},{50,40},{50,60},{-70,60},{-70,-20},{-61,-20}},
color={0,0,255},
smooth=Smooth.None));
connect(sensor.node, cap.node) annotation (Line(
points={{0,0},{0,0},{0,-20},{-20,-20}},
color={191,0,0},
smooth=Smooth.None));
end MultiDomainControl;

```

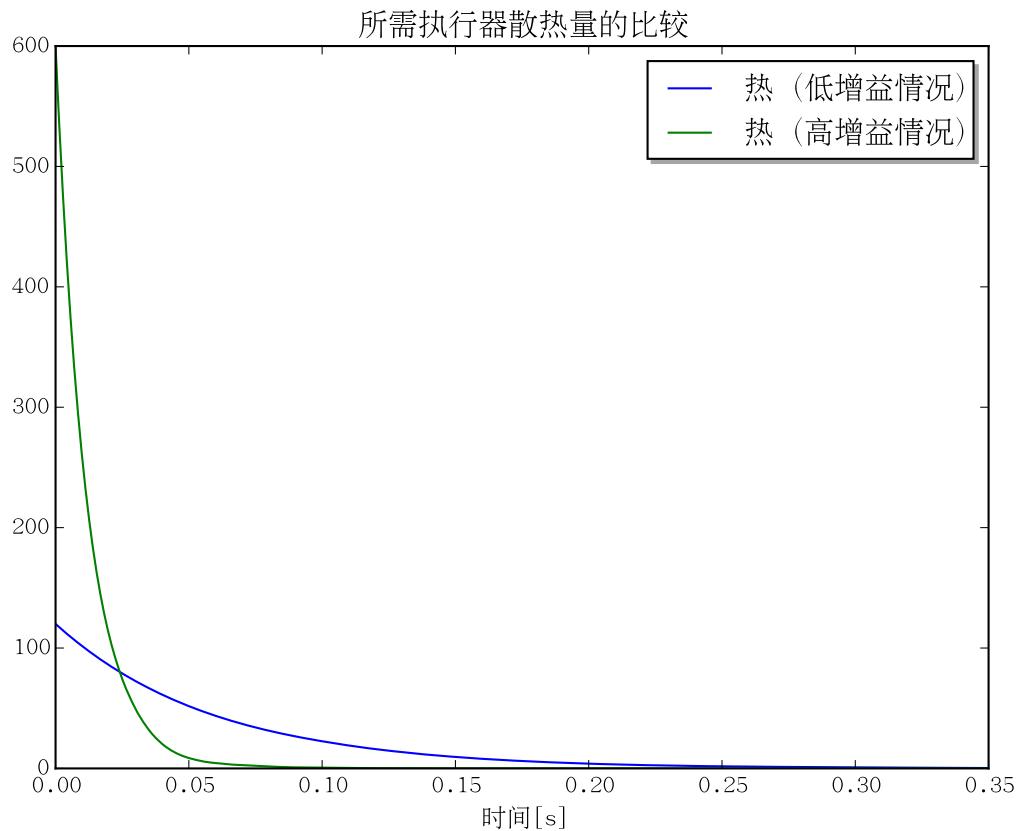
观察模型，我们可以看到，初始温度为 90 °C。而环境温度则为 25 °C。此外，该设定点温度（所需温度）为 30 °C。我们先前的例子里，系统温度最终变为环境温度。与之不同，此系统由于控制系统的影响，其温度应接近设定点温度。对上述系统进行仿真，我们可以得到以下温度响应：



我们可以增加控制器的“增益” $k$ 。这时，我们看到了不同的响应：



不过，我们可以从以下图中看到，我们的执行机构需要更大的热量输出。这是为了在第二种情况能得到更快的响应：



这仅仅是一个很简单的例子。但这演示了包含控制以及物理响应的系统可以如何让模型开发者允许探索两者对系统整体性能的影响。

## 结论

本节中，我们已经看到了如何定义因果性的 block 组件。而且，我们将其用在模拟物理系统以及与控制相关的行为。我们甚至看到了这些因果部件可以与非因果部件相结合，以产生一个“两全其美”的组合。这样一来，控制功能使用了因果部件来实现。而物理部件则使用非因果的部件。

### 第 7.1.8 节 化学组件

在最后一个基于组件建模的例子中，我们会再次回顾在向量与数组 (81) 章节内介绍的化学系统 (89)。不过，这次我们将创建各种效应的组件模型，并演示 Modelica 内的连接是如何自动确保种类的守恒。

## 种类

我们在本例中将要处理的物质类型由以下 enumeration 定义：

```
within ModelicaByExample.Components.ChemicalReactions.ABX;
type Species = enumeration(A, B, X);
```

请注意，这个定义在一个名为 ABX 的包内。这表示该组件模型要在涉及 A、B 和 X 的系统内正常工作。

## Mixture

下列 connector 定义也包含在 ABX 包内（这可以在 Interfaces 子包中找到）：

```
within ModelicaByExample.Components.ChemicalReactions.ABX.Interfaces;
connector Mixture
  Modelica.SIunits.Concentration C[Species];
  flow ConcentrationRate R[Species];
end Mixture;
```

在这里我们看到，我们的 Mixture 连接器使用浓度为穿越变量，浓度变化率为流变量。虽然，在这种情况下 flow 变量不是严格意义上的守恒量的流。但在本例中，由于所有反应都包含在相同的容器内，这样的定义足够了。

请注意，连接器里的 C 和 R 均为数组。而其下标由 enumeration 类型给出。我们之前就看到过枚举类（94）的这种用法。

## Solution

我们的第一个组件模型用于追踪控制体积内各种化学物质的浓度。正如前面所提到的，由于所有反应发生在同一体积内，我们实际上不需要指定控制体积的大小。

Solution 模型很简单。如同在本章前一部分（204）讨论过的 RegionalPopulation 模型，与模型唯一连接器相关联的横跨变量的变化率等于该连接器中的 flow 变量：

```
within ModelicaByExample.Components.ChemicalReactions.ABX.Components;
model Solution "A mixture of species A, B and X"
  Interfaces.Mixture mixture
    annotation (Placement(transformation(extent={{90,-10},{110,10}})));
    Modelica.SIunits.Concentration C[Species]=mixture.C
    annotation(Dialog(group="Initialization",showStartAttribute=true));
  equation
    der(mixture.C) = mixture.R;
end Solution;
```

## 反应

### Reaction

正如我们之前所看到的，该系统有三种化学反应。我们将研究的每个具体的反应，都会扩展自以下 partial 模型：

```
within ModelicaByExample.Components.ChemicalReactions.ABX.Interfaces;
partial model Reaction "A reaction potentially involving species A, B and X"
  parameter Real k "Reaction coefficient";
  Mixture mixture
    annotation (Placement(transformation(extent={{-110,-10},{-90,10}})));
  protected
    ConcentrationRate consumed[Species];
    ConcentrationRate produced[Species];
    Modelica.SIunits.Concentration C[Species] = mixture.C;
  equation
    consumed = -produced;
    mixture.R = consumed;
end Reaction;
```

我们看到，每个反应有反应系数 k 和 Mixture 类型的连接器 mixture。而 mixture 连接器则连接到反应发生的 Solution 部件中。内部向量值变量 consumed 和 produced 起到的作用类似于在本章前一部分介绍过的（206）SinkOrSource 内的 decline 和 growth 变量（即使用这些变量，我们就可以将每个反应的作用以一种直观的方法描述出来）。

### A+B->X

第一个完整反应模型中，我们将考虑的是一个 A 分子，一个 B 分子合成一个 X 分子。使用 Reaction 模型，我们可以如下对这种反应进行建模：

```
model 'A+B->X' "A+B -> X"
  extends Interfaces.Reaction;
protected
  Interfaces.ConcentrationRate R = k*C[Species.A]*C[Species.B];
equation
  consumed[Species.A] = R;
  consumed[Species.B] = R;
  produced[Species.X] = R;
end 'A+B->X';
```

该模型的第一个特点在于，它是由非字母数字字符组成的。具体来说，模型的名称中包含 +、-、>。只要名称是用单引号字符引用，这在 Modelica 是允许的。反应速率 R 与继承自 Reaction 模型的 consumed 以及 produced 变量配合使用。由此就可以清楚描述该反应中反应物和产物的方程式。

### A+B<-X

我们会考虑的下一个反应是将一个 X 分子转换成（回）一个 A 分子和一个 B 分子。这与之前的反应相反。该反应的 Modelica 代码为：

```
model 'A+B<-X' "A+B <- X"
  extends Interfaces.Reaction;
protected
  Interfaces.ConcentrationRate R = k*C[Species.X];
equation
  produced[Species.A] = R;
  produced[Species.B] = R;
  consumed[Species.X] = R;
end 'A+B<-X';
```

同样，方程清楚表达了哪些种类是反应物（即反应中的消耗），而那些是产物（即反应所产生的物质）。

### X+B->R+S

我们的最后一个反应将 X 分子和 B 分子转化成 R 分子和 S 分子：

```
model 'X+B->R+S' "X+B->R+S"
  extends Interfaces.Reaction;
protected
  Interfaces.ConcentrationRate R = k*C[Species.B]*C[Species.X];
equation
  consumed[Species.A] = 0;
  consumed[Species.B] = R;
  consumed[Species.X] = R;
end 'X+B->R+S';
```

我们不追踪 R 和 S 物质的浓度。因为，它们不过是副产物，不参与任何其它反应。本模型与前述模型同样遵循我们熟悉的模式。不同点在于，A 类物质不参与反应。

## 系统

我们可以将 Solution 模型和不同的反应模型相结合，如下：

```

within ModelicaByExample.Components.ChemicalReactions.Examples;
model ABX_System "Model of simple two reaction system"
  ABX.Components.Solution solution(C(each fixed=true, start={1,1,0})))
    annotation (Placement(transformation(extent={{-10,-10},{10,10}})));
  ABX.Components.'A+B->X' 'A+B->X'(k=0.1)
    annotation (Placement(transformation(extent={{30,30},{50,50}})));
  ABX.Components.'A+B<-X' 'A+B<-X'(k=0.1)
    annotation (Placement(transformation(extent={{30,-10},{50,10}})));
  ABX.Components.'X+B->R+S' 'X+B->R+S'(k=10)
    annotation (Placement(transformation(extent={{30,-50},{50,-30}})));
equation
  connect('A+B<-X'.mixture, solution.mixture) annotation (Line(
    points={{30,0},{10,0}}, color={0,0,0},
    smooth=Smooth.None));
  connect('X+B->R+S'.mixture, solution.mixture) annotation (Line(
    points={{30,-40},{20,-40},{20,0},{10,0}},
    color={0,0,0},
    smooth=Smooth.None));
  connect('A+B->X'.mixture, solution.mixture) annotation (Line(
    points={{30,40},{20,40},{20,0},{10,0}},
    color={0,0,0},
    smooth=Smooth.None));
end ABX_System;

```

注意 solution 组件内的修改语句如何用来设置 solution 组件内物质的初始浓度。另外，反应系数是用每个反应组分的修改语句指定的。最后，每个反应组分都连接到 solution.mixture 连接器。

Simulating this system for 10 seconds yields the following concentration trajectories:

```
#.. plot:: ../../plots/ABX.py # :include-source: no
```

## 结论

你可能会记得，在我们之前对这个化学系统的讨论中，推导出的方程组为：

$$\begin{aligned}\frac{d[A]}{dt} &= -k_1[A][B] + k_2[X] \\ \frac{d[B]}{dt} &= -k_1[A][B] + k_2[X] - k_3[B][X] \\ \frac{d[X]}{dt} &= k_1[A][B] - k_2[X] - k_3[B][X]\end{aligned}$$

每个方程表示特定种类的增量。方程右边的每项都用于计算该特定物种流入控制体积的净流量。哪怕牵涉相对少量的种类，手工构建该系统都会很容易引入错误。但通过使用面向组件方法，我们从不需要推导上述方程组。其结果是，这些方程会自动生成。通过自动化这个过程，我们能够避免许多潜在的错误。我们也不再需要花时间去找出并修复这些错误。

## 第 7.2 节 回顾

### 第 7.2.1 节 组件模型

本节我们将总结组件模型与此前创建的模型有何不同。这里讨论将分为两部分。第一部分将着眼于非因果建模。要如何才能为基于概要、面向组件的建模提供一个框架。这样，模型才能自动生成以及遵从守恒方程。第二部分将概述本章的主题会如何（主要是语法上）影响组件模型的定义。

不过，我们在深入讨论前，值得花些时间讨论术语。本章我们已经创建了两种不同类型的模型。第一种表示单独的效果（例如电阻、电容、弹簧、阻尼器）。另一种代表更复杂的组件（如电路、机制）。

而在讨论这些不同类模型间的差异前，下面我们来介绍一些术语以便能进行准确表述。组件模型是以可重用形式封装方程的模型。通过建立这样的模型，组件的实例就代替其所封装方程用在同样的位置。子

系统模型是由组件或其他子系统组成的模型。换句话说，子系统模型（一般）不包括方程。相反，它表示的是其它部件的总成。通常情况下，这些子系统模型通过在示意图内拖拽、连接部件模型以及子系统模型而创建的。组件模型是“无层级的”（不包含其他组件或子系统，只有方程）。相对地，子系统模型则是分层的。

我们会经常会将子系统模型称作为系统模型。系统模型是预期用于进行仿真的模型。在仿真时，Modelica 语言编译器会遍历模型的层次结构。然后，编译器会记下层次结构内的所有变量和方程式。这些都是将在仿真中使用的变量和方程式。当然，为了使其存在于唯一解，该系统模型（如同任何非 partial 模型一样）必须是平衡的（241）。

请注意，子系统模型可以包括方程式。Modelica 并没有规则禁止这样做。但是，大部分时候，模型倾向于或由方程，或由其他部件/子系统组成。避免在包含子组件或子系统的模型里放入方程其实是个好主意。因为这样做意味着在看子系统的示意图时，有关模型的某些信息将“看不见”。规则的其中一个例外是子系统中的 initial equation 区域。

现在我们完成了对术语的讨论。那么，我们开始深入讨论组件模型。

## 非因果建模

首先，我们会讨论非因果建模。我们在[连接器](#)（159）一章简单谈到了这个话题。在这里，我们将更全面地讨论非因果建模。

### 可组合性

非因果建模有两个非常大的优势。第一个是可组合性。在这里，可组合性是指我们能够将组件实例拖放并连接成几乎任何想要的配置。同时，在此过程中我们也无需担心“兼容性”。原因在于，非因果连接器的设计基础是物理兼容性，而非因果兼容性。这是可能的。因为非因果连接器的定义着眼于物理信息的交换，而非信息流的方向。其结果是，我们可以创建一个基于物理相互作用的组件模型，而无需任何关于信息交流性质（即方向性）的先验知识。

但可组合性也有其他的影响。我们不仅可以轻松通过通过拖放、连接组件去创建系统，但同时也很容易进行重新配置。将电路中的电流源替换为电压源会对系统的数学产生深远的影响（例如在系统表示框图时）。但在使用非因果方法建模时，这样的变化不会带来显著的影响。虽然底层的数学表达仍会改变，乃至会大量改变。但是，这对用户却无任何影响。因为数学表达是自动的编译过程中自动产生的。

最后，可组合的另一个特性在对于多领域系统的支持。事实上，Modelica 语言不仅支持不同工程领域（电、热、液压），它也支持多种建模的形式。模型开发者创建了框图、状态机、Petri 网等不同的模型苦库相对于在不同情况下使用特殊工具或编辑器，所有这些不同的领域和表述均可在 Modelica 里自由组合。

### 核算

**连接器** 非因果建模的另一个优点在于它会自动进行大量的核算。要明白这里进行了什么核算，让我们考虑以下 Modelica 标准库的旋转 connector 定义：

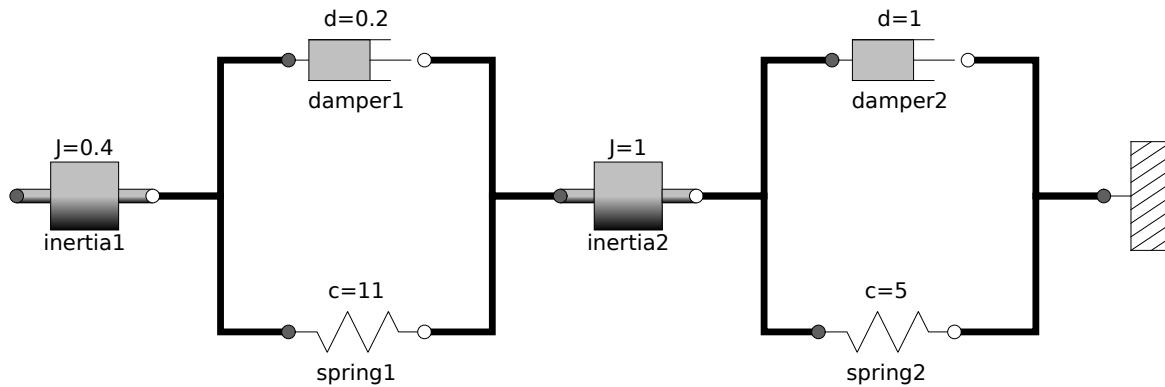
```
connector Flange_a "1-dim. rotational flange of a shaft (filled square icon)"
  Modelica.SIunits.Angle phi "Absolute rotation angle of flange";
  flow Modelica.SIunits.Torque tau "Cut torque in the flange";
  annotation(Icon(/* Filled gray circle */));
end Flange_a;

connector Flange_b "1-dim. rotational flange of a shaft (filled square icon)"
  Modelica.SIunits.Angle phi "Absolute rotation angle of flange";
  flow Modelica.SIunits.Torque tau "Cut torque in the flange";
  annotation(Icon(/* Gray circular outline */));
end Flange_b;
```

正如我们前面所讨论的，一个非因果连接器包括两种不同类型的变量：横跨变量及穿越变量。穿越变量前带有 flow 限定词。对于 Rotational 连接器而言，横跨变量是角位置 phi，而穿越变量则是扭矩 tau。

**正负号规则** 另外，回想前面讨论 Modelica 模型应遵守以下约定：连接器里穿越变量的正值表示物理量在流入与连接器相连的组件。这是一个重要的正负号惯例。不仅因为惯例确保了所有核算的正确性。而且该惯例会增加可组合性。因为这可以令（本质上对称的）组件，如弹簧、减震器等，在进行翻转后仍然有相同的功能。

**连接集合** 在我们介绍编译器所执行核算的细节前，我们需要引入一个连接集的概念。而为了介绍什么是连接集，请考虑以下的示意图：



请注意，该模型有 8 条连接：

```
equation
  connect(ground.flange_a, damper2.flange_b);
  connect(ground.flange_a, spring2.flange_b);
  connect(damper2.flange_a, inertia2.flange_b);
  connect(spring2.flange_a, inertia2.flange_b);
  connect(inertia2.flange_a, damper1.flange_b);
  connect(inertia2.flange_a, spring1.flange_b);
  connect(damper1.flange_a, inertia1.flange_b);
  connect(spring1.flange_a, inertia1.flange_b);
```

如果两条连接语句内有一个共同的连接器，它们就属于相同的连接集。如果一个连接器未连接到任何其他连接器，那么它就属于一个仅包含其本身的连接集。使用这个规则，我们可以组织连接器插入连接集如下：

- 连接集 #1
  - ground.flange\_a
  - damper2.flange\_b
  - spring2.flange\_b
- 连接集 #2
  - damper2.flange\_a
  - spring2.flange\_a
  - inertia2.flange\_b
- 连接集 #3
  - inertia2.flange\_a
  - damper1.flange\_b
  - spring1.flange\_b
- 连接集 #4
  - inertia1.flange\_b

- damper1.flange\_a
- spring1.flange\_a
- 连接集 #5
  - inertia1.flange\_a

请注意，这些连接集在图中由右至左出现。为了直观理解连接集，大家或者可以花时间将连接集内部件与模型简图一一对应。需要注意，flange\_a 连接器是实心圆，而 flange\_b 则为空心圆。

**生成方程** 这就是“核算”的开始。每个连接集都会自动生成特殊的公式。第一组自动方程都涉及到横跨变量。我们需要施加约束。从数学上讲，所有的横跨变量必须具有相同的值。此外，我们还会引入了一个公式。该公式会指出，连接集内所有穿越变量的总和必须为零。

对应上述连接集，有下列的自动生成公式：

```
// Connection Set #1
// Equality Equations:
ground.flange_a.phi = damper2.flange_b;
damper2.flange_b.phi = spring2.flange_b;
// Conservation Equation:
ground.flange_a.tau + damper2.flange_b.tau + spring2.flange_b.tau = 0;

// Connection Set #2
// Equality Equations:
damper2.flange_a.phi = spring2.flange_a.phi;
spring2.flange_a.phi = inertia2.flange_b.phi;
// Conservation Equation:
damper2.flange_a.tau + spring2.flange_a.tau + inertia2.flange_b.tau = 0;

// Connection Set #3
// Equality Equations:
inertia2.flange_a.phi = damper1.flange_b.phi;
damper1.flange_b.phi = spring1.flange_b.phi;
// Conservation Equation:
inertia2.flange_a.tau + damper1.flange_b.tau + spring1.flange_b.tau = 0;

// Connection Set #4
// Equality Equations:
inertia1.flange_b.phi = damper1.flange_a.phi;
damper1.flange_a.phi = spring1.flange_a.phi;
// Conservation Equation:
inertia1.flange_b.tau + damper1.flange_a.tau + spring1.flange_a.tau = 0;

// Connection Set #5
// Equality Equations: NONE
// Conservation Equation:
inertia1.flange_a.tau = 0;
```

请注意，对于一个空连接集（即连接集 # 5），由于集合中只有一个横跨变量，所以不产生描述相等关系的方程。守恒方程仍会产生。但方程仅包含一项。因此，方程相当于声明了一点。亦即，一个未连接的连接器内不会有任何量流出来。这也符合我们的物理直觉。

这一切有何物理意义？对于电路连接，这意味着每个连接都可以被视为连接器之间的“完美短路”。对于机械系统，连接则可被视为零惯性的完全刚性轴。总而言之，连接意味着每个连接器上的横跨变量均相等。而且，离开任何一个组件的守恒量必须进入另一个组件。没有任何守恒量会在组件间消失或者被储存。

**守恒** 这些方程造成两个重要的结果。首先，flow 变量自然而然地守恒。典型的 flow 变量是电流、扭矩、质量流率等等。因为这些量全部都是守恒量（即各为电荷、角动量和质量）的时间导数，上述公式令自动这些量自动守恒了。

但是，还有其他的量隐式地保持了守恒。具体来说，我们还可以保证能量守恒。对于所有这些领域，通过连接器的功率流都可以表示为穿越变量与横跨变量或其导数之积。其结果是，对于每个领域，利用由连接集自动生成的方程，我们可以很容易地推导出功率守恒方程。从上面的例子中我们知道，对第一个连接集我们有以下公式：

```
ground.flange_a.phi = damper2.flange_b;
damper2.flange_b.phi = spring2.flange_b;
ground.flange_a.tau + damper2.flange_b.tau + spring2.flange_b.tau = 0;
```

如果我们将最后一条方程与 ground.flange\_a 连接器的角速度  $\text{der}(\text{ground.flange\_a.phi})$  相乘，我们得到：

```
 $\text{der}(\text{ground.flange\_a.phi}) * \text{ground.flange\_a.tau}$ 
 $+ \text{der}(\text{ground.flange\_a.phi}) * \text{damper2.flange\_b.tau}$ 
 $+ \text{der}(\text{ground.flange\_a.phi}) * \text{spring2.flange\_b.tau} = 0;$ 
```

不过，我们也知道，连接集内的所有跨越变量是相等的。所以，其导数也必定相等。这意味着我们可以用其中一个替换其他的任何一个。做两次这样的替换，我们得到：

```
 $\text{der}(\text{ground.flange\_a.phi}) * \text{ground.flange\_a.tau}$ 
 $+ \text{der}(\text{damper2.flange\_b.phi}) * \text{damper2.flange\_b.tau}$ 
 $+ \text{der}(\text{spring2.flange\_b.phi}) * \text{spring2.flange\_b.tau} = 0;$ 
```

上式中的第一项就是通过 flange\_a 流入 ground 组件的功率。第二项是通过 flange\_b 流入 damper2 组件的功率。最后一项是通过 flange\_b 流入 spring2 组件的功率。由于这些代表了连接集内所有连接器中流动的功率，这意味着功率在连接集内守恒（即流出一个组件的所有功率必须流入另一个，没有丢失或被存储）。

**平衡的组件** 如果仔细观察前面的讨论，我们会在连接集内所生成的涉及非因果变量的方程式里发现一些有趣的点。但要看到这些点，我们首先需要重温我们此前对连接器和连接集的几个认识：

1. 一条连接只能从属于一个连接集。
2. 我们在前面讨论了非因果变量（167）。所以我们知道，每一个连接器内的穿越变量（即声明时带有 flow 限定词的变量），必须匹配一个横跨变量（即没有任何限定词的变量）。
3. 连接集产生的方程数等于连接组的连接器数乘以穿越—横跨变量对的数目。

记得非因果变量是成对出现的。这些变量所需的一半方程会自动通过连接产生（每对变量会生成一个方程）。这意味着变量所需的一半方程必须来自组件模型本身。

请记住，这里的讨论仅仅是关于连接器内的非因果变量。我们还需要考虑两种其他情况：

1. 组件模型内声明的变量（而不是在连接器上的）。
2. 连接器上的因果变量（即前面带有 input 或 output 限定词的变量）。

Modelica 的要求任何非 partial 模型均为平衡的。但这是什么意思？这意味着组件应该提供正确数量的方程（不比需要的更多，也不更少）。现在的问题是，如何计算所需方程的数目？

我们在对非因果变量的讨论里已经有了一定的基础。由于非因果变量所需的一半方程来自自动生成的方程，另一半方程必须来自含有这些连接器的组件模型之内。具体来说，该组件必须为其连接器的穿越—横跨变量对提供一个方程式。此外，组件还应该为其连接器具有 output 限定词的变量提供一个方程式（注意，该组件不需为其连接器具有 input 限定词的变量提供方程）。其理由是，组件可以假定所有 input 信号均为已知（由外部指定）。另外，组件要负责计算其声明的任何 output 信号。最后，组件内声明的任何（非 parameter）变量也必须对应一个公式。

总之，组件必须提供的方程式数总和为：

1. 其所有连接器内穿越—横跨变量对的数目
2. 声明中的组件模型的非 parameter 变量的数目。
3. 其所有连接器内 output 变量的数目。

请注意，这些方程可以（而且经常确实是）来自其继承的 partial 模型。

如果由组件所提供的方程数等于所需方程数，则该组件模型就成为是平衡的。

## 组件定义

本章里，我们讨论了如何创建组件模型。我们此前讨论了模型定义 (25) 应该包括的内容。而这一切从一开始就都没有改变，但关于组件模型有几件事情值得强调。

### 模块

首先，在讨论框图组件 (223) 的时候，我们介绍了 block 这种建模方法。block 是一种特殊的 model。这种模型的连接器只包含 input 和 output 的信号。

#### 有条件出现的变量/连接器

我们对可选地面连接器 (202) 的讨论中看到了另外一点。这就是条件声明。条件声明的条件表达式不能为时间的函数（即变量不能出现并在模拟期间消失）。相反，表达式必须是参数和常量的函数。这样编译器或仿真程序可以在仿真运行前判断变量是否应该存在。正如我们所看到的，这样的声明语法如下：

```
VariableType variableName(* modifications *) if conditional_expression;
```

换句话说，通过紧接着变量名称（以及添加到变量上的所有修改语句）添加 if 关键字条件表达式，就可以使该变量声明变为有条件的。当条件表达式为 true，有条件的变量才会存在。当条件表达式为 false 时，该变量就不会存在。

### 模型的适用范围

`assert` 要了解如何防止模型离开其适用范围，首先我们要解释一下 assert 函数。调用 assert 函数的语法是：

```
assert(conditional_expression, "Explanation of failure", assertLevel);
```

其中 conditional\_expression 是产生两种值，true 或 false，的表达式。false 值表示断言失败。我们马上会讨论失败的后果。第二个参数必须是 String，用以描述断言失败的原因。最后一个参数 assertLevel 的类型为 AssertionLevel（这在此前对 enumerations 的讨论中定义了）。这最后一个参数为可选。其默认值则为 AssertionLevel.error。

现在我们知道了如何使用 assert 函数。让我们来看看断言对模拟过程的影响。由此，我们可以明白断言的作用。

**定义模型的适用范围** 在创建组件 model（或者说任何 model）时，最好直接将公式使用范围添加到模型内。这可以通过添加 assert 调用来完成。assert 调用既可以加在 equation 区域，也可以 algorithm 区域。正如其名称所暗示的一样，这些断言语句要求某些条件必须始终为真。

如果一个模型中的公式仅在一定条件下准确或者适用，那么有一点很重要。这就是，上述限制必须包括在通过断言包含在模型内。否则，模型会默默地产生一个不正确的解。如果这个错误没有被发现，模型的仿真结果可能导致错误的决定。如果这个错误被发现了，这会破坏大家对模型的信任。所以，无论如何都要试图在模型中明示其局限性。

这样的断言语句会对模拟有什么影响？这值得花些考虑。模拟过程中会生成所谓的候选解。这些解有可能会就是最终的解。但也可能不是。求解器会提出不同的解。然后它检查以确保解的精确度满足一定数值容差。上述候选解就是在这个过程中产生的。而不够精确的候选解一般而言会以某种方法进一步被改良，直到求解器找到足够精确的解。

如果候选解违反了断言，那么该解会自动被认定为不准确。被违反的断言语句会自动触发解的改善过程。求解器会试图找到一个更准确、且希望不违反断言的新解。但是，如果上述改善过程找到一个足够精确的解（即精度在要求的可接受容差范围内），但该解仍违反系统中的某个断言，那么模拟环境将有两个选择。如果 assert 调用内的 level 参数是 AssertionLevel.error，那么仿真会终止。但如果 assert 调用内的 level 参数是 AssertionLevel.warning，那么用户会得到一条警告信息。警告内容就是断言内的描述。消息的传递在不同的仿真环境内可能会有所不同。要注意，level 参数的默认值（未在调用 assert 时提供 level 的话）是 AssertionLevel.error。

## 第 7.2.2 节 系统模型

下一章将深入讨论子系统 (249)。现在，我们只讨论到目前为止看到的几个子系统主题。

### 连接

我们在本章见到，组件和子系统模型间的一个区别在于，子系统模型包括 connect 语句。为了探讨 connect 语句的原理，让我们重温在讨论热力控制 (304) 时出现的 MultiDomainControl 例子。如果我们去掉所有标注（我们很快就会讨论这个内容），我们会得到一个如下所示的模型：

```
within ModelicaByExample.Components.BlockDiagrams.Examples;
model MultiDomainControl
  "Mixing thermal components with blocks for sensing, actuation and control"

  import Modelica.SIunits.Conversions.from_degC;

  parameter Real h = 0.7 "Convection coefficient";
  parameter Real m = 0.1 "Thermal maass";
  parameter Real c_p = 1.2 "Specific heat";
  parameter Real T_inf = from_degC(25) "Ambient temperature";
  parameter Real T_bar = from_degC(30.0) "Desired temperature";
  parameter Real k = 2.0 "Controller gain";

  Components.Constant setpoint(k=T_bar);
  Components.Feedback feedback;
  Components.Gain controller_gain(k=k);
  HeatTransfer.ThermalCapacitance cap(C=m*c_p, T0 = from_degC(90));
  HeatTransfer.Convection convection2(h=h);
  HeatTransfer.AmbientCondition amb(T_amb(displayUnit="K") = T_inf);
  Components.IdealTemperatureSensor sensor;
  Components.HeatSource heatSource;

equation
  connect(setpoint.y, feedback.u1);
  connect(feedback.y, controller_gain.u);
  connect(convection2.port_a, cap.node);
  connect(amb.node, convection2.port_b);
  connect(sensor.y, feedback.u2);
  connect(heatSource.node, cap.node);
  connect(controller_gain.y, heatSource.u);
  connect(sensor.node, cap.node);
end MultiDomainControl;
```

在我们前面对非因果建模 (238) 的讨论里，我们谈论了连接器内的非因果变量产生的方程式。但 connect 语句的作用取决于所连接变量的性质。MultiDomainControl 模型对我们的讨论很有用，因为该模型不只有非因果连接。

在考虑 MultiDomainControl 模型的某条特定连接前，让我们先阐述一下 connect 语句的实际作用。这里可能有一些复杂的情况。但为了简单和教学需要，在这里我们只讨论基本情况。

connect 语句连接了正正两个连接器。然后，语句按名字将每个连接器的每个相应变量“互相配对”。换句话说，语句将一个连接器的每个变量和另一连接器内的同名变量配对起来。

对每对变量，编译器首先确保这两个变量具有相同的类型（如 Real、Integer）。但是，产生的方程以及额外限制就取决于变量前的限定词了。下面列表涵盖了所有的基本情况：

- 穿越变量**—这是带有 flow 限定词的变量。我们在此前对非因果建模 (238) 讨论里就提到了，连接集的所有变量会生成一个守恒方程。
- 参数**—包含 parameter 限定词的变量不会产生任何方程式。相反，它会生成一个 assert 调用，确保两个变量之间值相同。这在某些情况下很有用。例如，当 connector 包括指定连接器内数组大小的 Integer 参数时，断言语句会确保数组大小相同。

- **输入**—带有 `input` 限定词的变量只能搭配带有 `input` 或 `output` 限定词的变量。假定这个要求得到满足，生成的等式会简单地为这两个变量建立相等关系。
- **输出**—带有 `output` 限定词的变量只能与具有 `input` 限定词的变量配对（即两个输出不能连接）。与输入变量的情况一样，这样的连接会产生相等关系。
- **横跨变量**—这些变量均没有任何限定词（和此前的情况不同）。正如我们对非因果建模（238）所讨论的一样，连接集会产生一系列方程，令有横跨变量相等。

在对框图组件（223）的讨论里，我们将 `input` 和 `output` 限定词表述为“因果性的”。事实上，`input` 和 `output` 限定词实际上并不指定执行计算的顺序。正如以上所讨论，这些限定词只是限制变量可以如何连接。除了以上提到的限制，还有另一个限制：一个连接组内只能有一个 `output` 信号（原因很明显）。

我们可以看到，在 MultiDomainControl 模型里已经包括了几个这样的例子。例如，

```
connect(setpoint.y, feedback.u1);
```

在这里，`output` 信号 `setpoint.y` 与 `input` 信号 `feedback.u1` 相连接。因此，此连接只涉及因果信号。但另一方面，我们有以下连接：

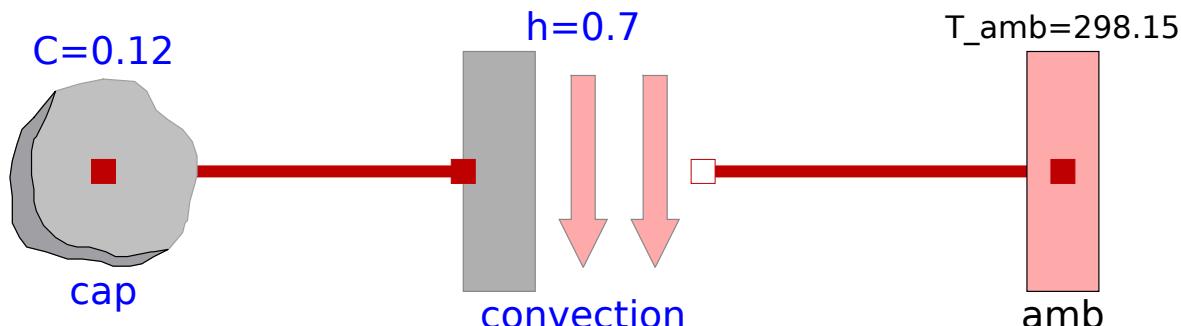
```
connect(heatSource.node, cap.node);
```

由此将得到此前讨论过（238）的那类守恒方程。

总之，使用 `connect` 语句是一种管理复杂任务（如生成守恒以及连续性方程）并为此生成公式的方法。而同时，语句也会检查并确保该连接是有意义（例如变量须具有相同的类型）。

## 简图

本章中，我们展示了 Modelica 子系统模型可以如何用图形方式表示，如：



产生这样图形所需的所有信息均包含在该 Modelica 模型里。虽然，这些信息已在本章中部分 Modelica 代码列举了出来。我们还没有真正讨论到底存储了什么信息、这些信息又是存储在了哪里。

渲染子系统简图需要三项信息：

- 用来表示每个组件的图标。
- 各组件的位置。
- 每个连接的路径

## 组件图标

每个组件的图标就是包含在为该组件定义 `Icon` 标注内的任何绘图图元。`Icon` 标注的细节在之前对图形标注（167）的讨论里涵盖了。

## 组件位置

我们知道要组件要怎样呈现。现在，我们需要知道要在哪里对组件进行绘制。这就需要 Placement 标注了。这个标注出现在本章中许多例子里，如：

