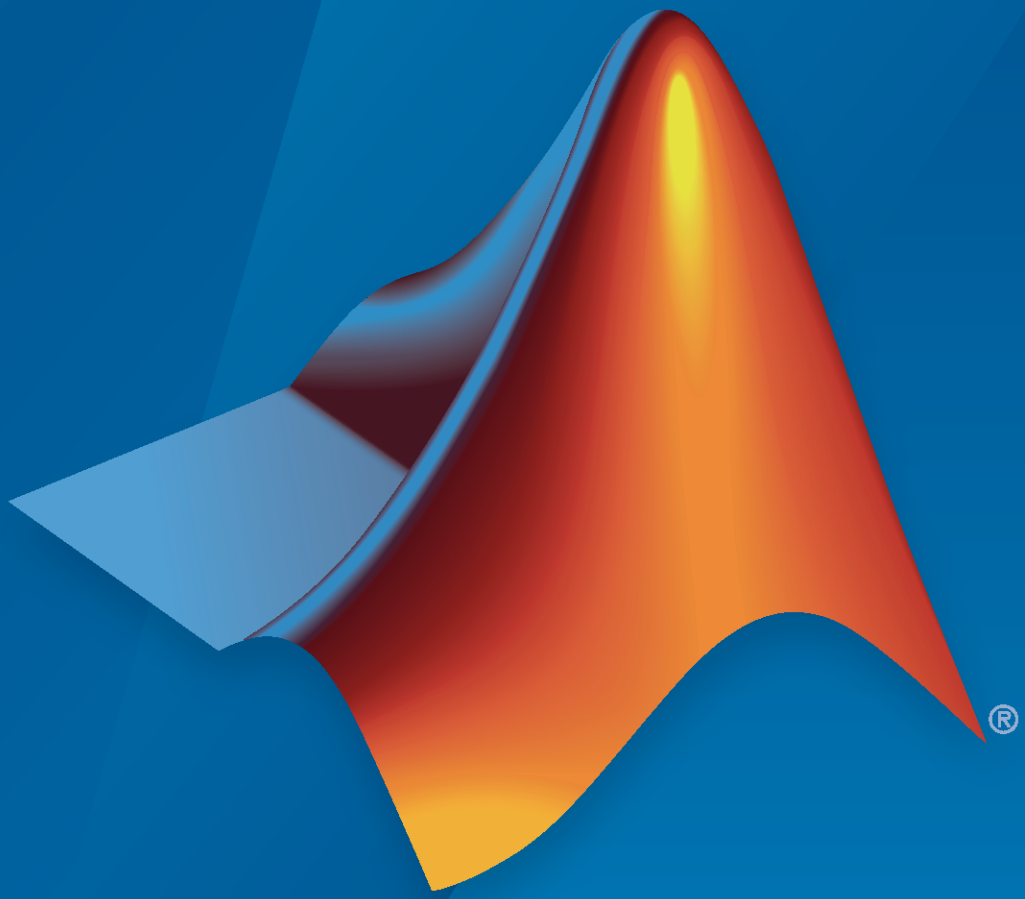
MATLAB®

面向对象的编程



MATLAB®

R2022b

MathWorks®

如何联系 MathWorks

最新动态: **www.mathworks.com**

销售和服务: **www.mathworks.com/sales_and_services**

用户社区: **www.mathworks.com/matlabcentral**

技术支持: **www.mathworks.com/support/contact_us**

电话: 010-59827000

迈斯沃克软件 (北京) 有限公司
北京市朝阳区望京东园四区 6 号楼
北望金辉大厦 16 层 1604

面向对象的编程

修订历史记录

2008 年 3 月	仅限在线版本	MATLAB 7.6（版本 2008a）中的新增内容
2008 年 10 月	仅限在线版本	MATLAB 7.7（版本 2008b）中的修订内容
2009 年 3 月	仅限在线版本	MATLAB 7.8（版本 2009a）中的修订内容
2009 年 9 月	仅限在线版本	MATLAB 7.9（版本 2009b）中的修订内容
2010 年 3 月	仅限在线版本	MATLAB 7.10（版本 2010a）中的修订内容
2010 年 9 月	仅限在线版本	MATLAB 7.11（版本 2010b）中的修订内容
2011 年 4 月	仅限在线版本	MATLAB 7.12（版本 2011a）中的修订内容
2011 年 9 月	仅限在线版本	MATLAB 7.13（版本 2011b）中的修订内容
2012 年 3 月	仅限在线版本	MATLAB 7.14（版本 2012a）中的修订内容
2012 年 9 月	仅限在线版本	MATLAB 8.0（版本 2012b）中的修订内容
2013 年 3 月	仅限在线版本	MATLAB 8.1（版本 2013a）中的修订内容
2013 年 9 月	仅限在线版本	MATLAB 8.2（版本 2013b）中的修订内容
2014 年 3 月	仅限在线版本	MATLAB 8.3（版本 2014a）中的修订内容
2014 年 10 月	仅限在线版本	MATLAB 8.4（版本 2014b）中的修订内容
2015 年 3 月	仅限在线版本	MATLAB 8.5（版本 2015a）中的修订内容
2015 年 9 月	仅限在线版本	MATLAB 8.6（版本 2015b）中的修订内容
2016 年 3 月	仅限在线版本	MATLAB 9.0（版本 2016a）中的修订内容
2016 年 9 月	仅限在线版本	MATLAB 9.1（版本 2016b）中的修订内容
2017 年 3 月	仅限在线版本	MATLAB 9.2（版本 2017a）中的修订内容
2017 年 9 月	仅限在线版本	MATLAB 9.3（版本 2017b）中的修订内容
2018 年 3 月	仅限在线版本	MATLAB 9.4（版本 2018a）中的修订内容
2018 年 9 月	仅限在线版本	MATLAB 9.5（版本 2018b）中的修订内容
2019 年 3 月	仅限在线版本	MATLAB 9.6（版本 2019a）中的修订内容
2019 年 9 月	仅限在线版本	MATLAB 9.7（版本 2019b）中的修订内容
2020 年 3 月	仅限在线版本	MATLAB 9.8（版本 2020a）中的修订内容
2020 年 9 月	仅限在线版本	MATLAB 9.9（版本 2020b）中的修订内容
2021 年 3 月	仅限在线版本	MATLAB 9.10（版本 2021a）中的修订内容
2021 年 9 月	仅限在线版本	MATLAB 9.11（版本 2021b）中的修订内容
2022 年 3 月	仅限在线版本	MATLAB 9.12（版本 2022a）中的修订内容
2022 年 9 月	仅限在线版本	MATLAB 9.13（版本 2022b）中的修订内容

目录

1 在 MATLAB 中使用面向对象的设计

2 基本示例

3 MATLAB 类概述

6 定义和组织类

7 值类或句柄类 - 如何选择

8 属性 - 存储类数据

9 方法 - 定义类操作

11 事件 - 发送和响应消息

12 如何构建类层次结构

15 常量属性

16 来自类元数据的信息

17 特化对象行为

在 MATLAB 中使用面向对象的设计

为什么使用面向对象的设计

本节内容

MATLAB 程序的编程方式

创建软件应用程序通常包括设计应用程序数据和实施对这些数据执行的操作。过程式程序将数据传递给函数，函数对数据执行必要的操作。面向对象的软件将数据和操作封装在各个对象中，这些对象通过各自的接口进行交互。

您可以通过 MATLAB 语言使用过程式编程方式和面向对象的编程方式来创建程序，并在程序中结合使用对象和普通函数。

过程式程序设计

在过程式编程中，您需要设计一系列步骤，通过执行这些步骤来实现所需状态。通常，您会将数据表示为结构体的单个变量或字段。您可以将操作实现为以变量为参数的函数。程序通常调用一系列函数，每个函数都接受传递的数据，然后返回修改后的数据。每个函数都会对数据执行一项或多项操作。

面向对象的程序设计

面向对象的程序设计包括：

- 确定要构建的系统或应用程序的组件
- 分析和识别模式，以确定哪些组件会重用或共享特征
- 基于相似性和差异性对组件进行分类

执行此分析后，您可以定义类来说明应用程序所使用的对象。

类和对象

类描述一组具有共同特征的对象。对象是类的特定实例。对象属性中包含的值使对象不同于同一类的其他对象。由类定义的函数（称为方法）实现类中所有对象通用的对象行为。

何时创建面向对象的程序

简单的编程任务可以用简单的函数来实现。然而，随着任务的规模和复杂度的增加，函数会变得更加复杂和难以管理。

当函数变得太大时，您可以将它们分成较小的函数，并将数据从一个函数传递到另一个函数。然而，随着函数数量的增加，设计和管理传递给函数的数据也变得困难和容易出错。在这种情况下，可以考虑将您的 MATLAB 编程任务转向面向对象的设计。

从对象的角度来理解问题

对某些问题来说，从对象的角度来考虑会更简单、更自然。将问题陈述中的名词视为要定义的对象，动词视为要执行的操作。

例如，我们可以设计类来表示货币借贷机构，如银行、抵押贷款公司、个人货币贷方等。上述各种类型的贷方很难用过程表示。但是，您可以将每类贷方表示为执行特定操作并包含特定数据的对象。设计对象的过程包括识别对您的应用程序很重要的贷方特征。

