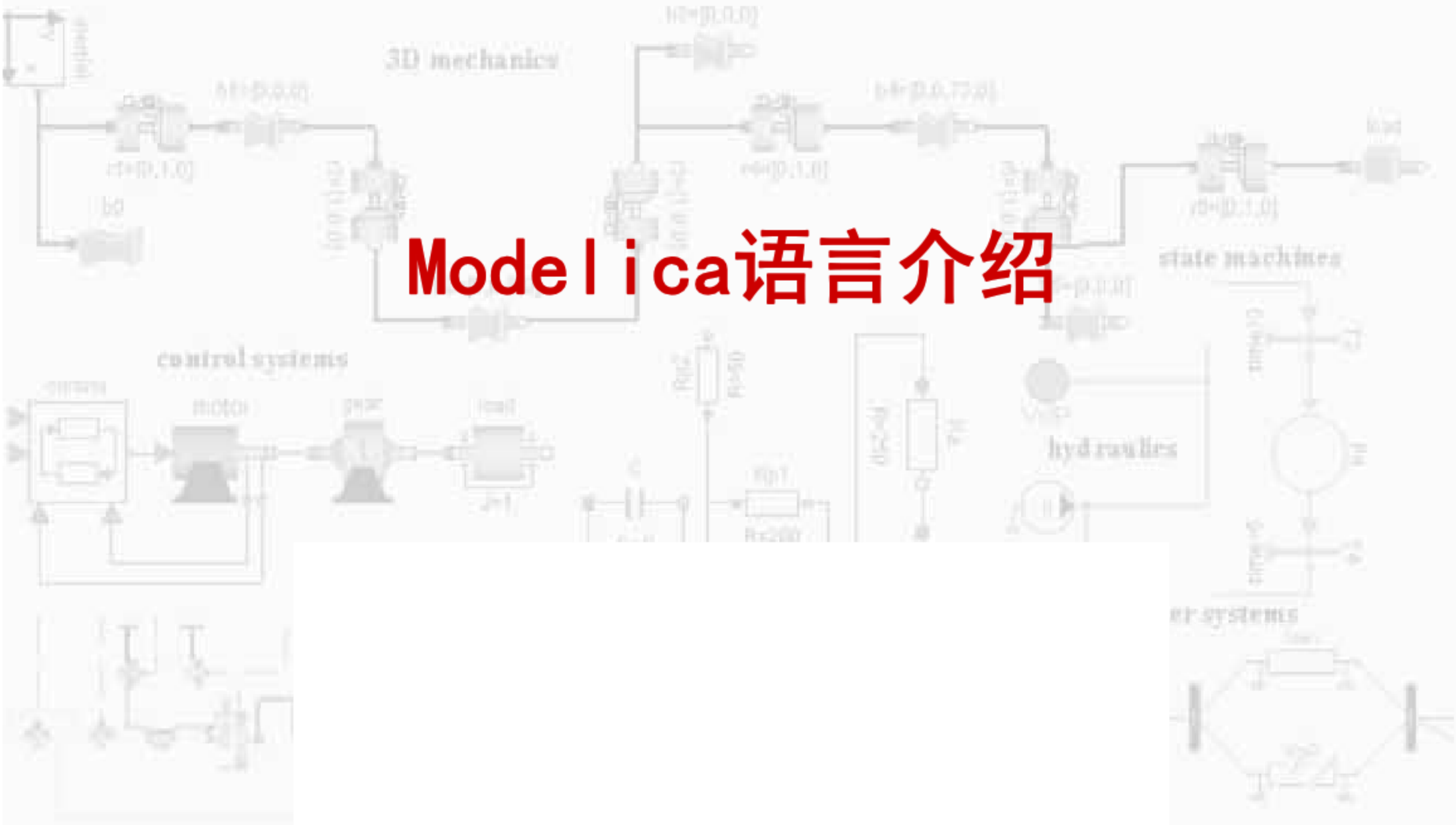
MWorks平台&Modelica语言系列

Modelica语言介绍



Modelica语言

- Modelica—A United Object-Oriented Language for Physical Systems Modeling
  - 统一的面向对象物理系统建模语言
- Modelica建模语言由瑞典Linköping的非赢利组织Modelica协会开发，是一种适用于大规模复杂异构物理系统建模的面向对象语言，可以免费使用。Modelica可以满足多领域建模需求，例如机电模型（机器人、汽车和航空应用中的机电系统包含了机械、电子、液压和控制子系统）、过程应用、电力发电和输送等。Modelica模型的数学描述是微分、代数和离散方程（组），相关的Modelica工具能够决定如何自动求解方程变量，因而无需手工处理。对具有超过10万个方程的大规模模型，可以使用专门的算法进行有效处理。Modelica适用于半实物仿真和嵌入式控制系统。

Modelica目标

- 统一面向对象的建模语言。
- 物理模型高度可复用性。
- 覆盖不同工程学科的物理构件。
- 支持LE/NLE/ODE/DAE/DE/PDE方程描述。
- 生成高效仿真代码。
- 开发构件库。

Modelica特点

- 基于方程的非因果建模
  - 相对于赋值语句，方程体现了不同的数据流向，能够更好地支持组件复用。
- 多领域建模
  - 能够描述电气、机械、热力学、液压、生物、控制等多领域模型或组件，已有大量可复用的领域库。
- 面向对象建模
  - 具有面向对象语言特征：类、泛型（C++模板）、子类型，允许组件复用和模型进化。
  - 提供良好的软组件模型，通过组件（接口）相互连接，快速搭建复杂物理系统。
- 连续-离散混合建模

主要内容

- 认识Modelica
- 面向对象的数学建模
  - 类及其实例
  - 继承
  - 泛型类
- 方程
- 非因果物理建模
- 基于组件建模
- 抽象类
- 数组
- 算法
- 函数
- 离散事件与混合建模
- 包
- 注解
- 命名惯例
- Modelica标准库(MSL)
- Modelica模型求解流程

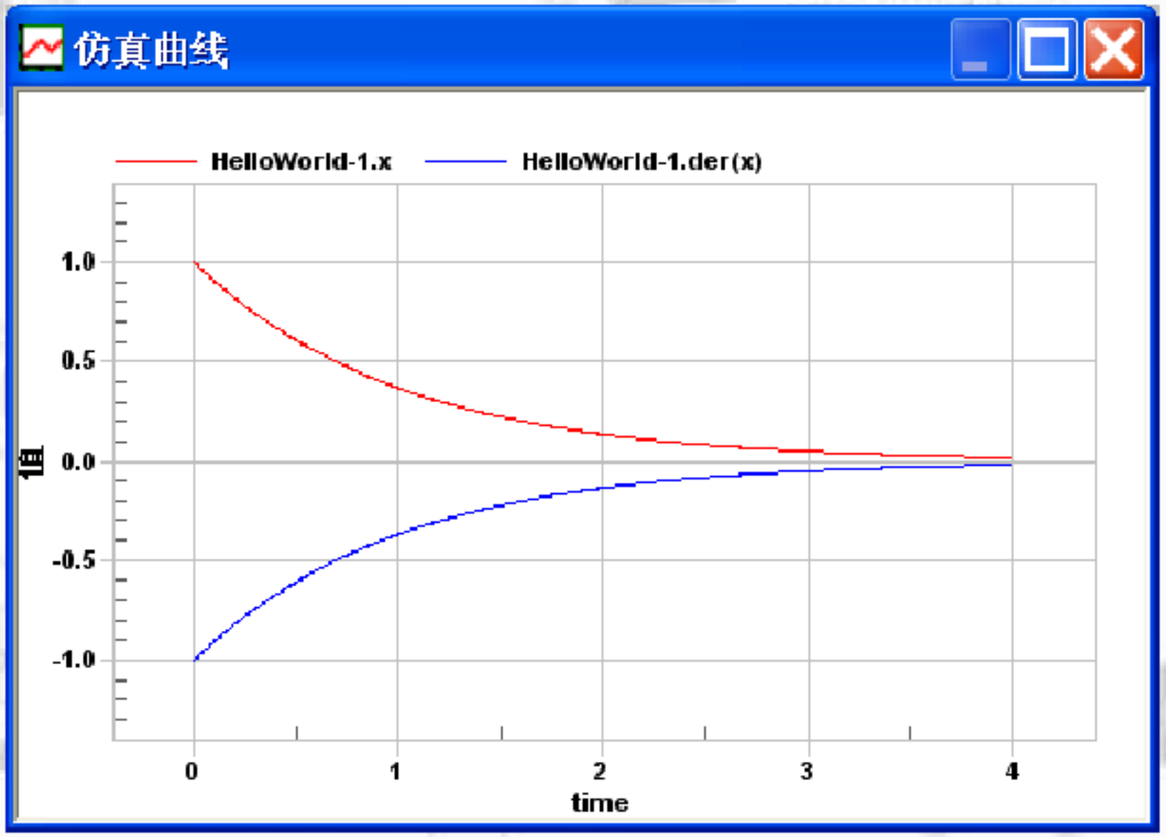
认识Modelica

- 例子：HelloWorld
- 例子：Pendulum
- 例子：DAEExample
- 例子：VanDerPol
- 基本的Modelica元素：
  - 变量（Variable）
  - 方程（Equation）
  - 常量（Constant）
  - 变量初值（start）
  - 注释（Comment）

例子：HelloWorld

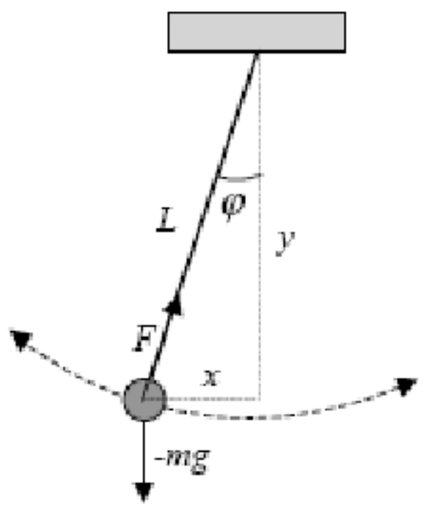
- Modelica模型组成:
  - 声明（变量、参数）
  - 方程（equation）

```
class HelloWorld
 Real x(start=1);
 parameter Real a=1;
equation
 der(x) = -a*x;
end HelloWorld;
```