```
Convection convection(h=0.7, A=1.0)
annotation (Placement(transformation(extent={{10,-10},{30,10}})));
```

Placement 标注不过建立了一个矩形区域，用以绘制组件的图标。与其他的图形标注（167）相同，我们可以将 Placement 标注用 record 定义表示：

```
record Placement
  Boolean visible = true;
  Transformation transformation "Placement in the diagram layer";
  Transformation iconTransformation "Placement in the icon layer";
end Placement;
```

visible 字段与在此前讨论 GraphicItem 标注时的作用一致。也就是说，此项用以控制组件是否被渲染。

transformation 字段定义了如何在示意图中呈现图标。若标注被认为是子系统图标的一部分时，iconTransformation 则定义了该标注此时的呈现方式。一般来说，iconTransformation 仅用于连接器上。因为连接器通常是出现在图标表示里的唯一组件。

Transformation 标注定义如下：

```
record Transformation
  Point origin = {0, 0};
  Extent extent;
  Real rotation(quantity="angle", unit="deg")=0;
end Transformation;
```

rotation 字段表示组件图标应旋转多少度。而 origin 字段则表明上述旋转所应围绕的位置。最后，extent 字段表示图标渲染区域的尺寸。

## 连接的显示

最后，我们讨论第三个话题，连接的显示。同样，决定如何渲染连接的标注已经出现在许多例子中。现在，我们终于会解释这些信息的意义了。考虑我们热力控制（304）例子中的下列 connect 声明：

```
connect(controller_gain.y, heatSource.u) annotation (Line(
  points={{41,40},{50,40},{50,60},{-70,60},{-70,-20},{-61,-20}},
  color={0,0,255},
  smooth=Smooth.None));
```

注意到 connect 语句后带有标注。尤其要注意，这是一个 Line 标注。我们此前已经讨论过 Line（169）。标注数据是在这里与当时的数据一致。

## 第 7.2.3 节 组件模型标注

本章例子中出现的几个标注先前已经介绍过了（例如在对图形标注（167）以及简图（244）的讨论里）。这里我们将介绍那些尚未介绍的标注，并讨论它们的作用。

defaultComponentName

### 类型：模型标注

defaultComponentName 标注用于模型定义中。其作用是定义该模型的实例应有的默认名称。图形工具会在将模型拖入简图时以此为其分配一个初始名称。

defaultComponentPrefixes

#### 类型：模型标注

defaultComponentName 标注定义将组件拖入图时使用的默认名称。而 defaultComponentPrefixes 则定义任何应自动纳入组件声明的**限定词**。此标注的值应该是一个**字符串**，字符串内容则是由空格分隔开的限定词列表。

组件在实例化时，图形化工具会去查找该组件相关的定义。目的是 defaultComponentPrefixes 标注是否有赋值。如果有，工具会提取该字符串列出的限定词，并立即将其添加为组件声明的限定词。

Dialog

#### 类型：声明标注

Dialog 标注用于在图形用户界面下帮助组织变量（一般而言用于 parameters）。此标注提供了超出编译模型所需的额外信息。这指示了图形工具在组件对话框中应包含哪些内容。

Dialog 标注的结构可以用以下 record 定义来表示：

```
record Dialog
  parameter String tab = "General";
  parameter String group = "Parameters";
  parameter Boolean enable = true;
  parameter Boolean showStartAttribute = false;
  parameter String groupImage = "";
  parameter Boolean connectorSizing = false;
end Dialog;
```

tab 字段是一个字符串。它表示此变量所应归属选项卡的名称。group 字段给出变量在**指定选项卡**内所属“组”的名称。enable 字段可以使用表达式，表明该参数何时应显示。showStartAttribute 字段可用于将（此变量的）start 属性值加入到用户界面。这样，用户就能够容易地指定 start 属性的值。该 groupImage 字段允许用户指定与 group 字段的组相关联的图像。最后，connectorSizing 仅用于声明指定连接器数组大小的整数参数。在这种情况下，如果 connectorSizing 字段值为 True，那么在影响该连接器所需大小的行为发生后，图形界面可以**选择**去更新标注参数的值。

DynamicSelect

#### 类型：声明标注

DynamicSelect 标注用于指定标注值可以如何依赖于仿真的解。例如，DynamicSelect 标注可以用来基于温度变化调整组件图标的颜色。DynamicSelect 有两个值与其相关联，即：

```
DynamicSelect(static_value, dynamic_value)
```

第一个是“静态”值。该值时会在没有可用模拟结果或者当特定工具不支持链接模拟结果到标注时是用。第二个值是“动态”值。这是一个表达式。表达式一般涉及标注声明所在作用域内的变量。该变量则可以从模拟结果得到。

preferredView

#### 类型：定义标注

preferredView 标注用于指定该定义在图形化的工具被选中时所应显示的“视图”。标注的可能取值为：

- “信息” — 显示与这个定义相关的任何文档。
- “文本” — 显示与此定义关联的 Modelica 代码。
- “简图” — 显示与此定义关联的示意图。

preferredView 标注的一个常见用途是用于创建某个特定 package 的文档。在这种情况下，package 包括 Documentation 标注。而其 preferredView 标注则设置为 info（从而令工具在访问定义直接显示文档）。

unassignedMessage

**类型：声明标注**

unassignedMessage 标注的值为字符串。标注了的声明没有找到相应方程以计算其值, unassignedMessage 标注的字符串值就可以作为编译器的诊断信息。



## 子系统

---

### 第 8.1 节 示例

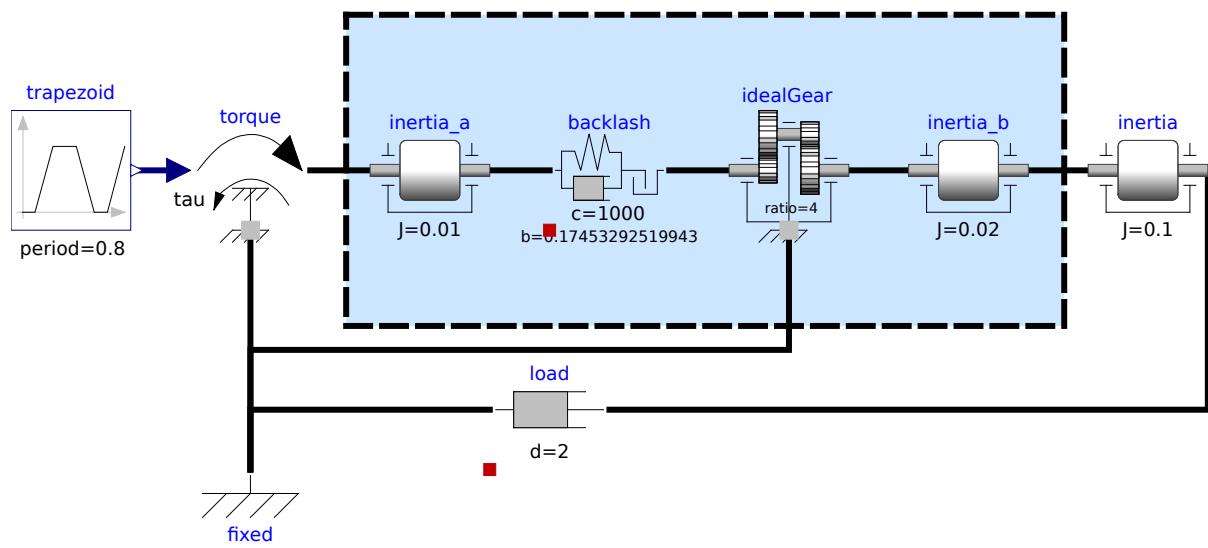
#### 第 8.1.1 节 齿轮总成

在本节中，我们将仔细观察对简单齿轮的建模。我们会考虑一系列因素，例如每个齿轮元件的惯性、齿之间存在的间隙等。最重要的当然还有两个旋转轴间的运动学关系了。我们将首先介绍一个如何创建上述总成的无层级模型的例子。然后，我们会讨论如何把上述无层级模型重构成能在不同场景下重用的子系统模型。

到现在为止，我们在创建组件模型时仅对一个物理作用进行建模，如惯性、柔顺性、阻性、对流等。正如已经多次提到的这样，这一般是值得提倡的。前提是在组件级建模。但许多现实世界的子系统综合了所有这些影响。在 Modelica 里，我们解决这个问题的方法就是建立可重复使用的子系统模型。当然，我们不会推倒重来，一个个地加入所有影响子系统模型的方程。相反，我们会重用已经开发了的组件模型。最终，子系统模型不过是预先存在的组件模型的特定组合而已。此外，我们会展示部件参数是如何连接到子系统参数上的。

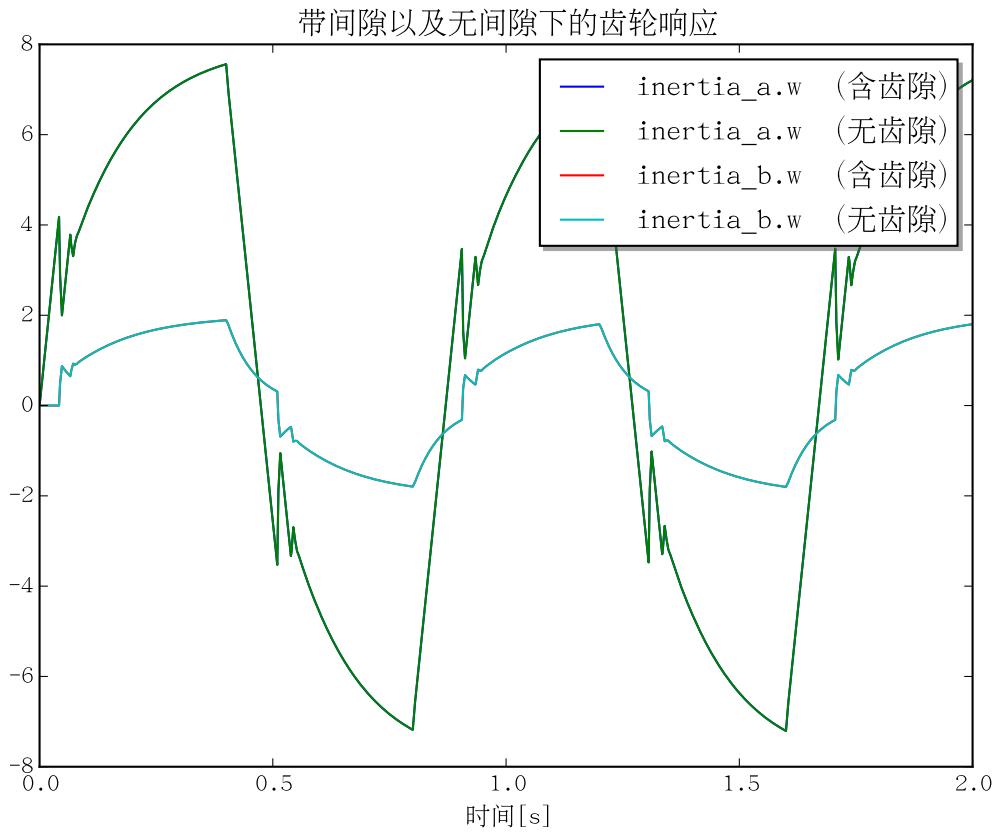
#### 无层级版本

如果对 Modelica 里创建可重用子系统模型的能力不甚熟悉的话，我们在开始时建立的 Modelica 模型可能看起来会是这样的：



该模型包括两个基本部件。虚线内的一部分模型表示齿轮本身的模型。它包括每个齿轮元件的惯性、两轮间的齿隙以及两轴之间的运动学关系。每个上述零部件均表示为一个单独的组件模型。虚线以外的另一部分模型表示我们正在运行的特定情景或实验。这包括一个以施加到齿轮上的扭矩曲线，以及被该齿轮驱动的下游负载。

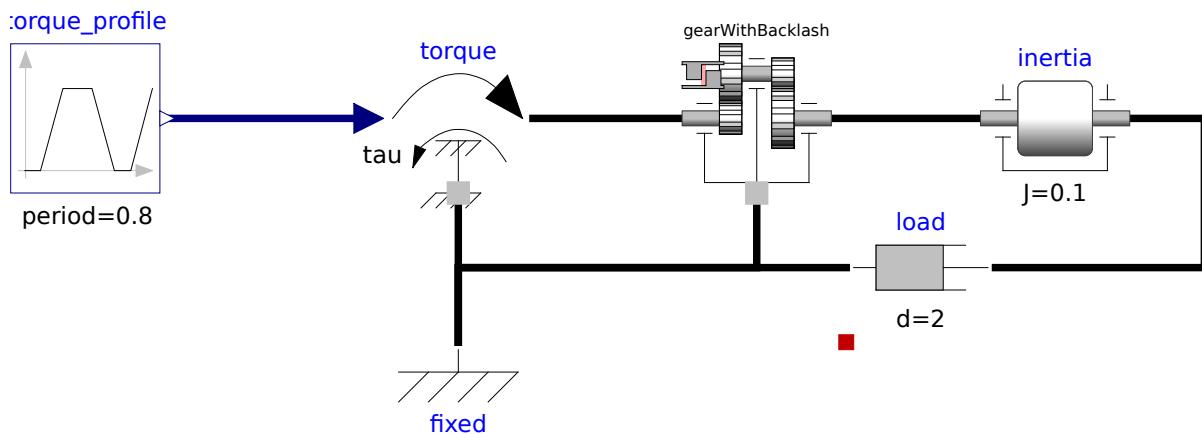
如果我们对这个系统进行仿真，那么会得到如下响应：



对这个系统必须要明白一点：虚线内的组件可能会在任何齿轮相关的应用里重复出现。事实上，例如在汽车变速器模型里上述组件模型就可能会多次重复出现。

### 有层级版本

因此，为了避免冗余（至于为什么要避免前面已经讨论过了），我们应该将虚线内的组件创建为可重复使用的子系统模型。在这种情况下，我们的示意图将如下所示：



在这种情况下，一系列用于表示齿轮的组件被替换为简图层内的单个实例。这是得益于，所有构成齿轮模型的部件模型组装为了以下子系统模型：

```

within ModelicaByExample.Subsystems.GearSubsystemModel.Components;
model GearWithBacklash "A subsystem model for a gear with backlash"
  extends Modelica.Mechanics.Rotational.Icons.Gear;
  import Modelica.Mechanics.Rotational.Components.*;

  parameter Boolean useSupport(start=true);
  parameter Modelica.SIunits.Inertia J_a
    "Moment of inertia for element connected to flange 'a'";
  parameter Modelica.SIunits.Inertia J_b
    "Moment of inertia for element connected to flange 'b'";
  parameter Modelica.SIunits.RotationalSpringConstant c
    "Backlash spring constant (c > 0 required)";
  parameter Modelica.SIunits.RotationalDampingConstant d
    "Backlash damping constant";
  parameter Modelica.SIunits.Angle b=0
    "Total backlash as measured from flange_a side";
  parameter Real ratio
    "Transmission ratio (flange_a.phi/flange_b.phi, once backlash is cleared)";

protected
  Inertia inertia_a(final J=J_a) "Inertia for the element 'a'"
  annotation (Placement(transformation(extent={{-80,-10},{-60,10}})));
  Inertia inertia_b(final J=J_b)
  annotation (Placement(transformation(extent={{40,-10},{60,10}}));
  ElastoBacklash backlash(final c=c, final d=d, final b=b) "Backlash as measured from flange_a"
  annotation (Placement(transformation(extent={{-40,-10},{-20,10}}));
  IdealGear idealGear(final useSupport=useSupport, final ratio=ratio)
  annotation (Placement(transformation(extent={{-10,-10},{10,10}}));

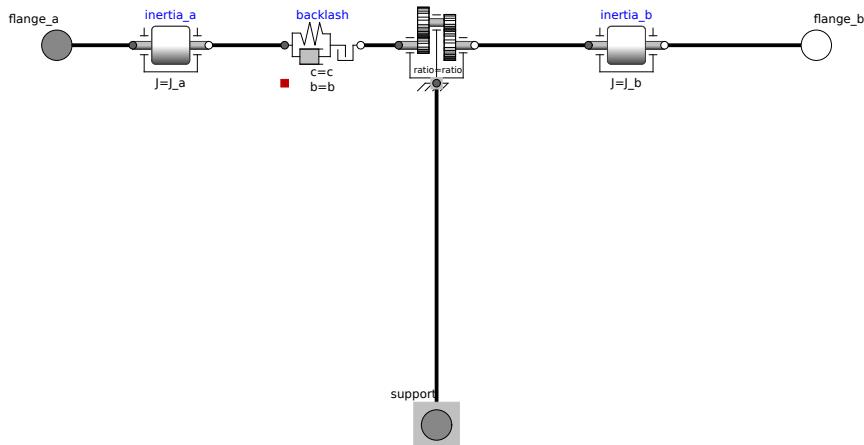
```

```

public
  Modelica.Mechanics.Rotational.Interfaces.Flange_a flange_a
  annotation (Placement(transformation(extent={-110,-10},{-90,10})));
  Modelica.Mechanics.Rotational.Interfaces.Flange_b flange_b
  annotation (Placement(transformation(extent={{90,-10},{110,10}}));
  Modelica.Mechanics.Rotational.Interfaces.Support support if useSupport
  annotation (Placement(transformation(extent={{-10,-110},{10,-90}}));
equation
  connect(flange_a, inertia_a.flange_a)
  annotation (Line(points={{-80,0},{-100,0}},
    color={0,0,0}, smooth=Smooth.None));
  connect(inertia_b.flange_b, flange_b)
  annotation (Line(points={{60,0},{100,0}},
    color={0,0,0}, smooth=Smooth.None));
  connect(idealGear.support, support)
  annotation (Line(points={{0,-10},{0,-100}},
    color={0,0,0}, smooth=Smooth.None));
  connect(idealGear.flange_b, inertia_b.flange_a)
  annotation (Line(points={{10,0},{40,0}},
    color={0,0,0}, smooth=Smooth.None));
  connect(backlash.flange_a, inertia_a.flange_b)
  annotation (Line(points={{-40,0},{-60,0}},
    color={0,0,0}, smooth=Smooth.None));
  connect(backlash.flange_b, idealGear.flange_a)
  annotation (Line(points={{-20,0},{-10,0}},
    color={0,0,0}, smooth=Smooth.None));
end GearWithBacklash;

```

渲染后的 GearWithBacklash 模型简图如下所示:

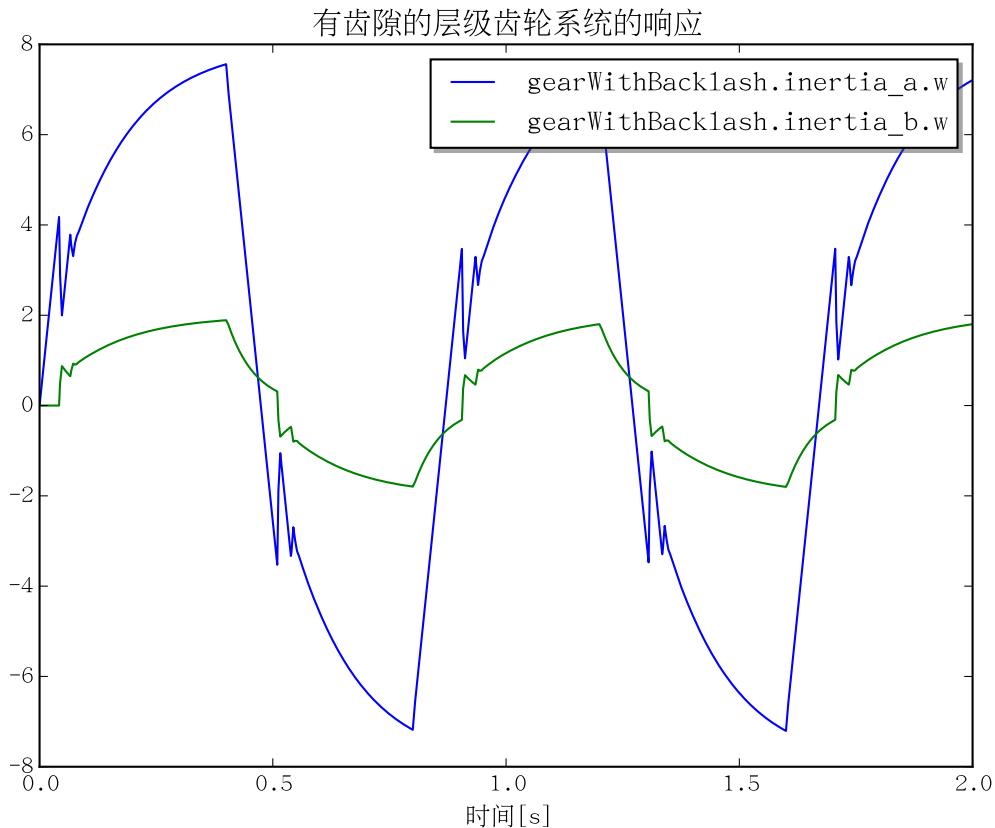


本模型做了不少的事情。首先,请注意这里有 useSupport 参数。我们在前面的章节中讨论过,此参数用以确定是否包括可选地面连接器 (202)。

注意,所有的子组件 (inertia\_a、inertia\_b、backlash 和 idealGear) 是 protected。只有连接器 (flange\_a、flange\_b 和 support) 和参数 (J\_a、J\_b、c、d、b、ratio) 是 public。这里的考虑是,用户唯一需要知道的是(或者说,应只能访问)连接器和参数。其他的一切是实现细节。模型的 protected 元素不能从外部引用。这可以防止模型在(用户不应该知道的)内部细节改变时不再有效。

还要注意有多少的参数,如 c, 是在子系统级别中指定的。之后,这些参数会通过层级结构传值给深层的参数(通常结合 final 限定词)。这样,组件的参数可以集中在子系统级别。而该模型的用户可以在一个地方(在子系统级别)看到所有相关参数。这就是所谓的传值(282)。本章后面,我们会对其进行更详细的讨论。

从下图中能看出,结果与此前无层级的版本别无二致:



## 结论

我们已经看到了如何利用组件模型可把方程化为可重用的组件。这避免了一次次手工输入方程这一繁琐、耗时且容易出错的过程。而当我们发现自己在不停将同样一系列模型放入类似的总成里时，同样的原则也适用。我们可以利用这个子系统模型的方法来创建可重用组件总成。将总成模型内部参数设定得令模型可以一次次的重用，而且重用时只需要改变参数

### 第 8.1.2 节 带迁移的掠食者猎物方程

在这一节中，我们将再次重温掠食者猎物模型。目的是了解在前面关于可重用组件模型的工作基础上，这个模型有什么变化。现在，我们将采取下一步行动，去创建各个地理区域的可重用（一般种群动力学）模型。然后，我们再用迁移模型将这些地域连接在一起。

#### 双种群的地区

所有在本节中的模型都会使用以下包括兔、狐狸种群的模型。而模型使用通常的掠食者猎物人口动态。其 Modelica 源代码为：

```
within ModelicaByExample.Subsystems.LotkaVolterra.Components;
model TwoSpecies "Lotka-Volterra two species configuration"
  // Import several component models from ModelicaByExample.Components.LotkaVolterra
  import ModelicaByExample.Components.LotkaVolterra.Components.RegionalPopulation;
  import ModelicaByExample.Components.LotkaVolterra.Components.Reproduction;
  import ModelicaByExample.Components.LotkaVolterra.Components.Starvation;
  import ModelicaByExample.Components.LotkaVolterra.Components.Predation;

  parameter Real alpha=0.1 "Birth rate";
```

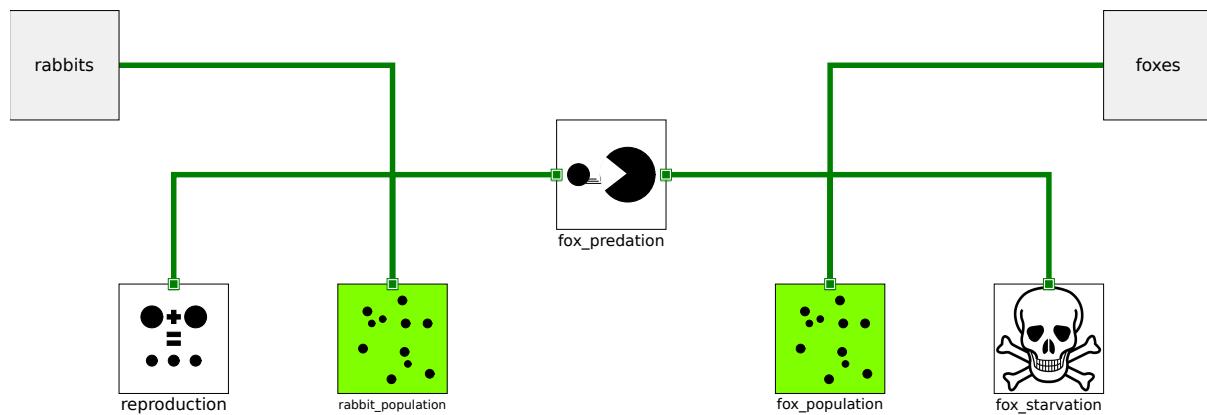
```

parameter Real gamma=0.4 "Starvation coefficient";
parameter Real initial_rabbit_population=10 "Initial rabbit population";
parameter Real initial_fox_population=10 "Initial fox population";
parameter Real beta=0.02 "Prey (rabbits) consumed";
parameter Real delta=0.02 "Predators (foxes) fed";

ModelicaByExample.Components.LotkaVolterra.Interfaces.Species rabbits
  "Population of rabbits in this region"
  annotation (Placement(transformation(extent={{-110,-10},{-90,10}})));
ModelicaByExample.Components.LotkaVolterra.Interfaces.Species foxes
  "Population of foxes in this region"
  annotation (Placement(transformation(extent={{90,-10},{110,10}})));
protected
  RegionalPopulation rabbit_population(
    initial_population=initial_rabbit_population,
    init=RegionalPopulation.InitializationOptions.FixedPopulation) "Rabbit population"
    annotation (Placement(transformation(extent={{-50,-60},{-30,-40}}));
  Reproduction reproduction(alpha=alpha) "Reproduction of rabbits"
    annotation (Placement(transformation(
      extent={{-10,-10},{10,10}},
      origin={-80,-50})));
  RegionalPopulation fox_population(
    init=RegionalPopulation.InitializationOptions.FixedPopulation,
    initial_population=initial_fox_population)
    annotation (Placement(transformation(extent={{30,-60},{50,-40}}));
  Starvation fox_starvation(gamma=gamma) "Starvation of foxes"
    annotation (Placement(transformation(extent={{70,-60},{90,-40}}));
  Predation fox_predation(beta=beta, delta=delta)
    "Foxes eating rabbits"
    annotation (Placement(transformation(extent={{-10,-30},{10,-10}}));
equation
  connect(reproduction.species, rabbit_population.species)
    annotation (Line(
      points={{-80,-40},{-80,-20},{-40,-20},{-40,-40}},
      color={0,127,0},
      smooth=Smooth.None));
  connect(fox_predation.a, rabbit_population.species)
    annotation (Line(
      points={{-10,-20},{-40,-20},{-40,-40}},
      color={0,127,0},
      smooth=Smooth.None));
  connect(fox_starvation.species, fox_population.species)
    annotation (Line(
      points={{80,-40},{80,-20},{40,-20},{40,-40}},
      color={0,127,0},
      smooth=Smooth.None));
  connect(fox_population.species, fox_predation.b)
    annotation (Line(
      points={{40,-40},{40,-20},{10,-20}},
      color={0,127,0},
      smooth=Smooth.None));
  connect(rabbit_population.species, rabbits)
    annotation (Line(
      points={{-40,-40},{-40,0},{-100,0}},
      color={0,127,0},
      smooth=Smooth.None));
  connect(fox_population.species, foxes)
    annotation (Line(
      points={{40,-40},{40,0},{100,0}},
      color={0,127,0},
      smooth=Smooth.None));
end TwoSpecies;

```

图表此组件呈现为:



此模型将作为本节随后区域人口动态模型的基础。

### 未连接区域

我们先会建立一个模型。模型由四个不相连的地域组成。对这样的模型其 Modelica 源代码很简单:

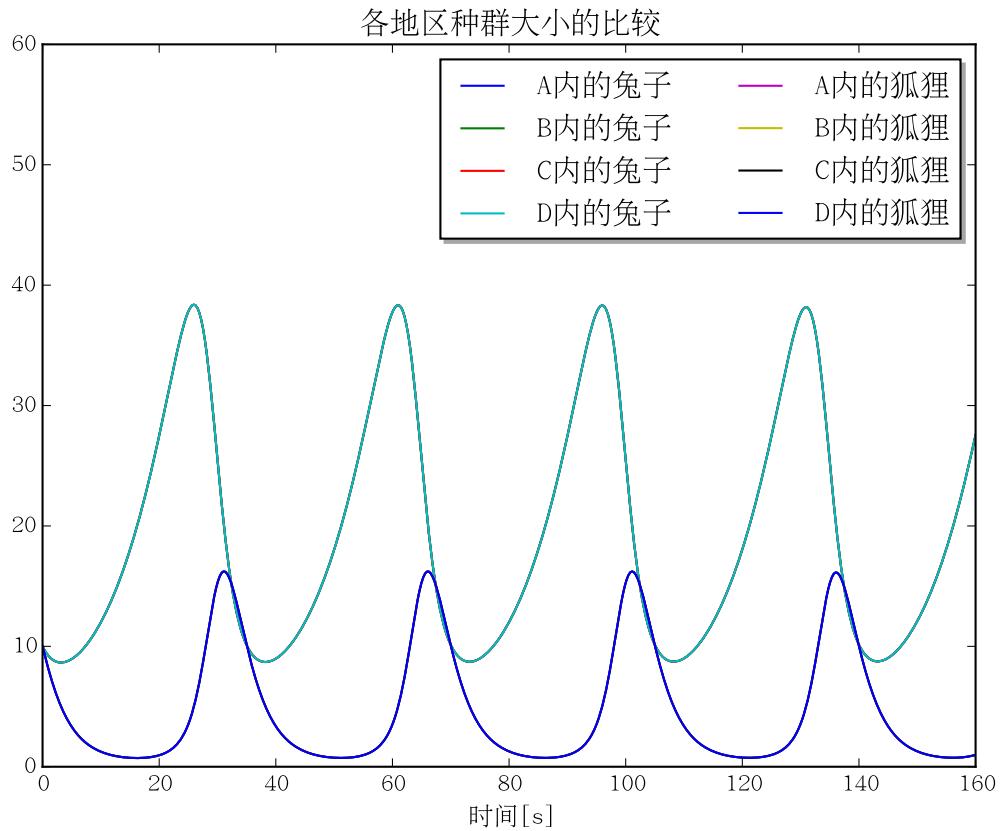
```

within ModelicaByExample.Subsystems.LotkaVolterra.Examples;
model UnconnectedPopulations "Several unconnected regional populations"
  Components.TwoSpecies A "Region A"
  annotation (Placement(transformation(extent={{-10,80},{10,100}})));
  Components.TwoSpecies B "Region B"
  annotation (Placement(transformation(extent={{-10,20},{10,40}})));
  Components.TwoSpecies C "Region C"
  annotation (Placement(transformation(extent={{-10,-40},{10,-20}})));
  Components.TwoSpecies D "Region D"
  annotation (Placement(transformation(extent={{-10,-100},{10,-80}})));
  annotation (experiment(StopTime=40, Intervals=0.008));
end UnconnectedPopulations;
  
```

此模型的简图也同样简单



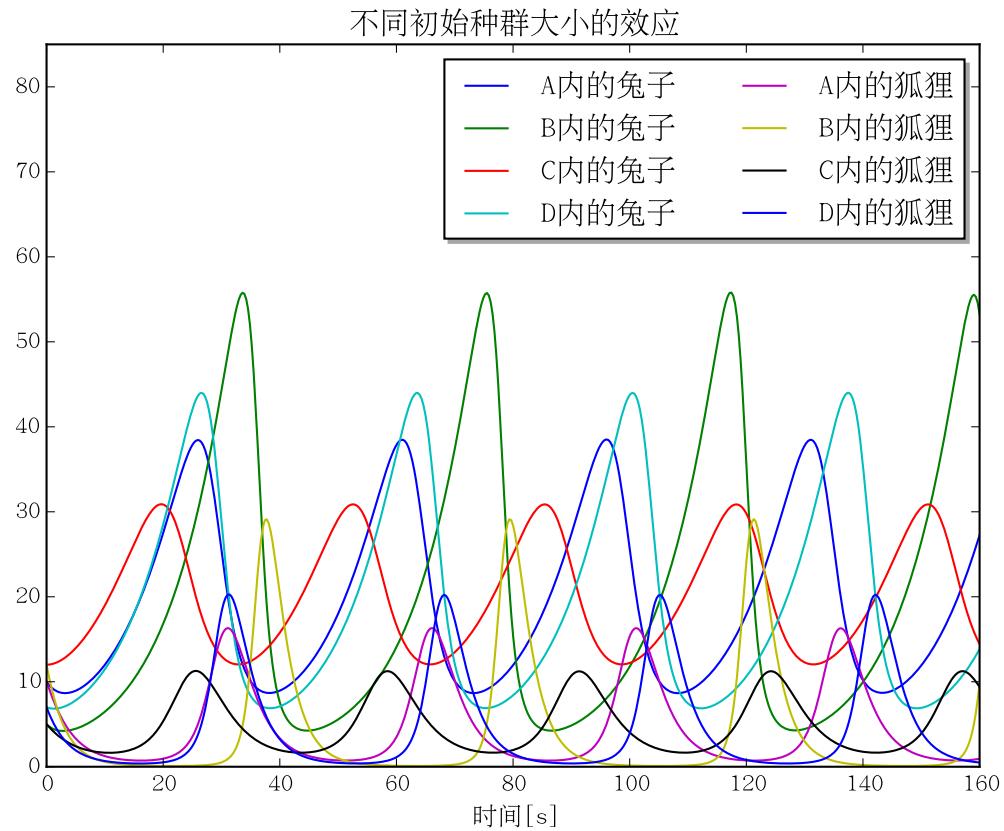
如果我们对该模型进行仿真，每个种群应遵循同一轨迹。因为各个种群的初始条件均是相同的。下面的图显示了这确实如此:



稍后我们将观察迁移的影响。但为了完全鉴别迁移的效果，我们需要对不同地域的发展趋势引进一定差异。所以，让我们修改 UnconnectedPopulations 模型的初始条件以期引进一些区域性的差异：

```
within ModelicaByExample.Subsystems.LotkaVolterra.Examples;
model InitiallyDifferent "Multiple regions with different initial populations"
  extends UnconnectedPopulations(
    B(initial_rabbit_population=5, initial_fox_population=12),
    C(initial_rabbit_population=12, initial_fox_population=5),
    D(initial_rabbit_population=7, initial_fox_population=7));
end InitiallyDifferent;
```

如果我们对该模型进行仿真，会看到各区域的种群动态有些许的差异。



## 迁移

现在我们已经模拟非相连区域里种群动态。下一步要研究的是迁移对这些动态可能造成的影响。

请考虑以下的 Modelica 迁移模型

```
within ModelicaByExample.Subsystems.LotkaVolterra.Components;
model Migration "Simple 'diffusion' based model of migration"
  parameter Real rabbit_migration=0.001 "Rabbit migration rate";
  parameter Real fox_migration=0.005 "Fox migration rate";
  ModelicaByExample.Components.LotkaVolterra.Interfaces.Species rabbit_a
    "Rabbit population in Region A"
    annotation (Placement(transformation(extent={{-70,90},{-50,110}})));
  ModelicaByExample.Components.LotkaVolterra.Interfaces.Species rabbit_b
    "Rabbit population in Region B"
    annotation (Placement(transformation(extent={{-70,-110},{-50,-90}}));
  ModelicaByExample.Components.LotkaVolterra.Interfaces.Species fox_a
    "Fox population in Region A"
    annotation (Placement(transformation(extent={{50,90},{70,110}}));
  ModelicaByExample.Components.LotkaVolterra.Interfaces.Species fox_b
    "Fox population in Region B"
    annotation (Placement(transformation(extent={{50,-110},{70,-90}}));
equation
  rabbit_a.rate = (rabbit_a.population-rabbit_b.population)*rabbit_migration;
  rabbit_a.rate + rabbit_b.rate = 0 "Conservation of rabbits";
  fox_a.rate = (fox_a.population-fox_b.population)*fox_migration;
  fox_a.rate + fox_b.rate = 0 "Conservation of rabbits";
  annotation ( Icon(graphics={
```

Rectangle(  
extent={{-100,100},{100,-100}},  
lineColor={0,127,0},