确定共同点

所有贷方有哪些共同点？例如，所有 **MoneyLender** 对象都可能具有 **loan** 方法和 **InterestRate** 属性。

确定差异

每个贷方有什么不同？一个贷方可以向企业提供贷款，而另一个贷方只向个人提供贷款。因此，对于不同类型的贷款机构，**loan** 操作可能需要有所不同。**MoneyLender** 基类的子类可特化 **loan** 方法的子类版本。每个贷方可以为其 **InterestRate** 属性设置不同值。

将共性析出为一个超类，并实现子类中每种类型的贷方的特性。

只添加必要的内容

这些机构从事的某些活动可能与您的应用程序无关。在设计阶段，请根据您的问题定义来确定对象必须包含的操作和数据。

对象管理内部状态

对象可提供结构体和元胞数组中无法提供的一些有用功能。例如，对象可以：

- 约束赋给任何给定属性的数据值
- 仅在查询属性时计算属性值
- 查询或更改任何属性值时广播通知
- 限制对属性和方法的访问

减少冗余

随着程序复杂度的增加，面向对象的设计的好处变得更加明显。例如，假设您将以下过程作为应用程序的一部分来实现：

1. 检查输入
2. 对第一个输入参数执行计算
3. 基于第二个输入参数转换步骤 2 的结果
4. 检查输出和返回值的有效性

您可以将此过程作为普通函数来实现。但是，假设您要在应用程序中的某个地方再次使用此过程，则除步骤 2 之外，您必须执行不同的计算。您可以复制并粘贴第一个实现，然后重写步骤 2。您也可以创建一个根据选项执行相应计算的函数，然后依此类推。然而，这些选项会导致代码更复杂。

面向对象的设计可以将公共代码析出为一个基类。该基类将定义所使用的算法，并实现在所有使用该代码的情况下都通用的内容。步骤 2 可以通过语法进行定义，但不进行实现，而将特化实现留给从这个基类派生的类。

步骤 1：

```
function checkInputs()
  % actual implementation
end
```

步骤 2：

```
function results = computeOnFirstArg()
  % specify syntax only
end
```

步骤 3：

```
function transformResults()
  % actual implementation
end
```

步骤 4：

```
function out = checkOutputs()
  % actual implementation
end
```

不必复制或修改基类中的代码。从基类派生的类会继承这些代码。继承能减少要测试的代码量，并将程序与对基本过程的更改隔离开来。

定义一致的接口

在面向对象的编程中，一种有用的编程方式是基于某个类来创建类似但更特化的类。此类定义公用接口。在您的程序设计中使用这种类可以：

- 确定特定目标的需求
- 将需求作为接口类编写到您的程序中

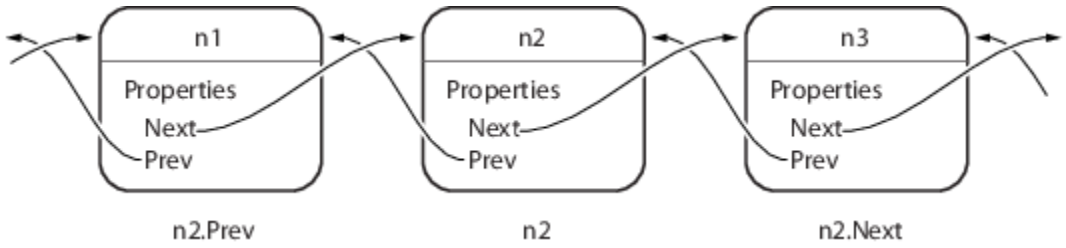
降低复杂度

对象减少了您使用组件或系统时必须具备的知识，从而降低了复杂度：

- 对象提供隐藏实现细节的接口。
- 对象实施控制对象交互方式的规则。

为了说明这些优点，我们以一个称为双链表的数据结构体的实现为例。有关实际的实现，请参阅“通过类实现链表” （第 3-20 页）。

以下是一个三元素列表的图：



要向列表中添加一个节点，请断开列表中的现有节点，插入新节点，然后适当地重新连接节点。以下是基本步骤：

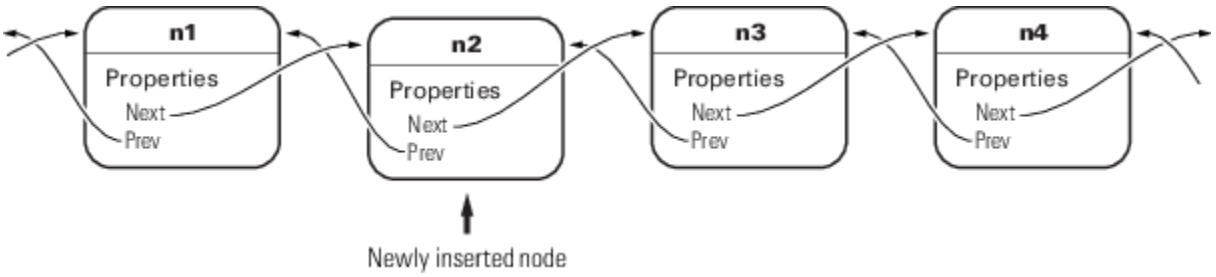
首先断开节点连接：

1 取消 **n2.Prev** 与 **n1** 的链接

2 取消 **n1.Next** 与 **n2** 的链接

现在创建新节点，连接它，并对原始节点重新编号：

1 将 **new.Prev** 链接到 **n1**
2 将 **new.Next** 链接到 **n3**（原来是 **n2**）
3 将 **n1.Next** 链接到 **new**（将是 **n2**）
4 将 **n3.Prev** 链接到 **new**（将是 **n2**）



方法如何执行这些步骤的详细信息封装在类设计中。每个节点对象都包含将自身插入列表中或从列表中删除自身的功能。

例如，在此类中，每个节点对象都有一个 **insertAfter** 方法。要向列表中添加节点，请创建节点对象，然后调用其 **insertAfter** 方法：

```
nnew = NodeConstructor;
nnew.insertAfter(n1)
```

由于节点类定义实现这些操作的代码，因此这段代码：

- 由类编写者以最佳方式实现
- 始终与最新版本的类保持一致
- 经过适当测试
- 会在从 MAT 文件加载对象时自动更新旧版对象。

对象方法会强制实施节点交互规则。采用这种设计，便无需在使用这些对象的应用程序中设计强制实施规则。这也意味着应用程序在其自身的过程实现中产生错误的可能性降低。

便于模块化

当您将系统分解成对象（汽车 –> 引擎 –> 燃油系统 –> 氧气传感器）时，您就根据自然边界划分了模块。类对代码模块化提供三个级别的控制：

- 公共 - 任何代码都可以访问此特定属性或调用此方法。
- 受保护 - 只有此对象的方法和从此对象的类派生的对象的方法才能访问此属性或调用此方法。
- 私有 - 只有对象自己的方法才能访问此属性或调用此方法。

重载的函数和运算符

定义类时，可以重载现有 MATLAB 函数来处理新对象。例如，MATLAB 串行端口类可重载 **fread** 函数，从连接到此对象所代表的端口的设备读取数据。如果您定义了用于表示数据的类，您可以为其定义各种运算，例如相等 (**eq**) 或加法 (**plus**)。

另请参阅

详细信息

- “类在 MATLAB 中的角色” (第 3-2 页)

句柄对象行为

本节内容

多个变量可以引用同一个句柄对象。因此，用户与句柄类的实例的交互不同于值类的实例。了解句柄对象的行为可以帮助您确定是实现句柄还是值类。本主题说明其中的一些交互。

有关句柄类的详细信息，请参阅“句柄类”。

什么是句柄?

句柄是特定类型的 MATLAB 对象。当变量保存句柄时，它实际上保存的是对相应对象的引用。

句柄对象允许多个变量引用同一个对象。句柄对象行为会影响复制句柄对象以及将它们传递给函数时发生的事情。

句柄的副本

句柄对象变量的所有副本都引用同一个底层对象。这种引用行为意味着，如果 **h** 标识某个句柄对象，则：

```
h2 = h;
```

创建另一个变量 **h2**，它引用与 **h** 相同的对象。

例如，MATLAB **audioplayer** 函数创建一个包含音频源数据的句柄对象，以重现特定的声音片段。**audioplayer** 函数返回的变量可标识音频数据，并允许您访问对象函数来播放音频。

MATLAB 软件包含音频数据，您可以加载这些数据并使用它们创建 **audioplayer** 对象。以下示例会加载音频数据、创建音频播放器，并播放音频：

```
load gong Fs y
gongSound = audioplayer(y,Fs);
play(gongSound)
```

假设您将 **gongSound** 对象句柄复制到另一个变量 (**gongSound2**)：

```
gongSound2 = gongSound;
```

变量 **gongSound** 和 **gongSound2** 是同一句柄的副本，因此引用的是同一音频源。使用任一变量访问 **audioplayer** 信息。

例如，通过为 **SampleRate** 属性赋一个新值来设置锣声音频源的采样率。首先获取当前采样率，然后设置新采样率：

```
sr = gongSound.SampleRate;
disp(sr)
```

```
8192
```

```
gongSound.SampleRate = sr*2;
```