例子：Pendulum

- Modelica变量前缀:
  - 常量（constant）
  - 输出（output）



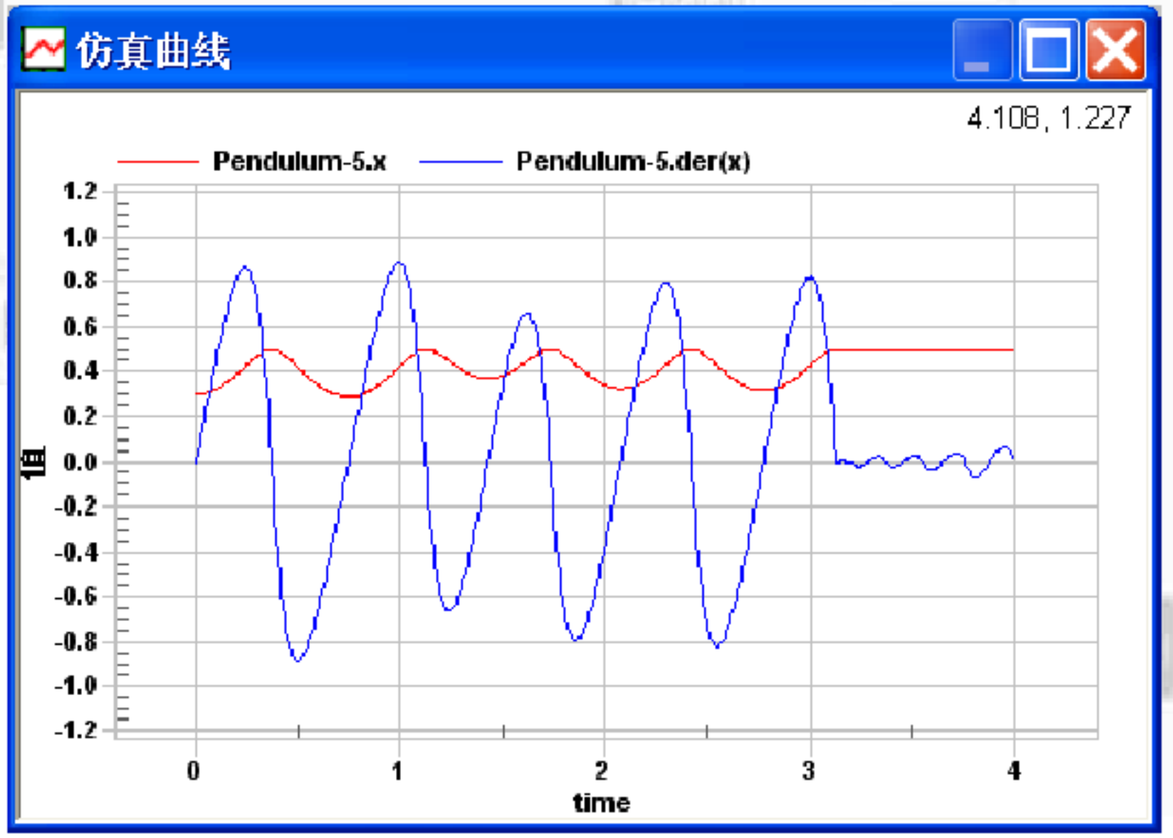
$$m\dot{v}_x = -\frac{x}{L}F$$

$$m\dot{v}_y = -\frac{y}{L}F - mg$$

$$\dot{x} = v_x$$

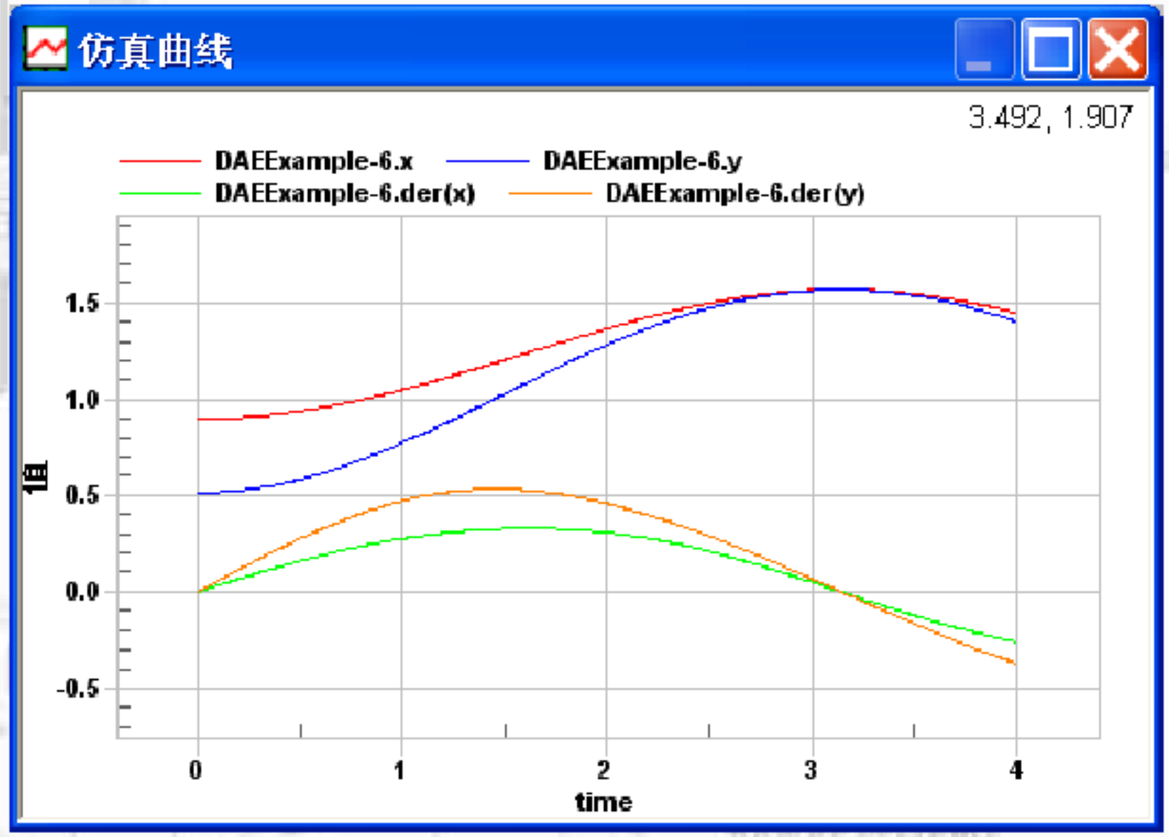
$$\dot{y} = v_y$$

$$x^2 + y^2 = L^2$$



例子：DAEExample

```
class DAEExample
 Real x(start=0.9);
 Real y;
equation
 der(y)+(1+0.5*sin(y))
   *der(x)=sin(time);
 x-y=exp(-0.9*x)  *cos(y);
end DAEExample;
```



变量数=方程数！

变量（Variable）

- Real
  - 双精度64位浮点数
- Integer
  - 32位整数
- enumeration(...)
- Boolean
  - true/false
- String
  - 文字串

```
enumeration ShirtSize = { Small,Middle,Large };
```

```
ShirtSize eSize = ShirtSize.Small;
```

常量（Constant）

- 用途:
  - 作为助记符增强可读性。
  - 易于维护。
- 命名常量（Named Constant）
  - constant Real PI=3.141592653589793;
  - constant String RedColor="Red";
  - constant Integer one=1;
- 参数常量（Parameter Constant）
  - 参数常量在仿真过程中是不变的。
  - parameter Real mass=20;

变量初值（start）

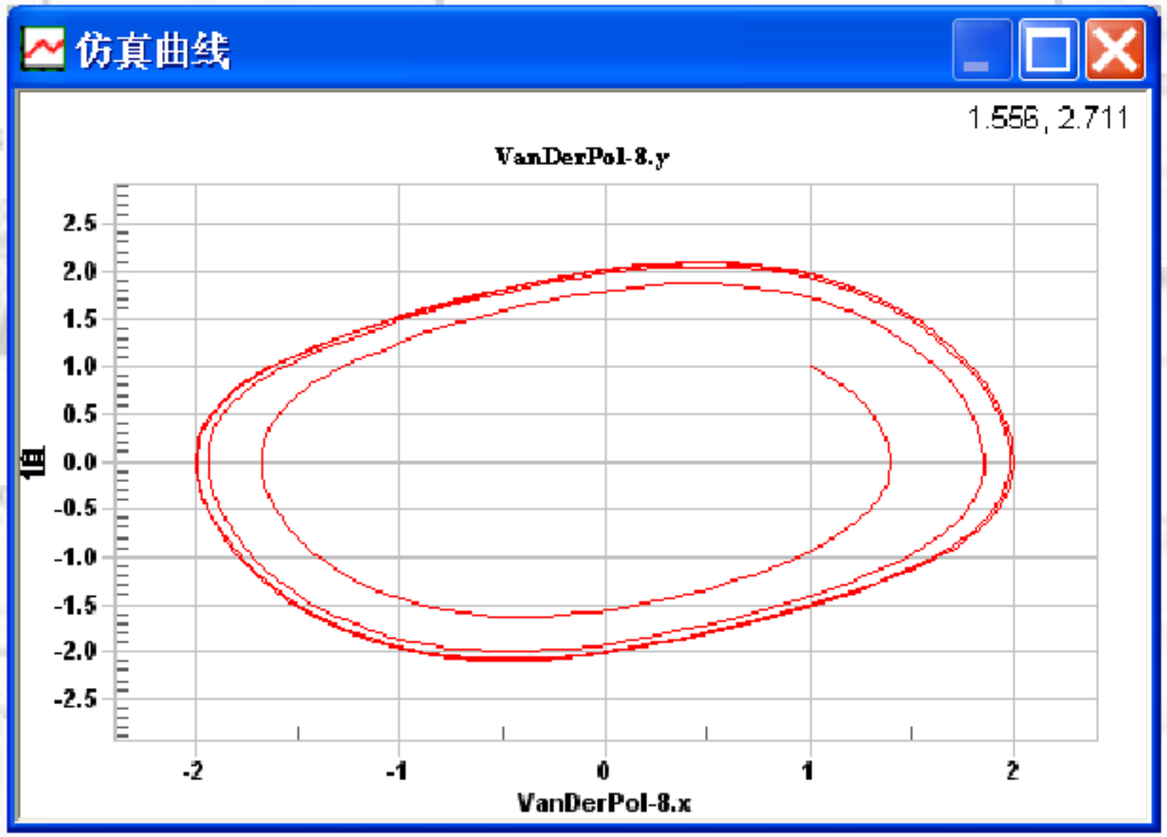
- 数值型变量默认0
- 布尔型变量默认false
- 字符串型变量默认空串
- 局部变量和函数返回值默认未定义
- 仿真初始值:
  - Real x(start=1) "Descriptive string for x";
  - Real y(start=1,fixed=true);

注释（Comment）

- comment（语法成分）
  - Real x(start=1) "Descriptive string for x";
- 单行注释
  - //...　（C++方式）
- 多行注释Real
  - /*...*/　（C方式）

例子：VanDerPol

- Example: VanDerPol
  - 注释（Comment）
  - 自定义X轴



国际单位制（SIunits）

- 7 国际标准基本单位:
  - kg, m, s, A, K, mol, cd
- 两个物理量只有在单位相同时才可以比较。

类型	单位	国际标准单位
Moment	N · m	$kg \cdot m^2/s^2$
Energy	J	$kg \cdot m^2/s^2$

标识符命名规则

- 标识符包括类名和组件名。
- 规则:
  - 以字母或下划线开始，后跟任意字母、数字或下划线，如: v1，_x1。
  - 以单引号包围的任意字母、数字或下划线，如: 2v′，2ndPt′。

面向对象的数学建模

- Modelica面向对象建模:
  - 强调陈述式结构和数学模型复用，通过层次式的组件连接和继承（Inheritance）实现结构化建模。
  - 系统的动态行为通过方程进行陈述式的描述（其他面向对象语言采用消息传递方式）。
  - 对象管理其中的实例变量和方程，并通过方程实现数据共享。

类和实例（Class and Instance）

- 类（Class）
- 实例（Instance）
- 初始化（Initialization）
- 受限类（Restricted Classes）
- 实例变型及其复用（Reuse by Modifier）
- 内置类型（Built-in Classes）
- 继承（Inheritance）
- 泛型类（Generic Class）

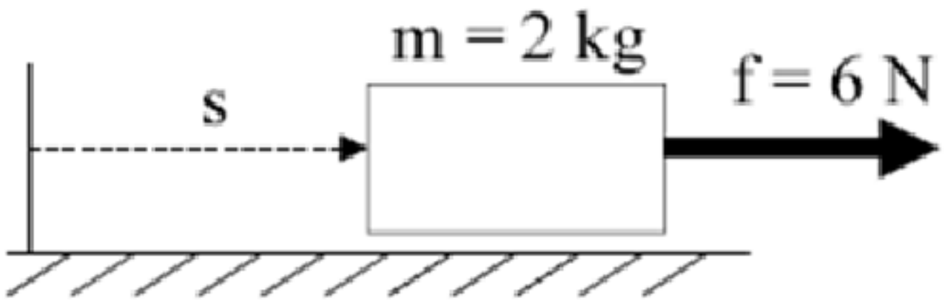
类（Class）

- Modelica类/模型的结构:
  - 变量：或称为实例、组件，对于时变变量，变量值记录求解过程中当前时刻的结果。
  - 方程：表示模型行为，与其他存在数据交互的模型方程一起决定求解过程（LE/NLE/ODE/DAE/DE/PDE）。
  - 模型可以是其他类的嵌套类（递归定义）。

```
class Point "Point in 3D space"
public
  Real x;
  Real y,z;
end Point;
```

```
Point pt;
pt.x := 20;
pt.x  = 20;
```

例子：MovingMass



$$m \cdot \ddot{s} = f$$

```
model MovingMass
  parameter Real m=2;
  parameter Real f=6;
  Real s;
  Real v;
equation
  v = der(s);
  m*der(v) = f;
end MovingMass;
```

实例（Instance）

- 类的实例化
  - 声明一个类的变量即创建实例。
- 理解主模型的概念

```
class Triangle
  Point point1;
  Point point2;
  Point point3;
end Point;
```

```
class Foo1
  ...
end Foo1;
class Foo2
  ...
end Foo2;
class Main
  Triangle pts;
  Foo1 f1;
end Main;
```

初始化（Initialization）

```
class Triangle
  Point point1(start={1, 2, 3});
  Point point2;
  Point point3;
end Point;
```

```
class Controler
  Real y;
equation
  der(y) = a*y+b*u;
Initial equation
  der(y) = 0;
end Controler;
```

```
class Main
  Triangle pts1
    (point1.start={1, 2, 3});
  Triangle pts2;
  Fool f1;
end Main;
```

```
der(y) = 0;
y = -(b/a)*u;
```

受限类（Restricted Classes）

- class:
  - model
  - connector
  - record
  - block
  - type
  - function
  - package
- 用途:
  - 更精确地表达模型含义
  - 增强易用性和可读性
  - 便于编译器作限制性检查
  - 可以改成class而不影响模型行为

受限类使用场景

类型	名称	作用
class	类	通用类
model	模型	陈述式模型
connector	连接器	组件之间的连接接口
record	记录	数据结构
block	框图	兼容基于框图的因果建模
type	类型	类型别名
function	函数	通过算法实现过程式建模
package	包	消除名字冲突和组织模型层次

受限类Model

- 不能用于连接。

```
model Pendulum
  ...
end Pendulum;
```

受限类Connector

- 类定义及其任何组件中不允许有方程。

```
connector Pin
  Voltage        v;
  flow Current   i;
end Pin;
```

受限类Record

- 只有公有（public）变量。
- 不能用于连接（connection）。
- 记录不能直接用来仿真。
- 成员变量不能声明为input/output、inner/outer、flow。

```
record Person
   Real    age;
   String  name;
end Person;
```

受限类Block

- 表示固定的因果及输入/输出关系的方框。
- 表示接口的每个组件必须是赋值等于输入或输出变量。
- 成员变量或组件，要么为参数或常量，要么为输入或输出变量，即只能带有前缀parameter/constant、input/output。

```
block MIMO
  parameter Integer in_n = 1;
  parameter Integer out_n = 1;
  input Real u[in_n];
  output Real y[out_n];
end MIMO;
```