```

    fillColor={255,255,255},
    fillPattern=FillPattern.Solid),
Text(
    extent={{-100,20},{100,-20}},
    lineColor={0,127,0},
    fillColor={255,255,255},
    fillPattern=FillPattern.Solid,
    textString="%name",
    origin={-120,0},
    rotation=90),
Bitmap(extent={{-84,82},{-36,-82}}, fileName=
    "modelica://ModelicaByExample/Resources/Images/rabbit.png"),
Bitmap(extent={{18,80},{94,-84}}, fileName=
    "modelica://ModelicaByExample/Resources/Images/fox.png")));
end Migration;

```

此模型利用两个连接区域内兔子和狐狸各自种群大小的区域差去求解迁移的速率。迁移的速率与各自种群大小在区域间之差成正比。换句话说，如果某区域内兔子数比另一区的多，兔子将从人口较多的区域移动到人口较少的区。这实际上是一个“扩散”迁移模型，不一定有生态学上的基础。引入模型的目的是提供一个例子去说明，要如何在各区域现有动态的基础上引入新的效应，以影响区域间的种群动态。

如果我们将先前未连接地区用迁移路径连接起来，如：

```

within ModelicaByExample.Subsystems.LotkaVolterra.Examples;
model WithMigration "Connect populations by migration"
extends InitiallyDifferent;
Components.Migration migrate_AB "Migration from region A to region B"
annotation (Placement(transformation(extent={{-10,50},{10,70}})));
Components.Migration migrate_BC "Migration from region B to region C"
annotation (Placement(transformation(extent={{-10,-10},{10,10}})));
Components.Migration migrate_CD "Migration from region C to region D"
annotation (Placement(transformation(extent={{-10,-70},{10,-50}})));
equation
connect(migrate_CD.rabbit_b, D.rabbits) annotation (Line(
    points={{-6,-70},{-6,-76},{-20,-76},{-20,-90},{-10,-90}},
    color={0,127,0},
    smooth=Smooth.None));
connect(migrate_CD.rabbit_a, C.rabbits) annotation (Line(
    points={{-6,-50},{-6,-44},{-20,-44},{-20,-30},{-10,-30}},
    color={0,127,0},
    smooth=Smooth.None));
connect(migrate_BC.rabbit_b, C.rabbits) annotation (Line(
    points={{-6,-10},{-6,-16},{-20,-16},{-20,-30},{-10,-30}},
    color={0,127,0},
    smooth=Smooth.None));
connect(migrate_CD.fox_b, D.foxes) annotation (Line(
    points={{6,-70},{6,-76},{20,-76},{20,-90},{10,-90}},
    color={0,127,0},
    smooth=Smooth.None));
connect(migrate_BC.fox_b, C.foxes) annotation (Line(
    points={{6,-10},{6,-16},{20,-16},{20,-30},{10,-30}},
    color={0,127,0},
    smooth=Smooth.None));
connect(migrate_CD.fox_a, C.foxes) annotation (Line(
    points={{6,-50},{6,-44},{20,-44},{20,-30},{10,-30}},
    color={0,127,0},
    smooth=Smooth.None));
connect(migrate_BC.fox_a, B.foxes) annotation (Line(
    points={{6,10},{6,16},{20,16},{20,30},{10,30}},
    color={0,127,0},
    smooth=Smooth.None));
connect(migrate_BC.rabbit_a, B.rabbits) annotation (Line(
    points={{-6,10},{-6,16},{-20,16},{-20,30},{-10,30}},
    color={0,127,0},
    smooth=Smooth.None));

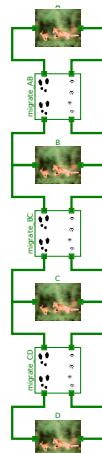
```

```

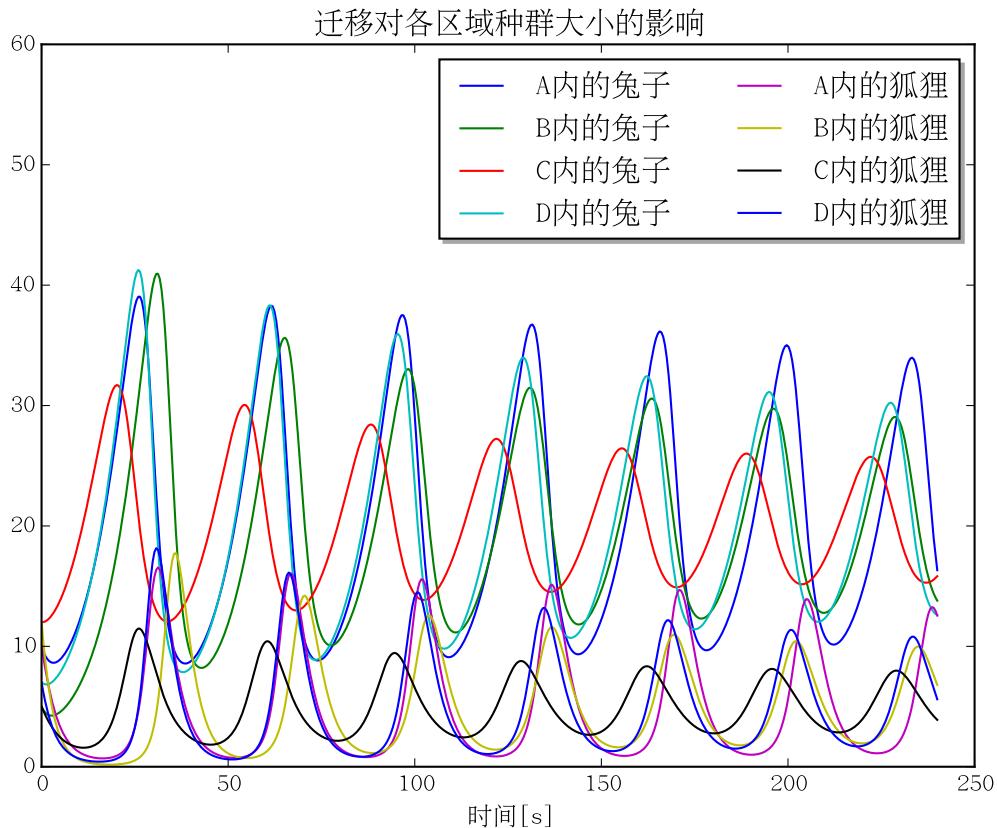
color={0,127,0},
smooth=Smooth.None));
connect(migrate_AB.fox_b, B.foxes) annotation (Line(
points={{6,50},{6,50},{6,44},{20,44},{20,30},{10,30}},
color={0,127,0},
smooth=Smooth.None));
connect(migrate_AB.rabbit_b, B.rabbits) annotation (Line(
points={{-6,50}, {-6,44}, {-20,44}, {-20,30}, {-10,30}},
color={0,127,0},
smooth=Smooth.None));
connect(migrate_AB.fox_a, A.foxes) annotation (Line(
points={{6,70},{6,76},{20,76},{20,90},{10,90}},
color={0,127,0},
smooth=Smooth.None));
connect(migrate_AB.rabbit_a, A.rabbits) annotation (Line(
points={{-6,70}, {-6,76}, {-20,76}, {-20,90}, {-10,90}},
color={0,127,0},
smooth=Smooth.None));
end WithMigration;

```

系统简图变为：



模拟这个系统后，我们可以看到不同区域内的人口动态在开始时并不同步。但是，最终不同的区域都会稳定成重复的行为模式：



## 结论

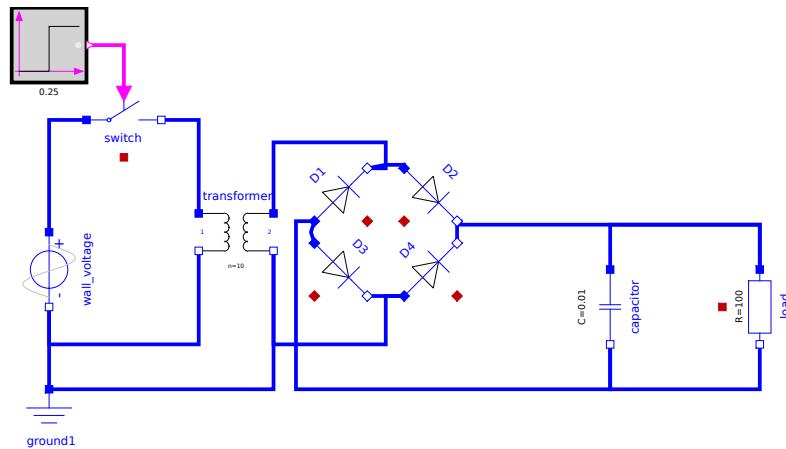
此前，我们把掠食者猎物方程化为代表捕食、饥饿和繁殖的组件。在本节中，我们能使用这些组件模型建立子系统模型，以表示在特定区域内的种群动态。在此基础上，我们将这些子系统连接为带层级的系统模型，以描述这些不同区域之间迁移的影响。

### 第 8.1.3 节 直流电源

在这一节中，我们将考虑如何在 Modelica 实现直流电源模型。我们将再一次展示，怎么重构无层级的系统模型，使得可重用的子系统模型能得到利用。

#### 无层级电源模型

在这里，我们的无层级系统模型是如下的直流电源电路：



模型的 Modelica 实现则如下：

```

within ModelicaByExample.Subsystems.PowerSupply.Examples;
model FlatCircuit "A model with power source, AC-DC conversion and load in one diagram"
import Modelica.Electrical.Analog;
Analog.Sources.SineVoltage wall_voltage(V=120, freqHz=60)
annotation (Placement(transformation(
  extent={{-10,-10},{10,10}},
  rotation=270, origin={-90,0})));
Analog.Ideal.IdealClosingSwitch switch(Goff=0)
annotation (Placement(transformation(extent={{-80,30},{-60,50}})));
Analog.Ideal.IdealTransformer transformer(Lm1=1, n=10, considerMagnetization=false)
annotation (Placement(transformation(extent={{-50,0},{-30,20}}));
Analog.Ideal.IdealDiode D1(Vknee=0)
annotation (Placement(
  transformation(
    extent={{-10,-10},{10,10}},
    rotation=45, origin={-12,20})));
Analog.Basic.Capacitor capacitor(C=1e-2)
annotation (Placement(
  transformation(
    extent={{-10,-10},{10,10}},
    rotation=270, origin={60,-10})));
Analog.Basic.Resistor load(R=100)
annotation (Placement(
  transformation(
    extent={{-10,-10},{10,10}},
    rotation=270, origin={100,-10})));
Modelica.Blocks.Sources.BooleanStep step(startTime=0.25)
annotation (Placement(transformation(extent={{-100,50},{-80,70}})));
Analog.Ideal.IdealDiode D2(Vknee=0)
annotation (Placement(
  transformation(
    extent={{-10,-10},{10,10}},
    rotation=-45, origin={12,20})));
Analog.Ideal.IdealDiode D3(Vknee=0)
annotation (Placement(
  transformation(
    extent={{-10,-10},{10,10}},
    rotation=-45, origin={-12,0})));
Analog.Ideal.IdealDiode D4(Vknee=0)
annotation (Placement(
  transformation(
    extent={{-10,-10},{10,10}},
    rotation=45, origin={12,0})));
Analog.Basic.Ground ground1
annotation (Placement(transformation(extent={{-100,-52},{-80,-32}})));

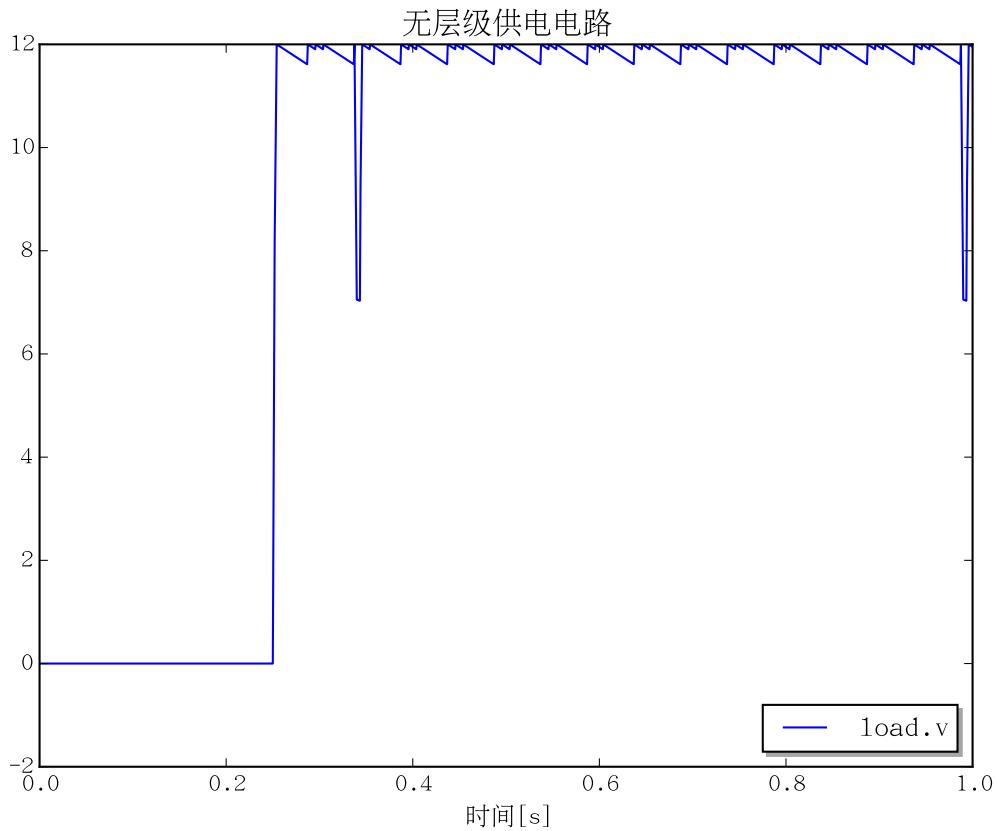
```

```

equation
  connect(switch.p, wall_voltage.p) annotation (Line(
    points={{-80,40},{-90,40},{-90,10}},
    color={0,0,255}, smooth=Smooth.None));
  connect(switch.n, transformer.p1) annotation (Line(
    points={{-60,40},{-50,40},{-50,15}},
    color={0,0,255}, smooth=Smooth.None));
  connect(step.y, switch.control) annotation (Line(
    points={{-79,60},{-70,60},{-70,47}},
    color={255,0,255}, smooth=Smooth.None));
  connect(D3.n, D4.p) annotation (Line(
    points={{-4.92893,-7.07107},{-2.46446,-7.07107},{-2.46446,-7.07107},{-1.09406e-006,-7.07107},{1.09406e-006,-7.07107},{4.92893,-7.07107}},
    color={0,0,255}, smooth=Smooth.None));
  connect(D1.n, D2.p) annotation (Line(
    points={{-4.92893,27.0711},{2.18813e-006,28},{4.92893,28},{4.92893, 27.0711}},
    color={0,0,255}, smooth=Smooth.None));
  connect(D1.p, D3.p) annotation (Line(
    points={{-19.0711,12.9289},{-20,10},{-19.0711,7.07107}},
    color={0,0,255}, smooth=Smooth.None));
  connect(D2.n, D4.n) annotation (Line(
    points={{19.0711,12.9289},{19.0711,11.4644},{19.0711,11.4644},{19.0711,10},
    {19.0711,7.07107},{19.0711,7.07107}},
    color={0,0,255}, smooth=Smooth.None));
  connect(transformer.p2, D1.n) annotation (Line(
    points={{-30,15},{-30,34},{0,34},{0,27.0711},{-4.92893,27.0711}},
    color={0,0,255}, smooth=Smooth.None));
  connect(D4.p, transformer.n2) annotation (Line(
    points={{4.92893,-7.07107},{0,-7.07107},{0,-20},{-30,-20},{-30,5}},
    color={0,0,255}, smooth=Smooth.None));
  connect(wall_voltage.n, transformer.n1) annotation (Line(
    points={{-90,-10},{-90,-20},{-50,-20},{-50,5}},
    color={0,0,255}, smooth=Smooth.None));
  connect(wall_voltage.n, ground1.p) annotation (Line(
    points={{-90,-10},{-90,-32}},
    color={0,0,255}, smooth=Smooth.None));
  connect(transformer.n2, ground1.p) annotation (Line(
    points={{-30,5},{-30,-32},{-90,-32}},
    color={0,0,255}, smooth=Smooth.None));
  connect(load.p, D2.n) annotation (Line(
    points={{100,0},{100,12},{20,12},{20,12.9289},{19.0711,12.9289}},
    color={0,0,255}, smooth=Smooth.None));
  connect(load.p, capacitor.p) annotation (Line(
    points={{100,0},{100,12},{60,12},{60,0}},
    color={0,0,255}, smooth=Smooth.None));
  connect(D1.p, capacitor.n) annotation (Line(
    points={{-19.0711,12.9289},{-24,12.9289},{-24,-32},{60,-32},{60,-20}},
    color={0,0,255}, smooth=Smooth.None));
  connect(load.n, capacitor.n) annotation (Line(
    points={{100,-20},{100,-32},{60,-32},{60,-20}},
    color={0,0,255}, smooth=Smooth.None));
end FlatCircuit;

```

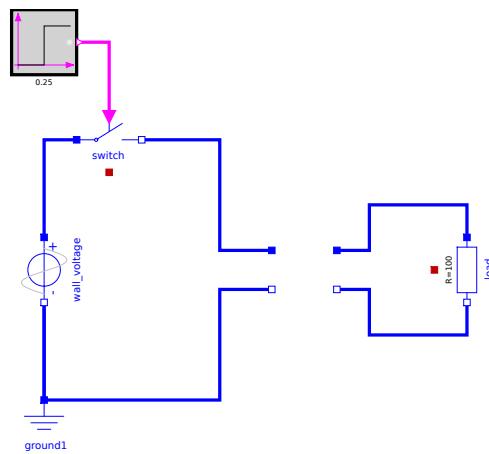
电源把交流输入电压（120V， 60Hz）经过整流后传入低通滤波器。对模型进行仿真后，我们可以看到 load 电阻上的电压如下：



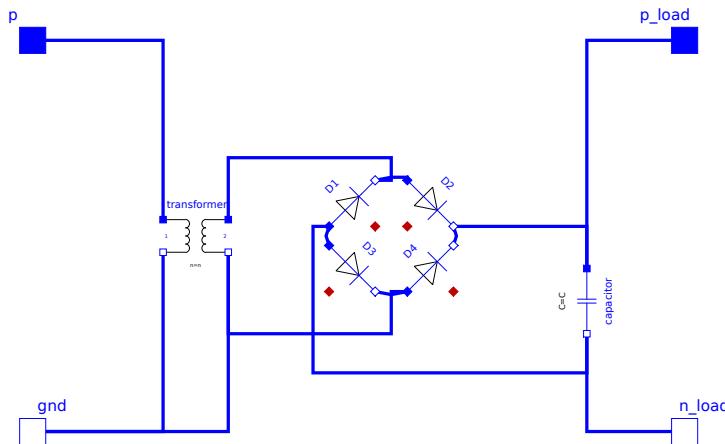
请注意，我们这里的目标是 12 伏的输出电压。但是，输出信号的质量会随着电源负载的增大而变差。在这个模拟里，负载最初为零（由于连接到电源的负载开关为开路）。但是，当开关闭合，电流开始流过负载（名为 load 的电阻）时，我们会开始看到一些错误。

### 有层级电源

再一次，我们以无层级版本为基础将一部分组件收集、整理成子系统模型，以改善系统。我们的系统级电路就变成了：



上述系统模型采用了 BasicPowerSupply 模型。此子模型的简图如下所示：



此可重用电源子系统模型的 Modelica 源代码为：

```

within ModelicaByExample.Subsystems.PowerSupply.Components;
model BasicPowerSupply "Power supply with transformer and rectifier"
import Modelica.Electrical.Analog;
parameter Modelica.SIunits.Capacitance C
  "Filter capacitance"
  annotation(Dialog(group="General"));
parameter Modelica.SIunits.Conductance Goff=1.E-5
  "Backward state-off conductance (opened diode conductance)"
  annotation(Dialog(group="General"));
parameter Modelica.SIunits.Resistance Ron=1.E-5
  "Forward state-on differential resistance (closed diode resistance)"
  annotation(Dialog(group="General"));
parameter Real n
  "Turns ratio primary:secondary voltage"
  annotation(Dialog(group="Transformer"));
parameter Boolean considerMagnetization=false
  "Choice of considering magnetization"
  annotation(Dialog(group="Transformer"));
parameter Modelica.SIunits.Inductance Lm1
  "Magnetization inductance w.r.t. primary side"
  annotation(Dialog(group="Transformer", enable=considerMagnetization));

Analog.Interfaces.NegativePin gnd
  "Pin to ground power supply"
  annotation (Placement(transformation(extent={{-110,-70},{-90,-50}})));
Analog.Interfaces.PositivePin p
  "Positive pin on supply side"
  annotation (Placement(transformation(extent={{-110,50},{-90,70}})));
Analog.Interfaces.PositivePin p_load
  "Positive pin for load side"
  annotation (Placement(transformation(extent={{90,50},{110,70}})));
Analog.Interfaces.NegativePin n_load
  "Negative pin for load side"
  annotation (Placement(transformation(extent={{90,-70},{110,-50}})));

protected
  Analog.Ideal.IdealTransformer transformer(
    final n=n, final considerMagnetization=considerMagnetization,
    final Lm1=Lm1)
    annotation (Placement(transformation(extent={{-60,-10},{-40,10}})));
  Analog.Ideal.IdealDiode D1(final Vknee=0, final Ron=Ron, final Goff=Goff)
    annotation (Placement(
      transformation(
        extent={{-10,-10},{10,10}},
        rotation=45,
        origin={-2,10})));

```

```

Analog.Basic.Capacitor capacitor(C=C)
annotation (Placement(
  transformation(
    extent={{-10,-10},{10,10}},
    rotation=270,
    origin={70,-20})));
Analog.Ideal.IdealDiode D2(final Vknee=0, final Ron=Ron, final Goff=Goff)
annotation (Placement(
  transformation(
    extent={{-10,-10},{10,10}},
    rotation=-45,
    origin={22,10})));
Analog.Ideal.IdealDiode D3(final Vknee=0, final Ron=Ron, final Goff=Goff)
annotation (Placement(
  transformation(
    extent={{-10,-10},{10,10}},
    rotation=-45,
    origin={-2,-10})));
Analog.Ideal.IdealDiode D4(final Vknee=0, final Ron=Ron, final Goff=Goff)
annotation (Placement(
  transformation(
    extent={{-10,-10},{10,10}},
    rotation=45,
    origin={22,-10})));
equation
connect(D3.n,D4. p) annotation (Line(
  points={{5.07107,-17.0711},{10,-18},{14.9289,-17.0711}},
  color={0,0,255},
  smooth=Smooth.None));
connect(D1.n,D2. p) annotation (Line(
  points={{5.07107,17.0711},{10,18},{14.9289,18},{14.9289,17.0711}},
  color={0,0,255},
  smooth=Smooth.None));
connect(D1.p,D3. p) annotation (Line(
  points={{-9.07107,2.92893},{-10,0},{-9.07107,-2.92893}},
  color={0,0,255},
  smooth=Smooth.None));
connect(D2.n,D4. n) annotation (Line(
  points={{29.0711,2.92893},{30,0},{29.0711,-2.92893}},
  color={0,0,255},
  smooth=Smooth.None));
connect(transformer.p2,D1. n) annotation (Line(
  points={{-40,5},{-40,24},{10,24},{10,17.0711},{5.07107,17.0711}},
  color={0,0,255},
  smooth=Smooth.None));
connect(D4.p,transformer. n2) annotation (Line(
  points={{14.9289,-17.0711},{10,-17.0711},{10,-30},{-40,-30},{-40,-5}},
  color={0,0,255},
  smooth=Smooth.None));
connect(D1.p, capacitor.n) annotation (Line(
  points={{-9.07107,2.92893},{-14,2.92893},{-14,-42},{70,-42},{70,-30}},
  color={0,0,255},
  smooth=Smooth.None));
connect(transformer.n1, gnd) annotation (Line(
  points={{-60,-5},{-60,-60},{-100,-60}},
  color={0,0,255},
  smooth=Smooth.None));
connect(transformer.n2, gnd) annotation (Line(
  points={{-40,-5},{-40,-60},{-100,-60}},
  color={0,0,255},
  smooth=Smooth.None));
connect(transformer.p1, p) annotation (Line(
  points={{-60,5},{-60,60},{-100,60}}),

```

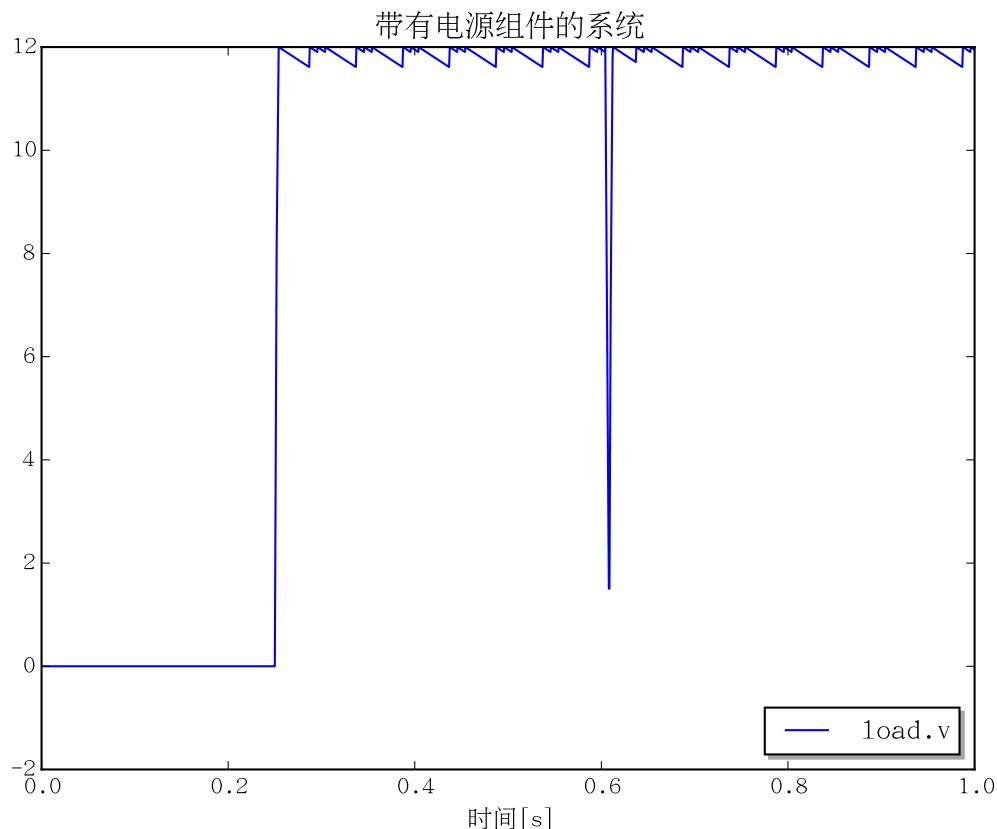
```

color={0,0,255},
smooth=Smooth.None));
connect(capacitor.p, D2.n) annotation (Line(
points={{70,-10},{70,2.92893},{29.0711,2.92893}},
color={0,0,255},
smooth=Smooth.None));
connect(capacitor.p, p_load) annotation (Line(
points={{70,-10},{70,60},{100,60}},
color={0,0,255},
smooth=Smooth.None));
connect(capacitor.n, n_load) annotation (Line(
points={{70,-30},{70,-60},{100,-60}},
color={0,0,255},
smooth=Smooth.None));
end BasicPowerSupply;

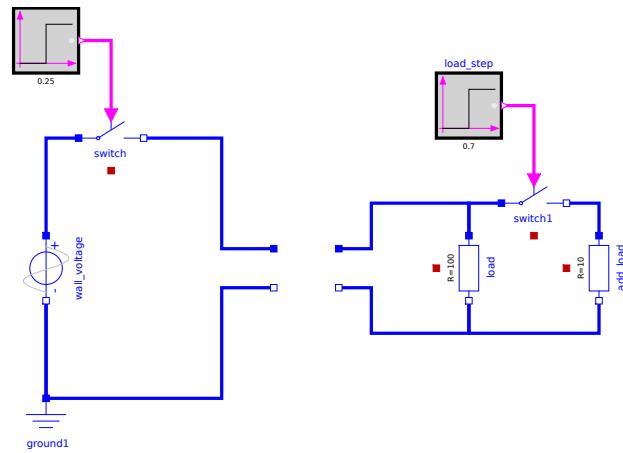
```

模型里有几个有趣的地方需要注意。首先，我们看到与先前相同的组织结构。参数和接口均为 public，而内部组件则为 protected。我们也看到用 Dialog ( 246) 标注来将参数组织为不同的组（这里是“General”和“Transformer”）。我们还看到 enable 标注上使用了 considerMagnetization 参数。只有 considerMagnetization 为真的情况下，Lm1 参数才会选择性地有效。

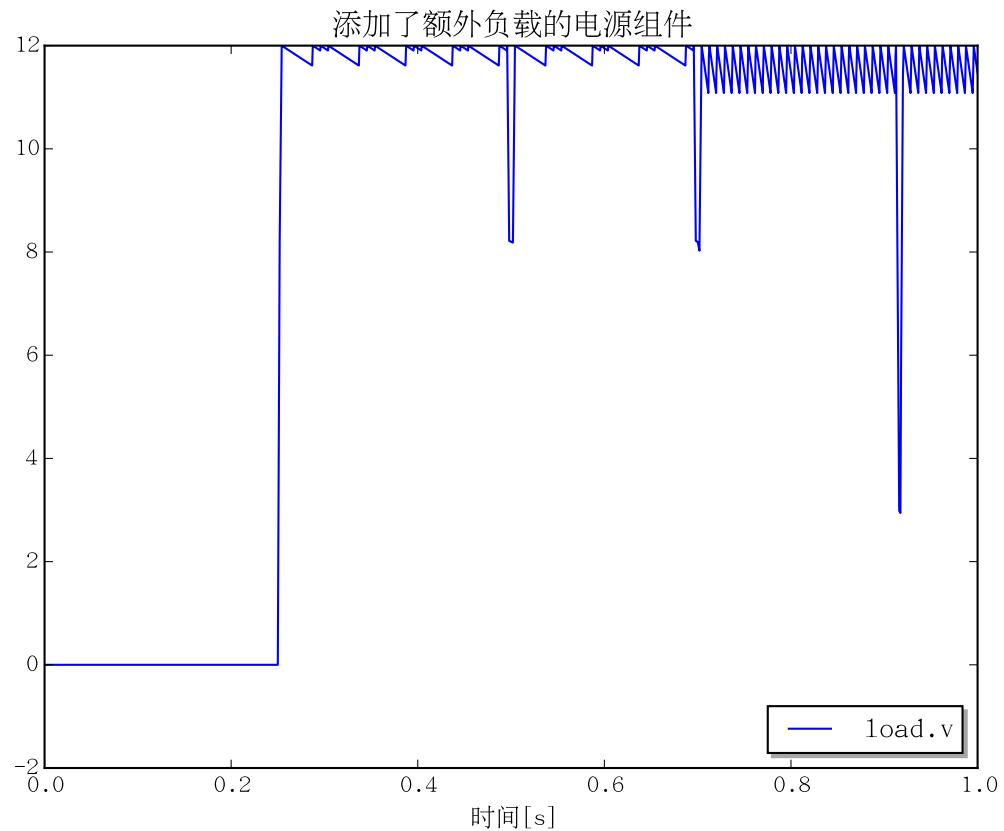
使用有层级系统模型，我们不出所料地得到与无层级版本相同的结果：



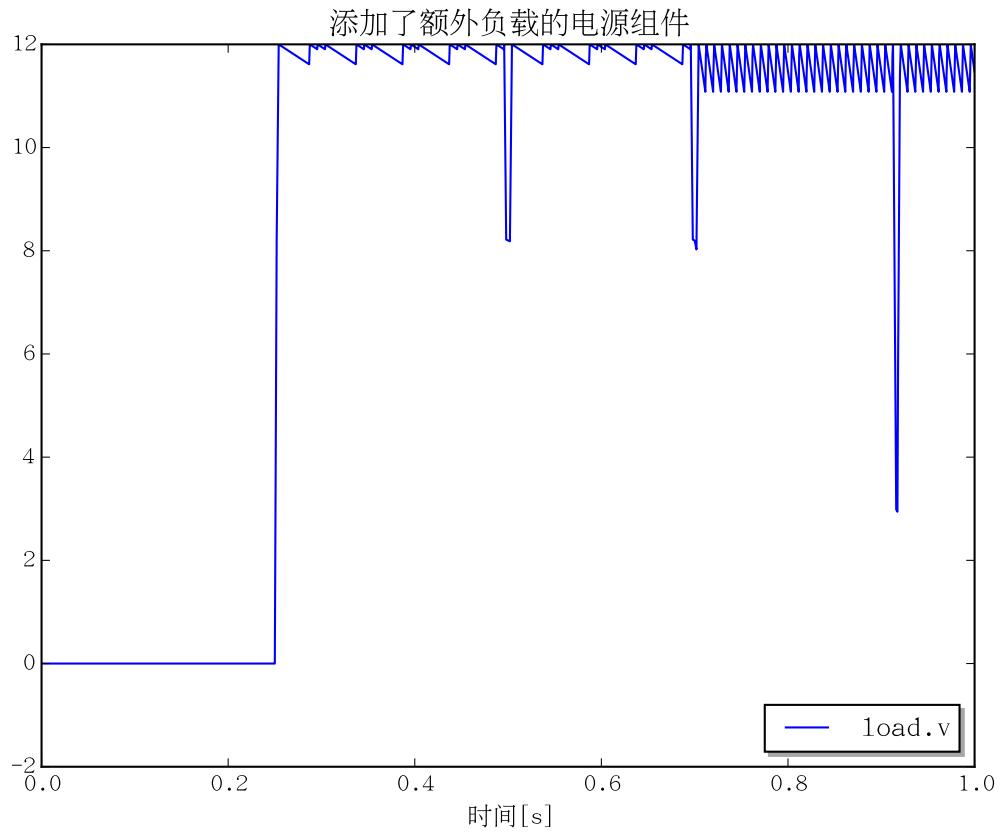
我们可以扩展系统模型，去包括额外的负载（会在一些延迟后有效）：



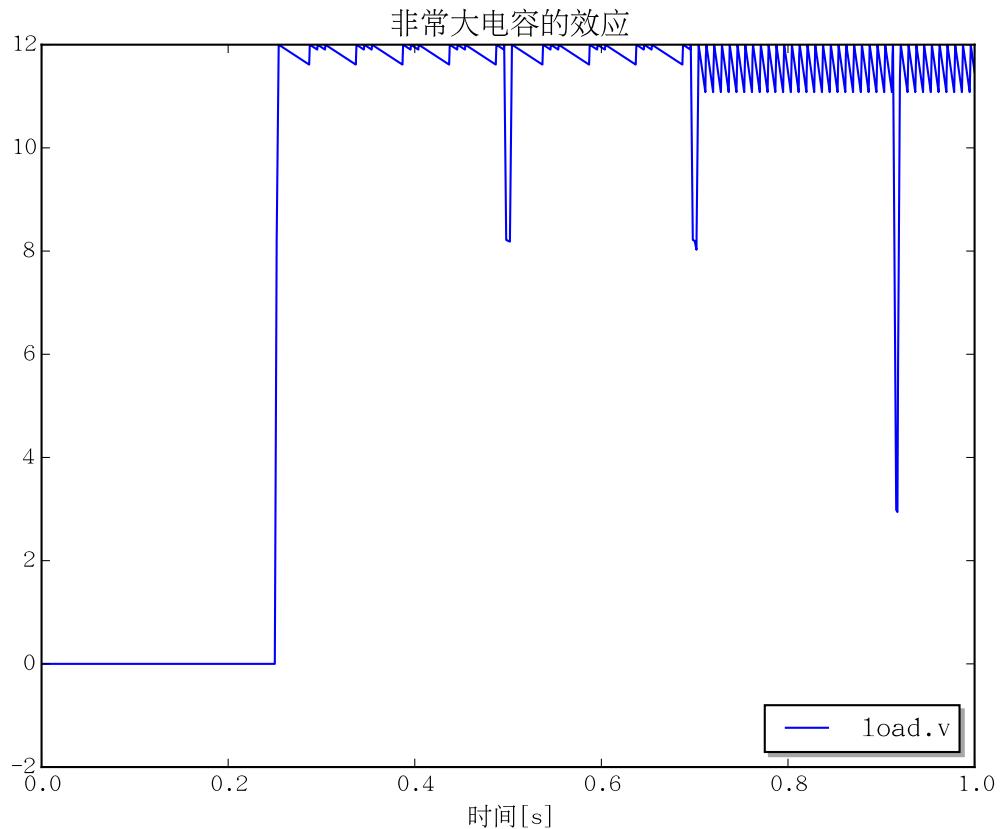
在这种情况下，如果我们模拟模型就可以看到额外负载将会对电源输出质量的影响：



通过增大电源内的电容就可以减少的电压波动的幅度，例如：



但是，如果我们增加的电容容量过多，我们会发现电源输出对负载变化的响应变得非常慢，如：



## 结论

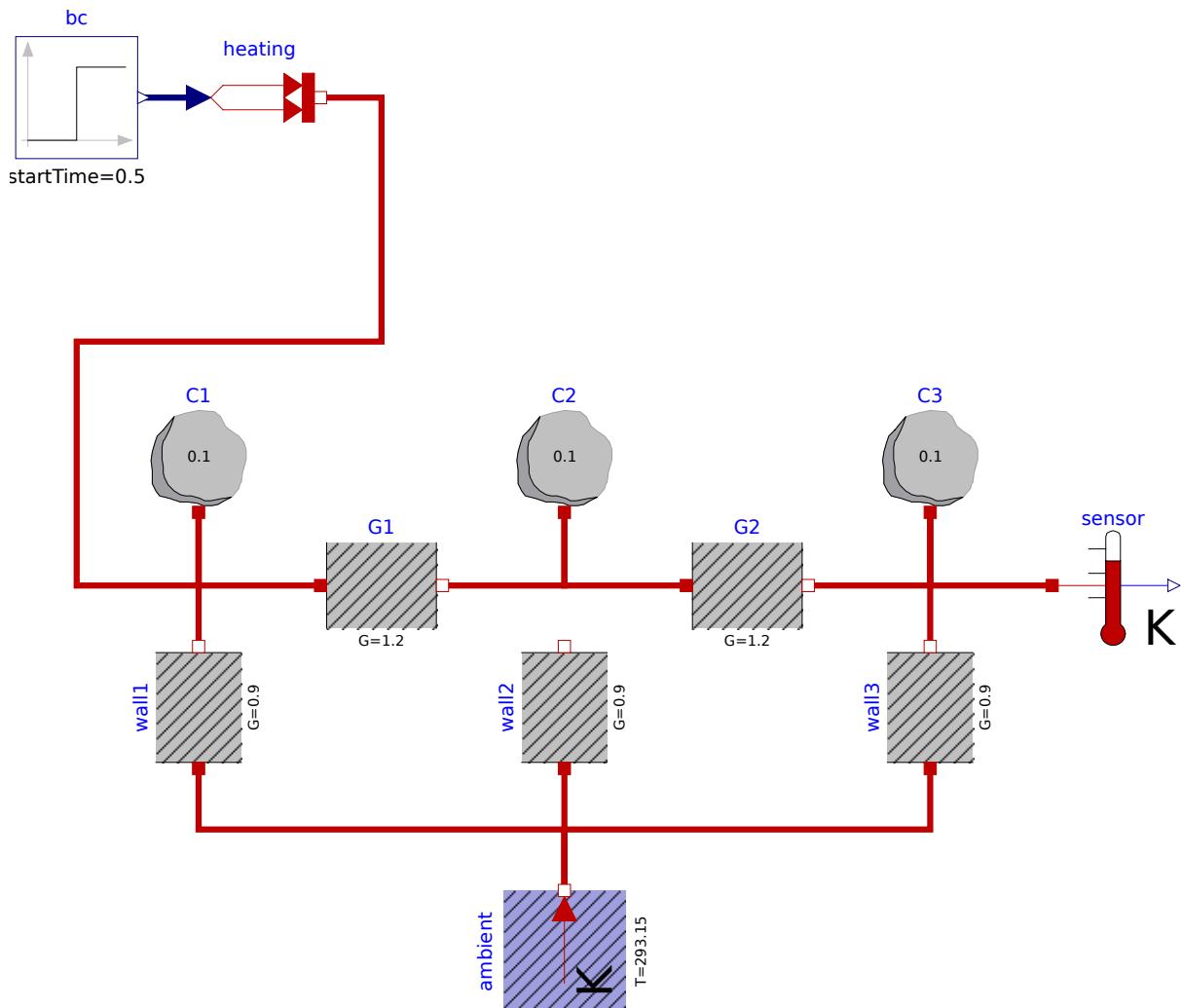
这个例子再次说明了组件的集合如何能组织成可重复使用的子系统模型。这个例子遵从了放置参数的最佳实践。而连接器在所得子系统模型的 public 区域。同时，内部组件则留在 protected 区域内。

### 第 8.1.4 节 空间分布的传热

下一个创建可重用子系统的例子会有些变化。在这一节中，我们不仅会与本章前面例子里一样，展示如何创建可重用的子系统模型。而且子系统模型将使用部件实例组成的数组。该数组的大小可被用于控制结果的空间分辨率。

#### 无层级系统

让我们开始像往常一样，从如下所示无层级系统级模型开始：



模型由热电容以及热电容之间的热导体组成。这里有 3 个热电容以及 5 个热导体。热量施加在该系统的左端。系统右端的温度传感器测量最右热容的温度变化。

模型的 Modelica 实现如下：