您可以使用 gongSound2 访问同一音频源：

```
disp(gongSound2.SampleRate)
```

```
16384
```

以新采样率播放锣声：

```
play(gongSound2)
```

在函数中修改过的句柄对象

将实参传递给函数时，函数会将变量从调用该函数的工作区复制到函数工作区的形参变量中。

将非句柄变量传递给函数不会影响调用方工作区中的原始变量。例如，myFunc 会修改名为 var 的局部变量，但当函数结束时，局部变量 var 便不再存在：

```
function myFunc(var)
  var = var + 1;
end
```

定义一个变量，并将其传递给 myfunc：

```
x = 12;
myFunc(x)
```

执行 myFunc(x) 后，x 的值没有更改：

```
disp(x)
```

```
12
```

myFunc 函数可以返回修改后的值，您可以将该值赋给同一个变量名称 (x) 或另一个变量。

```
function out = myFunc(var)
  out = var + 1;
end
```

修改 myfunc 中的值：

```
x = 12;
x = myFunc(x);
disp(x)
```

```
13
```

当参数是句柄变量时，函数只复制句柄，而不复制由该句柄标识的对象。两个句柄（原始句柄和本地副本）引用同一个对象。

当函数修改对象句柄引用的数据时，可以从调用工作区的句柄变量中访问这些更改，而不需要返回修改后的对象。

例如，使用 modifySampleRate 函数更改 audioplayer 的采样率：

```
function modifySampleRate(audioObj,sr)
  audioObj.SampleRate = sr;
end
```

创建一个 **audioplayer** 对象，并将其传递给 **modifySampleRate** 函数：

```
load gong Fs y
gongSound = audioplayer(y,Fs);
disp(gongSound.SampleRate)
```

```
8192
```

```
modifySampleRate(gongSound,16384)
disp(gongSound.SampleRate)
```

```
16384
```

modifySampleRate 函数不需要返回修改后的 **gongSound** 对象，因为 **audioplayer** 对象是句柄对象。

确定对象是否为句柄

句柄对象是 **handle** 类的成员。因此，您始终可以使用 **isa** 函数将对象标识为句柄。测试句柄变量时，**isa** 返回逻辑值 **true** (**1**)：

```
load gong Fs y
gongSound = audioplayer(y,Fs);
isa(gongSound,'handle')
```

要确定变量是否为有效的句柄对象，请使用 **isa** 和 **isvalid**：

```
if isa(gongSound,'handle') && isvalid(gongSound)
   ...
end
```

删除的句柄对象

删除句柄对象后，引用该对象的句柄变量仍可存在。这些变量将不再有效，因为它们所引用的对象已不再存在。对对象调用 **delete** 会删除对象，但不会清除句柄变量。

例如，创建一个 **audioplayer** 对象：

```
load gong Fs y
gongSound = audioplayer(y,Fs);
```

输出参数 **gongSound** 是一个句柄变量。调用 **delete** 删除该对象及其包含的音频源信息：

```
delete(gongSound)
```

但是，句柄变量仍然存在：

```
disp(gongSound)
```

```
handle to deleted audioplayer
```

whos 命令将 **gongSound** 显示为一个 **audioplayer** 对象：

```
whos
```

```
  Name          Size          Bytes  Class       Attributes
```

```
Fs           1x1            8  double
gongSound       1x1             0  audioplayer
y          42028x1         336224  double
```

注意 whos 命令返回的字节值不包括句柄引用的数据，因为许多变量可以引用相同的数据。

句柄 **gongSound** 不再引用有效对象，如 **isvalid** 句柄方法所示：

```
isvalid(gongSound)

ans =

  logical

   0
```

对已删除的句柄调用 **delete** 不起任何作用，也不会导致错误。您可以将同时包含有效和无效句柄的数组传递给 **delete**。MATLAB 会删除有效句柄，但遇到已无效的句柄时不会生成错误。

您不能通过无效的句柄变量来访问属性：

```
gongSound.SampleRate

Invalid or deleted object.
```

访问对象属性的函数和方法会导致错误：

```
play(gongSound)

Invalid or deleted object.
```

要删除变量 **gongSound**，请使用 **clear**：

```
clear gongSound
whos

  Name        Size          Bytes  Class    Attributes

  Fs          1x1               8  double
  y       42028x1          336224  double
```

另请参阅

详细信息

- “句柄类析构函数” （第 7-8 页）
- “句柄类和值类的比较” （第 7-2 页）

基本示例

创建简单类

本节内容

设计类

类的基本目的是定义封装数据的对象以及对该数据执行的操作。例如，**BasicClass** 定义一个属性和对该属性中的数据执行操作的两个方法：

- **Value** - 此属性包含存储在类对象中的数值数据
- **roundOff** - 此方法将属性值舍入到两位小数
- **multiplyBy** - 此方法将属性值乘以指定数值

用 **classdef ClassName...end** 模块开始类定义，然后在该模块中定义类属性和方法。以下是 **BasicClass** 的定义：

```
classdef BasicClass
  properties
    Value {mustBeNumeric}
  end
  methods
    function r = roundOff(obj)
      r = round([obj.Value],2);
    end
    function r = multiplyBy(obj,n)
      r = [obj.Value]*n;
    end
  end
end
```

有关类语法的汇总，请参阅 **classdef**。

要使用类，请执行以下操作：

- 将类定义保存在与该类同名的 **.m** 文件中。
- 创建该类的一个对象。
- 访问属性以将数据赋给属性。
- 调用方法以对这些数据执行操作。

创建对象

使用类名创建类的一个对象：

```
a = BasicClass
```

```
a =

  BasicClass with properties:

    Value: []
```

属性值最初为空。

访问属性

使用对象变量加点加属性名称的方式为 **Value** 属性赋值：

```
a.Value = pi/3;
```

要返回属性值，请使用不带赋值的圆点表示法：

```
a.Value
```

```
ans =

    1.0472
```

有关类属性的信息，请参阅“属性语法” （第 8-4 页）。

调用方法

对对象 **a** 调用 **roundOff** 方法：

```
roundOff(a)
```

```
ans =

    1.0500
```

将对象作为第一个参数传递给接受多个参数的方法，此处以 **multiplyBy** 方法的调用为例：

```
multiplyBy(a,3)
```

```
ans =

    3.1416
```

您也可以使用圆点表示法来调用方法：

```
a.multiplyBy(3)
```

使用圆点表示法时，没有必要将对象作为显式参数传递。该表示法使用圆点左侧的对象。

有关类方法的信息，请参阅“方法语法” （第 5-6 页）。

添加构造函数

类可以定义一个特殊的方法来创建类对象，称为构造函数。您可以使用构造函数方法将参数传递给构造函数，用以对属性赋值。**BasicClass** 的 **Value** 属性使用 **mustBeNumeric** 函数限制其可能的值。

以下是 **BasicClass** 类的构造函数。如果在调用构造函数时带有输入参数，此参数会被赋给 **Value** 属性。如果在调用构造函数时不带输入参数，则 **Value** 属性采用其默认值，即空 (**[]**)。

```
methods
   function obj = BasicClass(val)
      if nargin == 1
         obj.Value = val;
      end
   end
end
```

通过将此构造函数添加到类定义中，只需一个步骤即可创建对象并设置属性值：

```
a = BasicClass(pi/3)
```

```
a =

  BasicClass with properties:

    Value: 1.0472
```

构造函数还可以执行与创建类对象相关的其他操作。

有关构造函数的信息，请参阅“类构造函数方法” （第 9-12 页）。

方法向量化

MATLAB 支持运算向量化。例如，您可以向向量添加数字：

```
[1 2 3] + 2
```

```
ans =

     3     4     5
```

MATLAB 将数字 **2** 添加到数组 **[1 2 3]** 中的每个元素。要向量化算术运算符方法，请将 **obj.Value** 属性引用括在方括号中。

```
[obj.Value] + 2
```

此语法使方法能够处理对象数组。例如，使用索引赋值创建一个对象数组。

```
obj(1) = BasicClass(2.7984);
obj(2) = BasicClass(sin(pi/3));
obj(3) = BasicClass(7);
```

这两个表达式是等效的。

```
[obj.Value] + 2
[obj(1).Value obj(2).Value obj(3).Value] + 2
```

roundOff 方法是向量化的，因为属性引用用方括号括起来。

```
r = round([obj.Value],2);
```

由于 **roundOff** 是向量化的，它可以对数组进行操作。

```
roundOff(obj)

ans =

   2.8000    0.8700    7.0000
```

重载函数

类可以通过定义与现有 MATLAB 函数同名的方法来实现现有功能，例如加法。例如，假设您要添加两个 **BasicClass** 对象。这通常意味着将每个对象的 **Value** 属性的值相加。

下面是 MATLAB **plus** 函数的重载版本。它将 **BasicClass** 类的加法定义为属性值相加：

```
methods
  function r = plus(o1,o2)
    r = [o1.Value] + [o2.Value];
  end
end
```

通过实现名为 **plus** 的方法，您可以对 **BasicClass** 的对象使用“+”运算符。

```
a = BasicClass(pi/3);
b = BasicClass(pi/4);
a + b

ans =

   1.8326
```

通过向量化 plus 方法，您可以对对象数组执行运算。

```
a = BasicClass(pi/3);
b = BasicClass(pi/4);
c = BasicClass(pi/2);
ar = [a b];
ar + c

ans =

   2.6180    2.3562
```