受限类Type

- Type可以理解为类的别名。
- 用于扩展预定义类型、记录、数组。

```
type Double = Real;
type Vector3d = Real[3];
type Voltage = Real(...);
```

受限类Function

- 函数参数必须是输入变量或输出变量。
- 不能用于连接。
- 没有equation和initial algorithm子句，至多一个algorithm子句，可调用的函数需要有算法子句或外部函数接口。
- 不能调用内置函数der、initial、terminal、sample、pre、edge、change、reinit、delay、cardinality。

```
function MyFunc
  ...
end MyFunc;
```

受限类Package

- 建议只包含类和常量声明。
- 不能直接用来仿真。

```
package MyPack
   ...
end MyPack;
```

实例变型及其复用（Reuse by Modifier）

- Class是模型复用的前提，变型（Modification）修改声明方程，是模型复用的简洁有效手段。

```
model LowPassFilter
  parameter Real T=1;
  Real u, y(start=1);
equation
  T*der(y)+y = u;
end LowPassFilter;
```

```
model FiltersInSeries
  LowPassFilter F1(T=2), F2(T=3);
equation
  F1.u = sin(time);
  F2.u = F1.y;
end FiltersInSeries;
```

```
model ModifiedFiltersInSeries
  FiltersInSeries F12(F1(T=6), F2.T=11);
end ModifiedFiltersInSeries;
```

内置类型（Built-in Classes）

- Real、Integer、Boolean、String、enumeration(...)
  - Real x;
  - 值访问方式: x = x.value
- Voltage（电压）
  - class Voltage = Real(unit="V",min=-220,max=220);

内置属性

名称	含义
quantity	物理量纲
unit	单位
displayUnit	显示单位
value	数值
min	最小值
max	最大值
start	仿真初始值

继承（Inheritance）

- 基类（base class）、超类（superclass）
- 派生类（derived class）、子类（subclass）

```
record ColorData
  Real r;
  Real g;
  Real b;
end ColorData;
```

```
class Color
  extends ColorData;
equation
  r+g+b = 1;
end Color;
```

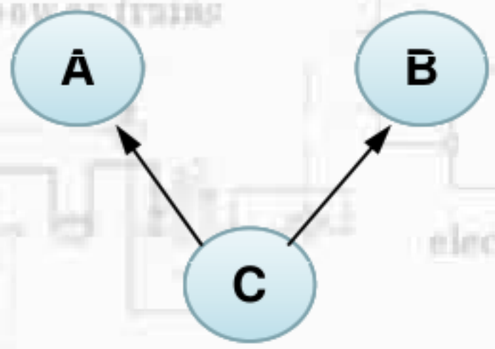
```
class Point
public
  Real x;
  Real y, z;
end Point;
```

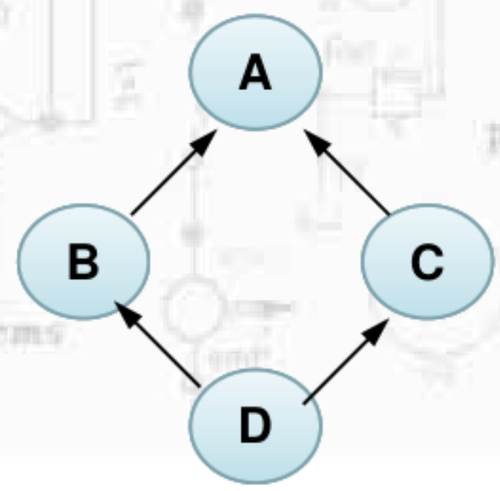
```
class ColorPoint
  Real x, y, z;
  Real r, g, b;
equation
  r+g+b = 1;
end ColorPoint;
```

```
class ColorPoint
  extends Point;
  extends Color;
end ColorPoint;
```

V型继承和菱形继承

- 变量继承:
  - 变量重名时，基类中的变量被覆盖。
  - 变量名称相同但类型不同时报错。
- 方程继承:
  - 方程语法上等价时，基类中的方程被覆盖。
  - 语法上不等价但数学上等价的方程不覆盖，可能会造成方程过约束。





Protected继承

```
class Color
  Real x,y,z;
equation
  r+g+b = 1;
end Color;
```

```
class Point
  Real x;
  Real y,z;
end Point;
```

```
class ColorPoint
public
  extends Point;
protected
  extends Color;
end ColorPoint;
```

```
class ColorPoint
public
  Real x,y,z;
protected
  Real r,g,b;
equation
  r+g+b = 1;
end ColorPoint;
```

泛型类（Generic Class）

- 泛型类也称为参数化的类，类似于C++的模板类
- 类参数可以理解为类的形参（注意区别前缀parameter），通过重新声明组件和类实现模型定制和模型重用。
- 类参数两种作用形式:
  - 类参数作用于实例：替换组件
  - 类参数作用于类型：替换类型
- Modelica关键字:
  - replacable
  - redeclare

类参数作用于实例

```
class C1
  replaceable GreenClass    pObj1;
  replaceable YellowClass   pObj2;
  replaceable GreenClass    pObj3;
  RedClass                  pObj4;
equation
  ...
end C1;
```

```
class C2
  RedClass     pObj1;
  GreenClass   pObj2;
  GreenClass   pObj3;
  RedClass     pObj4;
equation
  ...
end C2;
```

```
class C2 = C1(redeclare RedClass pObj1,
  redeclare GreenClass pObj2);
```

类参数作用于类型

```
class C1
  replaceable class ColoredClass =
    GreenClass;
  ColoredClass                p0bj1;
  replaceable YellowClass     p0bj2;
  ColoredClass                p0bj3;
  RedClass                    p0bj4;
equation
  ...
end C1;
```

```
class C2
  BlueClass      p0bj1;
  YellowClass    p0bj2;
  BlueClass      p0bj3;
  RedClass       p0bj4;
equation
  ...
end C2;
```

```
class C2 = C1(redeclare class ColoredClass =
  BlueClass);
```

方程（Equation）

- 方程表示模型的动态行为:
  - 传统语言中的赋值语句在Modelica中通常视为方程。
  - 属性赋值表现为方程。
  - 组件之间的连接（connect）将用于生成方程。
- Modelica单赋值原则:
  - 方程与变量数目相同。

- 方程vs赋值语句
- 产生方程的场景
- 方程语法形式
- 偏微分方程（PDE）

方程vs赋值语句

- 方程:
  - R*i = v
- 赋值语句:
  - i := v/R
  - v := R*i
  - R := v/i

产生方程的场景

- 正规方程（Normal Equations）
  - 出现在equation部分，包括特殊形式的connect方程。
- 声明方程（Declaration Equations）
  - 声明部分中的变量或常量赋值语句
  - constant Integer one = 1;
  - parameter Real mass = 20;
  - Real speed = 72;
- 变型方程（Modification Equations）
  - 用于修改（modify）属性。
- 初始方程（Initial Equations）
  - 用于仿真开始时刻求解DAE系统的初值。
  - 出现在initial equation部分，或带有start属性的变量声明。
  - Real speed(start=72);

方程语法形式

- 正规方程:

```
...
equation
...
<some other keyword>
...
```

- 初始方程:

```
...
initial equation
...
<some other keyword>
...
```

等式方程（Equality Equation）

```
expr1 = expr2;
(out1,out2) = func(exp1,exp2);
```

```
class EqualityEquations
  Real x,y,z;
equation
  (x, y) = func(1.0, 2.0); //正确!
  (x+1, 3.0) = func(1.0, 2.0); //错误!左边必须是变量
end EqualityEquations;
```

迭代方程（Repetitive Equation）

```
y = a[1] + a[2]*x + a[3]*x^2 + a[4]*x^3 + ... + a[n+1]*x^n
```

方程形式:

```
xpowers[1] = 1;
xpowers[2] = xpowers[1]*x;
...
xpowers[n+1] = xpowers[n]*x;
y = a * xpowers;
```

简化1:

```
for i in 1:n loop
  xpowers[i+1] = xpowers[i]*x;
end for;
```

简化2:

```
xpowers[2:n+1] = xpowers[1:n]*x;
```

连接方程（Connection Equation）

```
connect(connector1,connector2);
```

```
class RegComponents
  Component components[n];
equation
  for i in 1:n-1 loop
    connect(components[i].outlet,components[i+1].inlet);
  end for;
end RegComponents;
```

assert() and terminate()

```
assert(assert-expression, message-string)
terminate(message-string);
```

```
lowlimit = -5;
highlimit = 5;
assert(x >= lowlimit and x <= highlimit, "x out of limit");
```

```
when apollo.height < 0 then
  terminate("The moon lander touches the ground of the moon");
end when;
```

非因果物理建模

- 非因果建模是一种陈述式建模方式，意味着基于方程而不基于赋值语句。方程不管哪个变量是输入（已知）、哪个变量是输出（未知）；对于赋值语句，赋值符号左边总是输出，右边总数输入。
- 基于方程的模型其因果特性是不明确的，只有在方程系统求解时才确定固定变量的因果关系。
- 非因果建模适于表达复杂系统的物理结构，基于方程的Modelica模型也比传统包含赋值语句的模型具有更强的复用性。

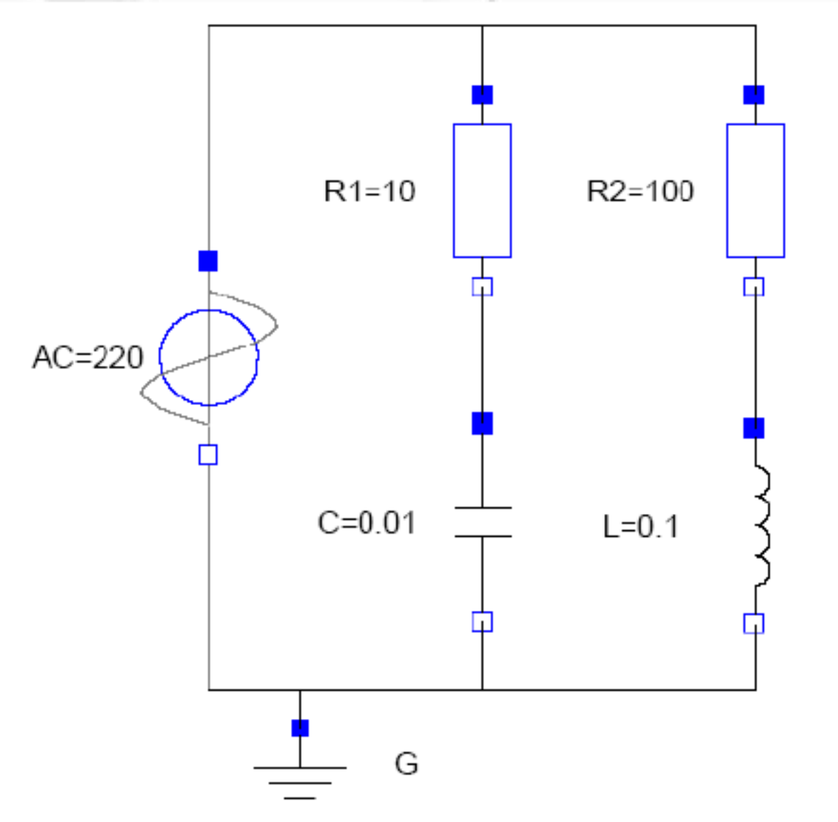
比较非因果建模与基于块图建模

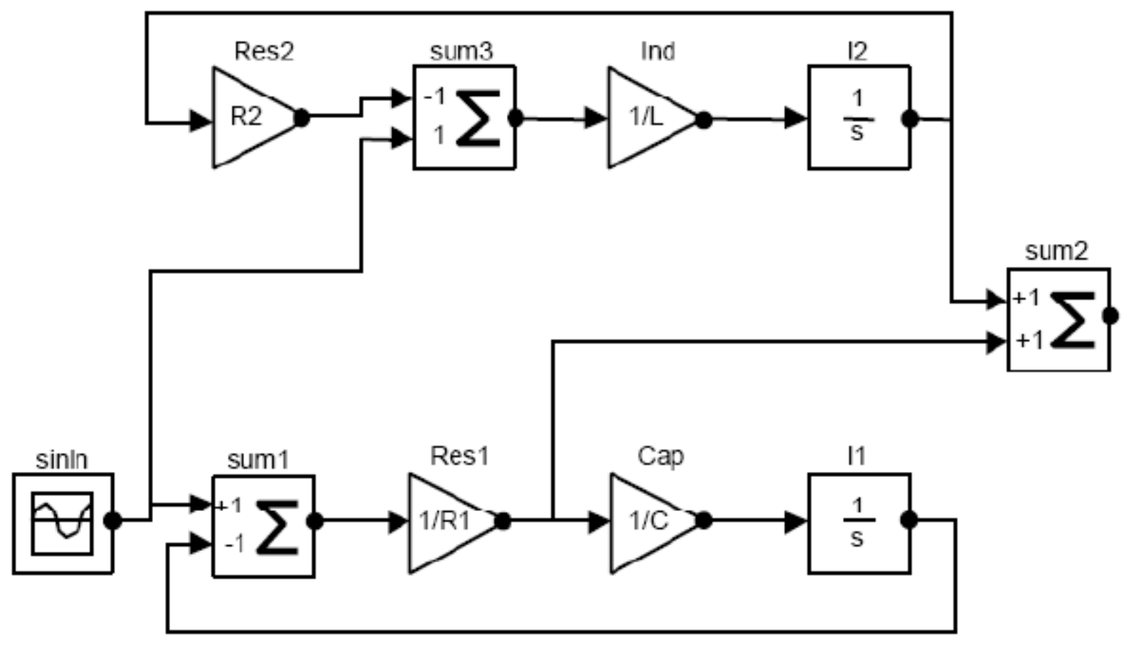
What to do?　　How to do?