```
within ModelicaByExample.Subsystems.HeatTransfer.Examples;
model FlatRod "Modeling a heat transfer in a rod in a without subsystems"
  Modelica.Thermal.HeatTransfer.Sources.PrescribedHeatFlow heating
    "Heating actuator"
```

```

annotation (Placement(transformation(extent={{-60,50},{-40,70}})));
Modelica.Blocks.Sources.Step bc(height=10, startTime=0.5) "Heat profile"
annotation (Placement(transformation(extent={{-90,50},{-70,70}})));
Modelica.Thermal.HeatTransfer.Components.HeatCapacitor C1(C=0.1, T(fixed=true))
annotation (Placement(transformation(
  extent={{-10,-10},{10,10}},
  origin={-60,2})));
Modelica.Thermal.HeatTransfer.Components.ThermalConductor G1(G=1.2)
annotation (Placement(transformation(extent={{-40,-30},{-20,-10}})));
Modelica.Thermal.HeatTransfer.Components.HeatCapacitor C2(C=0.1, T(fixed=true))
annotation (Placement(transformation(
  extent={{-10,-10},{10,10}},
  origin={0,2})));
Modelica.Thermal.HeatTransfer.Components.ThermalConductor G2(G=1.2)
annotation (Placement(transformation(extent={{20,-30},{40,-10}})));
Modelica.Thermal.HeatTransfer.Components.HeatCapacitor C3(C=0.1, T(fixed=true))
annotation (Placement(transformation(
  extent={{-10,-10},{10,10}},
  origin={60,2})));
Modelica.Thermal.HeatTransfer.Sensors.TemperatureSensor sensor
annotation (Placement(transformation(extent={{80,-30},{100,-10}})));
Modelica.Thermal.HeatTransfer.Components.ThermalConductor wall1(G=0.9)
annotation (Placement(transformation(
  extent={{-10,-10},{10,10}},
  rotation=90, origin={-60,-40})));
Modelica.Thermal.HeatTransfer.Sources.FixedTemperature ambient(T=293.15)
"Ambient temperature" annotation (Placement(transformation(
  extent={{-10,-10},{10,10}},
  rotation=90, origin={0,-80})));
Modelica.Thermal.HeatTransfer.Components.ThermalConductor wall2(G=0.9)
annotation (Placement(transformation(
  extent={{-10,-10},{10,10}},
  rotation=90, origin={0,-40})));
Modelica.Thermal.HeatTransfer.Components.ThermalConductor wall3(G=0.9)
annotation (Placement(transformation(
  extent={{-10,-10},{10,10}},
  rotation=90, origin={60,-40})));
equation
connect(bc.y, heating.Q_flow) annotation (Line(
  points={{-69,60},{-60,60}},
  color={0,0,127}, smooth=Smooth.None));
connect(C1.port, G1.port_a) annotation (Line(
  points={{-60,-8},{-60,-20},{-40,-20}},
  color={191,0,0}, smooth=Smooth.None));
connect(C2.port, G2.port_a) annotation (Line(
  points={{0,-8},{0,-20},{20,-20}},
  color={191,0,0}, smooth=Smooth.None));
connect(G2.port_b, C3.port) annotation (Line(
  points={{40,-20},{60,-20},{60,-8}},
  color={191,0,0}, smooth=Smooth.None));
connect(G1.port_b, C2.port) annotation (Line(
  points={{-20,-20},{0,-20},{0,-8}},
  color={191,0,0}, smooth=Smooth.None));
connect(heating.port, C1.port) annotation (Line(
  points={{-40,60},{-30,60},{-30,20},{-80,20},{-80,-20},{-60,-20},{-60,-8}},
  color={191,0,0}, smooth=Smooth.None));
connect(wall1.port_b, C1.port) annotation (Line(
  points={{-60,-30},{-60,-8}},
  color={191,0,0}, smooth=Smooth.None));
connect(wall1.port_a, ambient.port) annotation (Line(
  points={{-60,-50},{-60,-60},{0,-60},{0,-70}},
  color={191,0,0}, smooth=Smooth.None));
connect(wall2.port_a, ambient.port) annotation (Line(

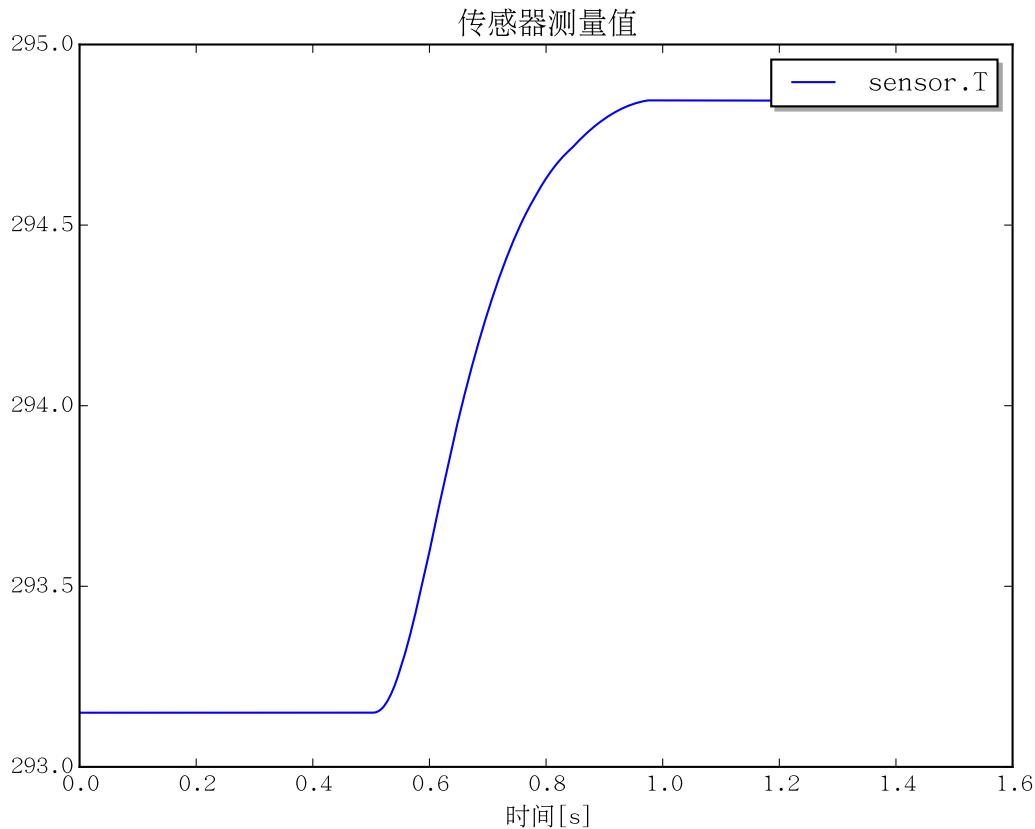
```

```

points={{0,-50},{0,-50},{0,-70}},
color={191,0,0}, smooth=Smooth.None));
connect(wall3.port_a, ambient.port) annotation (Line(
points={{60,-50},{60,-60},{0,-60},{0,-70}},
color={191,0,0}, smooth=Smooth.None));
connect(wall3.port_b, C3.port) annotation (Line(
points={{60,-30},{60,-8}},
color={191,0,0}, smooth=Smooth.None));
connect(sensor.port, C3.port) annotation (Line(
points={{80,-20},{60,-20},{60,-8}},
color={191,0,0}, smooth=Smooth.None));
connect(C2.port, wall2.port_b) annotation (Line(
points={{0,-8},{0,-19},{0,-30},{0,-30}},
color={191,0,0},
smooth=Smooth.None));
end FlatRod;

```

模拟这个系统，我们可以从下面的图内看到最右边热容的温度响应：



## 分段杆子系统

在无层级系统模型里，我们有 3 个热电容和 5 个热导体。这个配置代表的被等分为 3 段的杆、前述分段之间的热传导以及每个分段与环境的热传导。从理论上讲，我们可以把杆分为  $N$  段。然后，这些分段间有  $N - 1$  条热传导路径。而分段和环境之间则有  $N$  条热传导路径。

这里的配置为  $N = 3$ 。但我们可以创建一个以  $N$  为参数的子系统模型。换句话说，我们可以创建分为  $N$  等份子系统模型但要做到这一点，我们不能简单地将热容拖放到模型中，然后将其与热导体连接起来。因为我们不知道系统确切等分为几份。

相反，我们将使用组件数组来代表这一系列热容和热导体。所得到的 Rod 模型可以写为如下的 Modelica 代码：

```

within ModelicaByExample.Subsystems.HeatTransfer.Components;
model Rod "Modeling discretized rod"
  import HTC=Modelica.Thermal.HeatTransfer.Components;

  parameter Integer n(start=2,min=2) "Number of rod segments";
  parameter Modelica.SIunits.Temperature T0 "Initial rod temperature";
  Modelica.Thermal.HeatTransfer.Interfaces.HeatPort_a port_a
    "Thermal connector for rod end 'a'"
    annotation (Placement(transformation(extent={{-110,-10},{-90,10}})));
  Modelica.Thermal.HeatTransfer.Interfaces.HeatPort_b port_b
    "Thermal connector for rod end 'b'"
    annotation (Placement(transformation(extent={{90,-10},{110,10}})));
  parameter Modelica.SIunits.HeatCapacity C
    "Total heat capacity of element (= cp*m)";
  parameter Modelica.SIunits.ThermalConductance G_wall
    "Thermal conductivity of wall";
  parameter Modelica.SIunits.ThermalConductance G_rod
    "Thermal conductivie of rod";
  Modelica.Thermal.HeatTransfer.Interfaces.HeatPort_a ambient
    "Thermal connector for rod end 'a'"
    annotation (Placement(transformation(extent={{-10,-110},{10,-90}})));
protected
  HTC.HeatCapacitor capacitance[n](each final C=C/n, each T(start=T0, fixed=true))
  annotation (Placement(transformation(
    extent={{-10,-10},{10,10}},
    origin={-30,20})));
  HTC.ThermalConductor wall[n](each final G=G_wall/n)
  annotation (Placement(transformation(
    extent={{-10,-10},{10,10}},
    rotation=90, origin={-30,-20})));
  HTC.ThermalConductor rod_conduction[n-1](each final G=G_rod)
  annotation (Placement(transformation(extent={{-10,-10},{10,10}})));
equation
  for i in 1:n loop
    connect(capacitance[i].port, wall[i].port_b) "Capacitance to walls";
    connect(wall[i].port_a, ambient) "Walls to ambient";
  end for;
  for i in 1:n-1 loop
    connect(capacitance[i].port, rod_conduction[i].port_a)
      "Capacitance to next conduction";
    connect(capacitance[i+1].port, rod_conduction[i].port_b)
      "Capacitance to prev conduction";
  end for;
  connect(capacitance[1].port, port_a) "First capacitance to rod end";
  connect(capacitance[n].port, port_b) "Last capacitance to (other) rod end";
end Rod;

```

此模型有几个有趣的地方可以注意。首先，杆的等分数目由参数 n 表示：

```
parameter Integer n(start=2,min=2) "Number of rod segments";
```

参数 n 然后用在下列声明内。参数的目的是指定在杆内热容和热导元素的数目：

```

HTC.HeatCapacitor capacitance[n](each final C=C/n, each T(start=T0, fixed=true))
annotation (Placement(transformation(
  extent={{-10,-10},{10,10}},
  origin={-30,20})));
HTC.ThermalConductor wall[n](each final G=G_wall/n)
annotation (Placement(transformation(
  extent={{-10,-10},{10,10}},
  rotation=90, origin={-30,-20})));
HTC.ThermalConductor rod_conduction[n-1](each final G=G_rod)
annotation (Placement(transformation(extent={{-10,-10},{10,10}})));

```

需要注意的是，如果我们想对组件数组的每个组件进行修改，如加入  $G=G\_rod$ ，我们可以在修改上使用 each 限定词。我们将在本章节后面[修改语句 \(281\)](#)一节里讨论 each 限定词以及如何让修改应用于元件数组。

现在，我们已经声明了组件数组。然后，我们就可以使用 for 循环在 equation 断略把的电容和电导连接起来：

```
for i in 1:n loop
  connect(capacitance[i].port, wall[i].port_b) "Capacitance to walls";
  connect(wall[i].port_a, ambient) "Walls to ambient";
end for;
for i in 1:n-1 loop
  connect(capacitance[i].port, rod_conduction[i].port_a)
  "Capacitance to next conduction";
  connect(capacitance[i+1].port, rod_conduction[i].port_b)
  "Capacitance to prev conduction";
end for;
```

我们还需要将杆的端部连接到外部连接器，使该杆可和其他模型相连接：

```
connect(capacitance[1].port, port_a) "First capacitance to rod end";
connect(capacitance[n].port, port_b) "Last capacitance to (other) rod end";
```

这样，我们就能够创建任意次等分的分段杆模型。

## 空间分辨率

现在，我们为 Rod 模型设定了参数。大家可以看一下分段数量如何影响系统的响应。最终，我们应该看到的是，随着段数增大（或者每段变得更小），解会收敛。

首先，我们会通过考虑一个  $n=3$  的模型，即：

```
within ModelicaByExample.Subsystems.HeatTransfer.Examples;
model ThreeSegmentRod "Modeling a heat transfer using 3 segment rod subsystem"
  Modelica.Thermal.HeatTransfer.Sources.PrescribedHeatFlow heating
    "Heating actuator"
    annotation (Placement(transformation(extent={{-60,50},{-40,70}})));
  Modelica.Blocks.Sources.Step bc(height=10, startTime=0.5) "Heat profile"
    annotation (Placement(transformation(extent={{-90,50},{-70,70}}));
  Modelica.Thermal.HeatTransfer.Sensors.TemperatureSensor sensor
    annotation (Placement(transformation(extent={{80,-10},{100,10}})));
  Modelica.Thermal.HeatTransfer.Sources.FixedTemperature ambient(T=293.15)
    "Ambient temperature" annotation (Placement(transformation(
      extent={{-10,-10},{10,10}},
      rotation=90, origin={0,-80})));
  Components.Rod rod(n=3, C=0.3, G_wall=2.7, T0=293.15, G_rod=1.2)
    annotation (Placement(transformation(extent={{-20,-20},{20,20}})));
equation
  connect(bc.y, heating.Q_flow) annotation (Line(
    points={{-69,60}, {-60,60}},
    color={0,0,127}, smooth=Smooth.None));
  connect(heating.port, rod.port_a) annotation (Line(
    points={{-40,60}, {-30,60}, {-30,0}, {-20,0}},
    color={191,0,0},
    smooth=Smooth.None));
  connect(rod.ambient, ambient.port) annotation (Line(
    points={{0,-20}, {0,-70}, {0,-70}},
    color={191,0,0},
    smooth=Smooth.None));
  connect(rod.port_b, sensor.port) annotation (Line(
    points={{20,0}, {80,0}},
    color={191,0,0}),
```

```

    smooth=Smooth.None));
end ThreeSegmentRod;

```

然后，我们可以从这个模型扩展得到具有更多分段的模型，如：

```

within ModelicaByExample.Subsystems.HeatTransfer.Examples;
model SixSegmentRod "Rod divided into 6 pieces"
  extends ThreeSegmentRod(rod(n=6));
end SixSegmentRod;

```

```

within ModelicaByExample.Subsystems.HeatTransfer.Examples;
model TenSegmentRod
  extends SixSegmentRod(rod(n=10));
end TenSegmentRod;

```

```

within ModelicaByExample.Subsystems.HeatTransfer.Examples;
model OneHundredSegmentRod "Rod divided into 100 pieces"
  extends ThreeSegmentRod(rod(n=100));
end OneHundredSegmentRod;

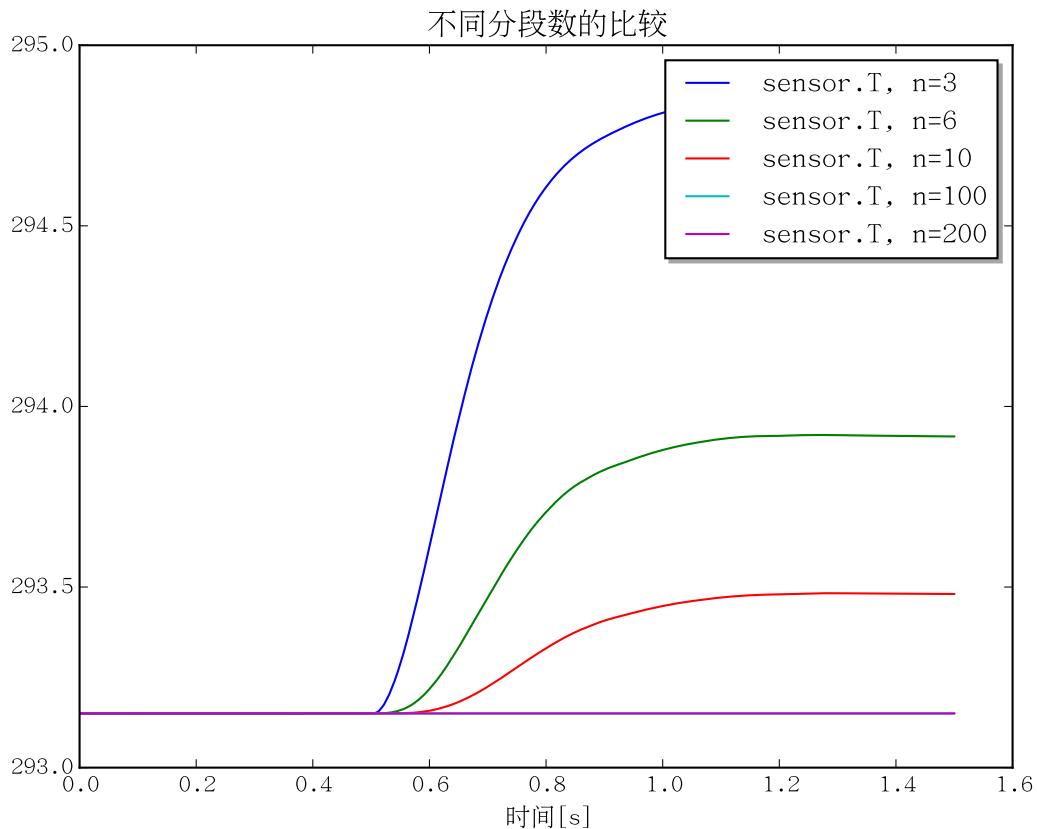
```

```

within ModelicaByExample.Subsystems.HeatTransfer.Examples;
model TwoHundredSegmentRod
  extends OneHundredSegmentRod(rod(n=200));
end TwoHundredSegmentRod;

```

如果我们模拟所有这些情况，我们会看到随着  $n$  变大，结果似乎收敛到一个共同的解  $n=10$  似乎提供了足够精确的解，而不需要引进太多多余的组件：



## 结论

在这一节中，我们已经看到了要如何使用组件数组建立任意大小的组件，并使用 for 循环将其连接在一起。

### 第 8.1.5 节 钟摆的间谐运动

在这一节中，我们将重新创建一个涉及摆的有趣实验[Berg] ( 331)。如果我们创建了具有不同固有频率的一系列摆，并让其在相同的位置开始运动。我们将看到它们会在不同的频率产生振荡。但是，如果我们从系统中删除所有能量耗散，所有的摆最终将在其最初的位置“团聚”。

一般来说，这些“团聚”事件间的周期为是系统中所有摆周期的最小公倍数。我们可以通过选择摆的长度来实现一个特定的“团圆”周期。

#### 钟摆模式

在本节中，我们可以像空间分布的传热 ( 270) 的例子一样用组件数组来建立子系统模型。但在创建摆的数组之前，我们需要先有一个单摆的模型。

```
within ModelicaByExample.Subsystems.Pendula;
model Pendulum "A single individual pendulum"
  import Modelica.Mechanics.MultiBody.Parts;
  import Modelica.Mechanics.MultiBody.Joints;

  parameter Modelica.SIunits.Position x;
  parameter Modelica.SIunits.Mass m "Mass of mass point";
  parameter Modelica.SIunits.Angle phi "Initial angle";
  parameter Modelica.SIunits.Length L "String length";
  parameter Modelica.SIunits.Diameter d=0.01;

  Parts.Fixed ground(r={0,0,x}, animation=false)
    annotation (Placement(
      transformation(
        extent={{-10,-10},{10,10}},
        rotation=270, origin={0,60})));
  Parts.PointMass ball(m=m, sphereDiameter=5*d)
    annotation (Placement(transformation(extent={{-10,-90},{10,-70}})));
  Parts.BodyCylinder string(density=0, r={0,L,0}, diameter=d)
    annotation (Placement(transformation(
      extent={{-10,-10},{10,10}},
      rotation=90,
      origin={0,-30})));
  Joints.Revolute revolute(phi(fixed=true, start=phi),
    cylinderDiameter=d/2, animation=false)
    annotation (Placement(
      transformation(
        extent={{-10,-10},{10,10}},
        rotation=90,
        origin={0,20})));
equation
  connect(string.frame_a, ball.frame_a) annotation (Line(
    points={{0,-40},{0,-40},{0,-80}},
    color={95,95,95},
    thickness=0.5,
    smooth=Smooth.None));
  connect(revolute.frame_b, ground.frame_b) annotation (Line(
    points={{0,30},{0,40},{0,40},{0,50}},
    color={95,95,95},
    thickness=0.5,
    smooth=Smooth.None));
```

```

connect(revolute.frame_a, string.frame_b) annotation (Line(
    points={{0,10},{0,10},{0,-20},{0,-20}},
    color={95,95,95},
    thickness=0.5,
    smooth=Smooth.None));
end Pendulum;

```

摆的组件显示如下：



### 系统模型

现在，有了单独的摆模型，我们就可以建立钟摆组成的系统。如果我们想要一个有  $n$  个摆的系统，而系统的完整周期为  $T$  秒。而第  $i$  个摆的长度可以用如下方式计算：

$$l_i = g_n \frac{T}{2\pi(X + (n - i))}$$

其中  $g_n$  是地球的引力常数。 $n$  是钟摆的数量。 $T$  是该系统完整循环的周期。而  $X$  是最长的摆锤在  $T$  秒的振荡次数。

Modelica 里我们可以建立上述系统，如下：

```

within ModelicaByExample.Subsystems.Pendula;
model System "A system of pendula"
  import Modelica.Constants.g_n;
  import Modelica.Constants.pi;

  parameter Integer n=15 "Number of pendula";
  parameter Modelica.SIunits.Position x[n] = linspace(0,(n-1)*0.05,n);
  parameter Modelica.SIunits.Time T = 54;
  parameter Modelica.SIunits.Time X = 30;
  parameter Modelica.SIunits.Length lengths[n] = { g_n*(T/(2*pi*(X+(n-i))))^2 for i in 1:n};
  parameter Modelica.SIunits.Angle phi0 = 0.5;

  Pendulum pendulum[n](x=x, each m=1, each phi=phi0, L=lengths)
    annotation (Placement(transformation(extent={{-10,-10},{10,10}})));
  inner Modelica.Mechanics.MultiBody.World world
    annotation (Placement(transformation(extent={{-80,-60},{-60,-40}})));
end System;

```

以下声明特别值得关注：

```
Pendulum pendulum[n](x=x, each m=1, each phi=phi0, L=lengths)
```

因为 `pendulum` 是有  $n$  个分量的数组，将有  $n$  个 `x`、`m`、`phi` 和 `L` 这些与摆相关的参数。举个例子，如果  $n=3$ ，那么模型的 `x` 将有 3 个值：`pendulum[1].x`、`pendulum[2].x` 和 `pendulum[3].x`。在 `pendulum` 声

明里，我们以不同方式对不同参数进行处理。对于  $m$ ，我们使用 `each m=1` 为每个钟摆赋相同的值。然而，对于  $L$ （和  $x$ ），我们提供数组以进行赋值。而 `lengths` 数组中的值使用前面介绍的钟摆长度公式来计算。我们将在本章后面（281）更完整地讨论如何将修改应用到组件数组。

如果对上述系统进行仿真，我们可以得到各摆轨迹的解。其如下所示：

我们可以从这个图看出，所有摆在每过 54 秒后都会回到初始位置。

## 结论

本节中，我们看到了如何才能使用、声明和修改组件数组。在本节的例子里，这允许我们指定系统内摆的数目并进行仿真。目的是观察这些摆在应用由前述方程式指定的不同长度时的特别行为。

## 第 8.2 节 回顾

### 第 8.2.1 节 子系统接口

本章的重点是组件模型可以如何被组织成可重用的子系统。正如我们在本章中许多例子里看到的，这例子总会出现一个常见的模式。为了进一步了解这种模式，让我们回顾子系统模型的各个方面。

#### 参数

一般而言，子系统的参数应为 `public`。通常情况下，除非 `model` 或 `block` 内定义的声明前带有 `protected` 关键字，否则所有声明都是公有的。在这种情况下，如果想通过在声明前添加 `public` 关键字去表明其为公共的话，也是可以的。换句话说：

```
model PublicAndProtected
  Real x; // This is public (because that is the default)
protected
  Real y1; // This is protected
  Real y2; // This is *also* protected
public
  Real z1; // This is public
  Real z2; // This is *also* public
equation
  // ...
end PublicAndProtected;
```

在本章中的例子中，经常可以见到参数定义要不是在定义的开始（这里参数默认为 `public`）就是在一系列明确标记 \ `public` 的声明里。

很显然，参数定义为公有，让子系统的用户可以进行访问。我们很快就会在对[修改语句](#)（281）以及[传值](#)（282）的讨论里看到，应如何将参数传值给较低级别的组件。但就目前而言，最主要的一点是要认识到参数声明是子系统设计模式的一部分。

#### 连接器

从某种意义上说，子系统模型和系统模型的区别就在于连接器。系统模型完整且可以立刻进行模拟。正因为如此，系统模型不希望外界任何元素影响，亦即不会有连接器。子系统可以封装组件的复杂层级结构（最小的单元为公式）。但是，由于子系统模型包括连接器，其实模型是作为较大系统的一部分来使用的。此外，正如我们在本章所看到的例子一样，由于连接器需要在连接到，他们应该是 `public`。

## 分层连接

由于子系统通常只会由组件或其他子系统组成，该子系统与外界的物理相互作用通常会重定向到内部的组件或子系统内。子系统边界的连接器实际上充当的是内部连接器的“代理”。我们在这一章已经多次看到这种模式。在 GearWithBacklash 模型里可以看到一个简单的例子：

```
connect(flange_a, inertia_a.flange_a)
```

请注意，connect 语句将子系统连接器实例 flange\_a 与子部件 inertia\_a 的连接器实例 inertia\_a.flange\_a。这是子系统模型的通用设计模式。而这个模式很容易识别。因为 connect 语句提到的连接器中的一个会包含“.”，而另一个则没有“.”。对于所有连接两个内部部件的“内部”连接，其 connect 语句提到的两个连接器里均会带有“.”。例如：

```
connect(idealGear.flange_b, inertia_b.flange_a)
```

## 分层连接公式生成

本书已多次提到 flow 变量的正负号规则（例如在我们对非因果连接（159）的讨论里）。flow 变量为正值表示该流在进入组件或子系统。同时，我们也指出，连接（243）生成方程式，使得连接对应的所有 flow 变量的和为零。

但是，处理分层子系统定义时，此规则有一个点不同。对于子系统，flow 正负号约定保持不变。所以，在连接器的 flow 变量正值还是代表流守恒量到该组件。连接集中所有 flow 变量会产生一个守恒方程这点也仍然成立。然而，在守恒方程中，内部组件连接器的 flow 变量将会与子系统连接器的 flow 变量具有相反的符号。

要理解这意味着什么，让我们考虑以下 GearWithBacklash 模型的两个 connect 语句：

```
connect(flange_a, inertia_a.flange_a)
annotation (Line(points={{-80,0},{100,0}},
color={0,0,0}, smooth=Smooth.None));
connect(idealGear.flange_b, inertia_b.flange_a)
annotation (Line(points={{10,0},{40,0}},
color={0,0,0}, smooth=Smooth.None));
```

从这些方程里，我们得到以下两个连接集：

- 连接集 # 1: flange\_a、inertia\_a.flange\_a
- 连接集 # 2: idealGear.flange\_b、inertia\_b.flange\_a

在每个连接集中都有一个 flow 变量 tau。使用前述的连接（243）规则，我们可能会预期 flow 变量将产生以下两个等式：

```
flange_a.tau + inertia_a.flange_a.tau = 0;
idealGear.flange_b.tau + inertia_b.flange_a = 0;
```

然而，在我们以往对连接（243）的讨论里，所有组件都是在同一层次的（例如 idealGear.flange\_b 和 inertia\_b.flange\_a 这样的）。但并不是所有子系统模型都是如此。而且，如上所述，若连接穿过不同层级，我们需要对不同层级内的连接器引入不同的符号。考虑到这一点，实际将产生的公式会是：

```
-flange_a.tau + inertia_a.flange_a.tau = 0;
idealGear.flange_b.tau + inertia_b.flange_a = 0;
```

请注意 flange\_a.tau 前面的减号。记住 flange\_a 充当 inertia\_a.flange\_a 的代理。以此为前提，则通过改变 flange\_a.tau 的符号，上述的第一个方程可以转化为：

```
inertia_a.flange_a.tau = flange_a.tau;
```

换句话说，通过改变 flange\_a.tau 符号，任何流过 flange\_a 守恒量也会流过 inertia\_a.flange\_a。而这正是我们在这种“代理”关系时所期望的行为。

## 实现

子系统模型通常只封装了某个组件集合。正如我们在本节已经讨论过的，子系统暴露的参数和连接器均为 public。子系统的用户将只能与这些公有部件进行交互。

子系统的实际内部细节为子系统的实现。对于子系统，实现通常由组件和其他子系统的集合组成。子系统实现里各个组件和子系统相互连接，且它们至少有一个连接器与子系统连接器相连。

正常情况下，最好不要让子系统的最终用户看到这些实现细节。要做到这一点，子系统中的所有非 parameter 声明通常会标记为 protected。这样做主要有两个原因。首先，这对子系统模型的用户隐藏了实现细节。作用是将接口简化为只有参数和连接器，避免了将用户真正需知道的内容与其不需要知道（甚至不应知道）的内容混在一起。将实现细节设定为 protected 另一个原因在于，这让开发者在之后能够灵活地提高或重构实现细节。若允许用户引用实现细节里的部件，这意味着用户就会（可能甚至是无意地）依赖于实现细节。其结果是，如果实现细节在以后发生改变，最终用户的模型就无法工作了。

### 第 8.2.2 节 组件数组

本章中有数个示例使用了组件数组。组件数组在某些情况下很有用。例如，当用户可能希望使用参数来“扩展”所使用部件的数量（正如我们在空间分布的传热（270）以及钟摆的间谐运动（276）的讨论里看到的一样）。

创建组件数组和我们前面在向量与数组（81）一节里讨论的标量数组创建实在没有任何不同。正如我们在这个例子中看到，

```
HTC.HeatCapacitor capacitance[n](each final C=C/n, each T(start=T0, fixed=true))
annotation (Placement(transformation(
  extent={{-10,-10},{10,10}},
  origin={-30,20})));
HTC.ThermalConductor wall[n](each final G=G_wall/n)
annotation (Placement(transformation(
  extent={{-10,-10},{10,10}},
  rotation=90, origin={-30,-20})));
HTC.ThermalConductor rod_conduction[n-1](each final G=G_rod)
annotation (Placement(transformation(extent={{-10,-10},{10,10}})));
```

创建组件数组的语法和用于其它类型的语法相同。要做的仅仅是声明的变量名称后加上维度。

然而，和标量不同，部件内会有其他声明。因此，在创建分量的数组时，数组中每个组件的结构都会被复制。Modelica 要求数组中的每一个元素都必须为相同的类型。这似乎是显而易见的。但这一一定程度上是因为我们还没有讨论过 replaceable 部件。我们会了解更多关于 replaceable 组件在下一章当我们谈论的 Modelica 的配置管理（320）特性。不过现在，我们只会简单地指出，只 redeclare 数组中的一个元素是不可能的。

正如在钟摆的间谐运动（276）讨论里简要地谈到的一样，我们在对组件数组进行修改语句（281）的时候，该组件中每个变量均被隐式地视为一个数组。举个例子，考虑下面的 record 定义：