相关信息

有关重载函数的信息，请参阅“在类定义中重载函数” （第 9-21 页）。

有关重载运算符的信息，请参阅“运算符重载” （第 17-5 页）。

BasicClass 代码列表

以下是添加本主题中讨论的功能后的 **BasicClass** 定义：

```
classdef BasicClass
   properties
```

```
        Value {mustBeNumeric}
    end
    methods
        function obj = BasicClass(val)
            if nargin == 1
                obj.Value = val;
            end
        end
        function r = roundOff(obj)
            r = round([obj.Value],2);
        end
        function r = multiplyBy(obj,n)
            r = [obj.Value] * n;
        end
        function r = plus(o1,o2)
            r = [o1.Value] + [o2.Value];
        end
    end
end
```

另请参阅

相关示例

- “类组件” （第 5-2 页）
- “验证属性值” （第 8-12 页）

MATLAB 类概述

类在 MATLAB 中的角色

本节内容
“类” （第 3-2 页）
“一些基本关系” （第 3-3 页）

类

在 MATLAB 语言中，每个值都有一个类。例如，使用赋值语句创建变量会构造适当类的变量：

```
a = 7;
b = 'some text';
s.Name = 'Nancy';
s.Age = 64;
whos
```

```
whos
  Name      Size            Bytes  Class     Attributes

  a         1x1                 8  double
  b         1x9                18  char
  s         1x1               370  struct
```

whos 等基本命令显示工作区中每个值的类。这些信息有助于 MATLAB 用户识别某些值是字符并显示为文本，而其他值是双精度数值，等等。有些变量（如结构体）可以包含多个不同的值类。

预定义的类

MATLAB 定义了构成该语言所用的基本类型的基础类。这些类包括数值、**logical**、**char**、**cell**、**struct** 和函数句柄。

用户定义的类

您可以创建自己的 MATLAB 类。例如，您可以定义类来表示多项式。该类可以定义通常与 MATLAB 类相关联的操作，如加法、减法、索引、在命令行窗口中显示等。这些操作需要执行多项式加法、多项式减法等等效操作。例如，当您将两个多项式对象相加时：

```
p1 + p2
```

plus 运算必须能够对多项式对象执行加法运算，因为多项式类定义了该运算。

在定义类时，可以重载特殊的 MATLAB 函数（如加法运算符的 **plus.m**）。MATLAB 在用户将这些运算应用于您的类的对象时，会调用这些方法。

有关创建这样的类的示例，请参阅“Representing Polynomials with Classes”。

MATLAB 类 - 关键术语

MATLAB 类使用以下词语来说明类定义的不同部分和相关概念。

- 类定义 - 说明每个类实例共有的内容。
- 属性 - 类实例的数据存储
- 方法 - 一些特殊函数，用于实现通常仅对类实例执行的运算

- 事件 - 由类定义并在发生某个特定操作时由类实例广播的消息
- 特性 - 修改属性、方法、事件和类的行为的值
- 侦听程序 - 在广播事件通知时，通过执行回调函数来响应特定事件的对象
- 对象 - 类的实例，包含存储在对象属性中的实际数据值
- 子类 - 从其他类派生并从这些类继承方法、属性和事件的类（您可以利用子类来重用派生它们的超类中定义的代码）。
- 超类 - 用作创建更具体定义的类（即子类）的基础的类。
- 包 - 定义类和函数命名作用域的文件夹

一些基本关系

本节讨论 MATLAB 类使用的一些基本概念。

类

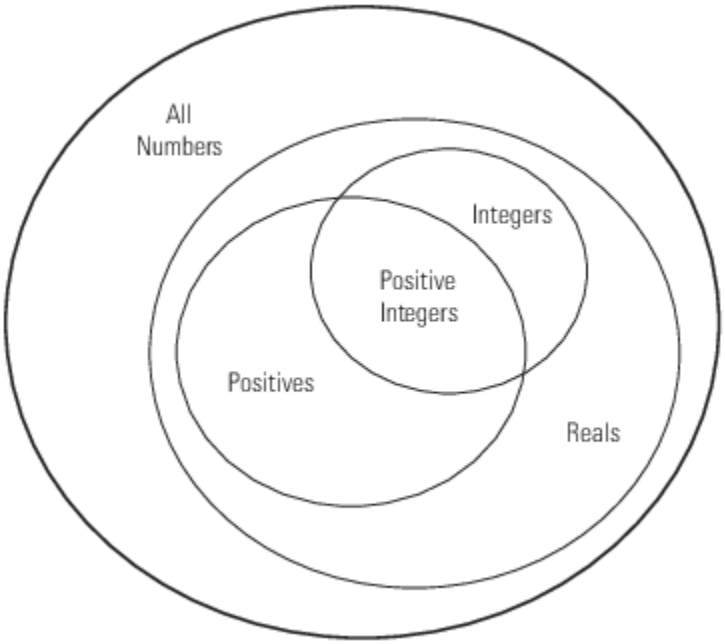
类是一种定义，它指定类的所有实例共享的某些特征。这些特征由定义类的属性、方法和事件以及修改每个类组件行为的特性值确定。类定义说明类的对象是如何创建和销毁的，对象包含什么数据，以及如何操作这些数据。

类的层次结构

有时，您可以基于现有类来定义新类。您可以通过这种方式在代表相似实体的新类中重用设计和方法。您可以通过创建子类来实现这种重用。子类定义的对象是超类所定义对象的子集。子类比超类更具体，并且可能会在从超类继承的那些组件中添加新的属性、方法和事件。

数学集可以帮助说明类之间的关系。在下图中，Positive Integers 是 Integers 的子集，也是 Positives 的子集。这三个集都是 Reals 的子集，而 Reals 又是 All Numbers 的子集。

Positive Integers 的定义有附加设定，即要求集的成员大于零。Positive Integers 合并了 Integers 和 Positives 的定义。得到的子集比超集更具体，因此定义也更窄，但子集仍然共享定义超集的所有特征。



“是”关系是一种用来确定基于现有超集定义特定子集是否合适的好方法。例如，以下每个陈述都成立：

- 正整数是整数
- 正整数是正数

如果“是”关系成立，则很可能您可以从代表某些更一般情况的一个或多个类中定义新类。

重用解

类通常组织成各个分类，以方便代码重用。例如，如果您定义一个类以实现与计算机串行端口的接口，它可能与用来实现与并行端口的接口的类相似。为了重用代码，您可以定义一个超类，该超类包含这两类端口共有的所有内容，然后从该超类派生子类，您只需要在子类中实现每个特定端口独有的内容。然后子类将继承超类的所有共有功能。

对象

类就像一个创建该类的特定实例的模板。此实例或对象包含由该类表示的特定实体的实际数据。例如，银行帐户类的实例是表示特定银行帐户的对象，具有实际的帐号和实际余额。该对象内置了执行由类定义的操作的能力，例如向帐户存款或从帐户余额中取款。

对象不仅仅是被动的数据容器。对象还可以主动管理包含的数据，如只允许执行某些操作，隐藏不需要公开的数据，以及防止外部客户端因执行对象设计中不包含的操作而误用数据。对象甚至还可以控制它们被销毁时所发生的行为。

封装信息

对象有一个很重要的特点：在您编写的软件中，您可以通过对象的属性和方法访问存储在对象中的信息，而无需知道这些信息如何存储，甚至也无需知道查询这些信息时是否涉及存储或计算。对象将访问对象的代码与方法和属性的内部实现隔离开来。您可以定义相应的类，来对不属于类的任何方法隐藏数据和操作。然后，您可以实现任何最适合预期用途的接口。

参考

[1] Shalloway, A., J. R. Trott, *Design Patterns Explained A New Perspective on Object-Oriented Design.*. Boston, MA: Addison-Wesley 2002.

[2] Gamma, E., R. Helm, R. Johnson, J. Vlissides, *Design Patterns Elements of Reusable Object-Oriented Software*. Boston, MA: Addison-Wesley 1995.

[3] Freeman, E., Elisabeth Freeman, Kathy Sierra, Bert Bates, *Head First Design Patterns*. Sebastopol, CA 2004.