陈述式模型只描述问题，不说明如何求解	陈述式	过程式	过程式模型的模型描述和求解是一体的
模型内部是非因果关系	非因果	因果	模型内部是因果关系
v = R*i	方程	赋值	v := R*i i := v/R R := v/i

图示非因果建模与基于块图建模

拓扑结构与物理结构一致！





基于组件建模

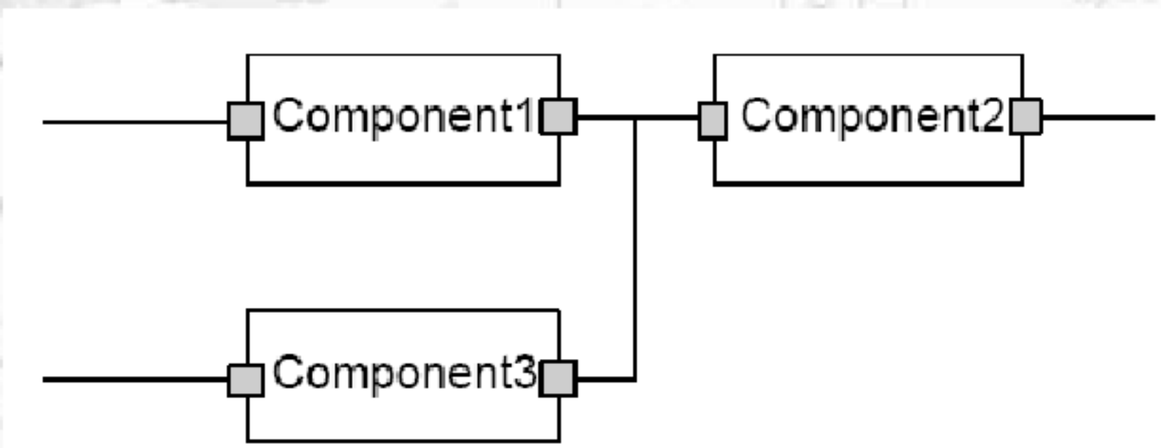
- 软组件模型（The Modelica Software Component Model）
  - 组件：类的实例
  - 连接机制：表示为连接图（Connection Diagram）
  - 组件框架：实现组件与连接机制，确保连接正常工作，并维护连接之间的约束关系。
- 组件（Components）
- 连接图（Connection Diagram）
- 连接器（Connectors）
- 连接（connect）
- 连接方程（Connection Equations）
- 隐式连接（Inner/Outer）

组件（Component）

- 组件是Modelica Class（预定义、MSL内置、自定义、…）的实例。
- 组件与其他组件或外部进行通信（或称之为耦合）是通过接口（Interface）进行的。
  - 端口（Port）
  - Modelica连接器（Connector）
- 组件定义:
  - 组件独立于使用环境来定义。
  - 组件只应包括连接器、必要的方程和局部变量。
  - 组件可嵌套其他互连的组件，并以此实现层次化建模。

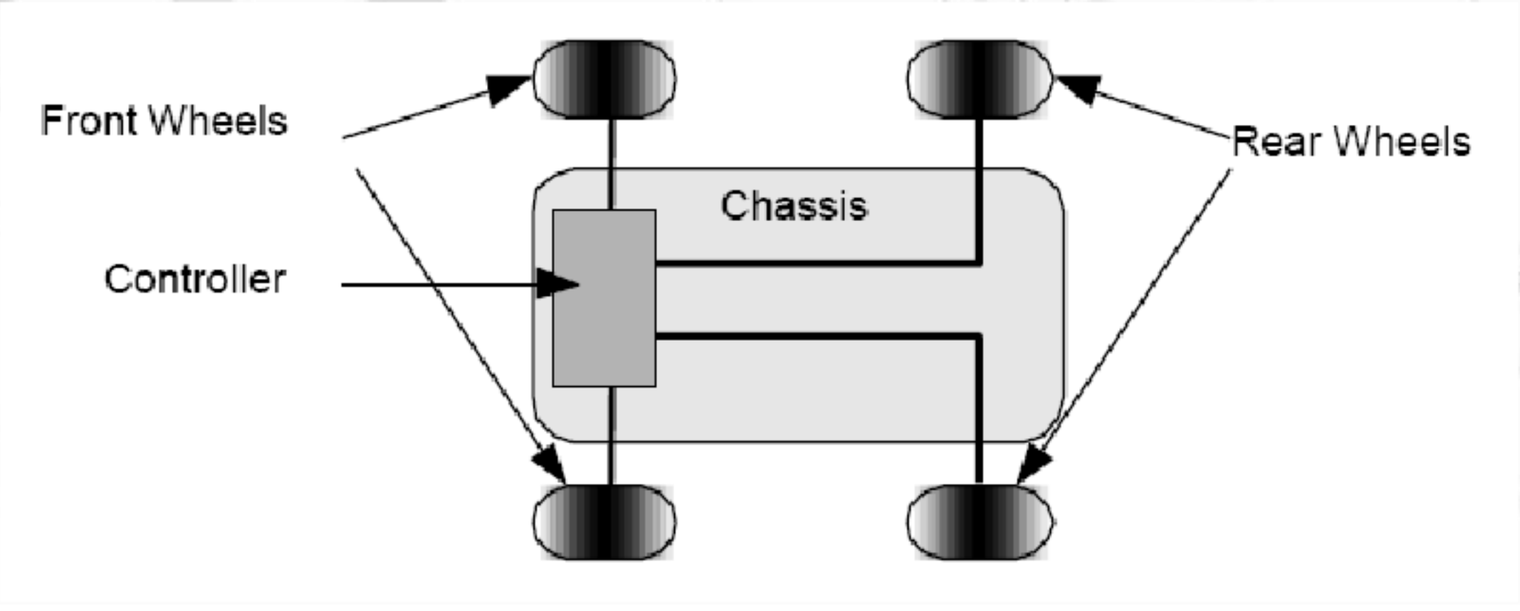
连接图（Connection Diagram）

- 连接图是组件及其连接的图示化表达。
  - 大矩形框表示一个物理组件，如电阻、电容、转换器、齿轮、阀等。
  - 连线对应物理连接，如电线、法兰、管道等。
  - 小矩形框表示连接器，其中的变量和方程用于实现组件之间的数据通信。



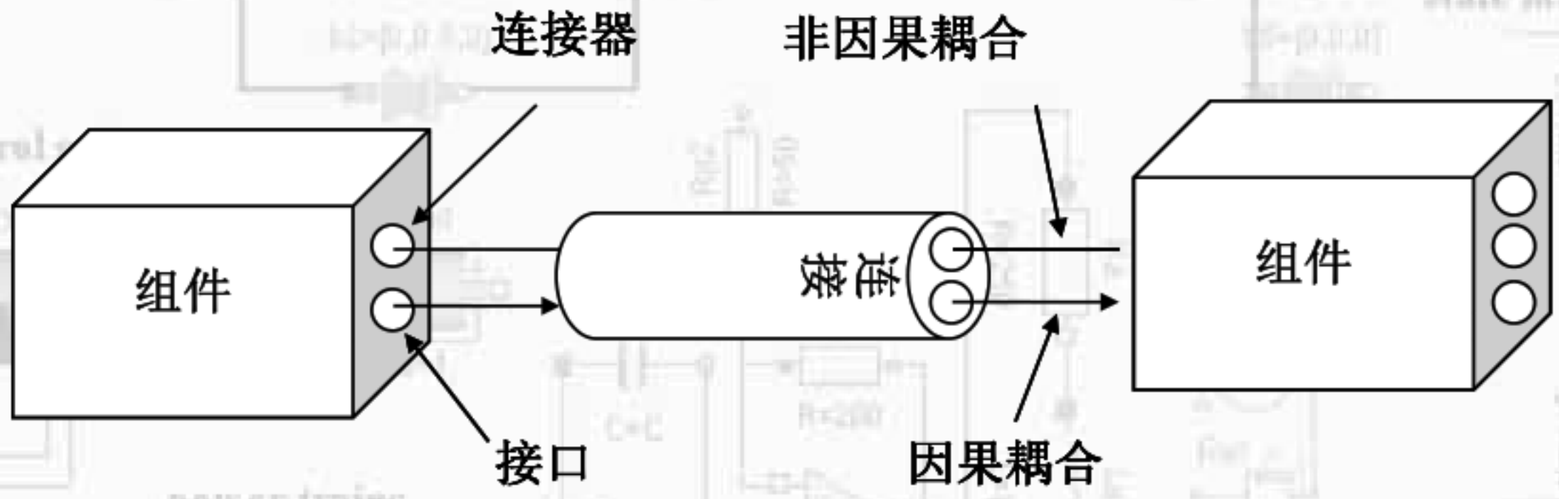
例子：Car

```
class Car "A car class to combine car components"
  Wheel          w1,w2,w3,w4   "四个车轮";
  Chassis        chassis       "底盘" ;
  CarController  controller    "控制器";
  ...
end Car;
```



连接器（Connector）

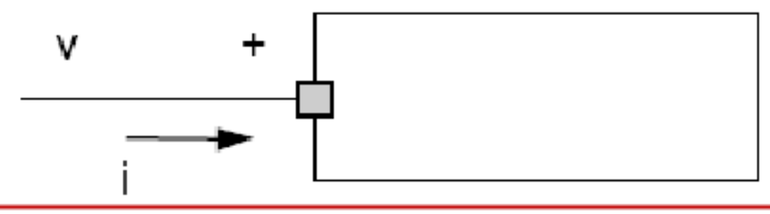
- 连接器是连接器类（connector）的实例，用于定义组件与外部的通信接口（变量）。



例子：Pin

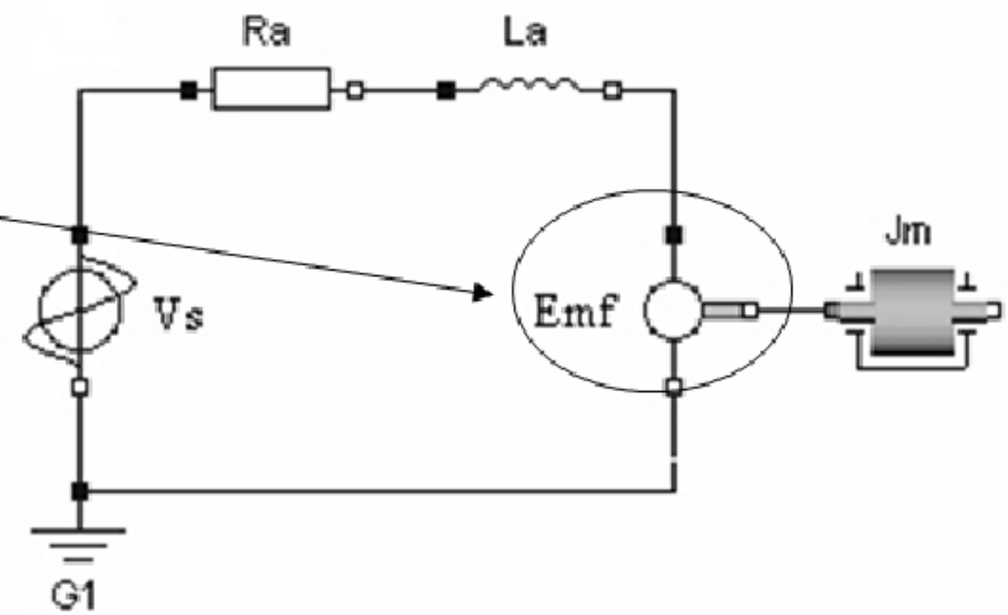
```
type Voltage = Real(unit="V");
type Current = Real(unit="A");
```

```
connector Pin
  Voltage      v;
  flow Current  i;
end Pin;
```



例子：EMF

```
model EMF
    parameter Real k = 1;
    Voltage v;
    Current i;
    AngularVelocity w;
    Pin p, n;
    Flange flange_b
equation
    v = p.v - n.v;
    0 = p.i + n.i;
    i = p.i;
    w = der(flange_b.phi);
    k*w = v;
    flange_b.tau = -k*i;
end EMF;
```



多领域建模的关键在于设计出组件转换器！

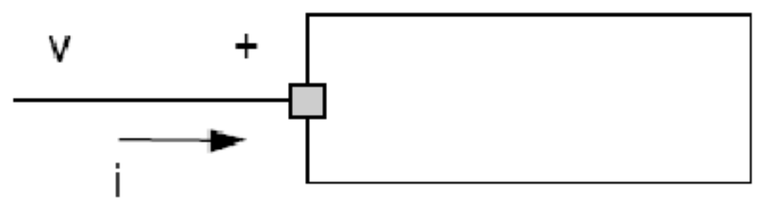
连接（Connection）

- 组件连接建立其连接器变量之间的当量关系。
  - 一般用于表现非因果关系，数据流向在建模时不确定。
  - 采用具有input或output属性的连接器建立因果连接。
- 连接在模型编译之后被转换成方程。