```
record Point
  Real x;
  Real y;
  Real z;
end Point;
```

假设我们声明 Point 组件的数组，如 Point p[5]。那么，任何对 p.x 的引用都会被视为有 5 个 Real 变量的数组，即 p[1].x、p[2].x、p[3].x、p[4].x 以及 p[5].x。这就是所谓的切片。最重要的一点是，如果我们省略下标（如 p.x），那么下标就对应所有的元素（在技术上说，下标维持在“未限定”状态。而随后则可以“限定”下标）。此外，如果提供的下标为一个范围（例如：、1:end、2:3），那么表达式则会解析为对应于该范围内所有索引的数组子集。上述内容对于含有组件数组的组件数组也一样成立。

下面的例子，说明了一些比较常见的情况：

```
record Vector3D
  Real x[3];
```

```

end Vector3D;

model ArrayExample
  Point p[2];
  Point q[2,3];
  Vector3D v[4];
equation
  p.x = {1, 2}; // p[1].x = 1, p[2].x = 2
  q[:,3].y = {4, 5}; // q[1,3].y = 4, q[2, 3].y = 5;
  q.x = [1, 2, 3; 4, 5, 6] // q[1,1].x = 1,
    // q[1,2].x = 2,
    // q[2,3].x = 6
  v.x[1] = {1, 2, 3, 4}; // v[1].x[1] = 1, v[2].x[1] = 2,
    // v[3].x[1] = 3, v[4].x[1] = 4
  v[:,].x[1] = {1, 2, 3, 4}; // v[1].x[1] = 1, v[2].x[1] = 2,
    // v[3].x[1] = 3, v[4].x[1] = 4
  v[2:3].x[1] = {2, 3}; // v[2].x[1] = 2, v[3].x[1] = 3
  v[1].x = {1, 2, 3}; // v[1].x[1] = 1, v[1].x[2] = 2,
    // v[1].x[3] = 3
end ArrayExample;

```

### 第 8.2.3 节 修改语句

此前，我们看到了应用在变量上的修改语句的例子。在某些情况下，这些修改应用在内建类型的属性 (27) 上，如：

```
Real x(start=2, min=1);
```

在另一些情况下，修改语句则应用在 model 实例上，用以改变该特定实例参数的值，例如：

```
StepVoltage Vs(V0=0, Vf=24, stepTime=0.5);
```

但需要指出，这样的修改语句可以向下修改不止一个层级。例如，考虑前面涉及 StepVoltage 组件的例子。我们也可以修改 StepVoltage 模型的 Vs 实例内与 Vf 参数相关联的 min 属性，如下：

```
StepVoltage Vs(V0=0, Vf(min=0), stepTime=0.5);
```

但是，如果我们想同时改变 Vf 参数的属性并且赋值呢？这种修改的语法为：

```
StepVoltage Vs(V0=0, Vf(min=0)=24, stepTime=0.5);
```

一个值得讨论的重要情况是，怎么如何对组件数组的执行修改。想象一下，我们声明了 StepVoltage 组件的如下数组：

```
StepVoltage Vs[5];
```

正如我们在对组件数组 (280) 的讨论中看到的一样，这不是合法的 Modelica 代码。但这些语句可以用于表示子系统内组件的集合。如果想给参数 Vf 赋值，我们有两个选择。第一是指定值的数组，例如：

```
StepVoltage Vs[5](Vf={24,26,28,30,32});
```

这将向量 {24,26,28,30,32} 的值分别赋给了 VS[1].Vf、VS[2].Vf、VS[3].Vf、VS[4].Vf 和 VS[5].Vf。另一个选择是给予该数组中的每个元素相同的值。我们可以使用和上面相同的数组初始化语法，如：

```
StepVoltage Vs[5](Vf={24,24,24,24,24});
```

但若的数组大小取决于 parameter 的话，问题就来了，如：

```
parameter Integer n;
StepVoltage Vs[n](Vf=/* ??? */);
```

如果我们试图用代码数组（如: {24,24,24}），那么这些代码就不会自动适应不同的 n。为了解决这个问题，我们可以使用 `fill` ( 97) 函数:

```
parameter Integer n;
StepVoltage Vs[n](Vf=fill(24, n));
```

这是个可以接受的方案。但想象一下，如果我们同时希望修改 Vf 的值以及 Vf 内的 min 属性呢？我们最终会得到这样的语句:

```
parameter Integer n;
StepVoltage Vs[n](Vf(min=fill(0,n))=fill(24, n));
```

一个个嵌套的修改会让这些定义会很快变得十分复杂。好在 Modelica 的包括一个处理这种情况的特性。通过在修改语句前添加 `each` 关键字，可以让修改语句影响每一个实例，例如:

```
parameter Integer n;
StepVoltage Vs[n](each Vf(min=0)=24);
```

修改语句是建模的重要部分。因为这些语句能让我们通过层级向下修改参数值。正如你在本节中的例子中所看到的一样，Modelica 语言提供了许多相关特性。这些特性能让应用在层次结构里的修改语句变得简单而功能强大。

## 第 8.2.4 节 传值

在建立子系统模型时，子系统很经常会包含一些会传递到其自身部件的参数。例如，考虑我们在对基本旋转组件 ( 185) 的讨论中使用的以下系统模型:

```
within ModelicaByExample.Components.Rotational.Examples;
model SMD
  Components.Damper damper2(d=1);
  Components.Ground ground;
  Components.Spring spring2(c=5);
  Components.Inertia inertia2(J=1,
    phi(fixed=true, start=1),
    w(fixed=true, start=0));
  Components.Damper damper1(d=0.2);
  Components.Spring spring1(c=11);
  Components.Inertia inertia1(
    J=0.4,
    phi(fixed=true, start=0),
    w(fixed=true, start=0));
equation
  // ...
end SMD;
```

有时候，我们会想在不同的上下文中使用这个模型，并改变一些部件参数的值，例如 d。我们可以让 d 成为子系统级别的参数，然后利用修改语句将值通过层次结构传递。结果应是如此:

```
within ModelicaByExample.Components.Rotational.Examples;
model SMD
  import Modelica.SIunits.*;
  parameter RotationalDampingConstant d;
  Components.Damper damper2(d=d);
  // ...
```

这里有一个问题。用户可能会去下层改变 `damper2.d` 的值，而不是在 SMD 模型修改 d 参数。要避免 d 参数和 `damper2.d` 参数的不同步（具有不同的值），我们可以使用 `final` 限定词将其永久绑定起来:

```
within ModelicaByExample.Components.Rotational.Examples;
model SMD
  import Modelica.SIunits.*;
  parameter final RotationalDampingConstant d;
```

```
Components.Damper damper2(final d=d);
// ...
```

通过添加 final 限定词，我们表明参数 damper2.d 的值不可以再修改。任何修改必须指向 d。

用相同方式代替在 SMD 模型里所有在代码里固定的数值，如下模型会得到很高的可重用性：

```
within ModelicaByExample.Components.Rotational.Examples;
model SMD
  import Modelica.SIunits.*;

  parameter RotationalDampingConstant d1, d2;
  parameter RotationalSpringConstant c1, c2;
  parameter Inertia J1, J2;
  parameter Angle phi1_init=0, phi2_init=0;
  parameter AngularVelocity w1_init=0, w2_init=0;

  Components.Damper damper2(final d=d2);
  Components.Ground ground;
  Components.Spring spring2(final c=c2);
  Components.Inertia inertia2(
    final J=J2,
    phi(fixed=true, final start=phi2_init),
    w(fixed=true, final start=w2_init));
  Components.Damper damper1(final d=d1);
  Components.Spring spring1(final c=c1);
  Components.Inertia inertia1(
    final J=J1,
    phi(fixed=true, final start=phi1_init),
    w(fixed=true, final start=w1_init));
equation
  // ...
end SMD;
```

如果我们想用某组特定的参数值，有两种可行的方法。一种方法是扩展上述的参数化模型，并在 extends 声明包括修改语句，如：

```
model SpecificSMD
  extends SMD(d2=1, c2=5, J2=1,
              d1=0.5, c1=11, J1=0.4,
              phi1_init=1);
```

需要注意的是，我们并不需要为 phi2\_init、w1\_init、w2\_init 添加修改语句，因为这些参数在声明都带有默认值。一般情况下，**只有在参数的默认值适用于绝大部分情况时，才应该设定**。这样做的原因在于如果参数没有默认值，大多数的 Modelica 编译器会生成警告，以提醒你必须进行赋值。但是，如果存在默认值，编译器会默默地使用默认值。如果默认值不合理或不典型，那么你会默默地在模型里引入一个不合理的值。

另一方面，我们回到关于传值的主题。另一个可用的方法将是实例化 SMD 模型，并在声明的变量使用修改语句去指定参数值，如：

```
SMD mysmd(d2=1, c2=5, J2=1,
           d1=0.5, c1=11, J1=0.4,
           phi1_init=1);
```

在下章架构模型（285）之前，我们暂时不会讨论哪种方法更好。



## 架构模型

---

在本书的开头，我们讨论了表述方程式并将其转化为仿真模型。仅仅是这一点就非常有趣了。原因是，我们不再需要当心如何去解由此得到的线性或非线性方程组，也不需要考虑对生成的微分方程组进行积分。然而，将复杂系统表述成一条条的方程并不是一种可扩展的方案。

因此，我们会探索 Modelica 里一些用以创建部件模型的特性。我们可以用上述特性重用这些方程，而不再需要在每次使用的时候将其重写一遍。这种做法不仅让以我们可以把预先创建的（且多半是测试过的）部件模型组合为系统，而且可以通过 Modelica 的标准图形标注支持用图形化方式组合和表示系统。

这种方法同样会有可扩展性问题。原因是用部件进行复杂建模需要大量的拖拽以及连接部件的操作。再者，在没有任何层级的情况下系统模型会变得庞大而复杂。这又在另一个层面限制可扩展性。为了解决这一问题，我们不考虑如何将可重用的方程组成模型，而是定义包含可重用子系统组成的模型。这种做法减少了在构建复杂系统模型时的所需的大量的拖拽以及连接部件的操作。

上述的步骤演示了如何在建立系统模型时减少那些繁琐、费时且可能很易出错的工作。本章介绍了此进程的最后一步，即：如何使用架构架构是指包含了预连接的子系统集合的模型。架构模型可以通过简单选择子系统的某个具体实施（模型）来组成系统。这样，我们不仅不用提供方程，而且也没有任何拖拽或连接组件、子系统的必要。相反，我们只需要为每个子系统所使用选定模型就可以进行仿真了。

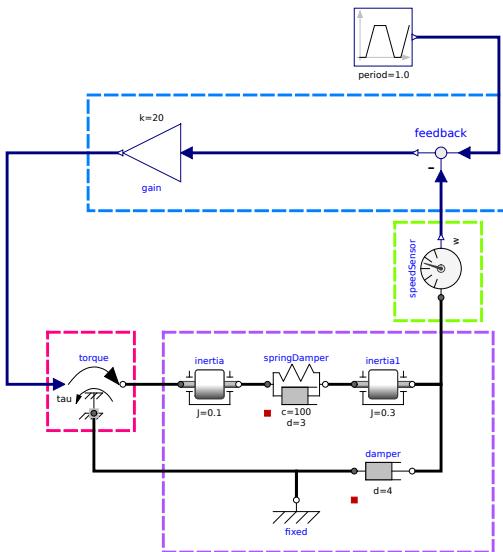
## 第 9.1 节 示例

### 第 9.1.1 节 传感器比较

我们会以一个例子开始对架构的研究。本例类似于我以前的书 [ItPMwM] ( 331) 里介绍过的另一个例子。在书里，我们将研究控制系统在使用几种不同的传感器模型时的性能。

#### 单层模型

我们的系统结构如下：



紫色框内组件代表受控对象的模型。在这种情况下，对象为经一个弹簧和阻尼器连接的两个转动惯量所组成的简单旋转系统。其中一个惯量由一个额外的阻尼器连接到旋转元素的地面参考系。绿框标示系统中的传感器。该传感器用于测量其中一个旋转轴的速度。类似地，紫框表示执行器。执行器对另一条轴（其速度没有测定的那条）施加扭矩。最后，蓝框内的所有组件代表了控制系统。该系统试图使所测速度尽可能地接近信号发生器在图的顶部所提供的设定值。

我们可以用 Modelica 代码表示如下：

```
within ModelicaByExample.Architectures.SensorComparison.Examples;
model FlatSystem "A rotational system with no architecture"
  Modelica.Mechanics.Rotational.Components.Fixed fixed
    annotation (Placement(transformation(extent={{10,-90},{30,-70}})));
  Modelica.Mechanics.Rotational.Components.Inertia inertia(J=0.1)
    annotation (Placement(transformation(extent={{-20,-50},{0,-30}}));
  Modelica.Mechanics.Rotational.Components.Inertia inertia1(J=0.3)
    annotation (Placement(transformation(extent={{40,-50},{60,-30}}));
  Modelica.Mechanics.Rotational.Sources.Torque torque(useSupport=true)
    annotation (Placement(transformation(extent={{-60,-50},{-40,-30}}));
  Modelica.Mechanics.Rotational.Components.SpringDamper springDamper(c=100, d=3)
    annotation (Placement(transformation(extent={{10,-50},{30,-30}}));
  Modelica.Mechanics.Rotational.Components.Damper damper(d=4)
    annotation (Placement(transformation(extent={{40,-80},{60,-60}}));
  Modelica.Mechanics.Rotational.Sensors.SpeedSensor speedSensor annotation (
    Placement(transformation(
      extent={{-10,-10},{10,10}}, rotation=90, origin={70,0})));
  Modelica.Blocks.Math.Feedback feedback annotation (Placement(transformation(
    extent={{10,-10},{-10,10}},
    origin={70,40})));
  Modelica.Blocks.Sources.Trapezoid trapezoid(period=1.0)
    annotation (Placement(transformation(extent={{40,70},{60,90}}));
  Modelica.Blocks.Math.Gain gain(k=20) annotation (Placement(transformation(
    extent={{-10,-10},{10,10}},
    rotation=180, origin={-30,40})));
equation
  connect(springDamper.flange_a, inertia.flange_b) annotation (Line(
    points={{10,-40},{0,-40}},
    color={0,0,0}, smooth=Smooth.None));
  connect(springDamper.flange_b, inertia1.flange_a) annotation (Line(
    points={{30,-40},{40,-40}},
    color={0,0,0}, smooth=Smooth.None));
  connect(torque.support, fixed.flange) annotation (Line(
    points={{-50,-50},{-50,-70},{20,-70},{20,-80}},
    color={0,0,0}, smooth=Smooth.None));
  connect(damper.flange_b, inertia1.flange_b) annotation (Line(
    points={{40,-80},{60,-60}},
    color={0,0,0}, smooth=Smooth.None));
```

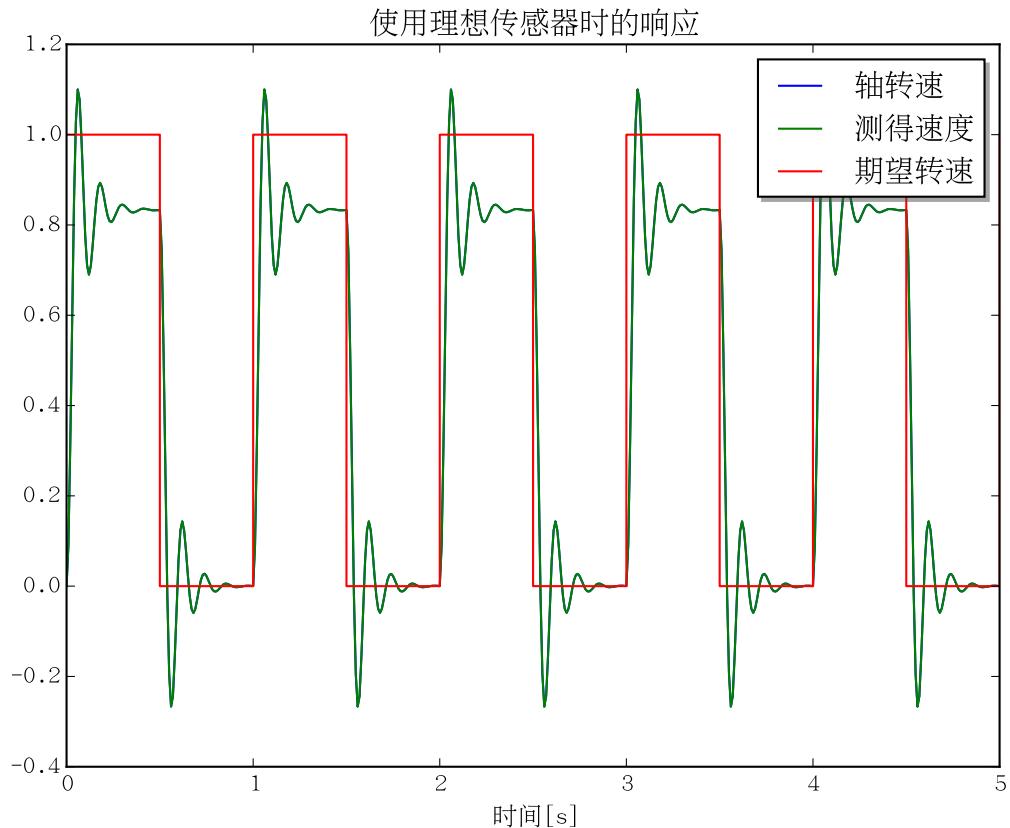
```

points={{60,-70},{70,-70},{70,-40},{60,-40}},
color={0,0,0}, smooth=Smooth.None));
connect(damper.flange_a, fixed.flange) annotation (Line(
points={{40,-70},{20,-70},{20,-80}},
color={0,0,0}, smooth=Smooth.None));
connect(torque.flange, inertia.flange_a) annotation (Line(
points={{-40,-40},{-20,-40}},
color={0,0,0}, smooth=Smooth.None));
connect(speedSensor.flange, inertia1.flange_b) annotation (Line(
points={{70,-10},{70,-40},{60,-40}},
color={0,0,0}, smooth=Smooth.None));
connect(feedback.y, gain.u) annotation (Line(
points={{61,40},{-18,40}},
color={0,0,127}, smooth=Smooth.None));
connect(gain.y, torque.tau) annotation (Line(
points={{-41,40},{-80,40},{-80,-40},{-62,-40}},
color={0,0,127}, smooth=Smooth.None));
connect(trapezoid.y, feedback.u1) annotation (Line(
points={{61,80},{90,80},{90,40},{78,40}},
color={0,0,127}, smooth=Smooth.None));
connect(speedSensor.w, feedback.u2) annotation (Line(
points={{70,11},{70,32}},
color={0,0,127}, smooth=Smooth.None));
annotation (
Diagram(graphics={
Rectangle(
extent={{-52,60},{94,20}}, lineColor={0,128,255},
pattern=LinePattern.Dash, lineThickness=0.5),
Rectangle(
extent={{54,16},{84,-18}}, lineColor={128,255,0},
pattern=LinePattern.Dash, lineThickness=0.5),
Rectangle(
extent={{-66,-22},{-36,-56}}, lineColor={255,0,128},
pattern=LinePattern.Dash, lineThickness=0.5),
Rectangle(
extent={{-26,-22},{88,-98}}, lineColor={170,85,255},
pattern=LinePattern.Dash, lineThickness=0.5}),
experiment(StopTime=4));
end FlatSystem;

```

请注意，在这个特定模型里，sensor 组件是 Modelica.Mechanics.Rotational.Sensors 包内 SpeedSensor 模型的一个实例。这是个报告精确解轨迹的“理想”传感器。换句话说，该传感器不引入任何一种测量误差。

事实是在模拟后我们看到，即便使用精确的速度测量，控制系统仍不太能够跟踪给定的速度轨迹：



我们会在稍后讨论为什么。但问题显然不是由测量误差带来的，因为所测量的速度是恰好等于实际轴速度。

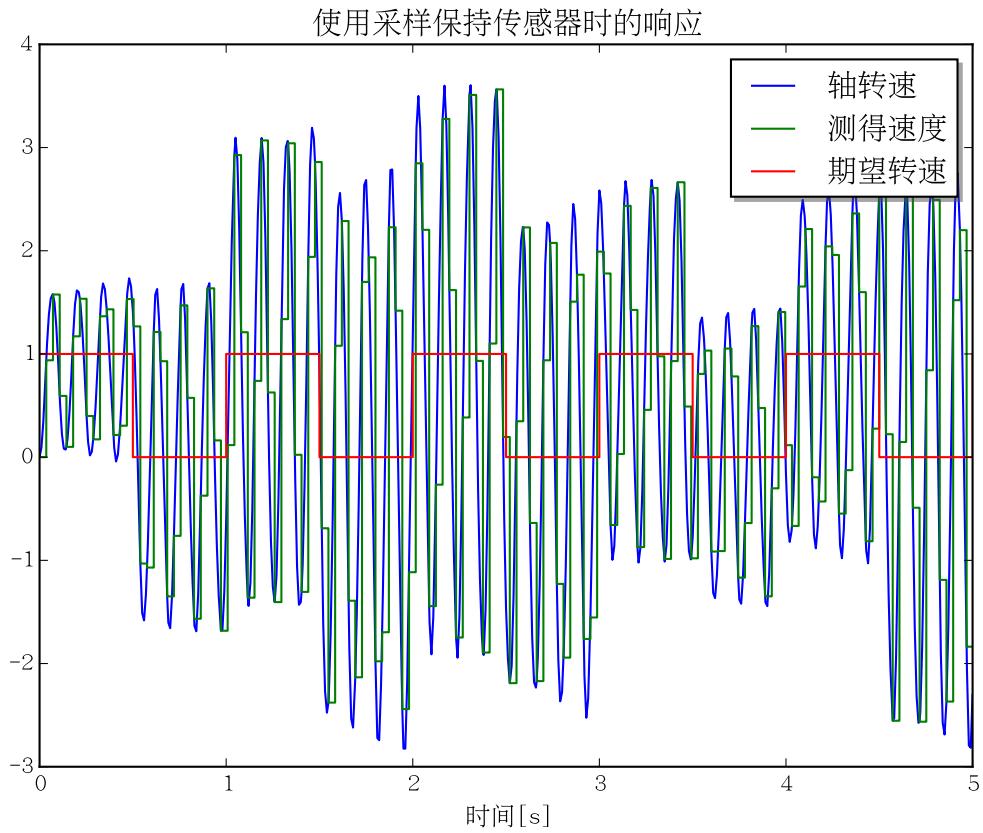
现在设想我们要用一个更现实的传感器模型，如之前编写的取样保持传感器（215）模型，以观察测量误差可能对系统性能有何额外影响。其中一点方法是将 FlatSystem 模型内的以下代码：

```
Modelica.Mechanics.Rotational.Sensors.SpeedSensor speedSensor annotation (
  Placement(transformation(
    extent={{-10,-10},{10,10}}, rotation=90, origin={70,0})));
Modelica.Blocks.Math.Feedback feedback annotation (Placement(transformation(
```

替换为：

```
Components.SpeedMeasurement.Components.SampleHold speedSensor(sample_rate=0.036)
annotation (Placement(transformation(
  extent={{-10,-10},{10,10}}, rotation=90, origin={70,0})));
Modelica.Blocks.Math.Feedback feedback annotation (Placement(transformation(
```

注意唯一的改变这里是 speedSensor 组件的类型。模拟这个系统，我们将看到控制系统以下的性能表现：

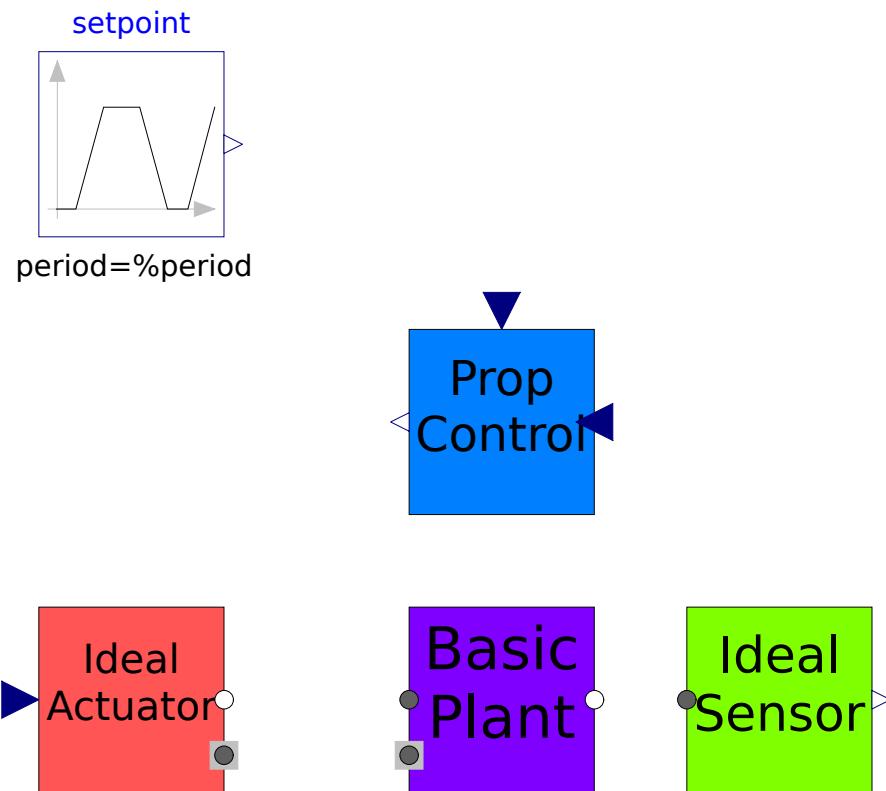


在这种情况下，我们可以系统性能每况愈下。虽然我们最初无法完全紧跟所需的速度，现在系统（由于测量误差）已变得不稳定。

### 带层级系统

在这里，我们想稍进一步探索上述性能问题，以一方面了解传感器的特性（如：sample\_rate）会如何影响系统性能。另一方面，我们也希望进一步考虑对于控制系统本身的改进。

如果我们要更换传感器、执行器和控制策略，那么第一步应该是将这些子系统组织为模型。如此，我们最终会得到了以下系统模型：



我们在系统级只会看到四个子系统。这些模型各自对应于刚才提到的子系统。我们的 Modelica 模型现在变得更为简单了，因为模型内仅有如下声明：

```
Implementation.BasicPlant plant
annotation (Placement(transformation(extent={{-10,-40},{10,-20}})));
Implementation.IdealActuator actuator
annotation (Placement(transformation(extent={{-50,-40},{-30,-20}})));
Implementation.IdealSensor sensor
annotation (Placement(transformation(extent={{20,-40},{40,-20}})));
Implementation.ProportionalController controller
annotation (Placement(transformation(extent={{-10,-10},{10,10}})));
Modelica.Blocks.Sources.Trapezoid setpoint(period=1.0)
annotation (Placement(transformation(extent={{-50,20},{-30,40}})));
```

每个子系统（plant、actuator、sensor 和 controller）是由 Implementations 包内的一个子系统模型来实现的。这是一种对模型的改进。原因是这意味着如果我们想改变控制器模型，我们可以简单通过改变与 controller 子系统所关联的的类型，然后系统就会采用另外这个子系统的另一种实现。相对于删除现有的控制器组件、拖拽新组件、然后（正确地！）重新连接所有关联部件这一系列繁复的步骤，这无疑是一种进步。

但是，我们在将传感器切换为采样保持版本后，仍然需要更改模型的文本，即：

```
Implementation.BasicPlant plant
annotation (Placement(transformation(extent={{-10,-40},{10,-20}})));
Implementation.IdealActuator actuator
annotation (Placement(transformation(extent={{-50,-40},{-30,-20}})));
Implementation.SampleHoldSensor sensor
annotation (Placement(transformation(extent={{20,-40},{40,-20}})));
Implementation.ProportionalController controller
annotation (Placement(transformation(extent={{-10,-10},{10,10}})));
Modelica.Blocks.Sources.Trapezoid setpoint(period=1.0)
annotation (Placement(transformation(extent={{-50,20},{-30,40}})));
```

此方案仍然有数个问题。首先，回想我们是通过更改类型名去改变子系统的实现。问题是：“什么类型的可以用在这里？”如果我将 sensor 子系统的类型改为 BasicPlant 呢？这会没有任何意义。但是，仅通过观察模型其实我们不会知道这点。但更大的问题是，为创建一个模型我们最终会得到两款几乎相同的模

型。正如我们在本书前面 (188) 所学到的一样，大家应该紧记 DRY (不要重复自己) 原则。而在这些模型中，我们看到了很多的冗余。

## 处理冗余

试想我们以 IdealSensor 模型为出发点建立层级模型：

```
within ModelicaByExample.Architectures.SensorComparison.Examples;
model HierarchicalSystem "Organizing components into subsystems"
  Implementation.BasicPlant plant
    annotation (Placement(transformation(extent={{-10,-40},{10,-20}})));
  Implementation.IdealActuator actuator
    annotation (Placement(transformation(extent={{-50,-40},{-30,-20}})));
  Implementation.IdealSensor sensor
    annotation (Placement(transformation(extent={{20,-40},{40,-20}})));
  Implementation.ProportionalController controller
    annotation (Placement(transformation(extent={{-10,-10},{10,10}})));
  Modelica.Blocks.Sources.Trapezoid setpoint(period=1.0)
    annotation (Placement(transformation(extent={{-50,20},{-30,40}})));
equation
  connect(actuator.shaft, plant.flange_a) annotation (Line(
    points={{-30,-30},{-10,-30}},
    color={0,0,0},
    smooth=Smooth.None));
  connect(actuator.housing, plant.housing) annotation (Line(
    points={{-30,-36},{-10,-36}},
    color={0,0,0},
    smooth=Smooth.None));
  connect(plant.flange_b, sensor.shaft) annotation (Line(
    points={{10,-30},{20,-30}},
    color={0,0,0},
    smooth=Smooth.None));
  connect(controller.command, actuator.tau) annotation (Line(
    points={{-11,0},{-70,0},{-70,-30},{-52,-30}},
    color={0,0,127},
    smooth=Smooth.None));
  connect(sensor.w, controller.measured) annotation (Line(
    points={{41,-30},{60,-30},{60,0},{10,0}},
    color={0,0,127},
    smooth=Smooth.None));
  connect(setpoint.y, controller.setpoint) annotation (Line(
    points={{-29,30},{0,30},{0,12}},
    color={0,0,127},
    smooth=Smooth.None));
end HierarchicalSystem;
```

现在，我们要建立这种模式的变体，去更改其中的 sensor 组件。

在前面，我们通过继承处理冗余。我们当然可以使用继承来将 HierarchicalSystem 内的子系统转移到另一个模型，然后改变此模型的参数，如：

```
model Variation1
  extends HierarchicalSystem(setpoint(startTime=1.0));
end Variation1;
```

但我们并不希望改变参数，要改变的是类型。假设我们能以某种方式“覆盖”以前的选择，这样就可以做到如下的事情：

```
model Variation2 "What we'd like to do (but cannot)"
  extends HierarchicalSystem(setpoint(startTime=1.0));
  Implementation.SampleHoldSensor sensor
    annotation (Placement(transformation(extent={{20,-40},{40,-20}})));
end Variation2;
```

这就是我们实际上想要做的。但是，这不是合法的 Modelica 代码。再者这个方法有若干个其他问题。首先，原模型开发人员可能不希望允许这样的变化。第二个问题是，我们最终会有（不同类型的，且不会少于）两个 sensor 组件。即使模型真的“覆盖”了 sensor 组件其任何先前的声明，另外一个问题，我们可能会键入名称错误的变量名，最后系统里有两个传感器。最后，我们仍然没有办法知道，将 sensor 改成 SampleHoldSensor 是否有意义。在这里这样改是有意义的，但在一般情况下我们如何保证呢？

幸运的是，有一种方式可以几乎达到我们的目的。但为了解决这些附带问题，我们需要更严格地考虑这个问题。

首先，我们需要指示该组件允许被更换。为了达到这一点，我们可以通过如下方式声明 sensor：

```
replaceable Implementation.IdealSensor sensor
annotation (Placement(transformation(extent={{20,-40},{40,-20}})));
```

使用了 replaceable 关键字表明，我们继承该模型时，该变量的类型是可以改变的（或者说，可以“重新声明”）。但要记住，这个模型也有如下的语句：

```
connect(plant.flange_b, sensor.shaft);
connect(sensor.w, controller.measured);
```

这就带来了一个问题。如果我们更换的 sensor 组件没有 w 连接器，那么会发生什么？在这种情况下，这个 connect 语句将产生错误。此时，我们会说这两种传感器型号不是插件兼容的。当对于 Y 的每个公有变量 X 都有一个具有相同的名字的对应变量，模型 X 插件兼容于 Y 模型。此外，在 X 内每个这样的变量本身必须是插件兼容它在 Y 内的对应变量。这样可以确保如果你将 Y 类型的组件变更为类型 X，你需要的一切（参数，连接器等）都将继续存在，并仍将是兼容的。不过，请注意，倘若 X 是插件兼容于 Y，这不意味着 Y 插件兼容于 X（我们将马上看到这样的例子）。

“插件兼容性”是非常重要的。原因是在一般情况下，我们希望确保任何重声明都是“安全的”。要做到这一点，我们必须确定在更改 sensor 组件时，使用的任何类型均插件兼容于原来的类型。在这种情况下，这意味着这个类型必须有一个 w 连接器（同样地，这个连接器必须插件兼容于重声明前的 w）。另外，这个类型必须有一个 shaft 连接器（同上，其必须插件兼容于以前的 shaft）。

所以接下来的问题是，我们的 SampleHoldSensor 能否实现满足插件兼容性要求？此模型是插件兼容于 IdealSensor 模型么？首先，让我们来看看 IdealSensor 模型：

```
within ModelicaByExample.Architectures.SensorComparison.Implementation;
model IdealSensor "Implementation of an ideal sensor"
  Modelica.Mechanics.Rotational.Interfaces.Flange_a shaft
    "Flange of shaft from which sensor information shall be measured"
    annotation (Placement(transformation(extent={{-110,-10},{-90,10}})));
  Modelica.Blocks.Interfaces.RealOutput w "Absolute angular velocity of flange"
    annotation (Placement(transformation(extent={{100,-10},{120,10}})));
protected
  Modelica.Mechanics.Rotational.Sensors.SpeedSensor idealSpeedSensor
    "An ideal speed sensor" annotation (Placement(transformation(
      extent={{-10,-10},{10,10}}));
```

此组件的公有接口包含两个连接器：w 和 shaft。观察 SampleHoldSensor 模型：

```
within ModelicaByExample.Architectures.SensorComparison.Implementation;
model IdealSensor "Implementation of an ideal sensor"
  Modelica.Mechanics.Rotational.Interfaces.Flange_a shaft
    "Flange of shaft from which sensor information shall be measured"
    annotation (Placement(transformation(extent={{-110,-10},{-90,10}})));
  Modelica.Blocks.Interfaces.RealOutput w "Absolute angular velocity of flange"
    annotation (Placement(transformation(extent={{100,-10},{120,10}})));
protected
  Modelica.Mechanics.Rotational.Sensors.SpeedSensor idealSpeedSensor
    "An ideal speed sensor" annotation (Placement(transformation(
      extent={{-10,-10},{10,10}}));
```

```
within ModelicaByExample.Architectures.SensorComparison.Implementation;
model SampleHoldSensor "Implementation of a sample hold sensor"
  parameter Modelica.SIunits.Time sample_rate(min=Modelica.Constants.eps);
```

```

Modelica.Mechanics.Rotational.Interfaces.Flange_a shaft
  "Flange of shaft from which sensor information shall be measured"
  annotation (Placement(transformation(extent={{-110,-10},{-90,10}})));
Modelica.Blocks.Interfaces.RealOutput w "Absolute angular velocity of flange"
  annotation (Placement(transformation(extent={{100,-10},{120,10}})));
protected
  Components.SpeedMeasurement.Components.SampleHold sampleHoldSensor(
    sample_rate=sample_rate)
  annotation (Placement(transformation(extent={{-10,-10},{10,10}})));

```

我们可以看到，其公有接口也包含连接器 w“和 ‘‘ shaft。此外，两者与 IdealSensor 模型上的连接器具有完全相同的类型。因此，SampleHoldSensor 模型插件兼容于 IdealSensor 模型。所以我们应该可以用 SampleHoldSensor 替换 IdealSensord 的实例。在替换后，我们的 connect 语句仍然有效。

那么，如果我们的 HierarchicalSystem 模型声明如下：

```

within ModelicaByExample.Architectures.SensorComparison.Examples;
model HierarchicalSystem "Organizing components into subsystems"
  replaceable Implementation.BasicPlant plant
    annotation (Placement(transformation(extent={{-10,-40},{10,-20}})));
  replaceable Implementation.IdealActuator actuator
    annotation (Placement(transformation(extent={{-50,-40},{-30,-20}})));
  replaceable Implementation.IdealSensor sensor
    annotation (Placement(transformation(extent={{20,-40},{40,-20}})));
  replaceable Implementation.ProportionalController controller
    annotation (Placement(transformation(extent={{-10,-10},{10,10}})));
  replaceable Modelica.Blocks.Sources.Trapezoid setpoint
    annotation (Placement(transformation(extent={{-50,20},{-30,40}})));
  // ...
end HierarchicalSystem;

```

然后我们可以以如下方式实现最初创建此模型变体的目的，而不必重复：