另请参阅

相关示例

- “创建简单类” （第 2-2 页）
- “开发协同工作的类” （第 3-5 页）
- “用类表示结构化数据” （第 3-12 页）
- “通过类实现链表” （第 3-20 页）

开发协同工作的类

本节内容

构建类

此示例讨论如何设计和实现两个通过事件和侦听程序进行交互的类。这两个类分别表示银行帐户和帐户管理器。

要设计表示银行帐户的类，首先要确定构成银行帐户的数据元素和操作。例如，银行帐户包含：

- 帐号
- 帐户余额
- 状态（open、closed 等）

您必须对银行帐户执行一些操作：

- 为每个银行帐户创建一个对象
- 存款
- 取款
- 生成语句
- 保存并加载 **BankAccount** 对象

如果余额太低，在您尝试取款时，银行帐户会发出通知。当此事件发生时，银行帐户会向其他用于侦听通知的实体广播通知。在本示例中，用一个简化版的帐户管理程序来执行此任务。

在本示例中，帐户管理程序确定所有银行帐户的状态。该程序监控帐户余额并为下列三个值之一赋值：

- **open** - 帐户余额为正值
- **overdrawn** - 帐户余额透支，但透支额不超过 200 美元。
- **closed** - 帐户余额透支超过 200 美元。

这些功能定义 **BankAccount** 和 **AccountManager** 类的要求。只包括满足您特定目标所需的功能。通过子类化 **BankAccount** 并向子类添加更多特定功能，以支持特殊类型的帐户。根据需要扩展 **AccountManager**，以支持新帐户类型。

指定类组件

类将数据存储在属性中，用方法实现操作，并支持用事件和侦听程序进行通知。以下部分说明 **BankAccount** 和 **AccountManager** 类如何定义这些组件。

类数据

类会定义以下属性来存储帐号、帐户余额和帐户状态：

- **AccountNumber** - 该属性存储标识帐户的编号。创建类的实例时，MATLAB 会为此属性赋值。只有 **BankAccount** 类方法可以设置此属性。**SetAccess** 属性是 **private**。
- **AccountBalance** - 该属性存储帐户的当前余额。存款和取款等类操作会为该属性进行相应赋值。只有 **BankAccount** 类方法可以设置此属性。**SetAccess** 属性是 **private**。
- **AccountStatus** - **BankAccount** 类定义该属性的默认值。每当 **AccountBalance** 的值低于 **0** 时，**AccountManager** 类方法便会更改该值。**Access** 特性指定只有 **AccountManager** 和 **BankAccount** 类可以访问该属性。
- **AccountListener** - 用于存储 **InsufficentFunds** 事件侦听程序。保存 **BankAccount** 对象不会保存此属性，因为加载对象时必须重新创建侦听程序。

类操作

以下方法实现类构建中定义的操作：

- **BankAccount** - 接受帐号和初始余额，以创建表示帐户的对象。
- **deposit** - 发生存款交易时，更新 **AccountBalance** 属性。
- **withdraw** - 发生取款交易时，更新 **AccountBalance** 属性。
- **getStatement** - 显示帐户信息。
- **loadobj** - 从 MAT 文件加载对象时，重新创建帐户管理侦听程序。

类事件

帐户管理程序会更改具有负余额的银行帐户的状态。要实现此操作，**BankAccount** 类会在取款导致负余额时触发一个事件。因此，触发 **InsufficientsFunds** 事件的行为在 **withdraw** 方法中发生。

要定义事件，请在 **events** 代码块中指定名称。通过调用 **notify** 句柄类方法触发事件。由于 **InsufficientsFunds** 不是预定义的事件，因此您可以用任何 **char** 向量命名它，并通过任何操作触发它。

BankAccount 类实现

请务必确保只有一组数据与 **BankAccount** 类的任一对象相关联。例如，您不希望对象的独立副本具有不同的帐户余额值。因此，请将 **BankAccount** 类当作句柄类来实现。给定句柄对象的所有副本都引用相同的数据。

BankAccount 类概要

BankAccount 类	讨论
`classdef BankAccount < handle`	句柄类，因为任何 **BankAccount** 实例都应该只有一个副本。“句柄类和值类的比较” （第 7-2 页）

`  properties (Access = ?AccountManager)`
`    AccountStatus = 'open'`
`  end`
	AccountStatus 包含由当前余额确定的帐户状态。访问仅限于 **BankAccount** 和 **AccountManager** 类。“类成员访问” （第 12-10 页）

BankAccount 类	讨论

properties (SetAccess = private)
  AccountNumber
  AccountBalance
end
	AccountManager 类方法访问的 AccountStatus 属性。

properties (Transient) AccountListener end	AccountNumber 和 AccountBalance 属性拥有私有 set 访问权限。 AccountListener property 是临时属性，因此不会保存侦听程序句柄。 请参阅“属性特性” （第 8-7 页）。
events InsufficientFunds end	类定义名为 InsufficentFunds 的事件。withdraw 方法在帐户余额变为负数时触发事件。 有关事件和侦听程序的信息，请参阅“事件”。
methods	普通方法的代码块。有关语法，请参阅“方法语法” （第 5-6 页）。

function BA = BankAccount(AccountNumber,InitialBalance)
  BA.AccountNumber = AccountNumber;
  BA.AccountBalance = InitialBalance;
  BA.AccountListener = AccountManager.addAccount(BA);
end
	构造函数用输入参数初始化属性值。
	
	AccountManager.addAccount 是 AccountManager 类的静态方法。为 InsufficientFunds 事件创建侦听程序，并将侦听程序句柄存储在 AccountListener 属性中。

function deposit(BA,amt)
  BA.AccountBalance = BA.AccountBalance + amt;
  if BA.AccountBalance > 0
  BA.AccountStatus = 'open';
  end
end
	deposit 调整 AccountBalance 属性的值。
	
	如果 AccountStatus 是 closed，但后续存款使 AccountBalance 变为正值，则 AccountStatus 将重置为 open。

function withdraw(BA,amt)
  if (strcmp(BA.AccountStatus,'closed')&& ...
  BA.AccountBalance < 0)
  disp(['Account ',num2str(BA.AccountNumber),...
  ' has been closed.'])
  return
  end
  newbal = BA.AccountBalance - amt;
  BA.AccountBalance = newbal;
  if newbal < 0
  notify(BA,'InsufficientFunds')
  end
end
	更新 AccountBalance 属性。如果取款导致帐户余额为负值，notify 将触发 InsufficentFunds 事件。
	
	有关侦听程序的详细信息，请参阅“事件和侦听程序语法” （第 11-9 页）。

function getStatement(BA)
  disp('----------')
  disp(['Account: ',num2str(BA.AccountNumber)])
  ab = sprintf('%0.2f',BA.AccountBalance);
  disp(['CurrentBalance: ',ab])
  disp(['Account Status: ',BA.AccountStatus])
  disp('----------')
end
end
	显示关于帐户的选定信息。普通方法代码块在此节结束。

methods (Static)	静态方法代码块的开始。请参阅“静态方法” （第 9-19 页）

BankAccount 类	讨论

`function obj = loadobj(s)`
`  if isstruct(s)`
`  accNum = s.AccountNumber;`
`  initBal = s.AccountBalance;`
`  obj = BankAccount(accNum,initBal);`
`  else`
`  obj.AccountListener = AccountManager.addAccount(s);`
`  end`
`end`
	loadobj 方法：
	• 如果加载操作失败，请从 **struct** 创建对象。
	• 使用新创建的 **BankAccount** 对象作为源来重新创建侦听程序。
	有关保存和加载对象的详细信息，请参阅“对象的保存和加载过程” （第 13-2 页）

`  end` `end`	静态方法代码块的结束 **classdef** 的结束

展开类代码

```
classdef BankAccount < handle
  properties (Access = ?AccountManager)
    AccountStatus = 'open'
  end
  properties (SetAccess = private)
    AccountNumber
    AccountBalance
  end
  properties (Transient)
    AccountListener
  end
  events
    InsufficientFunds
  end
  methods
    function BA = BankAccount(accNum,initBal)
      BA.AccountNumber = accNum;
      BA.AccountBalance = initBal;
      BA.AccountListener =  AccountManager.addAccount(BA);
    end
    function deposit(BA,amt)
      BA.AccountBalance = BA.AccountBalance + amt;
      if BA.AccountBalance > 0
        BA.AccountStatus = 'open';
      end
    end
    function withdraw(BA,amt)
      if (strcmp(BA.AccountStatus,'closed')&& BA.AccountBalance <= 0)
        disp(['Account ',num2str(BA.AccountNumber),' has been closed.'])
        return
      end
      newbal = BA.AccountBalance - amt;
      BA.AccountBalance = newbal;
      if newbal < 0
        notify(BA,'InsufficientFunds')
      end
    end
    function getStatement(BA)
      disp('----------------------')
      disp(['Account: ',num2str(BA.AccountNumber)])
      ab = sprintf('%0.2f',BA.AccountBalance);
      disp(['CurrentBalance: ',ab])
      disp(['Account Status: ',BA.AccountStatus])
      disp('----------------------')
    end
  end
  methods (Static)
    function obj = loadobj(s)
      if isstruct(s)
        accNum = s.AccountNumber;
        initBal = s.AccountBalance;
```

```
        obj = BankAccount(accNum,initBal);
      else
        obj.AccountListener = AccountManager.addAccount(s);
      end
    end
  end
end
```

构建 AccountManager 类

AccountManager 类旨在为帐户提供服务。对于 **BankAccount** 类，**AccountManager** 类会侦听导致余额下降到负值范围的取款。当 **BankAccount** 对象触发 **InsufficientsFunds** 事件时，**AccountManager** 会重置帐户状态。

AccountManager 类不存储数据，因此不需要属性。**BankAccount** 对象存储侦听程序对象的句柄。

AccountManager 执行以下两项操作：

- 根据取款结果，为每个帐户赋一个状态
- 通过监控帐户余额向系统添加帐户。

类组件

AccountManager 类实现以下两个方法：

- **assignStatus** - 将状态赋给 **BankAccount** 对象的方法。用作侦听程序回调。
- **addAccount** - 创建 **InsufficientFunds** 侦听程序的方法。