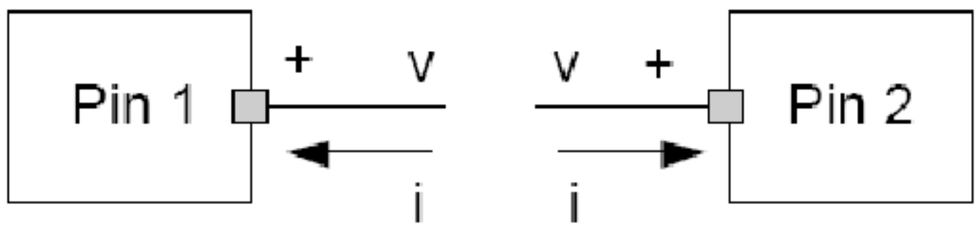
连接方程理论基础

- 连接器的变量耦合关系：
  - 对于非flow变量（也称势变量Cross，如电压、位移、转角）取等值（符合Kirchhoff第一定律）。
  - 对于flow变量（也称流变量Through，如电流、力、力矩）取和零（符合Kirchhoff第二定律）。

Voltage v;
flow Current i;
连接点上所有电压相等;
通过连接点（一般是正极）流入的所有电流之和为零。



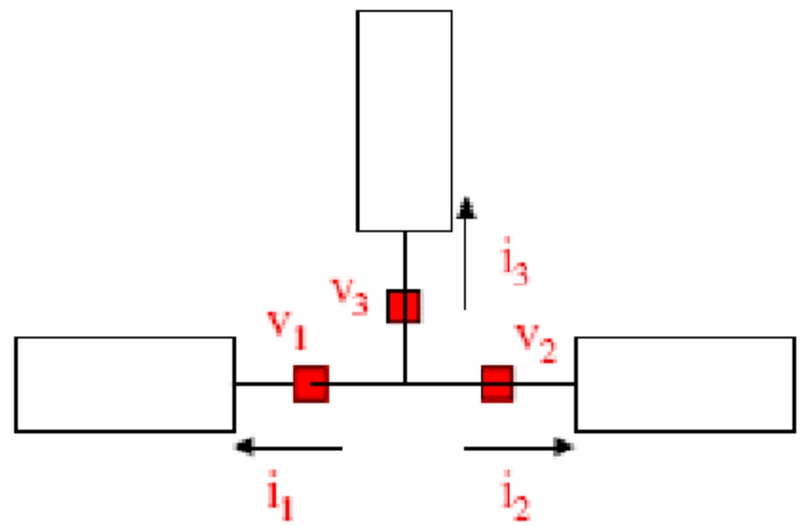
连接方程（Connection Equation）




```
connect(pin1, pin2);
```

```
pin1.v = pin2.v
pin1.i + pin2.i = 0;
```

多点连接方程




```
pin1.v = pin2.v = pin3.v
pin1.i + pin2.i + pin3.i = 0;
```

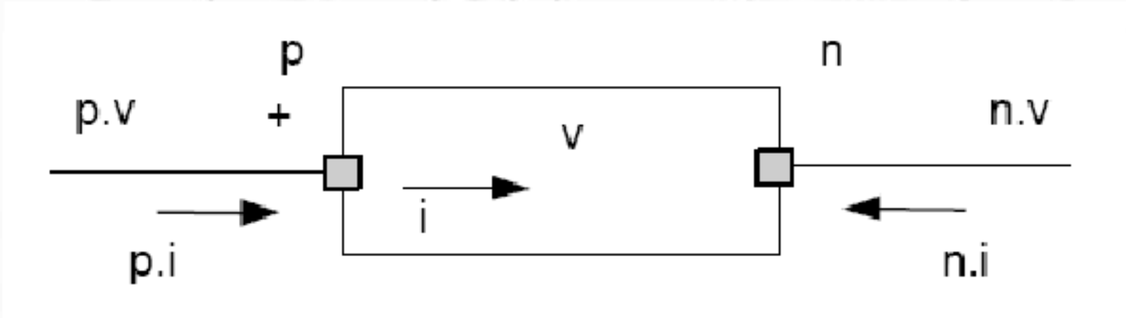
常用流变量和势变量

领域	势变量		流变量	
电学	电压	V	电流	i
一维平移	位移	s	力	f
一维转动	角度	φ	转距	τ
流体液压	压强	p	流速	$\dot{V}$
热力学	温度	T	热流	$\dot{Q}$
化学	化学势	μ	质点流量	$\dot{N}$

抽象类（Partial Class）

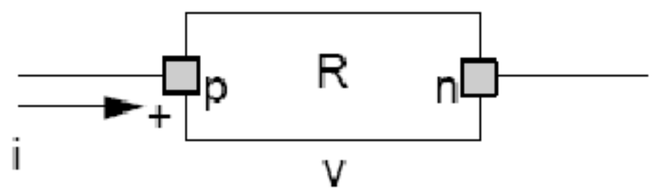
- 抽象类（Abstract Class）：
  - 捕捉共性属性。
  - 方程不完整（物理行为描述不完整）。

```
partial class TwoPin "Superclass of elements with 2 pins"
  pin  p, n;
  Voltage  v;
  Current  i;
equation
  v = p.v - n.v;
  0 = p.i+n.i;
  i = p.i;
end TwoPin;
```



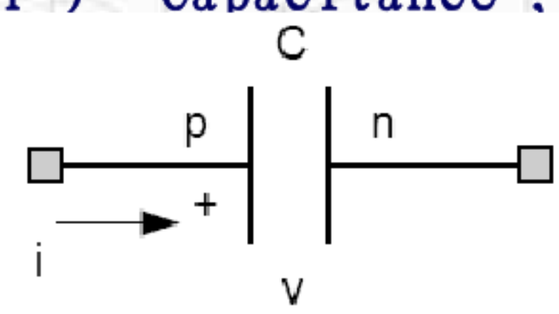
抽象类的复用（电阻与电容）

```
class Resistor "Ideal electrical resistor"
  extends TwoPin;
  parameter Real R(unit="Ohm") "Resistor";
equation
  R*i = v;
end Resistor;
```




```
v = p.v - n.v;
0 = p.i+n.i;
i = p.i;

R*i = v;
```

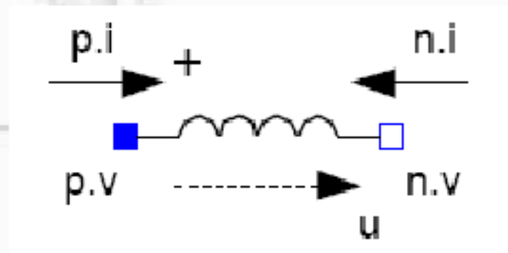
```
class Capacitor "Ideal electrical capacitor"
  extends TwoPin;
  parameter Real C(unit="F") "Capacitance";
equation
  C*der(v) = i;
end Capacitor;
```




```
v = p.v - n.v;
0 = p.i+n.i;
i = p.i;

C*der(v) = i;
```

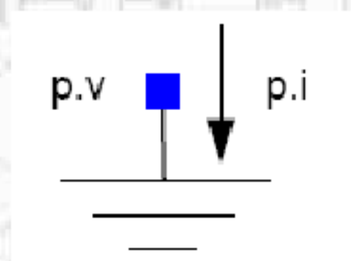
抽象类的复用（电感和接地）

```
class Inductor "Ideal electrical Inductor"
  extends TwoPin;
  parameter Real L(unit="H") "Inductance";
equation
  v = L*der(i);
end Inductor;
```




```
v = p.v - n.v;
0 = p.i+n.i;
i = p.i;

v = L*der(i);
```

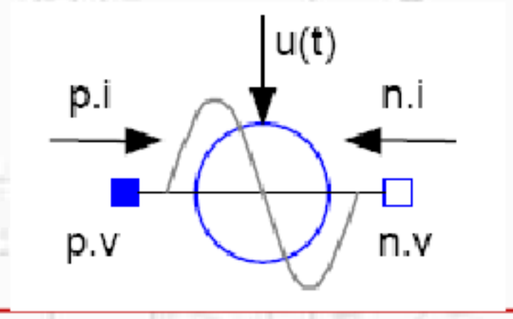
```
class Ground "Ground"
  Pin  p;
equation
  p.v = 0;
end Ground;
```




```
p.v = 0;
```

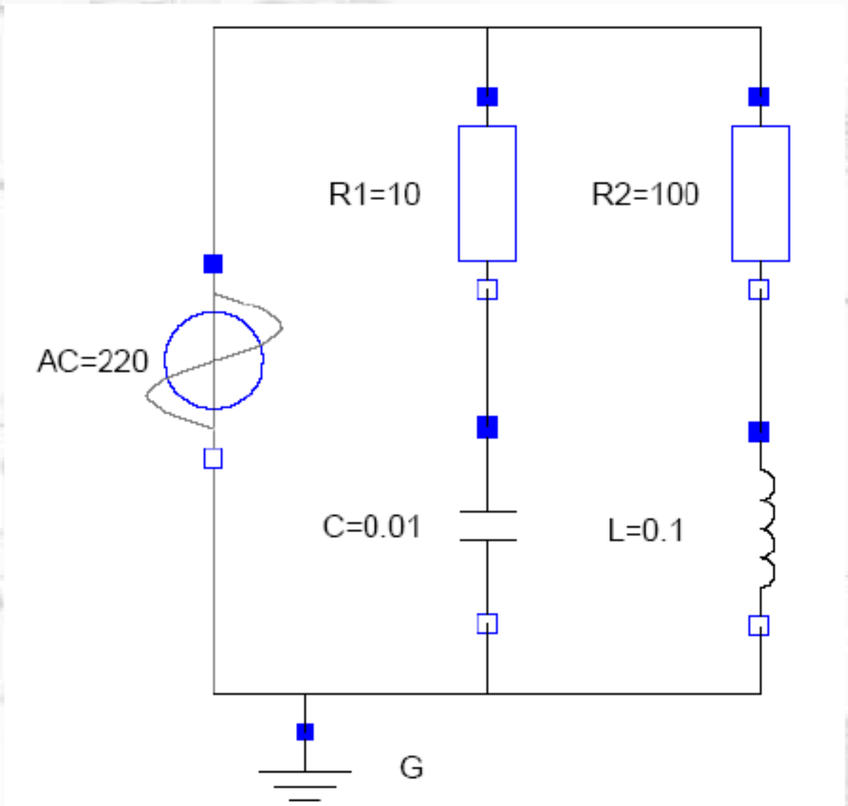
抽象类的复用（电源）

```
class VsourceAC "Sin-ware voltage source"
  extends TwoPin;
  parameter Voltage VA = 220 "幅值";
  parameter Real f(Unit= “Hz” ) = 50 "频率";
  constant Real PI = 3.141592653589793;
equation
  v = VA*sin(2*PI*f*time);
end VsourceAC;
```



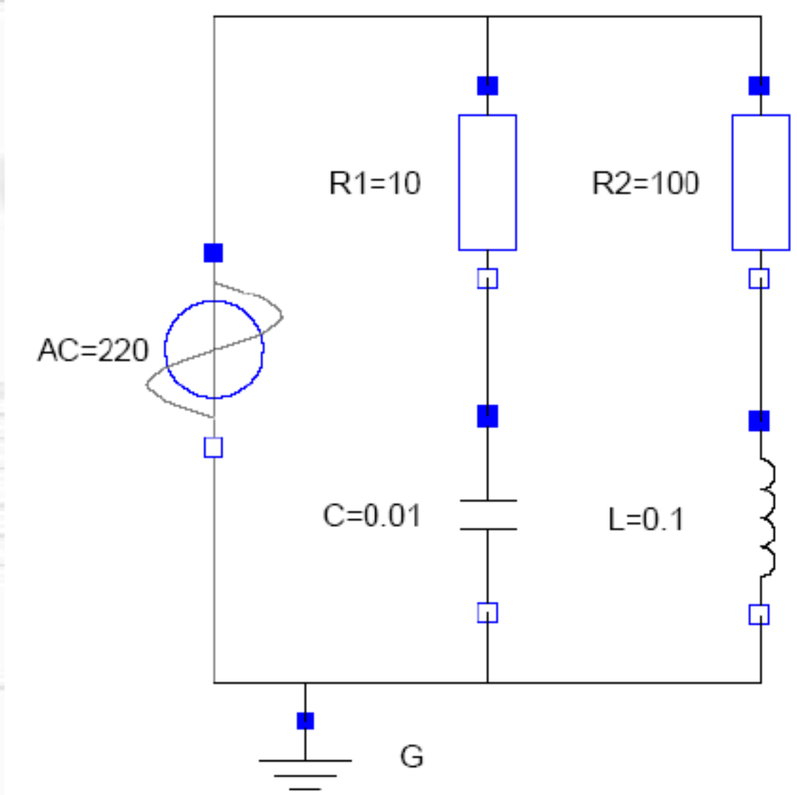

```
v = p.v - n.v;
0 = p.i+n.i;
i = p.i;

v = VA*
    sin(2*PI*f*
          time);
```