```

model Variation3 "DRY redeclaration"
  extends HierarchicalSystem(
    redeclare Implementation.SampleHoldSensor sensor
  );
end Variation3;

```

上述模型的还有几点值得注意。首先是重声明的语法和正常声明几乎相同。不同在于，其前面加上了 redeclare 关键字。还要注意的是，重新声明是 extends 子句的一部分。具体地讲，重声明语句如同其他所有修改一样，是一个在扩展子句内的更改。如果我们想既重声明 sensor 组件又要改变我们工作点的 startTime 参数，则两者均为 extends 子句内的更改，如：

```

model Variation3 "DRY redeclaration"
  extends HierarchicalSystem(
    setpoint(startTime=1.0),
    redeclare Implementation.SampleHoldSensor sensor
  );
end Variation3;

```

## 约束类型

回想一下，在本节的前面，SampleHoldSensor 模型的公有接口包括：

```

parameter Modelica.SIunits.Time sample_rate=0.01;
Modelica.Mechanics.Rotational.Interfaces.Flange_a shaft;
Modelica.Blocks.Interfaces.RealOutput w;

```

而 IdealSensor 模型的公有接口则只有：

```
Modelica.Mechanics.Rotational.Interfaces.Flange_a shaft;
Modelica.Blocks.Interfaces.RealOutput w;
```

如果重声明受限于新类型和原始类型的插件兼容性，那么我们可能会遇到下面的问题。万一我们在系统初始模型中使用了 SampleHoldSensor 传感器，即：

```
within ModelicaByExample.Architectures.SensorComparison.Examples;
model InitialSystem "Organizing components into subsystems"
  replaceable Implementation.BasicPlant plant;
  replaceable Implementation.IdealActuator actuator;
  replaceable Implementation.SampleHoldSensor sensor;
  replaceable Implementation.ProportionalController controller;
  replaceable Modelica.Blocks.Sources.Trapezoid setpoint;
equation
  // ...
  connect(plant.flange_b, sensor.shaft);
  connect(sensor.w, controller.measured);
  // ...
end InitialSystem;
```

进一步试想，我们想将 sensor 组件重定义为 dealSensor，如：

```
model Variation4
  extends InitialSystem(
    setpoint(startTime=1.0),
    redeclare Implementation.IdealSensor sensor // illegal
  );
end Variation4;
```

现在我们有一个问题。原来的 sensor 组件有个名为 sample\_rate 的参数。但是，我们试图把该组件替代没有这个参数的类型。换句话说，IdealSensor 模型不插件兼容于 SampleHoldSensor 模型。因为此新模型缺少一些原模型 SampleHoldSensor 具有的内容：sample\_rate。

但是，当我们观察 InitialSystem 模型的源代码后，我们看到了 sample\_rate 参数从未使用过。因此，并不存在真正的理由阻止类型转换。出于这个原因，Modelica 包括约束类型这个概念。

要了解重声明，重要的是需要清楚，其实有两个重要类型与原声明相关。第一个重要类型是原声明的类型。第二则是什么类型有可能，而且可让系统运行。这第二个类型被称为约束类型。因为只要任何重声明的类型插件兼容于约束类型，该模型应该仍然工作。这第二类叫做约束类型，因为只要任何重新声明是插件兼容约束类型，模型应该仍能正常工作。因此，上述的 InitialSystem 模型内原声明的类型是 SampleHoldSensor。但只要新类型的插件兼容 IdealSensor，该模型仍然能工作。

当我们表明一个组件是 replaceable 时，我们可以在最末加入 constrainedby 去指示约束型，如：

```
replaceable Implementation.SampleHoldSensor sensor
  constrainedby Implementation.IdealSensor;
```

上述声明的意思是 sensor 组件可以通过任何的插件兼容于 IdealSensor 的模型来重声明。但是如果不行重声明，那么默认会声明为 SampleHoldSensor 传感器。出于这个原因，声明 SampleHoldSensor 时使用的原始类型被称为默认类型。

注意，我们原来的 InitialSystem 模型的定义并没有指定约束类型。模型只指定了初始类型。在这种情况下，默认类型和约束类型被假定为初始类型。

在下一节讨论了如何使用自上而下架构驱动方法 (294) 去开发这样的系统模型时，我们将继续使用相同系统架构。

### 第 9.1.2 节 架构驱动方法

到目前为止，我们从扁平结构的建模方法为起点，逐渐在模型里使用 Modelica 的架构特性。我们会开始从上往下地用架构方法去建立系统。我们首先定义系统的结构，然后添加特定的子系统实现。

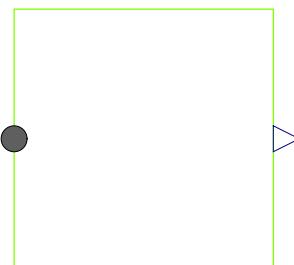
## 接口

我们希望首先建立顶层的架构模型，从而描述存在的子系统以及它们之间的连接。但是，我们需要把一些内容放入这个架构以表示我们的子系统。我们（暂时）不要直接为所有的子系统模型的建立实现。所以，我们从哪里开始？

答案是，我们会首先描述子系统的接口模型。还记得么？从本节前面里，重声明取决于约束类型，而且所有的实现必须和该约束类型具有一致性。其实接口模型基本上是约束类型的正式定义（即预期的公有接口）。不过接口模型并不包括实现细节。既然如此（即没有公式或子组件），因此接口模型是 partial 模型。但是，对于我们的目的这足够了。

让我们开始考虑 sensor 子系统的接口模型。我们已经详细讨论了公共接口。这是接口模型的 Modelica 定义以及图标：

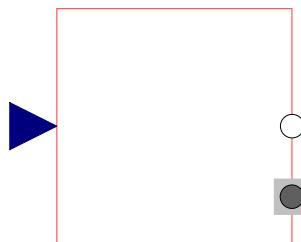
```
within ModelicaByExample.Architectures.SensorComparison.Interfaces;
partial model Sensor "Interface for sensor"
  Modelica.Mechanics.Rotational.Interfaces.Flange_a shaft
    "Flange of shaft from which sensor information shall be measured"
    annotation (Placement(transformation(extent={{-110,-10},{-90,10}})));
  Modelica.Blocks.Interfaces.RealOutput w "Absolute angular velocity of flange"
    annotation (Placement(transformation(extent={{100,-10},{120,10}})));
  annotation (Icon(graphics={Rectangle(extent={{-100,100},{100,-100}}}, lineColor={128,255,0})));
end Sensor;
```



这个 model 定义有几点需要注意。首先，正如我们刚才也提到，这种模型是 partial。这是因为这种模型仅具有连接器，但没有方程去帮助求解对这些连接器内的变量。另外值得注意的一点是，这些模型包含了批注。与连接器声明相关的标注说明这些连接器应该以如何形式表现。任何从这个 extends 的模型会继承这些标注。所以，连接器的位置必须在所有的实现里都有效。It also includes an Icon annotation. 模型还包括一个 Icon 标注。这实际上定义了最终实现的图标“背景”。换句话说，从此模型扩展的模型可以在 Sensor 模型图标的基础上中定义额外的图形。

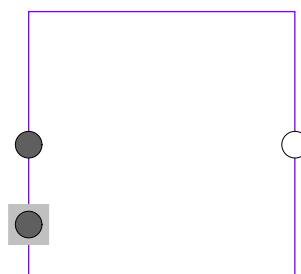
而执行器模型的接口定义唯一不同之处在于，它包括了两个旋转连接器。其中一个用以施加指令转矩，另一个则用以处理反作用转矩。倘若例如我们的执行器模型为电动马达，前者就是在转子的连接器，而后者是定子的连接器。除此以外，此模型非常类似于我们 Sensor 定义：

```
within ModelicaByExample.Architectures.SensorComparison.Interfaces;
partial model Actuator "Interface for actuator"
  Modelica.Mechanics.Rotational.Interfaces.Flange_b shaft "Output shaft"
    annotation (Placement(transformation(extent={{90,-10},{110,10}})));
  Modelica.Mechanics.Rotational.Interfaces.Support housing
    "Connection to housing"
    annotation (Placement(transformation(extent={{90,-70},{110,-50}})));
  Modelica.Blocks.Interfaces.RealInput tau "Input torque command"
    annotation (Placement(transformation(extent={{-140,-20},{-100,20}})));
  annotation (Icon(graphics={Rectangle(extent={{-100,100},{100,-100}}}, lineColor={255,85,85})));
end Actuator;
```



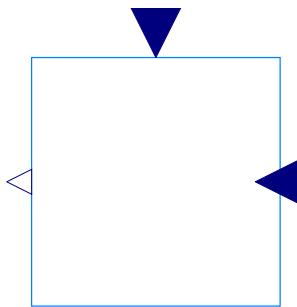
Plant 接口有三个旋转连接器。一个在“输入”侧（与执行器相连接）。一个在“输出”侧（与传感器相连接）。而最后一个用于“支持”侧（用以“固定”在其他物体上）：

```
within ModelicaByExample.Architectures.SensorComparison.Interfaces;
partial model Plant "Interface for plant model"
  Modelica.Mechanics.Rotational.Interfaces.Flange_b flange_b
    "Output shaft of plant"
    annotation (Placement(transformation(extent={{90,-10},{110,10}})));
  Modelica.Mechanics.Rotational.Interfaces.Flange_a flange_a
    "Input shaft for plant"
    annotation (Placement(transformation(extent={{-110,-10},{-90,10}})));
  Modelica.Mechanics.Rotational.Interfaces.Support_housing
    "Connection to mounting"
    annotation (Placement(transformation(extent={{-110,-70},{-90,-50}}));
    annotation (Icon(graphics={Rectangle(extent={{-100,100},{100,-100}}}, lineColor={128,0,255})));
  end Plant;
```



最后，我们有 Controller 接口定义：

```
within ModelicaByExample.Architectures.SensorComparison.Interfaces;
partial model Controller "Interface for controller subsystem"
  Modelica.Blocks.Interfaces.RealInput setpoint "Desired system response"
    annotation (Placement(transformation(
      extent={{-20,-20},{20,20}},
      rotation=270,
      origin={0,120})));
  Modelica.Blocks.Interfaces.RealInput measured "Actual system response"
    annotation (Placement(transformation(
      extent={{-20,-20},{20,20}},
      rotation=180,
      origin={100,0})));
  Modelica.Blocks.Interfaces.RealOutput command "Command to send to actuator"
    annotation (Placement(transformation(
      extent={{-10,-10},{10,10}},
      rotation=180,
      origin={-110,0})));
    annotation (Icon(graphics={Rectangle(extent={{-100,100},{100,-100}}}, lineColor={0,128,255})));
  end Controller;
```



无论控制器是如何实现的（如比例控制，PID 控制），Controller 必须有目标值 setpoint 的输入连接器、另一个用于测速的输入连接器 measured 和用以发送指令转矩 command 到执行器的输出连接器。

## 架构

指定了接口后，我们的架构模型可以写成如下形式：

```
within ModelicaByExample.Architectures.SensorComparison.Examples;
partial model SystemArchitecture
  "A system architecture built from subsystem interfaces"

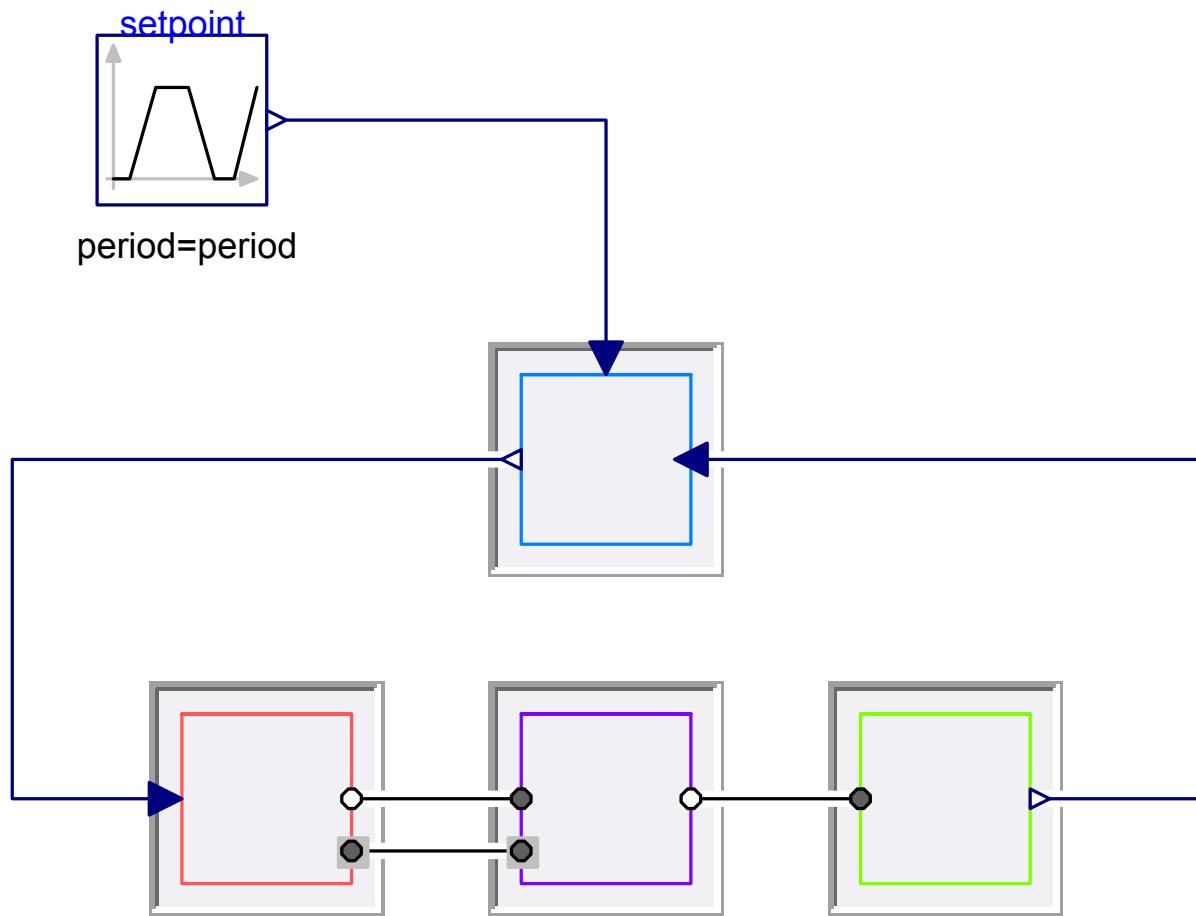
  replaceable Interfaces.Plant plant
  annotation (choicesAllMatching=true,
    Placement(transformation(extent={{-10,-50},{10,-30}})));
  replaceable Interfaces.Actuator actuator
  annotation (choicesAllMatching=true,
    Placement(transformation(extent={{-50,-50},{-30,-30}})));
  replaceable Interfaces.Sensor sensor
  annotation (choicesAllMatching=true,
    Placement(transformation(extent={{30,-50},{50,-30}})));
  replaceable Interfaces.Controller controller
  annotation (choicesAllMatching=true,
    Placement(transformation(extent={{-10,-10},{10,10}}));
    Modelica.Blocks.Sources.Trapezoid setpoint(period=1.0)
    annotation (choicesAllMatching=true,
      Placement(transformation(extent={{-60,30},{-40,50}}));
equation
  connect(actuator.shaft, plant.flange_a) annotation (Line(
    points={{-30,-40},{-10,-40}},
    color={0,0,0},
    smooth=Smooth.None));
  connect(actuator.housing, plant.housing) annotation (Line(
    points={{-30,-46},{-10,-46}},
    color={0,0,0},
    smooth=Smooth.None));
  connect(plant.flange_b, sensor.shaft) annotation (Line(
    points={{10,-40},{30,-40}},
    color={0,0,0},
    smooth=Smooth.None));
  connect(controller.measured, sensor.w) annotation (Line(
    points={{10,0},{70,0},{70,-40},{51,-40}},
    color={0,0,127},
    smooth=Smooth.None));
  connect(controller.command, actuator.tau) annotation (Line(
    points={{-11,0},{-70,0},{-70,-40},{-52,-40}},
    color={0,0,127},
    smooth=Smooth.None));
  connect(setpoint.y, controller.setpoint) annotation (Line(
    points={{-39,40},{0,40},{0,12}},
    color={0,0,127},
    smooth=Smooth.None));
```

```
end SystemArchitecture;
```

我们可以从上述 Modelica 代码看出，架构由四个 replaceable 子系统组成：plant，actuator，sensor 和 setpoint。所有这些声明只包括一种类型。正如我们先前在本节了解到的，该类型将（如我们所希望地）同时用作默认类型以及约束类型。此模型还包括子系统之间的所有连接。这样一来，模型完整描述了子系统以及其之间的联系。万事俱备，唯一缺少的就是去选择每个子系统的实现了。

请注意，SystemArchitecture 模型本身是 partial。这正是因为所有子系统的默认类型均为 partial，而我们还没有指定模型的实现。换句话说，模型不包含实现，因此（还）不能进行模拟。为表明这一点，我们标记模型为 partial。

正如我们的接口模型，这个模型也包含图形标注。这是因为我们不但指定了子系统，还指定了子系统的位置和连接的路径。我们的系统架构显式的效果如下：



## 实现

现在我们有了接口和架构，我们必须创建一些实现，用以“注入”到架构中。这些实现可以选择从现有的接口继承（从而避免冗余代码），或是仅确保在实现内的声明与接口插件兼容。显然在一般情况下从接口继承是一个更好的方法。但我们同时介绍这两种方法的例子，用以证明从接口继承并非必须。

这里有一些我们将在本节剩余部分使用的实现。请注意，尽管这些模型包括了多行的 Modelica 源代码，代码可以在图形化的 Modelica 环境内迅速生成（即通常不会手工键入这种模型代码）。

## 受控对象模型

```
within ModelicaByExample.Architectures.SensorComparison.Implementation;
model BasicPlant "Implementation of the basic plant model"
  parameter Modelica.SIunits.Inertia J_a=0.1 "Moment of inertia";
```

```

parameter Modelica.SIunits.Inertia J_b=0.3 "Moment of inertia";
parameter Modelica.SIunits.RotationalSpringConstant c=100 "Spring constant";
parameter Modelica.SIunits.RotationalDampingConstant d_shaft=3
  "Shaft damping constant";
parameter Modelica.SIunits.RotationalDampingConstant d_load=4
  "Load damping constant";

Modelica.Mechanics.Rotational.Interfaces.Support housing
  "Connection to mounting"
  annotation (Placement(transformation(extent={{-110,-70},{-90,-50}})));
Modelica.Mechanics.Rotational.Interfaces.Flange_a flange_a
  "Input shaft for plant"
  annotation (Placement(transformation(extent={{-110,-10},{-90,10}})));
Modelica.Mechanics.Rotational.Interfaces.Flange_b flange_b
  "Output shaft of plant"
  annotation (Placement(transformation(extent={{90,-10},{110,10}})));
protected
  Modelica.Mechanics.Rotational.Components.Fixed fixed
    annotation (Placement(transformation(extent={{-10,-80},{10,-60}}));
  Modelica.Mechanics.Rotational.Components.Inertia inertia(J=J_a)
    annotation (Placement(transformation(extent={{-40,-10},{-20,10}}));
  Modelica.Mechanics.Rotational.Components.Inertia inertia1(J=J_b)
    annotation (Placement(transformation(extent={{20,10},{40,10}}));
  Modelica.Mechanics.Rotational.Components.SpringDamper springDamper(c=c, d=
    d_shaft)
    annotation (Placement(transformation(extent={{-10,-10},{10,10}}));
  Modelica.Mechanics.Rotational.Components.Damper damper(d=d_load)
    annotation (Placement(transformation(extent={{20,-40},{40,-20}}));
equation
  connect(springDamper.flange_a, inertia.flange_b) annotation (Line(
    points={{-10,0},{-20,0}},
    color={0,0,0},
    smooth=Smooth.None));
  connect(springDamper.flange_b, inertia1.flange_a) annotation (Line(
    points={{10,0},{20,0}},
    color={0,0,0},
    smooth=Smooth.None));
  connect(damper.flange_b, inertia1.flange_b) annotation (Line(
    points={{40,-30},{50,-30},{50,0},{40,0}},
    color={0,0,0},
    smooth=Smooth.None));
  connect(damper.flange_a, fixed.flange) annotation (Line(
    points={{20,-30},{0,-30},{0,-70}},
    color={0,0,0},
    smooth=Smooth.None));
  connect(inertia1.flange_b, flange_b) annotation (Line(
    points={{40,0},{100,0}},
    color={0,0,0},
    smooth=Smooth.None));
  connect(inertia.flange_a, flange_a) annotation (Line(
    points={{-40,0},{-100,0}},
    color={0,0,0},
    smooth=Smooth.None));
  connect(fixed.flange, housing) annotation (Line(
    points={{0,-70},{0,-60},{-100,-60}},
    color={0,0,0},
    smooth=Smooth.None));
  annotation (Icon(graphics={
    Rectangle(

```

## 执行器模型

```

within ModelicaByExample.Architectures.SensorComparison.Implementation;
model IdealActuator "An implementation of an ideal actuator"
  Modelica.Mechanics.Rotational.Interfaces.Flange_b shaft "Output shaft"
    annotation (Placement(transformation(extent={{90,-10},{110,10}})));
  Modelica.Mechanics.Rotational.Interfaces.Support housing
    "Connection to housing"
    annotation (Placement(transformation(extent={{90,-70},{110,-50}})));
protected
  Modelica.Mechanics.Rotational.Sources.Torque torque(useSupport=true)
    annotation (Placement(transformation(extent={{-10,-10},{10,10}}));
  Modelica.Blocks.Interfaces.RealInput tau "Input torque command"
    annotation (Placement(transformation(extent={{-140,-20},{-100,20}}));
equation
  connect(torque.flange, shaft) annotation (Line(
    points={{10,0},{100,0}},
    color={0,0,0},
    smooth=Smooth.None));
  connect(torque.support, housing) annotation (Line(
    points={{0,-10},{0,-60},{100,-60}},
    color={0,0,0},
    smooth=Smooth.None));
  connect(torque.tau, tau) annotation (Line(
    points={{-12,0},{-120,0}},
    color={0,0,127},
    smooth=Smooth.None));
  annotation (Icon(graphics={
    Rectangle(
      extent={{-100,100},{100,-100}}),

```

```

within ModelicaByExample.Architectures.SensorComparison.Implementation;
model LimitedActuator "An actuator with lag and saturation"
  extends Interfaces.Actuator;
  parameter Modelica.SIunits.Time delayTime
    "Delay time of output with respect to input signal";
  parameter Real uMax "Upper limits of input signals";
protected
  Modelica.Mechanics.Rotational.Sources.Torque torque(useSupport=true)
    annotation (Placement(transformation(extent={{30,-10},{50,10}}));
  Modelica.Blocks.Nonlinear.Limiter limiter(uMax=uMax)
    annotation (Placement(transformation(extent={{-18,-10},{2,10}}));
  Modelica.Blocks.Nonlinear.FixedDelay lag(delayTime=delayTime)
    annotation (Placement(transformation(extent={{-70,-10},{-50,10}}));
equation
  connect(torque.flange, shaft) annotation (Line(
    points={{50,0},{100,0}},
    color={0,0,0},
    smooth=Smooth.None));
  connect(torque.support, housing) annotation (Line(
    points={{40,-10},{40,-60},{100,-60}},
    color={0,0,0},
    smooth=Smooth.None));
  connect(limiter.y, torque.tau) annotation (Line(
    points={{3,0},{28,0}},
    color={0,0,127},
    smooth=Smooth.None));
  connect(lag.u, tau) annotation (Line(
    points={{-72,0},{-120,0}},
    color={0,0,127},
    smooth=Smooth.None));
  connect(lag.y, limiter.u) annotation (Line(

```

```

    points={{-49,0},{-20,0}},
    color={0,0,127},
    smooth=Smooth.None));
annotation (Icon(graphics={
    Rectangle(
        extent={{-100,100},{100,-100}},
        lineColor={0,0,0},
        fillColor={255,85,85},

```

## 控制器模型

```

within ModelicaByExample.Architectures.SensorComparison.Implementation;
model ProportionalController "Implementation of a proportional controller"
  parameter Real k=20 "Controller gain";
  Modelica.Blocks.Interfaces.RealInput setpoint "Desired system response"
    annotation (Placement(transformation(
      extent={{-20,-20},{20,20}},
      rotation=270, origin={0,120})));
  Modelica.Blocks.Interfaces.RealInput measured "Actual system response"
    annotation (Placement(transformation(
      extent={{-20,-20},{20,20}},
      rotation=180, origin={100,0})));
  Modelica.Blocks.Interfaces.RealOutput command "Command to send to actuator"
    annotation (Placement(transformation(
      extent={{-10,-10},{10,10}},
      rotation=180, origin={-110,0})));
protected
  Modelica.Blocks.Math.Gain gain(k=k)
    annotation (Placement(transformation(
      extent={{-10,-10},{10,10}},
      rotation=180, origin={-50,0})));
  Modelica.Blocks.Math.Feedback feedback
    annotation (Placement(transformation(
      extent={{10,-10},{-10,10}})));
equation
  connect(feedback.y, gain.u) annotation (Line(
    points={{-9,0},{2,0},{2,0},{-38,0}},
    color={0,0,127}, smooth=Smooth.None));
  connect(feedback.u1, setpoint) annotation (Line(
    points={{8,0},{40,0},{40,60},{0,60},{0,120}},
    color={0,0,127},
    smooth=Smooth.None));
  connect(gain.y, command) annotation (Line(
    points={{-61,0}, {-80.5,0}, {-80.5,0}, {-110,0}},
    color={0,0,127},
    smooth=Smooth.None));
  connect(measured, feedback.u2) annotation (Line(
    points={{100,0},{60,0},{60,-40},{0,-40},{0,-8}},
    color={0,0,127}, smooth=Smooth.None));
end ProportionalController;

```

```

within ModelicaByExample.Architectures.SensorComparison.Implementation;
model PID_Controller "Controller subsystem implemented using a PID controller"
  extends Interfaces.Controller;
  parameter Real k "Gain of controller";
  parameter Modelica.SIunits.Time Ti "Time constant of Integrator block";
  parameter Modelica.SIunits.Time Td "Time constant of Derivative block";
  parameter Real yMax "Upper limit of output";
protected
  Modelica.Blocks.Continuous.LimPID PID(k=k, Ti=Ti, Td=Td, yMax=yMax)
    annotation (Placement(transformation(

```

```

    extent={{10,-10},{-10,10}})));
equation
  connect(setpoint, PID.u_s) annotation (Line(
    points={{0,120},{0,60},{40,60},{40,0},{12,0}},
    color={0,0,127},
    smooth=Smooth.None));
  connect(measured, PID.u_m) annotation (Line(
    points={{100,0},{60,0},{60,-40},{0,-40},{0,-12}},
    color={0,0,127},
    smooth=Smooth.None));
  connect(PID.y, command) annotation (Line(
    points={{-11,0},{-110,0}},
    color={0,0,127},
    smooth=Smooth.None));
annotation (Icon(graphics={
  Rectangle(
    extent={{-100,100},{100,-100}}),

```

### 传感器模型

```

within ModelicaByExample.Architectures.SensorComparison.Implementation;
model IdealSensor "Implementation of an ideal sensor"
  Modelica.Mechanics.Rotational.Interfaces.Flange_a shaft
    "Flange of shaft from which sensor information shall be measured"
    annotation (Placement(transformation(extent={{-110,-10},{-90,10}})));
  Modelica.Blocks.Interfaces.RealOutput w "Absolute angular velocity of flange"
    annotation (Placement(transformation(extent={{100,-10},{120,10}})));
protected
  Modelica.Mechanics.Rotational.Sensors.SpeedSensor idealSpeedSensor
    "An ideal speed sensor" annotation (Placement(transformation(
      extent={{-10,-10},{10,10}})));
equation
  connect(idealSpeedSensor.flange, shaft) annotation (Line(
    points={{-10,0},{-100,0}},
    color={0,0,0},
    smooth=Smooth.None));
  connect(idealSpeedSensor.w, w) annotation (Line(
    points={{11,0},{110,0}},
    color={0,0,127},
    smooth=Smooth.None));
  annotation (Icon(graphics={
    Rectangle(

```

```

within ModelicaByExample.Architectures.SensorComparison.Implementation;
model SampleHoldSensor "Implementation of a sample hold sensor"
  parameter Modelica.SIunits.Time sample_rate(min=Modelica.Constants.eps);
  Modelica.Mechanics.Rotational.Interfaces.Flange_a shaft
    "Flange of shaft from which sensor information shall be measured"
    annotation (Placement(transformation(extent={{-110,-10},{-90,10}})));
  Modelica.Blocks.Interfaces.RealOutput w "Absolute angular velocity of flange"
    annotation (Placement(transformation(extent={{100,-10},{120,10}})));
protected
  Components.SpeedMeasurement.Components.SampleHold sampleHoldSensor(
    sample_rate=sample_rate)
    annotation (Placement(transformation(extent={{-10,-10},{10,10}})));
equation
  connect(sampleHoldSensor.w, w) annotation (Line(
    points={{11,0},{110,0}},
    color={0,0,127},
    smooth=Smooth.None));
  connect(sampleHoldSensor.flange, shaft) annotation (Line(

```

```

points={{-10,0},{-100,0}},
color={0,0,0},
smooth=Smooth.None));
annotation ( Icon(graphics={
    Rectangle(

```

## 变体

### 基准配置

有了这些实现，我们可以创建完整的系统的一些不同实现。举个例子，为了实现我们最初的 FlatSystem 模型的行为，我们可以直接扩展 SystemArchitecture 模型，然后将每个子系统重新声明为与 FlatSystem 对应的子系统实现，即：

```

within ModelicaByExample.Architectures.SensorComparison.Examples;
model BaseSystem "System architecture with base implementations"
  extends SystemArchitecture(
    redeclare replaceable Implementation.ProportionalController controller,
    redeclare replaceable Implementation.IdealActuator actuator,
    redeclare replaceable Implementation.BasicPlant plant,
    redeclare replaceable Implementation.IdealSensor sensor);
end BaseSystem;

```

在这里，我们看到的 Modelica 指定配置的能力。注意了每个重声明都包括了 replaceable 限定词。这样做可以确保其他模型也可以继续重声明该部件。

如果我们希望 SystemArchitecture 模型使用上述部件作为默认实现，但同时仍使用接口作为约束类型，我们可以宣布 SystemArchitecture 子系统如下：

```

replaceable Implementation.BasicPlant plant constrainedby Interfaces.Plant
annotation (choicesAllMatching=true,
Placement(transformation(extent={{-10,-50},{10,-30}})));
replaceable Implementation.IdealActuator actuator constrainedby
Interfaces.Actuator
annotation (choicesAllMatching=true,
Placement(transformation(extent={{-50,-50},{-30,-30}})));
replaceable Implementation.IdealSensor sensor constrainedby Interfaces.Sensor
annotation (choicesAllMatching=true,
Placement(transformation(extent={{30,-50},{50,-30}})));
replaceable Implementation.ProportionalController controller constrainedby
Interfaces.Controller
annotation (choicesAllMatching=true,
Placement(transformation(extent={{-10,-10},{10,10}})));
replaceable Modelica.Blocks.Sources.Trapezoid setpoint(period=1.0) constrainedby
Modelica.Blocks.Interfaces.SO
annotation (choicesAllMatching=true,
Placement(transformation(extent={{-60,30},{-40,50}})));

```

### Variation1

如果我们希望创建 BaseSystem 模型的变体，可以用继承和修饰词来创建它们，如：

```

within ModelicaByExample.Architectures.SensorComparison.Examples;
model Variant1 "Creating sample-hold variant using system architecture"
  extends BaseSystem(redeclare replaceable
    Implementation.SampleHoldSensor sensor(sample_rate=0.01));
end Variant1;

```

请注意如何模型是如何在扩展 BaseSystem 配置后，仅仅改变 sensor 模型的。若我们对这个系统进行仿真，性能相当于原来单层模型（285）

但是，如果相反我们创建一个采样时间更长的配置，我们会发现系统变得不稳定（和在单层模型（285）内的对应模型完全一样）：

### Variation2

请注意，即使装有性能足够的传感器，在 Variant1 配置里控制器似乎收敛到了错误的稳态速度。这是因为我们只使用了比例增益控制器。但是，如果我们扩展 Variant1 模型，并添加一个 PID 控制器和一个更真实的、最大提供转矩量有限的执行器，即：

```
within ModelicaByExample.Architectures.SensorComparison.Examples;
model Variant2 "Adds PID control and realistic actuator subsystems"
  extends Variant1(
    redeclare replaceable Implementation.PID_Controller controller(
      yMax=15, Td=0.1, k=20, Ti=0.1),
    redeclare replaceable Implementation.LimitedActuator actuator(
      delayTime=0.005, uMax=10));
end Variant2;
```

我们会得到下面的仿真结果：

此外，如果用一段时间来调整 PID 调节器的增益，即：