实现 AccountManager 类

AccountManager 类将这两个方法均当作静态方法来实现，因为不需要 **AccountManager** 对象。这些方法对 **BankAccount** 对象进行操作。

不应将 **AccountManager** 实例化。将 **AccountManager** 类的功能与 **BankAccount** 类分隔开来可提供更大的灵活性和可扩展性。例如，这样做使您能够：

- 扩展 **AccountManager** 类，以支持其他类型的帐户，同时使单个帐户类保持简单和专门化。
- 更改帐户状态的判别条件，而不影响已保存和已加载的 **BankAccount** 对象的兼容性。
- 开发一个 **Account** 超类，该超类提取所有帐户的共性而不要求每个子类实现帐户管理功能。

AccountManager 类概要

AccountManager 类	讨论
classdef AccountManager	此类定义 **InsufficentFunds** 事件侦听程序和侦听程序回调。
methods (Static)	没有必要创建此类的实例，因此定义的方法是静态方法。请参阅“静态方法” （第 9-19 页）。

AccountManager 类	讨论

function assignStatus(BA)
  if BA.AccountBalance < 0
  if BA.AccountBalance < -200
  BA.AccountStatus = 'closed';
  else
  BA.AccountStatus = 'overdrawn';
  end
  end
end
	assignStatus 方法是 InsufficentFunds 事件侦听程序的回调。它根据 AccountBalance 属性的值来确定 BankAccount 对象的 AccountStatus 属性的值。
	
	BankAccount 类构造函数调用 AccountManager 的 addAccount 方法来创建和存储此侦听程序。

function lh = addAccount(BA)
  lh = addlistener(BA, 'InsufficientFunds', ...
  @(src, ~)AccountManager.assignStatus(src));
end
	addAccount 为 BankAccount 类定义的 InsufficentFunds 事件创建侦听程序。
	
	请参阅 “Control Listener Lifecycle”

end end	methods 和 classdef 的 end 语句。

展开类代码

```
classdef AccountManager
  methods (Static)
    function assignStatus(BA)
      if BA.AccountBalance < 0
        if BA.AccountBalance < -200
          BA.AccountStatus = 'closed';
        else
          BA.AccountStatus = 'overdrawn';
        end
      end
    end
    function lh = addAccount(BA)
      lh = addlistener(BA, 'InsufficientFunds', ...
        @(src, ~)AccountManager.assignStatus(src));
    end
  end
end
```

使用 BankAccount 对象

BankAccount 类虽然非常简单，但也能说明 MATLAB 类的行为方式。例如，创建一个具有帐号和 500 美元初始存款的 BankAccount 对象：

```
BA = BankAccount(1234567,500)
```

```
BA =

  BankAccount with properties:

    AccountNumber: 1234567
    AccountBalance: 500
   AccountListener: [1x1 event.listener]
```

使用 getStatement 方法检查状态：

```
getStatement(BA)
```

```
--------------------
Account: 1234567
CurrentBalance: 500.00
Account Status: open
--------------------
```

提取 600 美元，这会导致负的帐户余额：

```
withdraw(BA,600)
getStatement(BA)

--------------------
Account: 1234567
CurrentBalance: -100.00
Account Status: overdrawn
--------------------
```

600 美元的取款触发了 **InsufficientsFunds** 事件。根据 **AccountManager** 类定义的当前判别条件，相应的状态为 **overdrawn**。

再提取 200 美元：

```
withdraw(BA,200)
getStatement(BA)

--------------------
Account: 1234567
CurrentBalance: -300.00
Account Status: closed
--------------------
```

现在，侦听程序已将 **AccountStatus** 设置为 **closed**，进一步的取款尝试将被阻止而不会触发事件：

```
withdraw(BA,100)

Account 1234567 has been closed.
```

如果通过存款将 **AccountBalance** 恢复为正值，则 **AccountStatus** 将返回到 open 状态，并重新允许取款：

```
deposit(BA,700)
getStatement(BA)

--------------------
Account: 1234567
CurrentBalance: 400.00
Account Status: open
--------------------
```

用类表示结构化数据

本节内容

将对象作为数据结构体

以下示例定义用于存储具有特定结构的数据的类。使用一致的结构存储数据，从而更轻松地创建函数来操作数据。MATLAB **struct** 可以包含说明特定数据元素的字段名称，这是一种有用的数据组织方式。但是，类可以定义数据存储（属性）和对该数据执行的操作（方法）。本示例说明了这些优点。

示例背景

对于本示例，数据表示拉伸应力/应变测量值。这些数据用于计算各种材料的弹性模数。简而言之，应力是施加在材料上的力，应变是由此产生的形变。其比率决定材料的特性。虽然这种方法是一个大大简化了的流程，但足以满足本示例的目的。

数据的结构

下表说明了数据的结构。

数据	描述
Material	**char** 标识被测材料的类型的向量
SampleNumber	测试样本的编号
Stress	表示测试过程中施加给样本的应力的数值向量。
Strain	表示所施加应力对应值处应变的数值向量。
Modulus	定义在测材料弹性模数的数值，该数值由应力和应变数据计算得出

TensileData 类

以下示例从类的简单实现开始，并以此实现为基础来说明特性如何增强类的有用性。

该类的第一个版本仅提供数据存储。该类为每个必需的数据元素定义一个属性。

```
classdef TensileData
  properties
```

```
    Material
    SampleNumber
    Stress
    Strain
    Modulus
  end
end
```

创建实例并赋给数据

以下语句将创建一个 **TensileData** 对象，并为其进行数据赋值：

```
td = TensileData;
td.Material = 'Carbon Steel';
td.SampleNumber = 001;
td.Stress = [2e4 4e4 6e4 8e4];
td.Strain = [.12 .20 .31 .40];
td.Modulus = mean(td.Stress./td.Strain);
```

类相对于结构体的优势

像处理任何 MATLAB 结构体一样处理 **TensileData** 对象（在以前的语句中为 **td**）。但是，相对于使用通用数据结构（例如 MATLAB **struct**），将特化数据结构体定义为类具有以下优势：

- 用户不小心拼错字段名称时会收到错误消息。例如，键入以下内容：

  ```
  td.Modulus = ...
  ```

  会直接在结构体中添加一个字段。但如果 **td** 是 **TensileData** 类的一个实例，则将返回错误。
- 类易于重用。一旦定义了类，就可以轻松地使用添加新属性的子类来扩展它。
- 类易于识别。类有名称，因此您可以使用 **whos** 和 **class** 函数与工作区浏览器来识别对象。使用类名，可以轻松地用有意义的名称来引用记录。
- 赋值时，类可以验证各个字段值，包括类或值。
- 类可以限制对字段的访问，例如，允许读取某个特定字段，但不能更改该字段。

将属性限制为特定值

可以通过定义属性 set 访问方法，将属性限制为特定值。MATLAB 每次设置属性值时都会调用 set 访问方法。

Material 属性 set 函数

Material 属性 set 方法将属性的赋值限制为以下字符串之一：**aluminum**、**stainless steel** 或 **carbon steel**。

将此函数定义添加到方法代码块中。

```
classdef TensileData
  properties
    Material
    SampleNumber
    Stress
    Strain
```

```
    Modulus
  end
  methods
    function obj = set.Material(obj,material)
      if (strcmpi(material,'aluminum') ||...
          strcmpi(material,'stainless steel') ||...
          strcmpi(material,'carbon steel'))
        obj.Material = material;
      else
        error('Invalid Material')
      end
    end
  end
end
```

当尝试设置 **Material** 属性时，MATLAB 在设置属性值之前会调用 **set.Material** 方法。

如果该值与可接受值匹配，函数便将属性设置为该值。set 方法中的代码可以直接访问属性，以避免递归调用属性 set 方法。

例如：

```
td = TensileData;
td.Material = 'brass';
```

```
Error using TensileData/set.Material
Invalid Material
```

用构造函数简化接口

通过添加一个构造函数来简化与 **TensileData** 类的接口，该构造函数：

- 使您能够将数据作为参数传递给构造函数
- 为属性赋值

该构造函数是一个与类同名的方法。

```
methods
  function td = TensileData(material,samplenum,stress,strain)
    if nargin > 0
      td.Material = material;
      td.SampleNumber = samplenum;
      td.Stress = stress;
      td.Strain = strain;
    end
  end
end
```

使用以下语句创建一个完整填充了数据的 **TensileData** 对象：

```
td = TensileData('carbon steel',1,...
    [2e4 4e4 6e4 8e4],...
    [.12 .20 .31 .40]);
```

按需计算数据

如果某个属性的值取决于其他属性的值，请使用 **Dependent** 特性定义该属性。MATLAB 不存储从属属性的值。当访问从属属性时，将使用从属属性 get 方法来确定属性值。当显示对象属性或作为显式查询的结果时，可以进行访问。

计算模数

TensileData 对象不存储 **Modulus** 属性的值。构造函数没有 **Modulus** 属性值的输入参数。**Modulus** 的值：

- 根据 **Stress** 和 **Strain** 属性的值计算得出
- 在 **Stress** 或 **Strain** 属性的值发生变化时也必定会改变