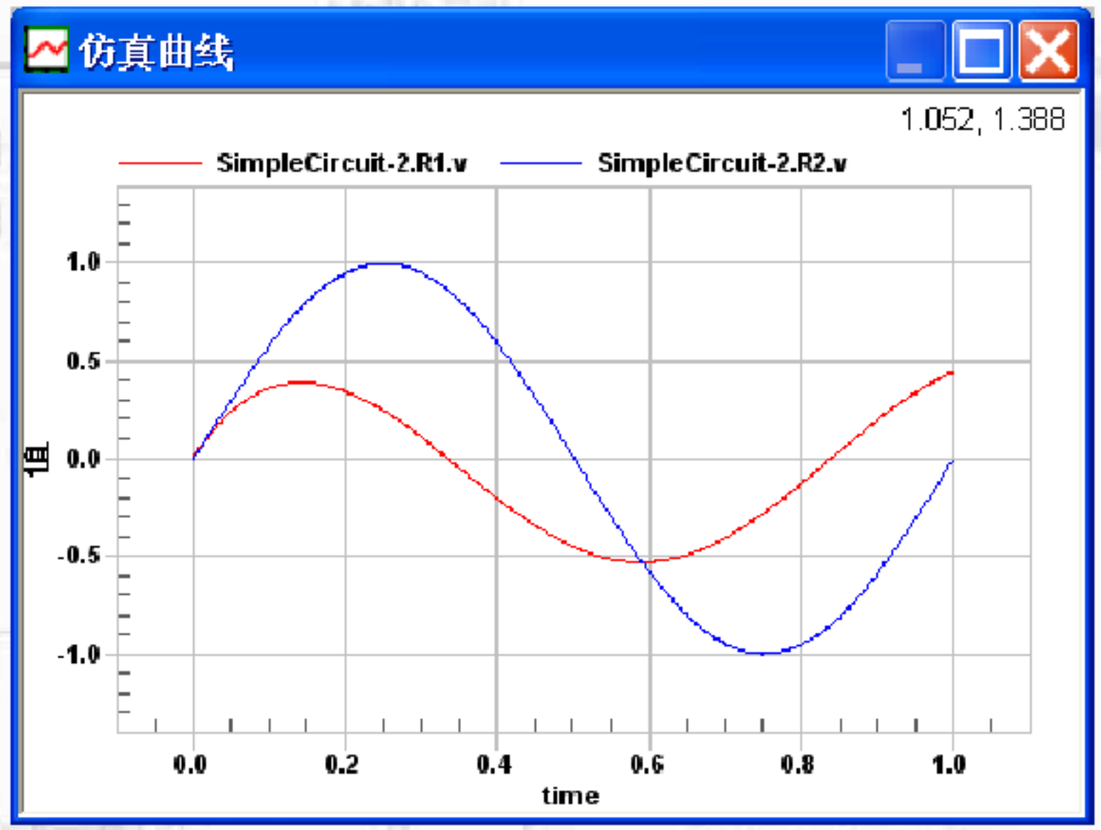
例子：SimpleCircuit




```
model SimpleCircuit
  Resistor R1(R=10), R2(R=100);
  Capacitor C(C=0.01);
  Inductor L(L=0.1);
  VsourceAC AC;
  Ground G;
equation
  connect(AC.p, R1.p);
  connect(R1.n, C.p);
  connect(C.n, AC.n);
  connect(R1.p, R2.p);
  connect(R2.n, L.p);
  connect(L.n, C.n);
 connect(AC.n, G.p);
end SimpleCircuit;
```

简单电路模型仿真


R1=10
R2=100
AC=220
C=0.01
L=0.1
G


仿真曲线
1.052, 1.388
SimpleCircuit-2.R1.v
SimpleCircuit-2.R2.v
1.0
0.5
0.0
-0.5
-1.0
值
0.0
0.2
0.4
0.6
0.8
1.0
time

数组（Array）

- Modelica数组源于Matlab，但比Matlab更灵活。
  - 数组声明
  - 数组下标（Array Index）
  - 数组四则运算
  - 数组函数
  - 数组连接（Array Concatenation）
  - 数组切分（Array Slice）

数组声明

- Java声明方式:
  - Real[3] position_vector = {1,2,3};
  - Real[3,3] IdentityMat = {{1,0,0},{0,1,0},{0,0,1}};
  - Real[3,3,3] arr3d;
- C声明方式:
  - Real position_vector[:] = {1,2,3};
  - Real position_vector[3] = {1,2,3};
  - Real IdentityMat[3,3] = {{1,0,0},{0,1,0},{0,0,1}};
  - Real arr3d[3,3,3];
- 混合:
  - Real[3] IdentityMat[3] = {{1,0,0},{0,1,0},{0,0,1}};
  - Real[3,2] arr4d[4,5]; //等价于Real[4,5,3,2]

数组下标（Array Index）

- A[i, j, k, ...]
  - 下限: 1-based
  - 上限: size(A, k)
- 子矩阵: A(i1: i2, j1: j2)
- 数组方程:
  - 声明: Real position_vector[3] = {1, 2, 3};
  - E1: position_vector[1] = 1;
  - E2: position_vector[2] = 2;
  - E3: position_vector[3] = 3;

数组四则运算

- 适用于Real和Integer数组，针对标量、向量、矩阵和多维数组。
  - 声明：Real position_vector[3] = {1,2,3};
  - postion_vector*2 = {2,4,6}
  - 数组+/-数组：要求维度相同；
  - 数组*/标量：逐个元素相*/
  - 向量*向量：点积（结果为标量）
  - 矩阵*向量/矩阵*矩阵：矩阵乘法（限制维度）
  - 数组求幂：要求满足相乘条件（因此只针对方阵）

数组函数

- 内置函数:
  - transpose(A): 转置（第1维和第2维）
  - zeros(n1,n2,n3): 全部置0
  - ones(n1,n2,n3): 全部置1
  - fill(s,n1,n2,n3): 全部置s
  - identity(n): n维单位矩阵
  - min(A): 返回最小值
  - max(A): 返回最大值
  - sum(A): 返回所有元素的和
- 标量函数作用于数组:
  - cos({1,2,3})={cos(1),cos(2),cos(3)}

数组连接（Array Concatenation）

- 连接运算符:
  - cat(k, A, B, C, ...)
  - cat(1, {2, 3}, {5, 8, 4}) = {2, 3, 5, 8, 4}
- 分号运算符:
  - [A; B; C]
- 逗号运算符:
  - [A, B, C]

```
[expr11, expr12, ... expr1n;
 expr21, expr22, ... expr2n;
 ...
 exprm1, exprm2, ... exprmn]
```

```
[1, 2;
 3, 4]
```

```
[{{1}}, {{2}};
 {{3}}, {{4}}]
```

```
[{{1, 2}};
 {{3, 4}}]
```

```
{{1, 2}},
 {{3, 4}}
```

```
[1]
```

```
{{1}}
```

数组切分（Array Slice）

```
[a11, a12, a13;
 a21, a22, a23;
 a31, a32, a33]
```

```
a[1,1:3] = [a11, a12, a13]
```

```
a[1:3,1] = [a11;a21; a31]
```

```
a[2:3,2:3] = [a22,a23; a32,a33]
```

算法构造

- 算法用于描述过程式的物理行为。
  - 算法（Algorithm）
  - 语句（Statement）
  - 函数（Function）
  - 函数调用
  - MIMO函数
  - 外部函数（External Function）
  - 算法vs函数（Algorithm Viewed as Function）
  - 运算符重载（Operator Overloading）

算法（Algorithm）

- 算法用于描述过程式的物理行为。
- 算法由关键字algorithm标识，包含一系列赋值语句。

```
...
algorithm
...
<some other keyword>
...
```

- 算法的语义类似函数:
  - 区分输入变量和输出变量。

```
equation
  x = y*2;
  z = w;
algorithm
  x1 := z+x;
  x2 := y - 5;
  x3 := x2+y;
equation
  u = x1+x2;
  ...
```

语句（Statements）

- 分支语句:
  - if-then-else
- 循环语句:
  - for-loop
  - while-loop
- When语句
  - 只在条件从false变成true的瞬时计算一次

分支语句

```
if <condition> then
   <statements>
else if <condition> then
   <statements>
else
   <statementss>
end if;
```

```
sum := 0;
n := size(a,1);
while n > 0 then
  if a[n] > 0 then
    sum := sum+a[n];
  else if a[n] > -1 then
    sum := sum-a[n]-1;
  else
    sum := sum-a[n];
  end if;
  n := n-1;
end while;
```

For-loop语法形式

```
for <iteration-variable> in <iteration-set-expression> loop
  <statement1>
  <statement2>
  ...
end for;
```

```
for k in 1:10+2 loop //k取值: 1,2,3,...,12
for i in {1,3,6,7} loop //i取值: 1,3,6,7
for r in 1.0 :1.5 :5.5 loop //r取值: 1.0, 2.5, 4.0, 5.5
```

例子：计算n次多项式

```
y := a[1] + a[2]*x + a[3]*x^2 + a[4]*x^3 + ... + a[n+1]*x^n
```

```
algorithm
  y := 0;
  xpower := 1;
  for i in 1:n+1 loop
    y := y+a[i]*xpower;
    xpower := xpower*x;
  end for;
  ...
```

When语句

- When-语句只在条件从false变成true的瞬时计算一次。

```
when <conditions> then
  <statements>
elsewhen <conditions> then
  <statements>
end when;
```

```
when x > 2 then
  y1 := sin(x);
  y3 := 2*x + y1 + y2;
end when;
```

```
when {sample(0,2), x < 5} then
  y1 := sin(x);
  y3 := 2*x + y1 + y2;
end when;
```

```
when sample(0,2) or x < 5 then
  y1 := sin(x);
  y3 := 2*x + y1 + y2;
end when;
```

函数（Functions）

- 函数在数学建模时应用普遍
  - 内置函数：abs、abs、sqrt、mod、div、…
  - 四则运算：+ - * /
- 自定义函数：
  - 只包含一个algorithm
  - 形参用input关键字声明
  - 函数输出用output关键字声明
  - 函数形参不能在函数内部赋值

例子： polynomialEvaluator

```
function polynomialEvaluator
  input Real a[;];
  input Real x := 1.0;
  output Real y;
protected
  Real xpower;
algorithm
  y := 0;
  xpower := 1;
  for i in 1:size(a,1) loop
    y := y+a[i]*xpower;
    xpower := xpower*x;
  end for;
end polynomialEvaluator;
```

函数调用

- 函数调用方式:
  - 按位置传参（参数类型、参数数目等与实参一致）

```
y := polynomialEvaluator({1,2,3,4},21};
```

  - 按名字传参（此时参数顺序不重要）

```
y := polynomialEvaluator(a={1,2,3,4},x=21};
y := polynomialEvaluator(x=21,a={1,2,3,4}};
```

  - 输入参数有默认值时可省略

```
y := polynomialEvaluator(a={1,2,3,4}};
```

MIMO函数

```
function f
  input Real x;
  input Real y;
  output Real r1;
  output Real r2;
  output Real r3;
  ...
end f;
```

```
algorithm
  (a, b, c) := f(1.0, 2.0);
  ...
```

```
equation
  (a, b, c) = f(1.0, 2.0);
  ...
```

外部函数（External Functions）

- Modelica允许声明外部C 或Fortran 函数

```
function log
  input Real x;
  output Real y;
  external;
end log;
```

- 外部函数选项:
  - 输出参数、局部数组，外部函数形参顺序，显式声明数组内存布局（row-major/column-major）、…

外部C函数

```
function polynomialMultiply
  input Real a[:],b[:];
  output Real c[:] = zeros(size(a,1) + size(b,1) -1);
external "C"
end polynomialMultiply;
```

C函数声明:

```
extern void polynomialMultiply(double const *, int, double const *,
  int, double *, int);
```

例子： leastSquares

```
function LeastSquares "Solves a linear least squares problem"
  input Real A[:,:];
  input Real B[:,:];
  output Real Ares[size(A,1),size(A,2)] := A;
  output Real x[size(A,2),size(B,2)];
protected
  Integer lwork = min(size(A,1),size(A,2))+
    max(max(size(A,1),size(A,2)),size(B,2))*32;
  Real work[lwork];
  Integer info;
  String transposed="NNNN";
external "FORTRAN 77" dgels(transposed, 100, size(A,1), size(A,2),
  size(B,2), Ares, size(A,1), B, size(B,1), work, lwork, info);
end LeastSquares;
```

算法vs函数

```
algorithm
  y := x;
  z := 2*y;
  y := z+y;
  ...
```

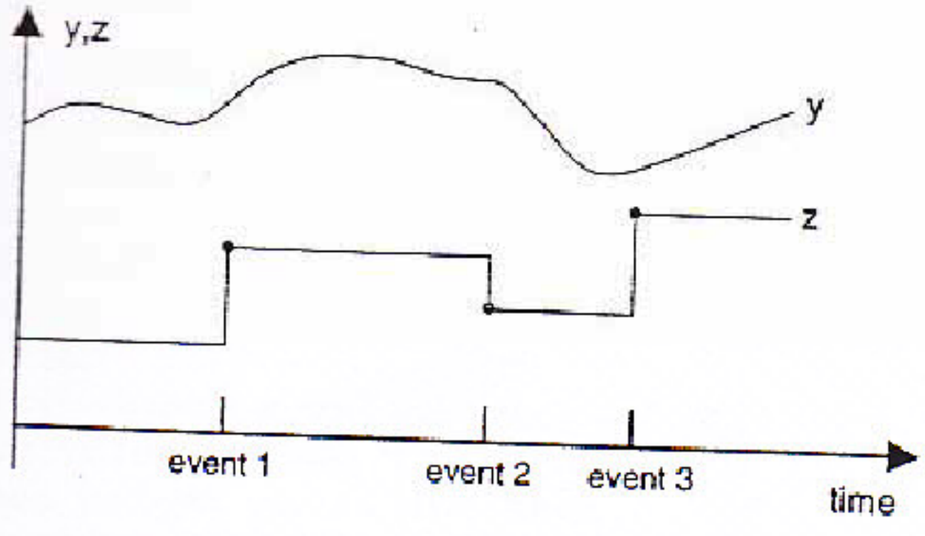
改写成函数调用:

```
(y, z) = f(x);
...
```

```
function f
  input Real x;
  output Real y, z;
algorithm
  y := x;
  z := 2*y;
  y := z+y;
  ...
end f;
```