```
within ModelicaByExample.Architectures.SensorComparison.Examples;
model Variant2_tuned "A tuned up version of variant 2"
  extends Variant2(
    controller(yMax=50, Ti=0.07, Td=0.01, k=4),
    actuator(uMax=50),
    sensor(sample_rate=0.01));
end Variant2_tuned;
```

那么，我们将得到更好的仿真结果：

### 结论

我们就此完成了对上述架构的讨论。这个例子的重点是，我们可以非常容易地通过用架构探索系统的备选配置。但除了易用性外（意思是我們能很快完成这些事情），我们也能够确保一定程度的正确性，因为每个新配置都不需要增加连接。相反，用户只需要指定在各个子系统里使用的实现。而且在这时，Modelica 编译器可以进行检查，以确保用户选择的实现插件兼容于指定架构的限制类型。

## 第 9.1.3 节 热力控制

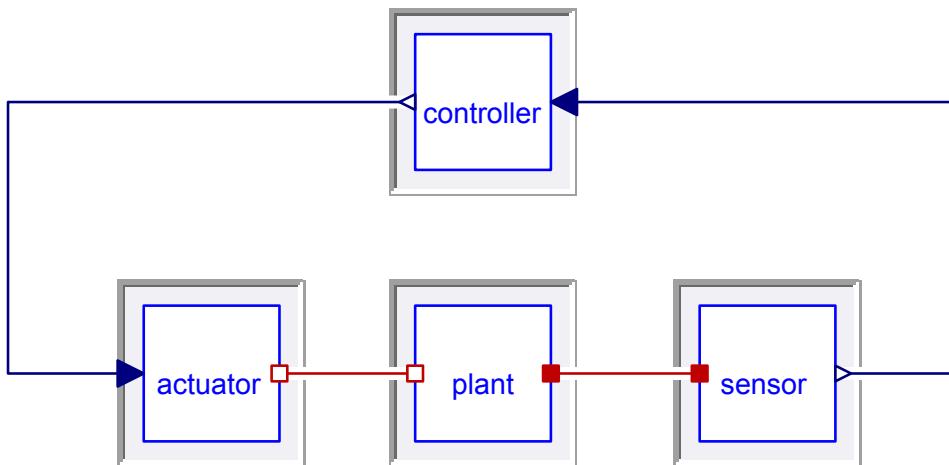
在这一章中，我们将考虑另一个系统。这个系统包括了受控对象、控制器、传感器和执行器。这个应用场景是分为三区的房子的热控制。受控对象将是房子本身。传感器为温度传感器。执行器则为房内的火炉。利用这些模型，我们将探索不同的控制策略。

我们也将如同上节一般，遵循架构驱动的方法来构建该系统。然而，在开始时我们会使用一组接口。然后，在讨论其局限性后，我们重新使用不同的方法，以获得更大的灵活性。

### 初步方法

#### 架构

让我们从下列架构开始：



我们在这里看到了和上一节同样的基本组件：受控对象模型、传感器、控制器和执行器。其实，这是一个非常典型的架构。在某些情况下，人们可以受控对象模型分解成几个子系统和/或加入多个控制器和控制回路。只是，许多闭环系统的控制问题具有类似的结构。

各种应用里的不同点一般在于，上述部件之间交换的特定信号有不同。在本例里，我们可以从上述结构原理图上看出，接口定义如下：

- 执行器接收到目标温度，然后通过与受控对象的热连接注入热
- 传感器模型也带有热连接器（和受控对象相连）。而且它还包含了测量温度的输出信号。
- 受控对象有两个热连接。一个热连接代表炉子热量和受控对象的连接。另一个是传感器的位置。
- 控制器以（从传感器）测得的温度作为输入，并输出一个指定热输出（到执行器）

此基本系统的 Modelica 代码如下：

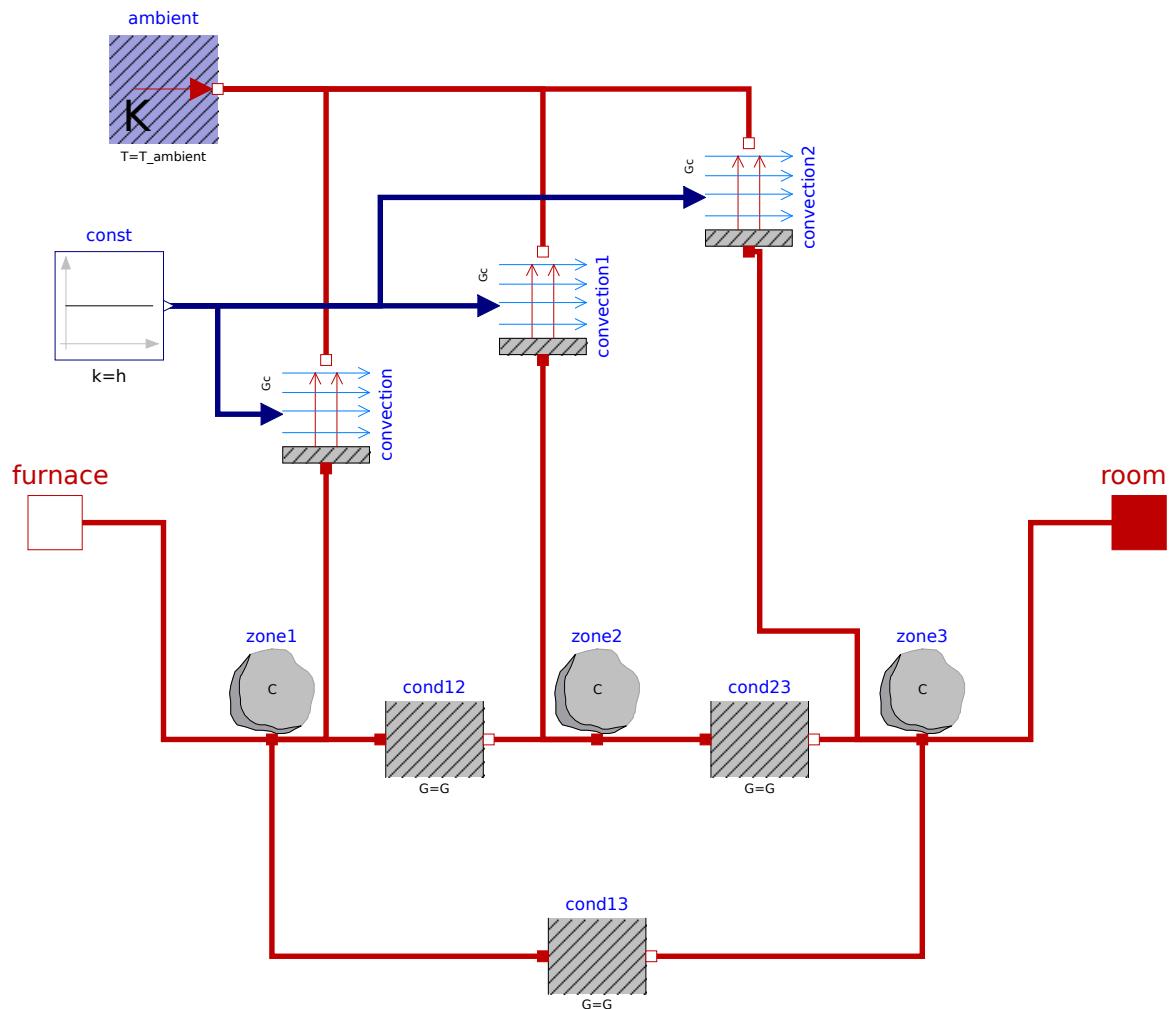
```

within ModelicaByExample.Architectures.ThermalControl.Architectures;
partial model BaseArchitecture "A basic thermal architecture"
  replaceable Interfaces.PlantModel plant
    annotation (Placement(transformation(extent={{-10,-10},{10,10}})));
  replaceable Interfaces.ControlSystem controller
    annotation (Placement(transformation(extent={{-10,30},{10,50}})));
  replaceable Interfaces.Sensor sensor
    annotation (Placement(transformation(extent={{32,-10},{52,10}})));
  replaceable Interfaces.Actuator actuator
    annotation (Placement(transformation(extent={{-50,-10},{-30,10}})));
equation
  connect(plant.room, sensor.room) annotation (Line(
    points={{10,0},{32,0}},
    color={191,0,0}, smooth=Smooth.None));
  connect(sensor.temperature, controller.temperature) annotation (Line(
    points={{53,0},{70,0},{70,40},{12,40}},
    color={0,0,127}, smooth=Smooth.None));
  connect(actuator.furnace, plant.furnace) annotation (Line(
    points={{-30,0},{-10,0}},
    color={191,0,0}, smooth=Smooth.None));
  connect(controller.heat, actuator.heat) annotation (Line(
    points={{-11,40},{-70,40},{-70,0},{-52,0}},
    color={0,0,127}, smooth=Smooth.None));
end BaseArchitecture;
  
```

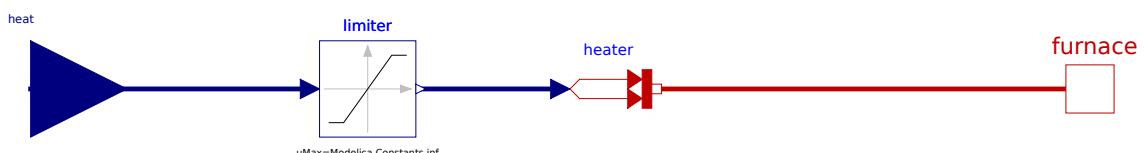
## 初始实现

### 受控对象

受控对象的模型如下：

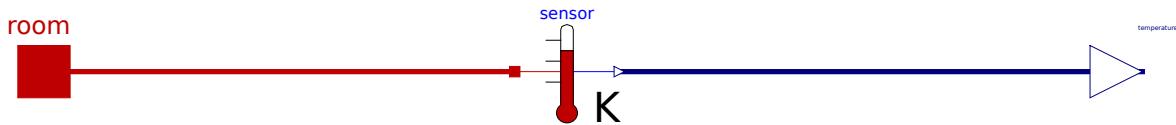


我们在这里可以看到，其中暖炉加热的区域与测温的第三区是分隔开的。暖炉模型是一个简单的热源：



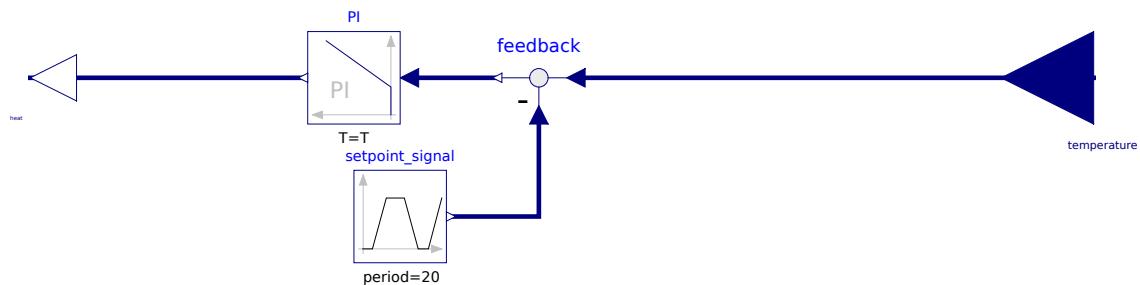
执行器以指令热量水平作为输入，然后注入相应的热量到系统里。

类似地，传感器也很简单：



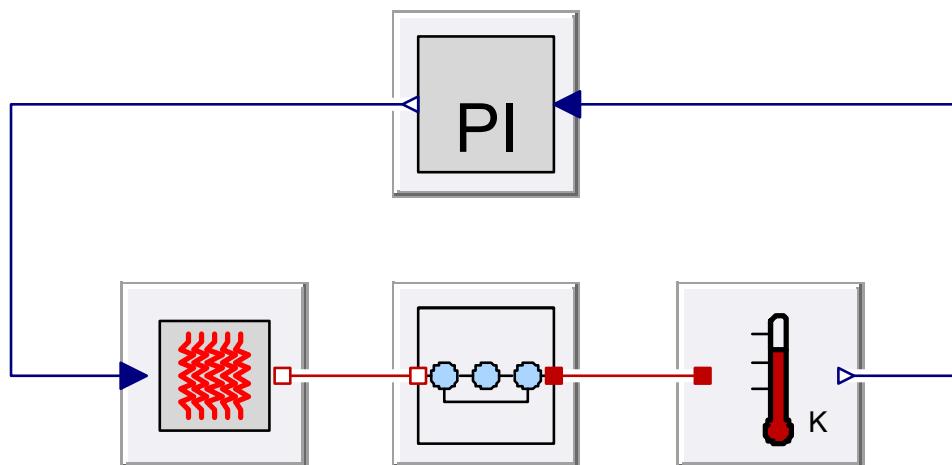
这种传感器不引入任何误差。相反，这个传感器提供精确温度这一连续信号。

我们将使用以下 PI 控制器来控制温度：



### 初步结果

将上述实现填充入架构，模型看起来如下：



请注意各个子系统的图标已经改变了。这是因为当我们进行了 redeclare 后，子系统便会使用相关联的新类型的图标。这个系统的 Modelica 语言代码是：

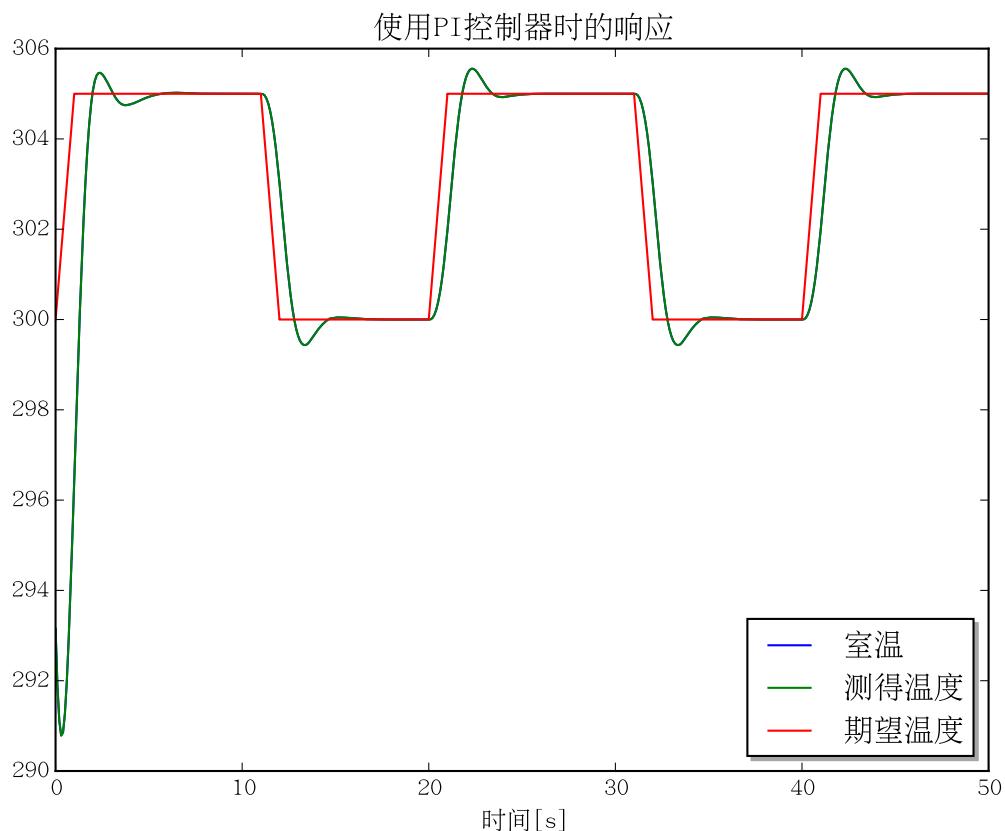
```
within ModelicaByExample.Architectures.ThermalControl.Examples;
model BaseModel "Base model using a conventional architecture"
  extends Architectures.BaseArchitecture(
```

```

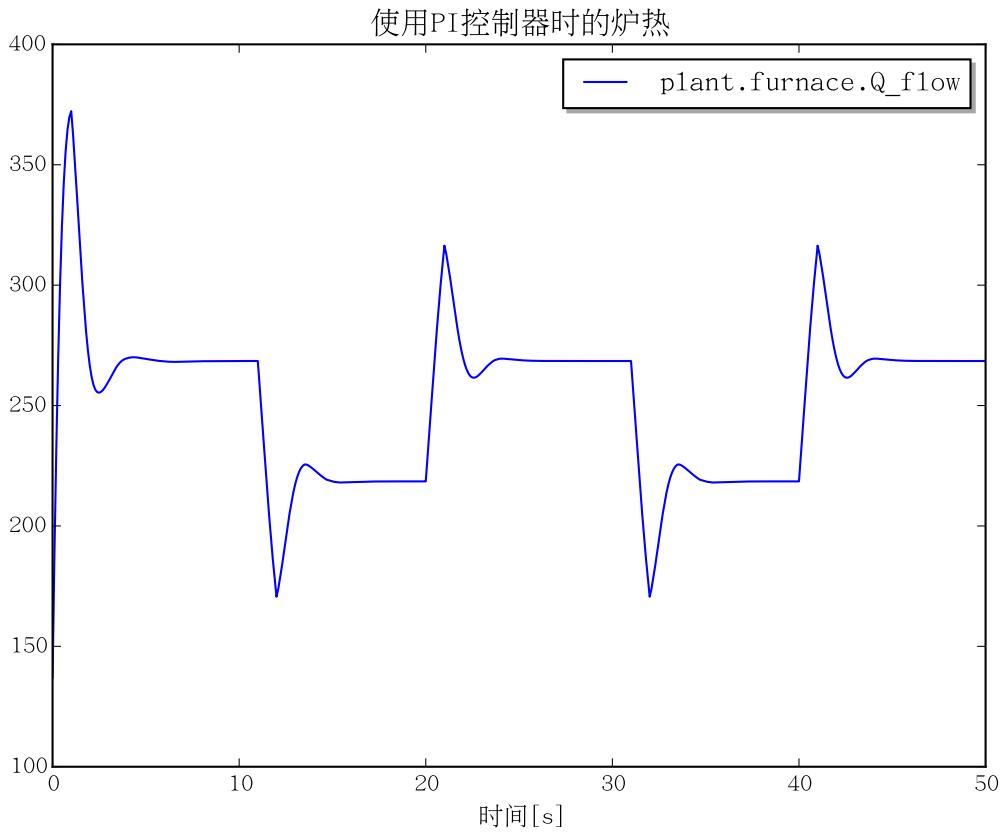
redeclare Implementations.ThreeZonePlantModel plant(
  C=2, G=1, h=2, T_ambient=278.15),
redeclare
  ModelicaByExample.Architectures.ThermalControl.Implementations.ConventionalPIControl
  controller(setpoint=300, T=1, k=20),
  redeclare Implementations.ConventionalActuator actuator,
  redeclare Implementations.ConventionalSensor sensor);
end BaseModel;

```

如果我们对系统进行仿真，会得到以下的结果：



我们可以看到，这种方法效果非常好。实现这种程度控制所需的暖炉热量如下：

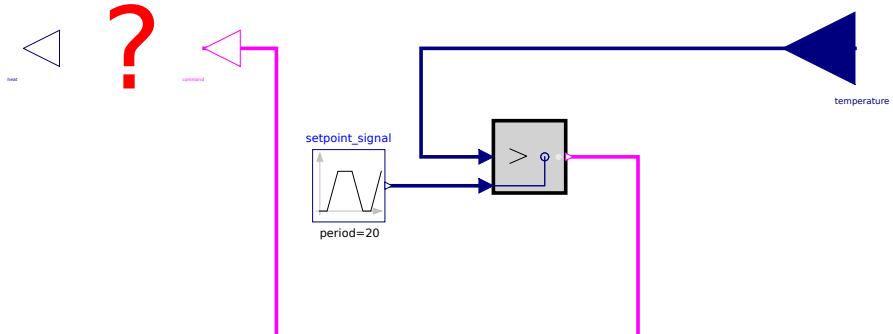


## 开关控制

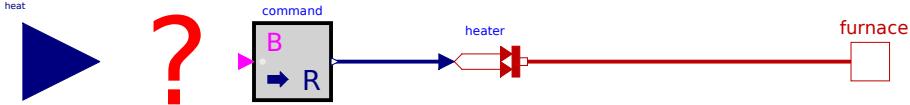
到目前为止，这种做法好像已经相当成功。我们有一个很好的架构，我们可以用其来考虑不同的执行器、传感器、控制器甚至受控对象模型。我们开发的控制系统在这种情况下性能似乎不错。

但有一点值得注意的是，在这种情况下暖炉热量必须为连续。不过，家庭取暖系统通常不采用这种类型的控制策略。相反，这类系统倾向于使用所谓的开关控制。在这种控制策略里，暖炉是非“开”即“关”。

我们有这个灵活的架构。为了解决这个问题，我们也许应该创建控制器和执行器模型的实现。其中控制器命令是一个布尔值，去指示是否暖炉的开启或关闭。但是，如果我们开始这么做的话，很快就会遇到以下问题：



需要注意的是，从我们的控制器输出 Boolean 值，但我们的 ControlSystem 接口里的命令信号 heat 却为 Real 值。我们在执行器处也有同样的问题：



该接口提供 Real 值的执行机构。但我们再一次看到，如果暖炉要求“开启”或“关闭”命令时，我们就有不匹配。

所以，问题就变成：如何处理不同子系统要求不同接口这种情况？

## 可扩展方法

expandable 连接器定义解决了这个问题。通过这种方法，不论控制策略输出 Boolean 或 Real，我们的子系统接口都不会改变。在这种情况下，改变的是连接器实例中的内容。

要了解这些 expandable 连接器是如何工作的，我们将重新设计架构模型。新架构包括 expandable 连接器，然后我们会考虑架构模型如何可用于连续以及“开关”的控制策略。

## 可扩展连接器

让我们可以构造更灵活架构模型的关键功能是 expandable connector。举个例子，先前我们定义了如下的 Actuator 接口：

```
within ModelicaByExample.Architectures.ThermalControl.Interfaces;
partial model Actuator "Actuator subsystem interface"

Modelica.Blocks.Interfaces.RealInput heat "Heating command" annotation (
  Placement(transformation(
    extent={{-20,-20},{20,20}},
    origin={-120,0})));
Modelica.Thermal.HeatTransfer.Interfaces.HeatPort_b furnace
  "Connection point for the furnace"
  annotation (Placement(transformation(extent={{90,-10},{110,10}})));
end Actuator;
```

此接口包含两个连接器，heat 连接器和 furnace 连接器。furnace 连接器为热连接器，它使暖炉与受控对象产生热相互作用。heat 连接器为来自控制器的 Real 值输入信号。该信号指定所需的热量输出大小。事实上，在我们切换到需要 Boolean 信号的控制方法时，接口模型的上述信号为 Real 值这点正是问题所在。为了解决这个问题，我们将使用下面的接口定义执行器：

```
within ModelicaByExample.Architectures.ThermalControl.Interfaces;
partial model Actuator_WithExpandableBus
  "Actuator subsystem interface with an expandable bus"

ExpandableBus bus
  annotation (Placement(transformation(extent={{-110,-10},{-90,10}})));

Modelica.Thermal.HeatTransfer.Interfaces.HeatPort_b furnace
  "Connection point for the furnace"
  annotation (Placement(transformation(extent={{90,-10},{110,10}})));
end Actuator_WithExpandableBus;
```

这里我们看到 furnace 连接器仍然存在。但是 heat 连接器已经不见了。相反，它已被替换为新的连接器实例：类型为 ExpandableBus 的 bus。ExpandableBus 连接器的定义是：

```
within ModelicaByExample.Architectures.ThermalControl.Interfaces;
  expandable connector ExpandableBus "An example of an expandable bus connector"
end ExpandableBus;
```

换句话说，**连接器是空的**。但重要的是 expandable 限定词的存在。如果总线总是需要某些特定信号，这些信号就应该列在其连接器的定义之内。而事实上，ExpandableBus 类没有任何变量或子连接器。这意味着总线内的信息量没有最低要求。不过总线可以通过**扩展**来包含更多的信息。

当然，我们可以使用继承来添加新信号。但是，继承会引入了一个新类型。而接口定义中使用的类型规定了连接器的类型。所以，通过继承并不能真正有效创造更复杂的接口。

注意，Modelica 内并有一个没有正式的“总线”定义。这个术语经常用于上述情况去表示携带多条信息的连接器。

可扩展连接器的功能比较特殊。可扩展总线的信号由其上的连接决定。通过向可扩展总线添加连接，信号便会隐含地添加在连接器上。这时的 Modelica 编译会观察与上述连接器相连的所有连接器，并将这些连接器全部扩展。这样一来，所有的连接器才可以互相匹配。稍后，我们将进一步介绍上述过程。但在这之前，我们需要首先有讨论的实际模型。

受控对象模型的接口没有受使用 expandable 连接器的影响。但传感器和控制器的接口如下：

```
within ModelicaByExample.Architectures.ThermalControl.Interfaces;
partial model Sensor_WithExpandableBus
  "Sensor subsystem interface using an expandable bus"

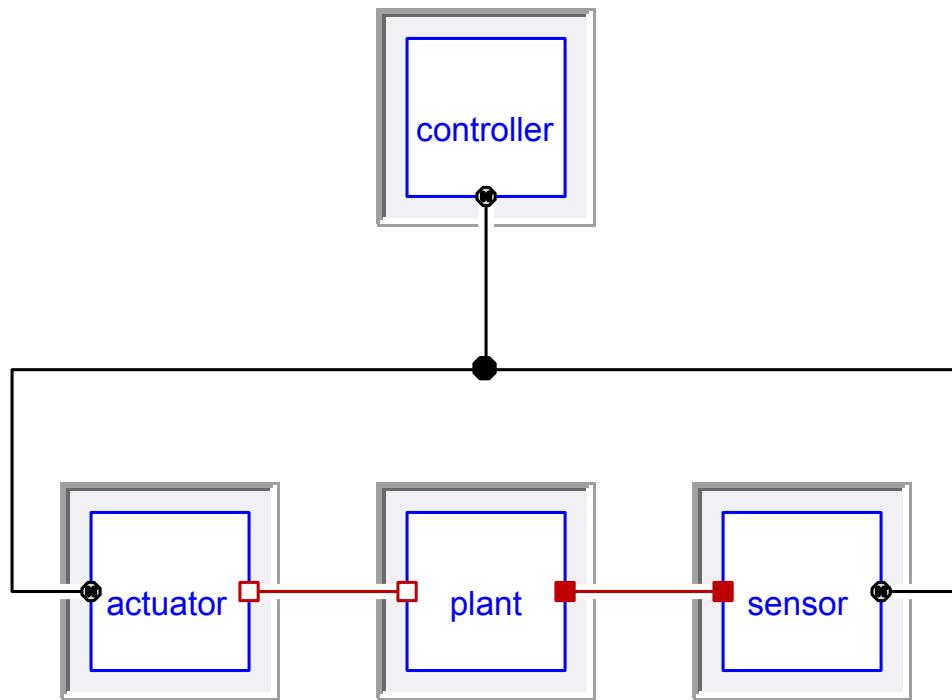
  Modelica.Thermal.HeatTransfer.Interfaces.HeatPort_a room
    "Thermal connection to room"
    annotation (Placement(transformation(extent={{-110,-10},{-90,10}})));

  ModelicaByExample.Architectures.ThermalControl.Interfaces.ExpandableBus bus
    annotation (Placement(transformation(extent={{90,-10},{110,10}})));
end Sensor_WithExpandableBus;
```

```
within ModelicaByExample.Architectures.ThermalControl.Interfaces;
partial model ControlSystem_WithExpandableBus
  "Control system interface using an expandable bus connector"
  ExpandableBus bus annotation (Placement(transformation(extent={{-10,-110},{10,-90}})), iconTransformation(extent={{-10,-110},{10,-90}}));
end ControlSystem_WithExpandableBus;
```

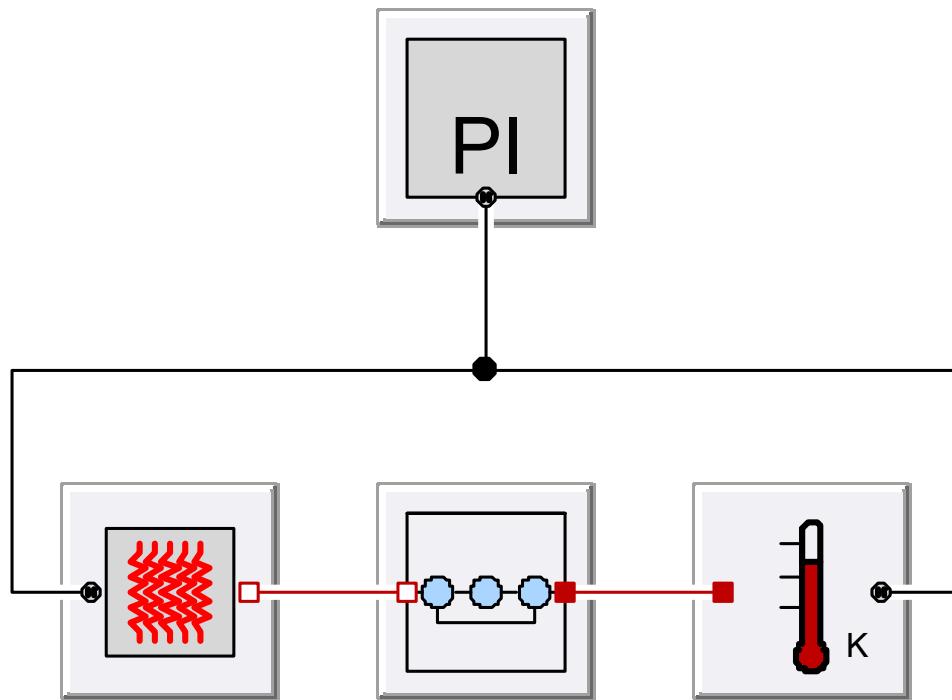
请注意控制器接口变得多么简单。这是因为通过使用 expandable 连接器，我们可以把从传感器接收到的温度测量以及发送到执行器的热量命令放在**在同一总线上**。因此，我们只需要一个接口。开发人员仍可以直接使用多个总线去组织信号，使信号更好地反映物理存在或避免混乱。在此，我们使用单一连接器的目的纯粹是为了告诉读者这样做是可能的。

使用可扩展的连接器，我们可以创建以下改进版的架构：

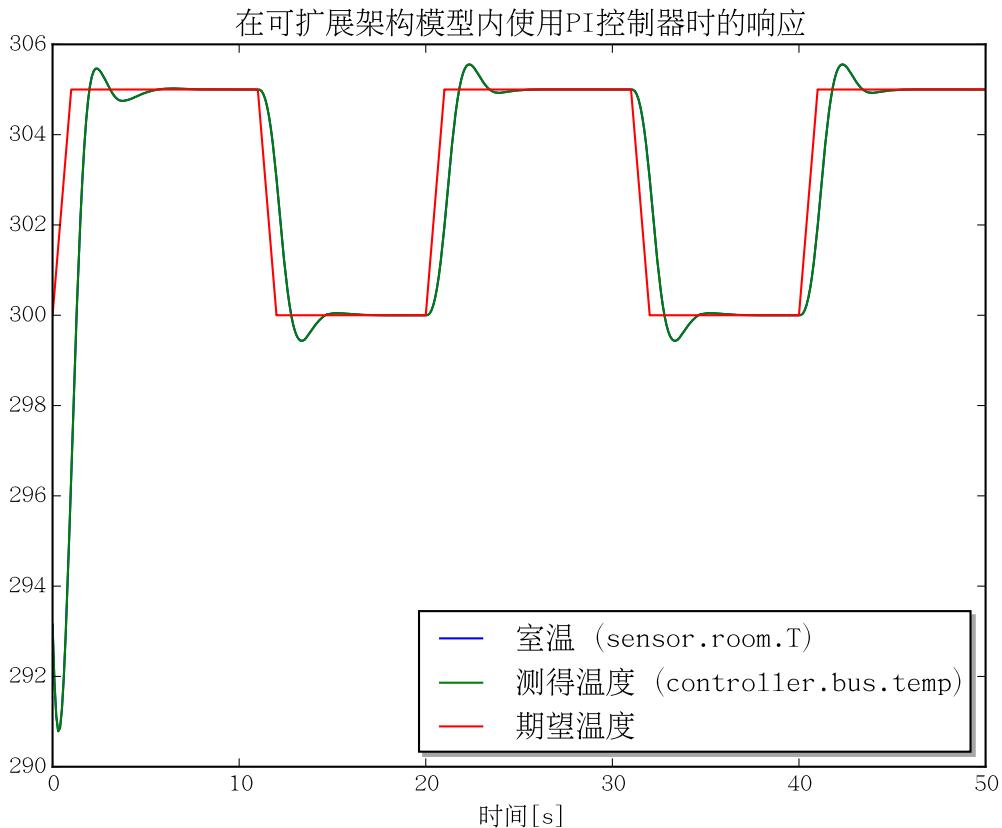


### 可扩展实现

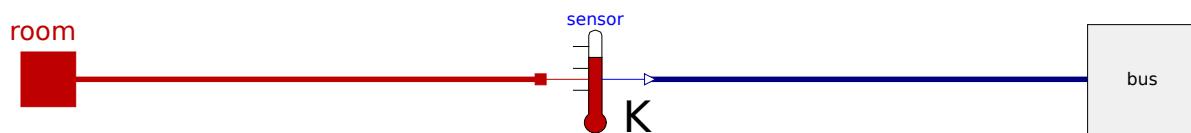
有了这个更灵活的架构，让我们先重建原有的连续控制系统配置：



若我们绘制这个系统的仿真结果，会得到如下的响应：



请注意，所测温度对应于信号 controller.bus.temp。其中 bus 是可扩展的连接器的一个实例。进一步回忆的 ExpandableBus 定义不含有一个叫 temperature 的信号。所以现在的问题是，该信号是怎么去到连接器上的？问题就出在传感器模型的实现。传感器模型的框图如下：



对应的 Modelica 代码是：

```
within ModelicaByExample.Architectures.ThermalControl.Implementations;
model TemperatureSensor "Temperature sensor using an expandable bus"
  extends Interfaces.Sensor_WithExpandableBus;
protected
  Modelica.Thermal.HeatTransfer.Sensors.TemperatureSensor sensor
  annotation (Placement(transformation(extent={{-10,-10},{10,10}})));
equation
  connect(sensor.T, bus.temperature) annotation (Line(
    points={{10,0},{100,0}},
    color={0,0,127},
```

```

smooth=Smooth.None));
connect(room, sensor.port) annotation (Line(
points={{-100,0},{-10,0}},
color={191,0,0},
smooth=Smooth.None));
end TemperatureSensor;

```

重要的是高亮显示的那行。

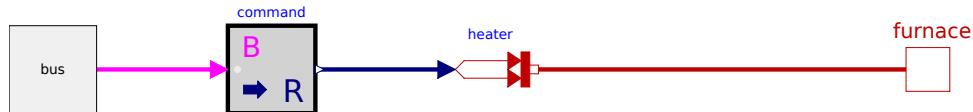
在该图中，我们可以看到，温度传感器组件的输出信号连接在总线上。但是，我们观察 connect 声明可以看出，信号不仅仅是连接到总线上。此信号连接在总线上一个名为 temperature 的量上。temperature 连接器并不存在于 ExpandableBus 的定义里。相反，**它是由 connect 语句本身创建的！**这也恰恰是 expandable 限定词所允许的。

一般情况下，我们不希望所有连接器均为 expandable。若我们先验地知道所有信号的名称和类型，那么就应该明确的列出。这样做会允许 Modelica 语言的编译器进行一些重要的检查，以确保模型的正确性。值得注意的是，添加所述 expandable 限定词到连接器会使不小心创建无用信号成为可能（例如由于打错字）。这种错误在没有添加限定词的时候，本来可以由编译器找出。

## 重配置

现在，我们已经演示了如何使用可扩展的方法对系统的连续控制版本进行建模。现在，让我们重新关注到“开关”版本。

我们已经看到如何配置可扩展连接器版的温度传感器子系统。剩下的就是控制器和执行器的模型。执行器模型框图如下所示：



同样，看 Modelica 的代码时要注意对 bus 连接器信号的引用：

```

within ModelicaByExample.Architectures.ThermalControl.Implementations;
model OnOffActuator "On-off actuator implemented with an expandable bus"
  extends Interfaces.Actuator_WithExpandableBus;
  parameter Real heating_capacity "Heating capacity of actuator";
protected
  Modelica.Thermal.HeatTransfer.Sources.PrescribedHeatFlow heater
    annotation (Placement(transformation(extent={{-10,-10},{10,10}})));
  Modelica.Blocks.Math.BooleanToReal command(realTrue=heating_capacity,
    realFalse=0)
    annotation (Placement(transformation(extent={{-60,-10},{-40,10}})));
equation
  connect(heater.port, furnace) annotation (Line(
    points={{10,0},{100,0}}, color={191,0,0},
    smooth=Smooth.None));
  connect(command.y, heater.Q_flow) annotation (Line(
    points={{-39,0}, {-10,0}}, color={0,0,127},
    smooth=Smooth.None));
  connect(command.u, bus.heat_command) annotation (Line(
    points={{-62,0}, {-100,0}}, color={255,0,255},
    smooth=Smooth.None));

```

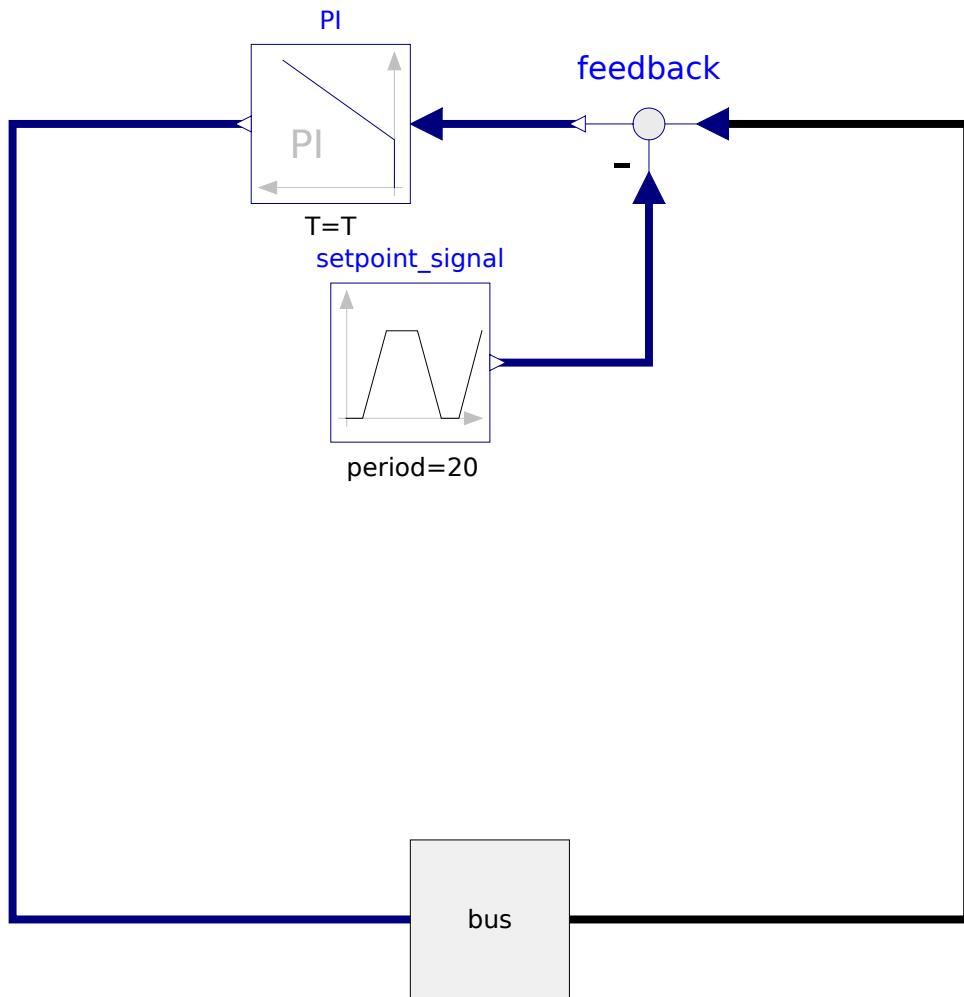
```

smooth=Smooth.None));
end OnOffActuator;

```

再次，注意强调的那行。该行引用 bus 连接器的名为 heat\_command 的元素。和上面一样，该信号不存在于 ExpandableBus 的定义内。该信号是隐式创建的。因为它在上述高亮显示 connect 语句中被引用了。

由传感器模型我们可以看到，测得的温度输出到 bus 连接器的一个名为 temperature 的 Real 信号里。由执行器模型我们则看到，执行器需要的控制器命令来自一个名为 heat\_command 的 Boolean 信号。因此，我们应该会看到控制器模型使用这两个信号。控制器的框图如下：



但图内不包含足够的细节去让我们得知所引用 bus 连接器信号的准确名称。为此，我们需要看实际的源代码：