因此，最好仅在需要时计算 **Modulus** 属性的值。使用属性 **get** 访问方法计算 **Modulus** 的值。

Modulus 属性 get 方法

Modulus 属性取决于 **Stress** 和 **Strain**，因此其 **Dependent** 特性为 **true**。将 **Modulus** 属性放在单独的 **properties** 代码块中，并设置 **Dependent** 特性。

get.Modulus 方法计算并返回 **Modulus** 属性的值。

```
properties (Dependent)
  Modulus
end
```

仅使用默认特性在 **methods** 代码块中定义属性 get 方法。

```
methods
  function modulus = get.Modulus(obj)
    ind = find(obj.Strain > 0);
    modulus = mean(obj.Stress(ind)./obj.Strain(ind));
  end
end
```

此方法在消除分母数据中的零后，计算应力与应变数据的平均比率。

查询属性时，MATLAB 会调用 **get.Modulus** 方法。例如，

```
td = TensileData('carbon steel',1,...
    [2e4 4e4 6e4 8e4],...
    [.12 .20 .31 .40]);
td.Modulus
```

```
ans =
  1.9005e+005
```

Modulus 属性 set 方法

要设置 **Dependent** 属性的值，该类必须实现属性 set 方法。不需要允许显式设置 **Modulus** 属性。但您可以使用 set 方法提供自定义的错误消息。**Modulus** set 方法引用当前属性值，然后返回错误：

```
methods
  function obj = set.Modulus(obj,~)
    fprintf('%s%d\n','Modulus is: ',obj.Modulus)
    error('You cannot set the Modulus property');
  end
end
```

显示 TensileData 对象

TensileData 类重载 **disp** 方法。此方法控制命令行窗口中的对象显示。

disp 方法显示 **Material**、**SampleNumber**、**Modulus** 属性的值。它不显示 **Stress** 和 **Strain** 属性数据。这些属性包含在命令行窗口中不容易查看的原始数据。

disp 方法使用 **fprintf** 在命令行窗口中显示格式化的文本：

```
methods
  function disp(td)
    fprintf(1,...
      'Material: %s\nSample Number: %g\nModulus: %1.5g\n',...
      td.Material,td.SampleNumber,td.Modulus);
  end
end
```

绘制应力对应变的图的方法

查看应力/应变数据图有助于确定材料在所施加的拉力范围内的行为。**TensileData** 类会重载 MATLAB **plot** 函数。

plot 方法创建应力对应变数据的线性图，并添加标题和轴标签，从而生成一个拉力数据记录的标准化图：

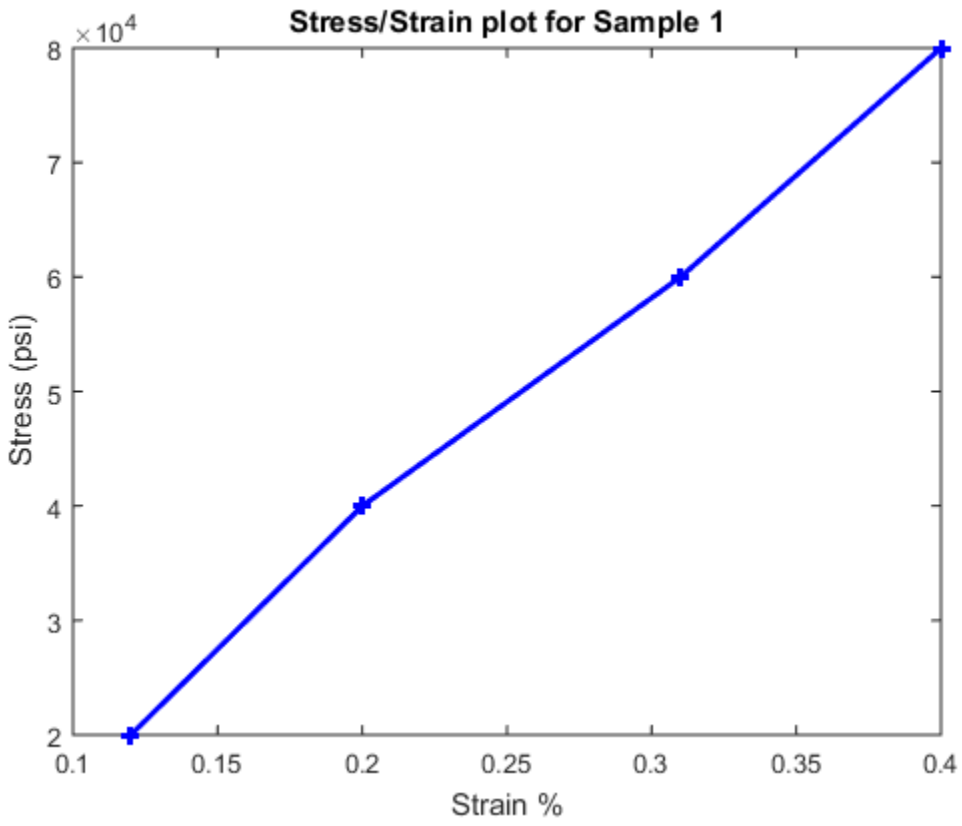
```
methods
  function plot(td,varargin)
    plot(td.Strain,td.Stress,varargin{:})
    title(['Stress/Strain plot for Sample',...
      num2str(td.SampleNumber)])
    ylabel('Stress (psi)')
    xlabel('Strain %')
  end
end
```

该方法的第一个参数是包含数据的 **TensileData** 对象。

该方法将参数的变量列表 (**varargin**) 直接传递给内置 **plot** 函数。**TensileData plot** 方法允许您传递线条设定符参数或属性名称-值对组。

例如：

```
td = TensileData('carbon steel',1,...
    [2e4 4e4 6e4 8e4],[.12 .20 .31 .40]);
plot(td,'-+b','LineWidth',2)
```



TensileData 类概要

示例代码	讨论
`classdef TensileData`	值类支持对象的独立副本。有关详细信息，请参阅“句柄类和值类的比较” （第 7-2 页）

`properties`
`  Material`
`  SampleNumber`
`  Stress`
`  Strain`
`end`
	请参阅“数据的结构” （第 3-12 页）

`properties (Dependent)`
`  Modulus`
`end`
	查询时计算 **Modulus**。有关此代码的信息，请参阅“按需计算数据” （第 3-14 页）。
	
	有关一般信息，请参阅“从属属性的 set 和 get 方法” （第 8-32 页）

`methods`	有关方法的一般信息，请参阅“Ordinary Methods”

示例代码	讨论

`function td = TensileData(material,samplenum,...`
`  stress,strain)`
`  if nargin > 0`
`  td.Material = material;`
`  td.SampleNumber = samplenum;`
`  td.Stress = stress;`
`  td.Strain = strain;`
`  end`
`end`
	有关此代码的信息，请参阅“用构造函数简化接口” （第 3-14 页）。
	
	有关构造函数的一般信息，请参阅“类构造函数方法” （第 9-12 页）

`function obj = set.Material(obj,material)`
`  if (strcmpi(material,'aluminum') ||...`
`  strcmpi(material,'stainless steel') ||...`
`  strcmpi(material,'carbon steel'))`
`  obj.Material = material;`
`  else`
`  error('Invalid Material')`
`  end`
`end`
	限制 **Material** 属性的可能值。
	
	有关此代码的信息，请参阅“将属性限制为特定值” （第 3-13 页）。
	
	有关属性 set 方法的一般信息，请参阅“属性 set 方法” （第 8-27 页）。

`function m = get.Modulus(obj)`
`  ind = find(obj.Strain > 0);`
`  m = mean(obj.Stress(ind)./obj.Strain(ind));`
`end`
	查询时计算 **Modulus** 属性。
	
	有关此代码的信息，请参阅“Modulus 属性 get 方法” （第 3-15 页）。
	
	有关属性 get 方法的一般信息，请参阅“属性 set 方法” （第 8-27 页）。

`function obj = set.Modulus(obj,~)`
`  fprintf('%s%d\n','Modulus is: ',obj.Modulus)`
`  error('You cannot set Modulus property');`
`end`
	为 **Dependent Modulus** 属性添加 set 方法。有关此代码的信息，请参阅“Modulus 属性 set 方法” （第 3-15 页）。
	
	有关属性 set 方法的一般信息，请参阅“属性 set 方法” （第 8-27 页）。

`function disp(td)`
`  fprintf(1,'Material: %s\nSample Number: %g\nModulus: %1.5g\n',...`
`  td.Material,td.SampleNumber,td.Modulus)`
`end`
	重载 **disp** 方法以显示特定属性。
	
	有关此代码的信息，请参阅“显示 TensileData 对象” （第 3-16 页）
	
	有关重载 disp 的一般信息，请参阅“Overloading the disp Function”

`function plot(td,varargin)`
`  plot(td.Strain,td.Stress,varargin{:})`
`  title(['Stress/Strain plot for Sample',...`
`  num2str(td.SampleNumber)])`
`  ylabel('Stress (psi)')`
`  xlabel('Strain %')`
`end`
	重载 **plot** 函数以接受 **TensileData** 对象和应力对应变的图。
	