离散事件与混合建模

- 离散行为场景:
  - 变量值可能只在某个时间点发生变化（电源开/关）
  - 变量的变化不是连续的（离合器分离/结合）
  - 尽管不是瞬态变化，但变化时间区间小（阀门开/关）
- 离散事件:
  - 离散变量在特定时间点数值发生变化
  - 离散事件的存在使得DAE系统求解更加困难



离散变量（Discrete Variables）

- 离散变量:
  - 带有discrete前缀的Real变量
  - Integer、Boolean、enumeration(...)
  - When-结构中的连续变量
- 离散建模方法:
  - If-结构
  - When-结构

If-结构

- 单行If结构:

```
if expr then expr else expr
```

- 多分支If结构:

```
if <condition> then
    <statements>
elseif <condition> then
    <statesments>
else
    <statesments>
end if
```

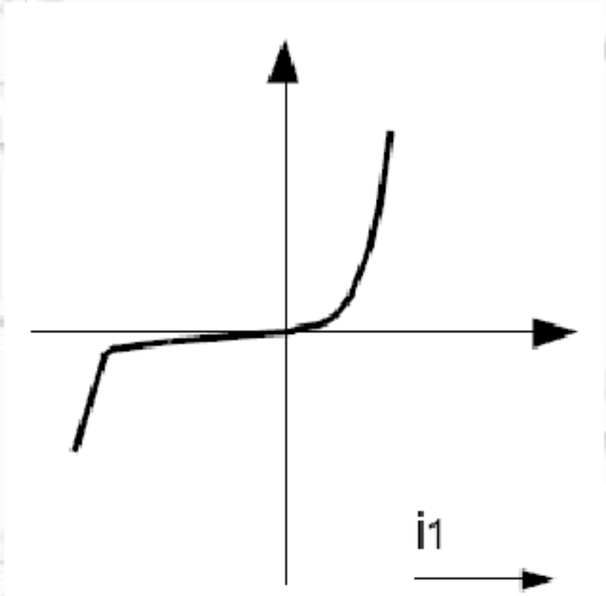
条件方程（If-Equations）

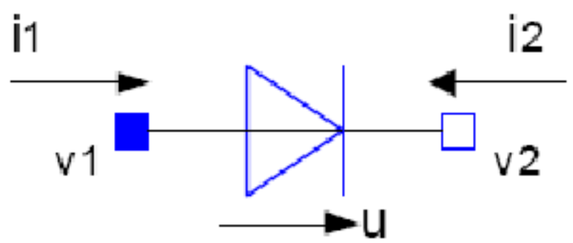
- 要求if-, elseif-, else-分支方程数量相同。

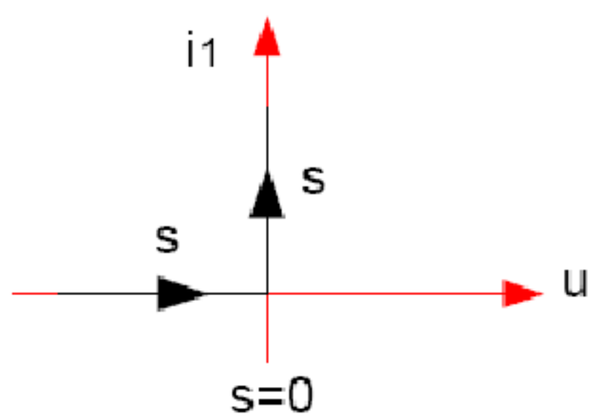
```
if <condition> then
  <equations>
elseif <condition> then
  <equations>
else
  <equations>
end if;
```

```
model MoonLanding
  ...
equation
  if (time<thrustDecreaseTime) then
    apollo.thrust = force1;
  elseif (time<thrustEndTime) then
    apollo.thrust = force2;
  else
    apollo.thrust = 0;
  end if;
  ...
end MoonLanding;
```

例子：Real/Ideal Diode








```
model Diode "Ideal diode"
  extends TwoPin;
  Real s;
  Boolean off;
equation
  off = s < 0;
  if off then v=s else v=0;
    //条件方程
  i = if off then 0 else s;
    //条件表达式
end Diode;
```

When-方程

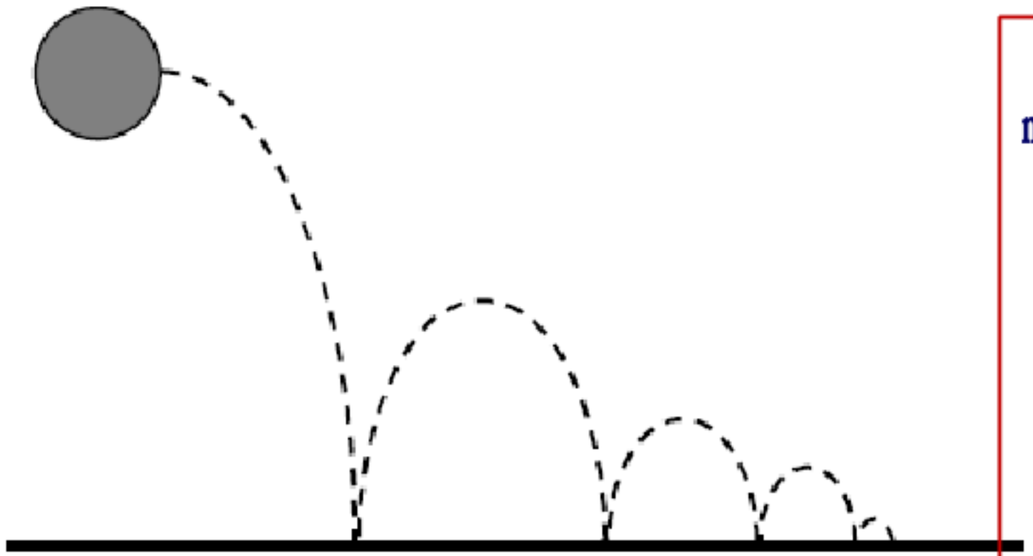
- When-方程语法:

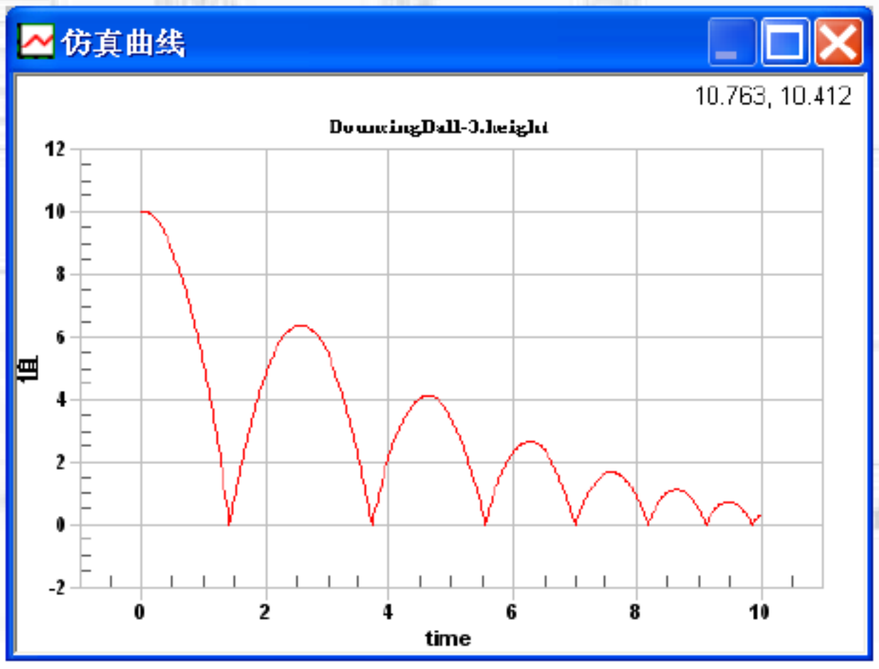
```
when {condition1, condition2, ...} then
    <equations>
end when;
```

- When-方程只在条件从false变成true的瞬时生效一次。
  - 这一点与条件表达式和条件方程不同。

```
when x > 2 then
  y1 = sin(x);
  y3 = 2*x + y1+y2;
end when;
```

例子：BouncingBall



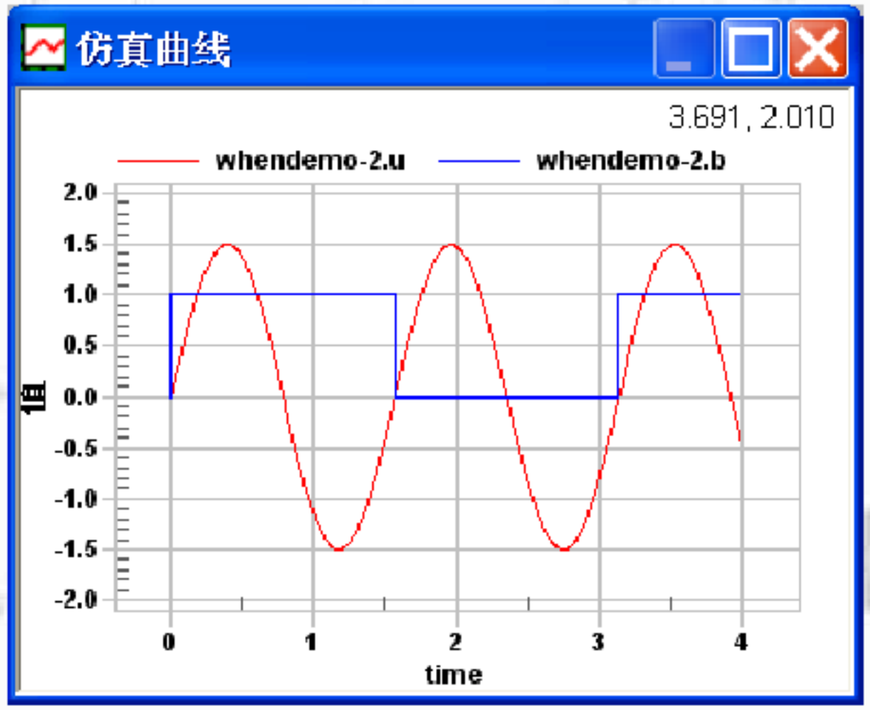



```
model BouncingBall
 constant Real g=9.81;
 parameter Real c=0.8;
 Real height(start=10);
 Real velocity(start=0);
 annotation
  (experiment(StopTime=10));
equation
 der(height) = velocity;
 der(velocity) = -g;
 when height <= 0 then
  reinit(velocity, -c*(velocity));
 end when;
end BouncingBall;
```

If与When的区别

- If-方程在条件为true的整个区间内都是有效的，变量在整个区间内连续求值。
- When-方程仅在条件从false变成true的瞬时生效一次，变量只在事件发生时被求值，并在整个区间保持不变。

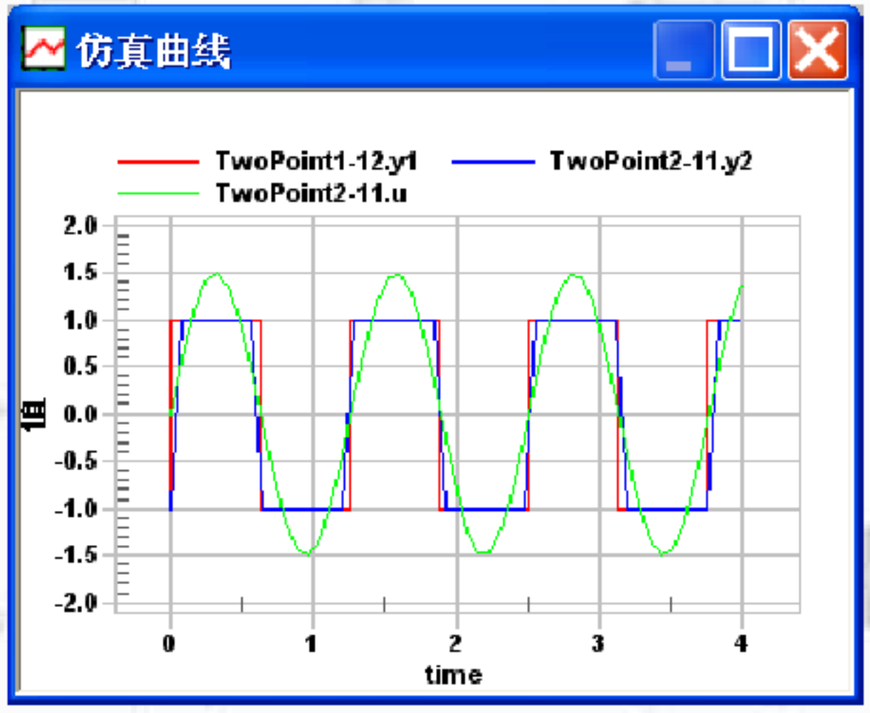
```
model WhenDemo
  parameter Real A=1.5,w=4;
  Real u;
  Boolean b;
equation
  u = A*Modelica.Math.sin(w*time);
  when u > 0 then
    b = not pre(b);
  end when;
end WhenDemo;
```