```

within ModelicaByExample.Architectures.ThermalControl.Implementations;
model ExpandablePIControl "PI controller implemented with an expandable bus"
  extends Interfaces.ControlSystem_WithExpandableBus;
  parameter Real setpoint "Desired temperature";
  parameter Real k=1 "Gain";
  parameter Modelica.SIunits.Time T "Time Constant (T>0 required)";
protected
  Modelica.Blocks.Sources.Trapezoid setpoint_signal(
    amplitude=5, final offset=setpoint, rising=1,
    width=10, falling=1, period=20)
    annotation (Placement(transformation(extent={{-20,-40},{0,-20}})));
  Modelica.Blocks.Math.Feedback feedback
    annotation (Placement(transformation(extent={{30,-10},{10,10}})));
  Modelica.Blocks.Continuous.PI PI(final T=T, final k=-k)

```

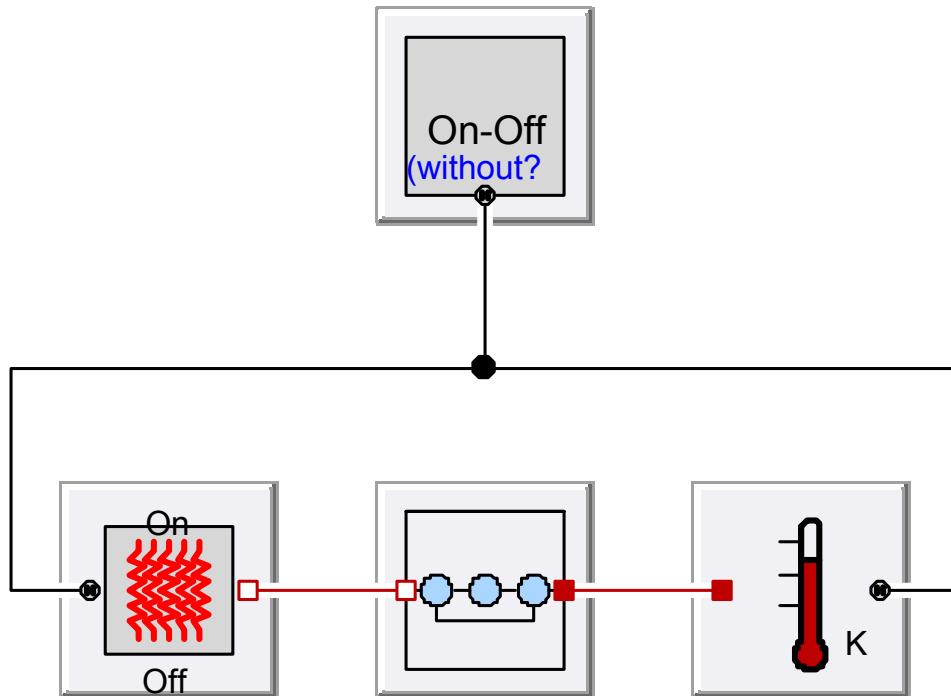
```

annotation (Placement(transformation(extent={{-10,-10},{-30,10}})));
equation
  connect(setpoint_signal.y, feedback.u2)
    annotation (Line(
      points={{1,-30},{20,-30},{20,-8}},
      color={0,0,127}, smooth=Smooth.None));
  connect(PI.u,feedback.y) annotation (Line(
    points={{-8,0},{11,0}},
    color={0,0,127}, smooth=Smooth.None));
  connect(bus.temperature, feedback.u1) annotation (Line(
    points={{0,-100},{60,-100},{60,0},{28,0}},
    color={0,0,0}, smooth=Smooth.None));
  connect(PI.y, bus.heat) annotation (Line(
    points={{-31,0},{-60,0},{-60,-100},{0,-100}},
    color={0,0,127}, smooth=Smooth.None));
end ExpandablePIControl;

```

再次，注意高亮行。这些 connect 语句不仅隐式将 temperature 和 heat\_command 信号加入 bus 连接器，而且这两个名称传感器和执行器模型所需的信号相匹配。

将所有子系统组合起来，我们得到系统的框图如下：

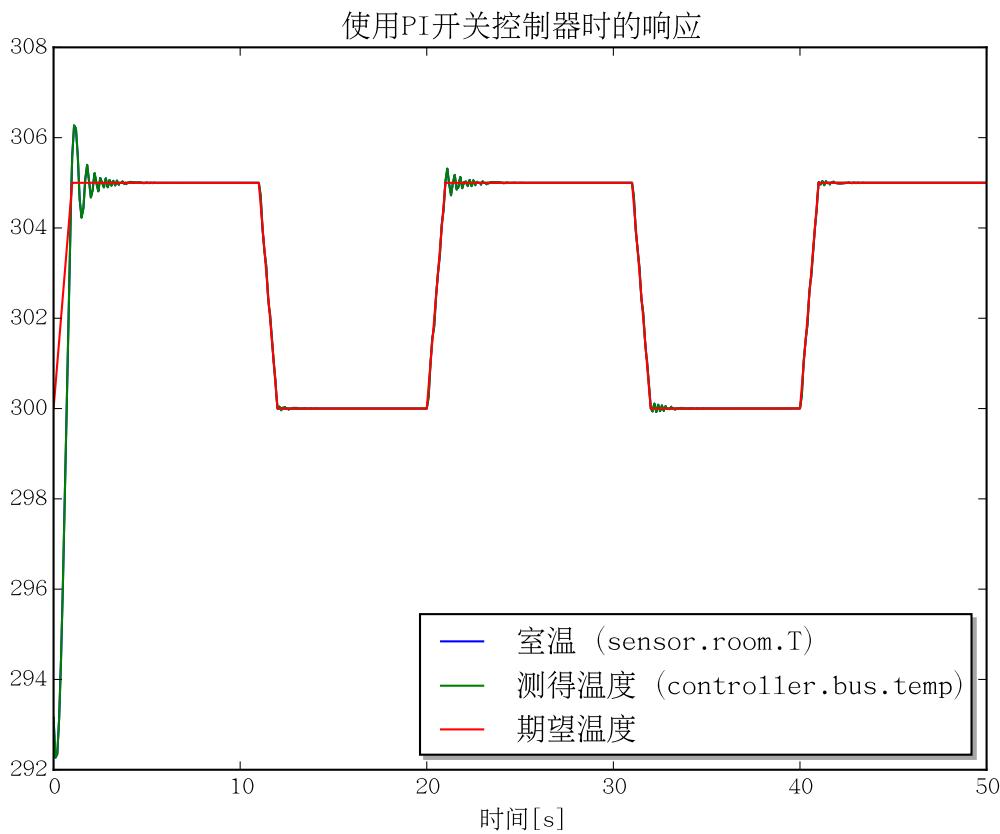


系统模型的源代码甚为简单：

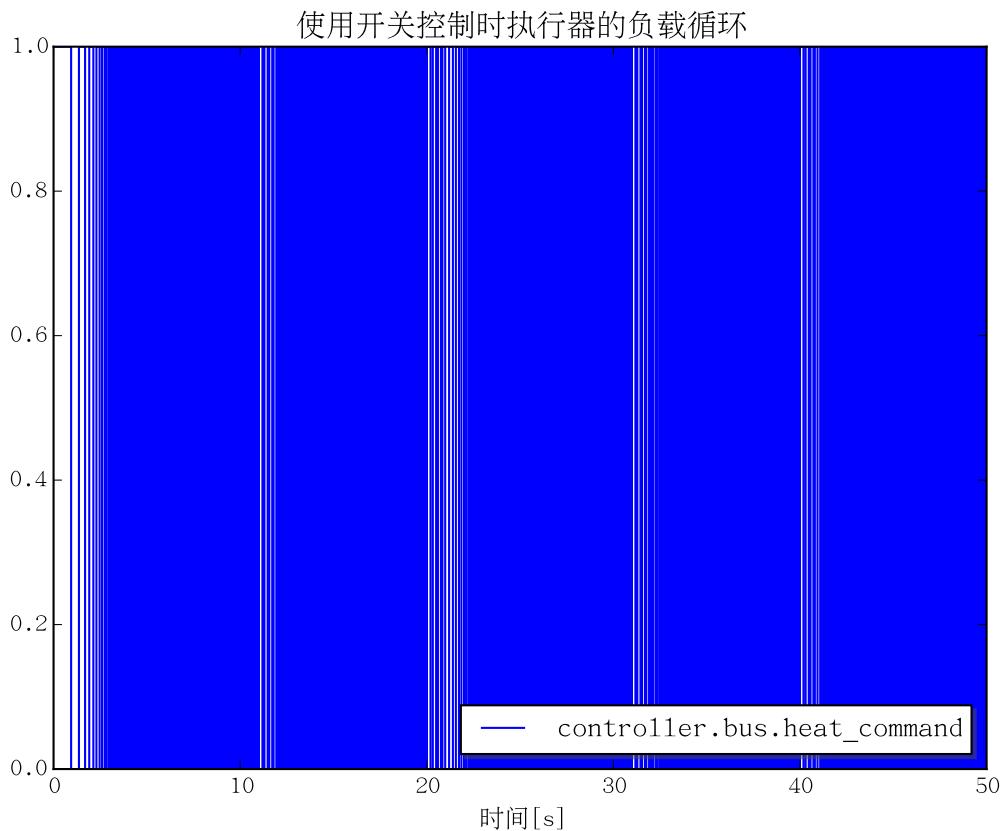
```

within ModelicaByExample.Architectures.ThermalControl.Examples;
model OnOffVariant "Variation with on-off control"
  extends ExpandableModel(
  redeclare replaceable
    Implementations.OnOffActuator actuator(heating_capacity=500),
  redeclare replaceable
    Implementations.OnOffControl controller(setpoint=300));
end OnOffVariant;

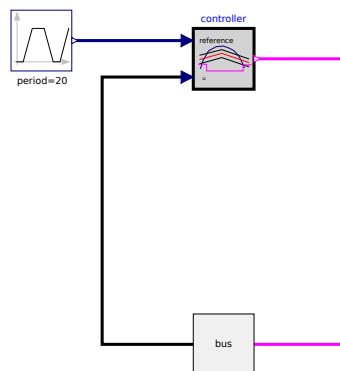
```



然而，这些模型仍有一个问题。若我们观察暖炉的负载循环，就可以更清楚地看出这个问题：



这正是我们在前面小节滞回 (68) 中展示的相同问题。正因为控制策略缺乏任何迟滞，我们才所看到的暖炉不断打开和关闭。如果再加上迟滞，控制器模型就变为：

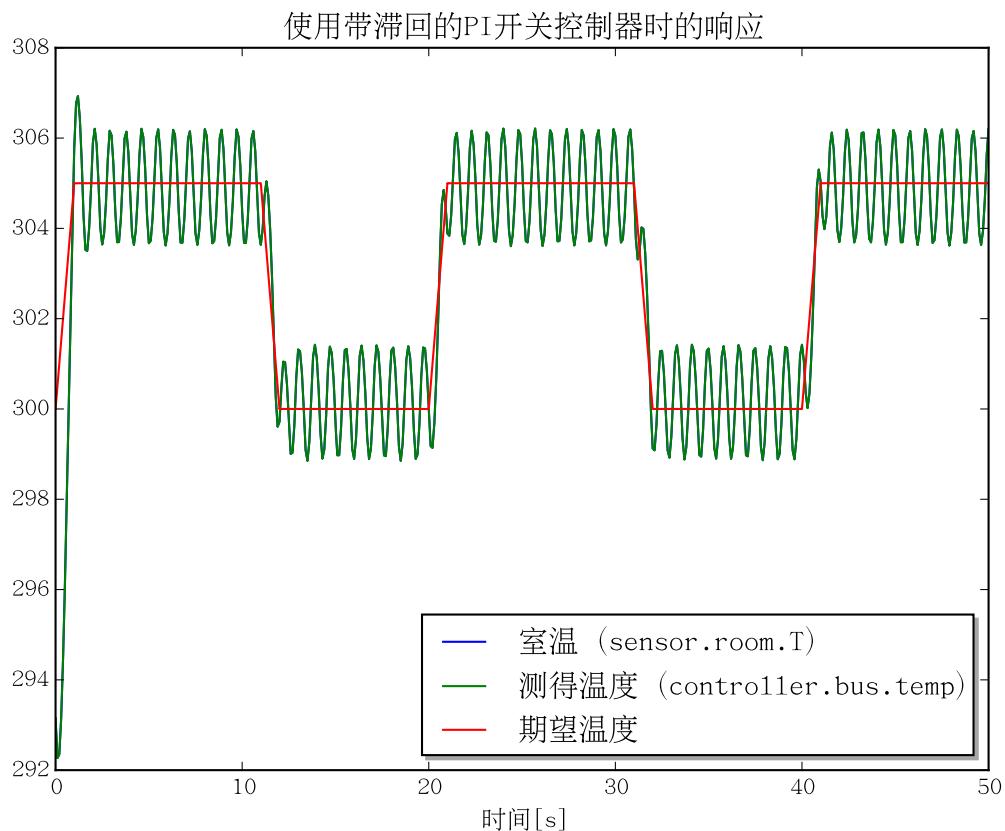


其余部分均保持不变。我们将使用相同的传感器和执行器模型。我们仍然使用相同的总线信号，因为这仍然是一个开关控制器。所以，系统级模型的唯一变化（相对于 OnOffVariant 模型）就是所使用的不同控制器模型。我们可以看到，Modelica 的这些配置管理功能可以很好地在系统模型内表述这点：

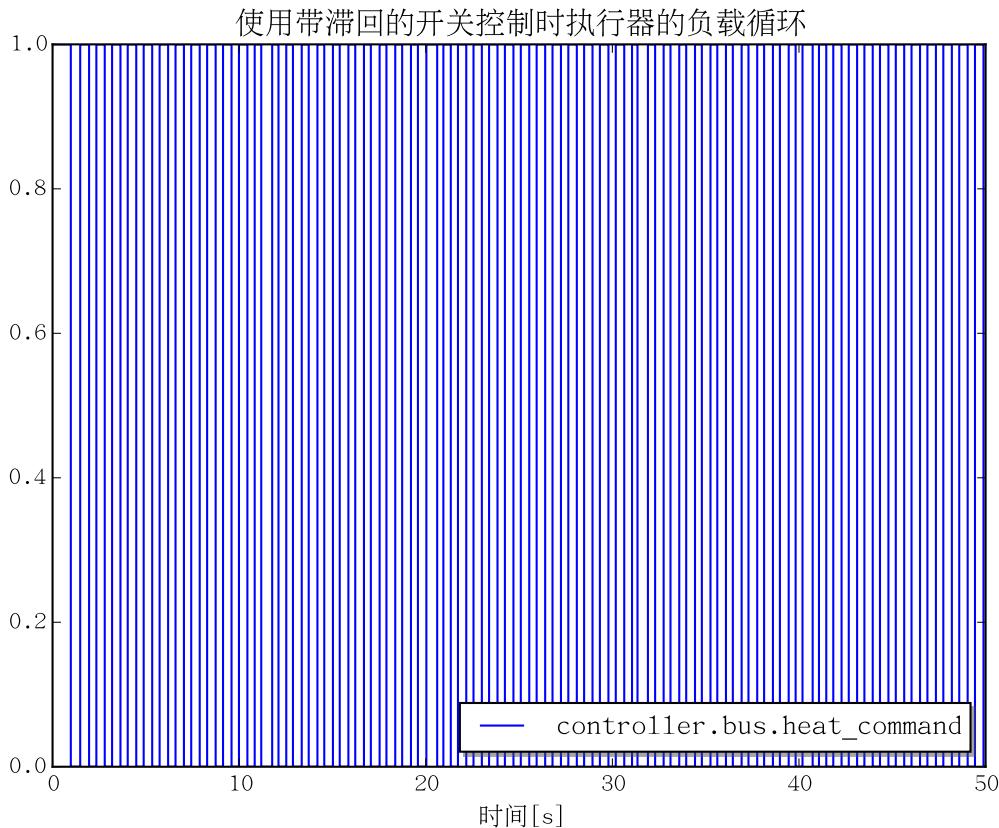
```

within ModelicaByExample.Architectures.ThermalControl.Examples;
model HysteresisVariant "Using on-off controller with hysteresis"
  extends OnOffVariant(redeclare Implementations.OnOffControl_WithHysteresis
    controller(setpoint=300, bandwidth=1));
end HysteresisVariant;
  
```

使用迟滞控制，我们的模拟结果如下：



但最重要的区别是，滞后不会导致之前在开关控制器中看到的抖振：



## 结论

这是我们能够使用 Modelica 配置管理功能，采取基于架构的方法来建立系统模型的第二个例子。当存在基于相同架构有很多变体需要分析时，架构方法非常有用。使用 redeclare 特性，能够容易地替换子系统的不同设计，或者对应不同的工程分析使用具有所需细节度的子系统。

在本例中，我们看到了对比标准连接器而言 expandable 连接器如何提供了更大的灵活性。不过，这也带有一定的风险。Modelica 语言编译器通常会进行类型检查。但在使用了可扩展连接器后，类型检测就不那么严格了。

## 第 9.2 节 回顾

### 第 9.2.1 节 接口和实现

#### 概念定义

我们在本章提出的两个例子里，接口定义用作架构定义过程的一部分。术语“接口”并非来自 Modelica。这个词是计算机语言中的一个常用术语。在 Modelica 里，我们可以认为接口是定义了所有外部可见细节的模型。你也可以把一个接口当作没有任何内部细节的“壳”。出于这个原因，接口模型几乎总是带有 partial 标记。

另一个重要的概念就是“实现”。这是从计算机语言世界借来的另一个词。接口用于简单地描述模型外部可见的部分。而实现除此以外还包括了内部细节。实现包括真正实现该接口所需的信息。某些情况下，该模型可能只是接口的部分实现（这种情况下，它也应该标记为 partial）。而在另一些情况下，该模型可以代表特定子系统的架构。其中的进一步实现细节则在推迟到模型层级另一级（这是 partial 模型的另一种情况）。不过大部分的时间，这些实现将是特定子系统的完整（非 partial）模型。

## 插件兼容性

当我们谈论接口和实现所要考虑的重中之重就是插件兼容性的概念。我们在[传感器比较 \(285\)](#)例子已经讨论过，模型 X 插件兼容于模型 Y 的条件是，每个 Y 的公共变量都能在 X 找到具有相同名称的对应公共变量。此外，每个 X 的上述变量本身必须是插件兼容于其在 Y 内的对应变量。这可以确保如果你将 Y 类型的组件更换为 X 类型的组件，你需要的一切（参数，连接器等）将依然存在、并仍将兼容。**不过，请注意如果 X 插件兼容于 Y，这不意味着 Y 插件兼容于 X**（我们稍后会看到）。

一般来说，大多数情况我们关注的插件兼容性问题是围绕着给定的实施是否插件兼容于给定的接口。正如我们在这些例子里看到的一样（我们也马上会回顾），Modelica 的配置管理功能依赖于接口和实现间的关系。而配置管理的流程则是围绕插件的兼容性为中心的。

## 结论

必须要知道，我们不但应该以接口与实现模型为基础进行思考。而且，我们也要正式创建接口模型，并区分接口模型与具体实现。因为，在创建架构驱动的模型时，上述思路都非常有用。

### 第 9.2.2 节 配置管理

#### replaceable

真正让 Modelica 实现配置管理特性的是 replaceable 关键字。此关键词是用来标记其类型在之后可以改变（或“重新声明”）的模型部件。一种理解 replaceable 的方法是，它允许模型开发人员在模型里定义插孔。这些模型要么一张“白纸”（接口模型就是模型里定义的原始类型），或者最少是“可配置的”。

使用 replaceable 关键字的优点是，它允许不通过重新接线而创建新模型。这不仅规定了未来模型的结构框架（可以确保命名惯例、共同接口等被遵守），这也有助于减少模型开发过程中的一种潜在错误：即创建连接。

要让组件变成可替换所必须要做的唯一事情就是在声明前添加 replaceable 关键字，即：

```
replaceable DeclaredType variableName;
```

其中 DeclaredType 是名为 variableName 的变量其初始类型。在这样的声明中，我们可以赋予 variableName 变量一个新类型（我们即将讨论详细的做法）。但任何 variableName 使用的新类型必须**插件兼容**（320）于原类型。

#### constrainedby

正如我们刚才提到的，默认情况下任何 replaceable 组件的新类型必须插件兼容于最初类型的。但这种情况其实不一定必须的。我们先前对[约束类型 \(293\)](#)的讨论已经指出，我们能够同时为变量指定一个缺省类型，以及一个独立约束类型，令替换的新类型必须与之兼容。

指定另外约束类型需要使用到 constrainedby 关键字。使用 constrainedby 关键字语法如下：

```
replaceable DefaultType variableName constrainedby ConstrainingType;
```

其中 variableName 是前面变量声明的名字，DefaultType 代表 variableName 类型。ConstrainingType 表示约束类型。如前所述，variableName 变量使用的任何新类型必须插件兼容于约束类型。另外，当然 DefaultType 也必须插件兼容于约束类型了。

#### constrainedby vs. extends

旧版本的 Modelica 不包含 constrainedby 关键字。有相同功能的是 extends 关键字。但有人认为，继承和插件的兼容性截然不同，足矣用一个单独的关键字去减少混乱。所以，如果一段 Modelica 代码里在本该出现 constrainedby 关键字的地方（即在 replaceable 声明的后面）你却看到了 extends 关键字。

redeclare

所以现在我们知道，通过使用 `replaceable` 关键字，可以让我们在之后改变一个变量的类型。改变类型又称为“重新声明”变量（或者说声明为不同的类型）。出于这个原因，用 `redeclare` 关键字指示重声明甚为恰当。若我们有以下的系统模型：

```
model System
  Plant plant;
  Controller controller;
  Actuator actuator;
  replaceable Sensor sensor;
end System;
```

此 `System` 内只有传感器是 `replaceable`。因此，其他各子系统（即 `plant`、`controller`、`actuator`）的类型不能改变。

如果我们想在扩展这个模型时使用不同的 `sensor` 子系统模型，那我们会以如下方式使用 `redeclare` 关键字：

```
model SystemVariation
  extends System(
    redeclare CheapSensor sensor
  );
end SystemVariation;
```

这告诉了 Modelica 语言编译器是在 `SystemVariation` 模型的上下文内，`sensor` 子系统应该是 `CheapSensor` 模型的一个实例，而非（一般情况下默认的）`Sensor` 模型。但是，`CheapSensor` 模型（或重声明时选择的任何其他类型）必须插件兼容于该变量的约束类型。

`redeclare` 语句的语法和正常的声明真的几乎完全一样。唯一不同之处在于语句前面的 `redeclare` 关键字。很明显，重新声明的任何变量必须事先曾被声明（即你不能使用此语法来声明一个变量，只能重声明它）。

你重定义一个组件时，新的重声明会取代前面的重声明。这点**非常重要**。例如，下面的重声明之后：

```
redeclare CheapSensor sensor;
```

`sensor` 组件**不能再能被替换**。原因是新的声明不包括 `replaceable` 关键字。其结果是，此关键词虽然此前在最初的模型有定义，在所有以后的模型却如同不存在了。如果我们希望部件以保持可替换，重新声明将需要写为：

```
redeclare replaceable CheapSensor sensor;
```

此外，如果我们重新将变量声明成可替换的，我们还可以选择**重新声明约束类型**，如下：

```
redeclare replaceable CheapSensor sensor constrainedby NewSensorType;
```

但是，原始约束类型哪怕在这种情况下仍然能发挥作用。原因是新类型 `NewSensorType` 必须插件兼容于原始约束类型。用编程语言的术语，我们能够收缩类型（减少插件兼容类型的种类），而不能拓宽类型（将此前并非插件兼容的类型变为插件兼容）。

此前在讨论**组件数组**（280）时，我们指出了重新声明数组中的各个元素并不可能。相反，重声明必须应用到整个数组。换句话说，如果我们最初声明如下：

```
replaceable Sensor sensors[5];
```

然后可以重新声明数组，例如：

```
redeclare CheapSensor sensors[5];
```

但要注意，重新声明影响 `sensors` 数组中的每个元素。仅仅重新定义一个元素是不可能的。

## 修改

可替换性带来的一个重要问题出现在有重声明时进行修改的情况下。要理解这个问题，考虑下面的例子。

```
replaceable SampleHoldSensor sensor(sample_rate=0.01)
constrainedby Sensor;
```

现在, 如果我们要以如下语句重定义 sensor, 会发生什么情况呢?

```
redeclare IdealSensor sensor;
```

sample\_rate 值会消失么? 我们希望如此。因为 IdealSensor 模型可能不具有名为 sample\_rate 的 parameter 以供修改。

然而, 让我们考虑另外一种情况:

```
replaceable Resistor R1(R=100);
```

现在想象我们有另外一个电阻器模型 SensitiveResistor。而这个模型插件兼容于 Resistor (即它有一个名为 R 的 parameter)。但模型也包括一个额外的参数 dRdT, 以指示电阻的 (线性) 温度敏感度。我们可能想要做这样的事情:

```
redeclare SensitiveResistor R1(dRdT=0.1);
```

R 在这种情况下会如何呢? 在这里, 我们实际上希望保持 R 的值, 让其在整个重声明仍然存在。否则, 我们就需要不断重申这点, 即:

```
redeclare SensitiveResistor R1(R=100, dRdT=0.1);
```

这就会违反 DRY 原则。其结果将是, 在 R 原始值的任何改变都会被任意的重声明所覆盖。

因此, 我们已经看到了两个有效用例。在一种情况下, 我们不希望在重声明后保留修改。在另一个例子里, 我们则想保留修改。幸好, Modelica 语言可以方法可以表达这两种需求。正常的 Modelica 语义可用于第一种情况。在重定义时, 原声明中的所有修改都会被去除。至于第二种情况呢? 解决方案是应用约束类型上的修改。因此, 以前面的电阻为例, 原来的声明将要修改为:

```
replaceable Resistor R1 constrainedby Resistor(R=100);
```

在这里我们单独地明确列出默认类型 Resistor 以及约束类型 Resistor(R=100)。原因是约束类型现在包含了修改。这样将修改移动到约束类型后, 该修改会自动应用到原来的声明以及随后的重声明里。因此, 在这种情况下, 电阻器实例 R1 的 R 值为 100。即便修改不直接加在变量名后。而此外, 如果我们进行前述的重声明, 即:

```
redeclare SensitiveResistor R1(dRdT=0.1);
```

R=100 修改也会自动应用于此

总之, 如果你想修改只应用于特定的声明不影响随后的重声明, 那么应该将修改附加在变量名。如果你想让修改持续影响随后的重声明, 就应将其附加到约束型后。

## 重定义

另外, replaceable 关键字不仅可以和声明相关联, 也可以与定义有关。此功能的主要用途是一次性改变多个组件的类型。举个例子, 假设电路模型有数个不同的电阻元件:

```
model Circuit
  Resistor R1(R=100);
  Resistor R2(R=150);
  Resistor R4(R=45);
  Resistor R5(R=90);
  // ...
equation
  connect(R1.p, R2.n);
  connect(R1.n, R3.p);
  // ...
end Circuit;
```

现在想象一下，我们希望这个模型有一个版本使用普通的 Resistor 组件。而另一个版本里，每个电阻均为 SensitiveResistor 模型的实例。为此，其中一种方法是定义我们的 Circuit 如下：

```
model Circuit
  replaceable Resistor R1 constrainedby Resistor(R=100);
  replaceable Resistor R2 constrainedby Resistor(R=150);
  replaceable Resistor R4 constrainedby Resistor(R=45);
  replaceable Resistor R5 constrainedby Resistor(R=90);
  // ...
equation
  connect(R1.p, R2.n);
  connect(R1.n, R3.p);
  // ...
end Circuit;
```

但在这种情况下，带有 SensitiveResistor 组件的电路将定义为：

```
model SensitiveCircuit
  extends Circuit(
    redeclare SensitiveResistor R1(dRdT=0.1),
    redeclare SensitiveResistor R2(dRdT=0.1),
    redeclare SensitiveResistor R3(dRdT=0.1),
    redeclare SensitiveResistor R4(dRdT=0.1)
  );
end SensitiveCircuit;
```

注意，我们不必另外指定电阻值。原因是电阻的设置是附加在 Circuit 模型的约束类型上的。但我们必须一而再地改变每个电阻模型然后指定 dRdT，即便取值均是一样的。这多少有些无趣。不过，Modelica 提供了一种方法去一次性修改所有部件。首先，在模型内定义一个本地类型：

```
model Circuit
  model ResistorModel = Resistor;
  ResistorModel R1(R=100);
  ResistorModel R2(R=150);
  ResistorModel R4(R=45);
  ResistorModel R5(R=90);
  // ...
equation
  connect(R1.p, R2.n);
  connect(R1.n, R3.p);
  // ...
end Circuit;
```

这样做是为了帮 Resistor 创建别名 ResistorModel。这本身并不能帮助我们一次性改变每个电阻的类型。但通过将 ResistorModel 变为 replaceable 则可以做到这点：

```
model Circuit
  replaceable model ResistorModel = Resistor;
  ResistorModel R1(R=100);
  ResistorModel R2(R=150);
  ResistorModel R4(R=45);
  ResistorModel R5(R=90);
  // ...
equation
  connect(R1.p, R2.n);
  connect(R1.n, R3.p);
  // ...
end Circuit;
```

若我们的 Circuit 模型定义如上，那么就可以创建如下的 SensitiveCircuit 模型：

```
model SensitiveCircuit
  extends Circuit(
    redeclare ResistorModel = SensitiveResistor(dRdT=0.1)
```

```
 );
end SensitiveCircuit;
```

所有的电阻元件类型仍为 ResistorModel，我们没有必要重声明任何一个元素。通过将 ResistorModel 类型定义改变为 SensitiveResistor( $dRdT=0.1$ )，我们做到了重新定义什么是 ResistorModel。注意， $dRdT=0.1$  这一更改将应用到类型为 ResistorModel 的所有组件。从技术上讲，这不是组件类型的重声明，而是一个类型的重定义。但是，我们再次使用了 redeclare 关键字。

有趣的是，这些重定义仍然有默认类型和约束类型的概念。重定义类型的一般语法为：

```
replaceable model AliasType = DefaultType(...) constrainedby ConstrainingType(...);
```

正如可替换组件的情况一样，任何对默认类型 DefaultType 进行的修改，只会在不重新定义 AliasType 的情况下有效。不过，任何对约束型 ConstrainingType 进行的修改，都会在重定义之后会继续有效。此外，AliasType 必须总是插件兼容于约束类型。

虽然相对于可更换组件，语言的这个特性使用并不太频繁。但这个特性可以节省时间，并有助于避免在特定情况下的错误。

## 选择

这部分重点是配置管理。我们已经了解到，约束类型决定 redeclare 时可选的模型。如果由一个模型开发人员同时创建架构模型以及所有兼容的实现，那么他会知道需要使用什么约束类型匹配潜在的配置。

但如果你在使用的是由别人开发的架构模型呢？你如何确定存在哪些可能模型呢？好在 Modelica 语言规范包括一些标准标注以解决这一问题。

### choices

choices 标注允许原模型的开发者将给定声明和一系列修改相关联。在最简单的情况下，用户可以为特定参数指定不同的值：

```
parameter Modelica.SIunits.Density rho
annotation(choices(choice=1.1455 "Air",
choice=992.2 "Water"));
```

在本例，模型开发者为 rho 参数列出了几个用户可能使用的赋值。而上述每个选项均为对 rho 变量的修改。此信息常用在图形化的 Modelica 工具里，目的是为用户提供智能化的选项。

这个功能也可以很容易地在配置管理的情况下使用。请看下面的例子：

```
replaceable IdealSensor sensor constrainedby Sensor
annotation(
choices(
choice=redeclare SampleHoldSensor sensor
"Sample and hold sensor",
choice=redeclare IdealSensor sensor
"An ideal sensor"));
```

同样，模型开发者可以将一组可能的修改连同声明一起。在使用图形工具时，这些 choice 值可为用户提供一套合理的选项去配置系统。

### choicesAllMatching

不过，这里有一个问题。要明确列出所有这些选择不仅麻烦，而且适合的模型选项也可能会改变。毕竟，其他开发人员（原始模型开发者以外的人）也可能去创建某个特定约束型的实现。为何不让用户在配置自己的系统时可以选择见到所有的合规选项呢？

好在 Modelica 正正包含了这样的标注。这个标注就是 `choicesAllMatching`。只要在给定的声明（或 `replaceable` 定义）将此标注设置为 `True`，Modelica 工具就会因此而查找所有合规的选项，并将其展现在用户界面上。例如，

```
replaceable IdealSensor sensor constrainedby Sensor
annotation(choicesAllMatching=true);
```

通过添加此标注，建模工具会在用户用图形界面重新配置他们的模型时找到所有合规的重声明模型。这大大可以增加架构模型的实用性。原因是，此标注在稍微增加模型开发者的工作量的前提下为用户提供了全方位的选择。

## 结论

在本节中，我们讨论了 Modelica 的配置管理功能。本功能与 Modelica 其他的方面有着相同目标：促进重用，提高生产效率，并确保正确性。Modelica 的包括许多功能强大的选项去进行组件重声明以及类型重定义。将这点与 `choicesAllMatching` 标注结合，模型可以用一些明确的选项去支持大量的配置。

## 第 9.2.3 节 可扩展连接器

### 定义

正如我们在本章热力控制 (304) 例子里看到的，要建构一个架构模型时很难定义一个接口去涵盖所有重要配置。在这种情况下，一种解决方法就是用 `expandable` 连接器。

可扩展连接器使用与正常连接器定义完全相同的语法（见前面连接器定义 (167) 的讨论）。唯一不同之处在于定义前加入了 `expandable` 限定词，如：

```
expandable connector ConnectorName "Description of the connector"
  // Declarations for connector variables
end ConnectorName;
```

倘若我们希望的话，就可以忽略 `expandable` 限定词而把这个连接器当成一个正常的连接器来使用。上述使用所依赖的假设是，将可扩展连接器作为正常连接器来使用的组件其方程数目正确，使得任何组件模型均为平衡的组件 (241)。

但是，如果用户将信号连接到某个有特定名称的元素，而该元素不是连接定义内的一部分，那么该新元素将添加到连接器的该实例里。此新元素的类型将和与其相连的连接器相同的。最终，该元素将被添加到连接集 (239) 内任何其他连接器的实例上。

这样，`expandable` 连接器定义便可以“增长”出原来没有的附加信号。这样，可扩展连接器的任何接口定义就具有灵活性，可以在初始连接器定义的基础上交换更多的信息。正如我们在热力控制 (304) 例子里看到的一样，这个功能在当接口因应控制器的不同而显著改变时非常有用。

到目前为止，可扩展连接器最常用的用例是用向连接器添加额外 `input` 和 `output` 信号。这通常出现在传感器、执行器这些子系统的实现细节改变时。因为控制器与上述部件配对所需信息会因此而改变。例如，凸轮相位控制的内燃机将需要来自控制器的凸轮角指令值。但是，如果发动机没有这个功能则没有这样的必要。

不过，`expandable` 连接器也可以连接非因果的子连接器，以此为子系统附加新的物理相互作用。例如，某些子系统可能不包含热效应，但另一些则包含。包含热效应的子系统可以加入非因果连接器，从而允许其它子系统与之进行热交换。

### 结论

可扩展接口定义规定了连接器所拥有变量的最小集。使用可扩展接口的前提是，所有元件模型必须让连接器内定义变量数目与方程数目相平衡。此外，附加的变量可以随时添加至 `expandable` 连接器内。为此，整个系统模型仍要平衡。可扩展连接器一般用于与接口定义共同使用。这样就可以定义架构模型所需最小接口，以便通过选择子系统的不同实现来实现扩展。



# 第三部分

## 索引及表格



- genindex
- search



---

参考文献

---

- [Lotka] Lotka, A.J., “Contribution to the Theory of Periodic Reaction”, J. Phys. Chem., 14 (3), pp 271–274 (1910)
- [Volterra] Volterra, V., Variations and fluctuations of the number of individuals in animal species living together in Animal Ecology, Chapman, R.N. (ed), McGraw-Hill, (1931)
- [Guldberg] C.M. Guldberg and P. Waage, "Studies Concerning Affinity" C. M. Forhandlinger: Videnskabs-Selskabet i Christiana (1864), 35
- [Elmqvist] “Fundamentals of Synchronous Control in Modelica”, Hilding Elmqvist, Martin Otter and Sven-Erik Mattsson <http://www.ep.liu.se/ecp/076/001/ecp12076001.pdf>
- [Elmqvist] “Fundamentals of Synchronous Control in Modelica”, Hilding Elmqvist, Martin Otter and Sven-Erik Mattsson <http://www.ep.liu.se/ecp/076/001/ecp12076001.pdf>
- [Berg] Richard E. Berg, “Pendulum waves: A demonstration of wave motion using pendula” <http://dx.doi.org/10.1119/1.16608>
- [ItPMwM] Michael M. Tiller, “Introduction to Physical Modeling with Modelica” <http://www.amazon.com/Introduction-Physical-Modeling-International-Engineering/dp/0792373677>