例子：事件与noEvent

- 一般情况下，If-结构和When-结构中的方程和算法语句会在条件满足时触发事件。
- noEvent操作符可使事件失效，即不触发事件。

```
model TwoPoint
  parameter Real w=5, A=1.5;
  Real u,y1,y2;
equation
  u = A*Modelica.Math.sin(w*time);
  y1 = if u > 0 then 1 else -1;
  y2 = if noEvent(u > 0) then 1
    else -1;
end TwoPoint;
```

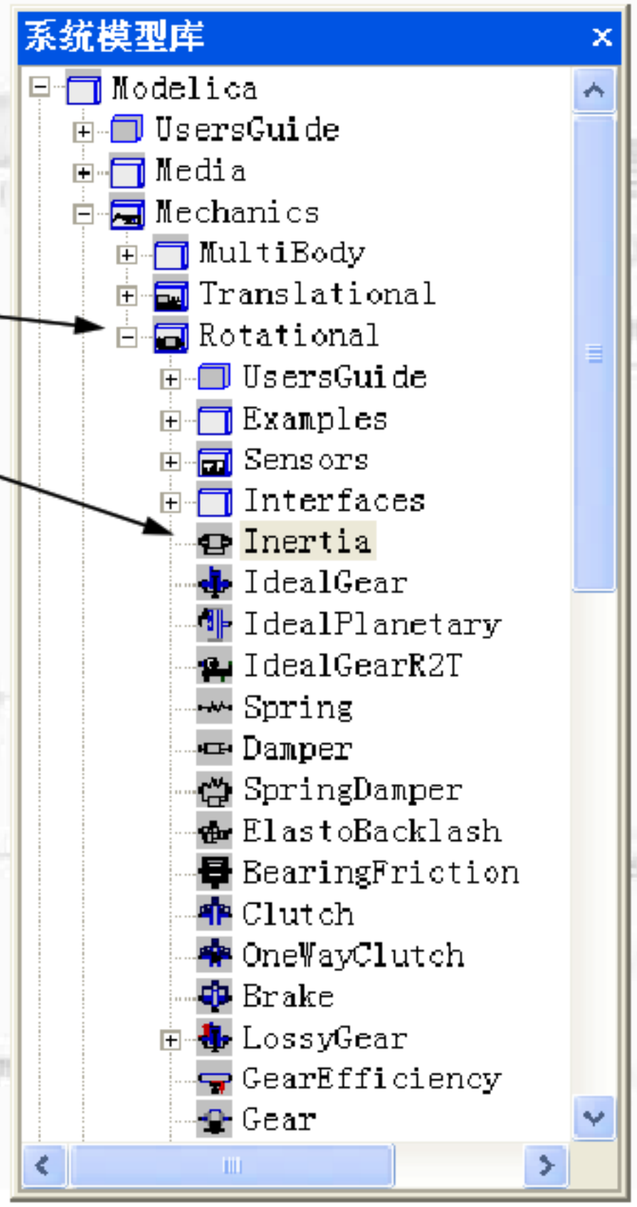


包（Packages）

- 包是一种受限类。
- 用途:
  - 有效地组织模型层次，便于查找。
  - 消除名字冲突。
- 包的物理组织形式:
  - 局部嵌套：模型以局部类的形式储存于包中。
  - 结构化实体：通过文件目录结构储存模型。

局部嵌套

```
文件: Rotational.mo
package Modelica
  package Mechanics
    package Rotational
      model Inertia
        ...
      end Inertia;
      model Torque
        ...
      end Torque;
        ...
      end Rotational;
    end Mechanics;
    ...
end Modelica;
```



结构化实体

```
...\Library\Modelica
    \Blocks
    \Electrical
    \Mechanics
        \package.mo
        \Rotational.mo
        \Translational.mo
```

文件: ...\Mechanics\Rotational.mo

```
package Rotational
  model Inertia
    ...
  end Inertia;
  model Torque
    ...
  end Torque;
  ...
end Rotational;
```

```
package Mechanics
  ...
end Mechanics;
```

文件: ...\Mechanics\package.mo

包的导入

- 限定性导入

```
import Modelica.Siunits.Voltage;
```

- 非限定性导入

```
import Modelica.Siunits.*;
```

- 通过短名引入

```
import SI = Modelica.Siunits;
```

- 局部包（Local Packages）

例子：Import

```
package MyPack
 import Modelica.SIunits.*;
 class foo;
  Voltage  v;
  Modelica.SIunits.Ampere  i;
 end foo;
end MyPack;
```

```
encapsulated package MyPack
 import SI = Modelica.SIunits;
 import Modelica;
 class foo;
  SI.Voltage  v;
  Modelica.SIunits.Ampere  i;
 end foo;
end MyPack;
```

注解（Annotations）

- 注解包含模型中必要的辅助信息，增强了代码可读性，提高模型组件的易用性。
- Modelica规范只声明了注解的语法格式，注解的含义由Modelica平台进行解释。
- 多数情况下，去掉注解文本不影响模型仿真结果。

注解的使用场景

- 图示化建模
- 提供选项
- 外部函数声明

```
model Resistor
 annotation(Icon(
  Rectangle(extent=[- 70, 30; 70, - 30]),
  Line(points=[- 90, 0; - 70, 0]),
  ...
 ));
 ...
end Resistor;
```

```
annotation(Choices(
  choice=1 "P",choice=2 "PI",choice=2 "PID"));
```

```
annotation(arrayLayout = "columnMajor");
```

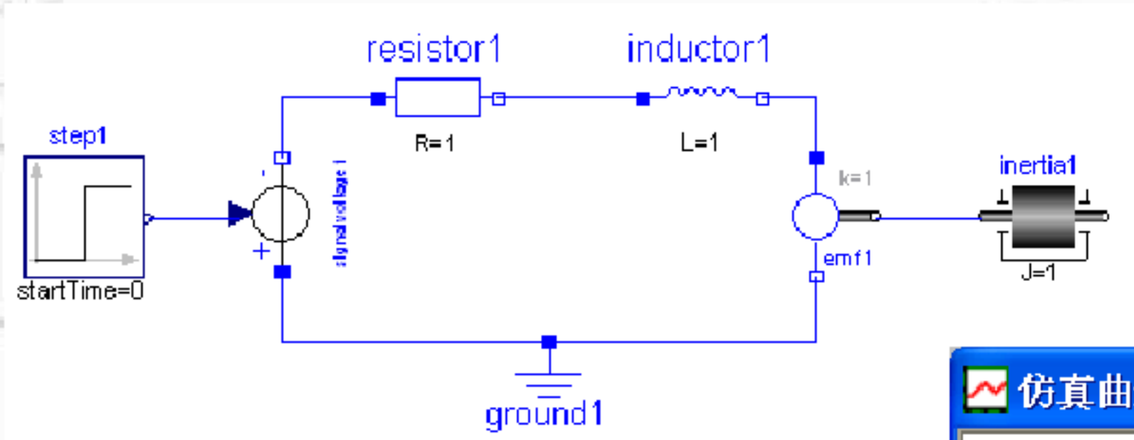
命名惯例（Naming Conventions）

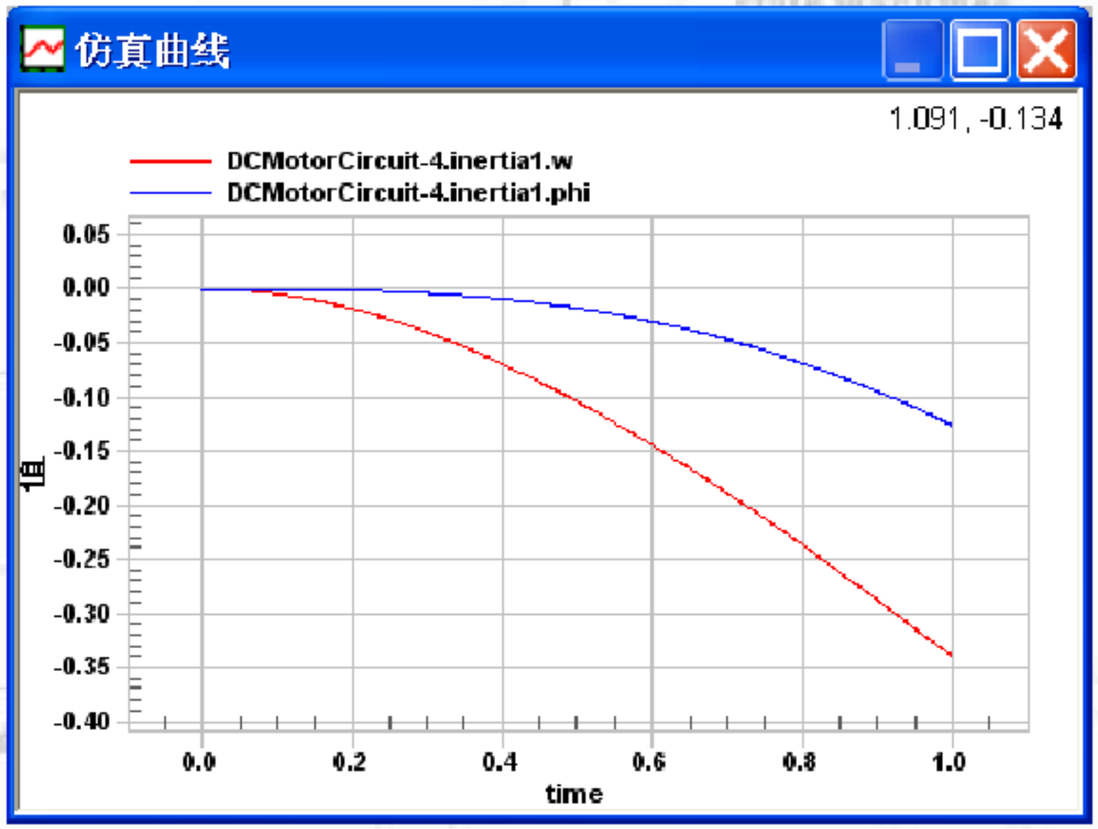
- 建议构建Modelica模型时遵循一定的命名惯例
  - 减少命名冲突
  - 改善可读性
- 命名惯例:
  - 类型和类名首字母大写，如: Voltage。
  - 变量名首字母小写，如: body; 只有一个字母时除外，如: T 表示温度Temperature。
  - 多个单词连用时每个词的首字母大写，如: ElectricalCurrent、bodyPart。
  - 下划线用于分隔不同分类的名字，如: body_low_up。
  - 成对使用的连接器与MSL一致，如: 电气p、n; 机械flange_a、flange_b; …。

Modelica标准库（MSL）

库名	内容
Modelica.Constants	物理和数学常量
Modelica.Icons	图标
Modelica.Math	数学函数
Modelica.SIunits	SI单位制
Modelica.Electrical	电气
Modelica.Block	控制
Modelica.Mechanical.Translational	机械（一维平移）
Modelica.Mechanical.Rotational	机械（一维旋转）
Modelica.Mechanical.MultiBody	机械（多体）
Modelica.Thermal	热学

例子：DCMotorCircuit





Modelica模型求解流程

Modelica源文件 → 编译 → 分析 → 优化 → C代码生成 → 仿真 → 后处理

- 编译将复杂物理的模型平坦化，铲平模型的层次结构，将模型转化为一组平坦的方程、常量、参数和变量。
- 分析优化阶段检查方程是否是相容，若不相容则找出问题根源，以便用户校正模型；若相容则进行简化减少方程个数，为仿真求解做准备。
- 仿真求解时，根据方程的参数依赖关系，结合数值求解包提供的函数，形成模型的求解流程和控制策略，并生成C代码求解器，通过编译运行C程序实现模型求解。

主要内容回顾

- 认识Modelica
- 面向对象的数学建模
  - 类及其实例
  - 继承
  - 泛型类
- 方程
- 非因果物理建模
- 基于组件建模
- 抽象类
- 数组
- 算法
- 函数
- 离散事件与混合建模
- 包
- 注解
- 命名惯例
- Modelica标准库(MSL)
- Modelica模型求解流程

Modelica特点回顾

- 基于方程的非因果建模
  - 相对于赋值语句，方程体现了不同的数据流向，能够更好地支持组件复用。
- 多领域建模
  - 能够描述电气、机械、热力学、液压、生物、控制等多领域模型或组件，已有大量可复用的领域库。
- 面向对象建模
  - 具有面向对象语言特征：类、泛型（C++模板）、子类型，允许组件复用和模型进化。
  - 提供良好的软组件模型，通过组件（接口）相互连接，快速搭建复杂物理系统。
- 连续-离散混合建模