	“绘制应力对应变的图的方法” （第 3-16 页）

`  end` `end`	**methods** 和 **classdef** 的 **end** 语句。

展开类代码

```
classdef TensileData
  properties
    Material
    SampleNumber
    Stress
    Strain
  end
  properties (Dependent)
```

```
    Modulus
  end

  methods
    function td = TensileData(material,samplenum,stress,strain)
      if nargin > 0
        td.Material = material;
        td.SampleNumber = samplenum;
        td.Stress = stress;
        td.Strain = strain;
      end
    end

    function obj = set.Material(obj,material)
      if (strcmpi(material,'aluminum') ||...
          strcmpi(material,'stainless steel') ||...
          strcmpi(material,'carbon steel'))
        obj.Material = material;
      else
        error('Invalid Material')
      end
    end

    function m = get.Modulus(obj)
      ind = find(obj.Strain > 0);
      m = mean(obj.Stress(ind)./obj.Strain(ind));
    end

    function obj = set.Modulus(obj,~)
      fprintf('%s%d\n','Modulus is: ',obj.Modulus)
      error('You cannot set Modulus property');
    end

    function disp(td)
      sprintf('Material: %s\nSample Number: %g\nModulus: %1.5g\n',...
        td.Material,td.SampleNumber,td.Modulus)
    end

    function plot(td,varargin)
      plot(td.Strain,td.Stress,varargin{:})
      title(['Stress/Strain plot for Sample ',...
        num2str(td.SampleNumber)])
      xlabel('Strain %')
      ylabel('Stress (psi)')
    end
  end
end
```

另请参阅

详细信息

- “类组件” （第 5-2 页）

通过类实现链表

本节内容

类定义代码

有关类定义代码列表，请参阅 “dlnode 类概要” （第 3-22 页）。

要使用该类，请创建一个名为 **@dlnode** 的文件夹，并将 **dlnode.m** 保存到该文件夹中。**@dlnode** 的父文件夹必须位于 MATLAB 路径上。或者，将 **dlnode.m** 保存到路径文件夹。

dlnode 类设计

dlnode 是用于创建双链表的类，其中每个节点均包含：

- 数据数组
- 下一个节点的句柄
- 上一个节点的句柄

每个节点都有相应的方法，使该节点能够：

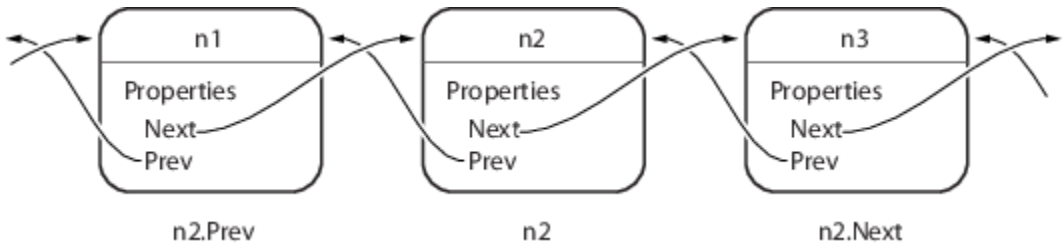
- 插入到链表中的指定节点之前
- 插入到链表中的特定节点之后
- 从列表中删除

类属性

dlnode 类将每个节点实现为具有以下三个属性的句柄对象：

- **Data** - 包含此节点的数据
- **Next** - 包含列表中下一个节点的句柄 (**SetAccess = private**)
- **Prev** - 包含列表中上一个节点的句柄 (**SetAccess = private**)

下图显示包含三个节点 **n1**、**n2** 和 **n3** 的列表。它还显示节点如何引用下一个节点和上一个节点。



类方法

dlnode 类实现以下方法：

- **dlnode** - 构造一个节点，并将作为输入传递的值赋给 **Data** 属性
- **insertAfter** - 在指定的节点之后插入此节点
- **insertBefore** - 在指定的节点之前插入此节点
- **removeNode** - 从列表中删除此节点，并重新连接其余节点
- **clearList** - 高效删除大型列表
- **delete** - 删除列表时 MATLAB 调用的私有方法。

创建双链表

通过将节点数据传递给 **dlnode** 类构造函数来创建节点。例如，以下语句将创建三个节点，分别包含数据值 **1**、**2**、**3**：

```
n1 = dlnode(1);
n2 = dlnode(2);
n3 = dlnode(3);
```

使用为此目的设计的类方法将这些节点构建为一个双链表：

```
n2.insertAfter(n1) % Insert n2 after n1
n3.insertAfter(n2) % Insert n3 after n2
```

现在，这三个节点就链接在一起了：

```
n1.Next % Points to n2
```

```
ans =

  dlnode with properties:

    Data: 2
    Next: [1x1 dlnode]
    Prev: [1x1 dlnode]
```

```
n2.Next.Prev % Points back to n2
```

```
ans =

  dlnode with properties:

    Data: 2
    Next: [1x1 dlnode]
    Prev: [1x1 dlnode]
```

```
n1.Next.Next % Points to n3
```

```
ans =

  dlnode with properties:

    Data: 3
    Next: []
    Prev: [1x1 dlnode]
```

```
n3.Prev.Prev % Points to n1

ans =

  dlnode with properties:

    Data: 1
    Next: [1x1 dlnode]
    Prev: []
```

为什么要对链表使用句柄类?

每个节点都是唯一的，因为不可能有两个节点可以在同一个节点之前或之后。

例如，节点对象 **node** 在其 **Next** 属性中包含下一个节点对象 **node.Next** 的句柄。同样，**Prev** 属性包含前一节点 **node.Prev** 的句柄。使用上一节中定义的三节点链表，您可以证实以下语句为 true：

```
n1.Next == n2
n2.Prev == n1
```

现在假设您将 **n2** 赋给 **x**：

```
x = n2;
```

则以下两个等式也为 true：

```
x == n1.Next
x.Prev == n1
```

但是，由于节点的每个实例都是唯一的，因此列表中只有一个节点能够满足与 **n1.Next** 相等的条件，并且具有 **Prev** 属性，该属性包含 **n1** 的句柄。因此，**x** 必须指向与 **n2** 相同的节点。

必须有一种方式让多个变量引用同一个对象。MATLAB **handle** 类为 **x** 和 **n2** 提供了引用同一节点的方式。

句柄类定义 **eq** 方法（使用 **methods('handle')** 列出句柄类方法），该方法支持将 == 运算符用于所有句柄对象。

相关信息

有关句柄类的详细信息，请参阅“句柄类和值类的比较” （第 7-2 页）。

dlnode 类概要

本节说明 **dlnode** 类的实现。

示例代码	讨论
`classdef dlnode < handle`	“dlnode 类设计” （第 3-20 页） “为什么要对链表使用句柄类？” （第 3-22 页） “句柄类和值类的比较” （第 7-2 页）

示例代码	讨论
properties Data end	“dlnode 类设计” （第 3-20 页）

properties (SetAccess = private)
  Next = dlnode.empty
  Prev = dlnode.empty
end
	“属性特性” （第 8-7 页）：**SetAccess**.
	
	将这些属性初始化为 **empty dlnode** 对象。
	
	有关属性的一般信息，请参阅“属性语法” （第 8-4 页）

methods	有关方法的一般信息，请参阅“类设计中的方法” （第 9-2 页）

function node = dlnode(Data)
  if (nargin > 0)
  node.Data = Data;
  end
end
	创建单个节点（未连接）只需要数据。
	
	有关构造函数的一般信息，请参阅“构造函数的指导原则” （第 9-13 页）

function insertAfter(newNode, nodeBefore)
  removeNode(newNode);
  newNode.Next = nodeBefore.Next;
  newNode.Prev = nodeBefore;
  if ~isempty(nodeBefore.Next)
  nodeBefore.Next.Prev = newNode;
  end
  nodeBefore.Next = newNode;
end
	将节点插入到双链表中指定节点之后，或者链接两个指定节点（如果还没有列表）。为 **Next** 和 **Prev** 属性正确赋值。
	
	“插入节点” （第 3-26 页）

function insertBefore(newNode, nodeAfter)
  removeNode(newNode);
  newNode.Next = nodeAfter;
  newNode.Prev = nodeAfter.Prev;
  if ~isempty(nodeAfter.Prev)
  nodeAfter.Prev.Next = newNode;
  end
  nodeAfter.Prev = newNode;
end
	将节点插入到双链表中指定节点之前，或者链接两个指定节点（如果还没有列表）。此方法为 **Next** 和 **Prev** 属性正确赋值。
	
	请参阅“插入节点” （第 3-26 页）

function removeNode(node)
  if ~isscalar(node)
  error('Nodes must be scalar')
  end
  prevNode = node.Prev;
  nextNode = node.Next;
  if ~isempty(prevNode)
  prevNode.Next = nextNode;
  end
  if ~isempty(nextNode)
  nextNode.Prev = prevNode;
  end
  node.Next = = dlnode.empty;
  node.Prev = = dlnode.empty;
end
	删除节点并修复列表，以便其余节点正确连接。**node** 参数必须为标量。
	
	如果不存在对节点的引用，MATLAB 会将其删除。
	
	“删除节点” （第 3-27 页）

示例代码	讨论

`function clearList(node)`
`  prev = node.Prev;`
`  next = node.Next;`
`  removeNode(node)`
`  while ~isempty(next)`
`  node = next;`
`  next = node.Next;`
`  removeNode(node);`
`  end`
`  while ~isempty(prev)`
`  node = prev;`
`  prev = node.Prev;`
`  removeNode(node)`
`  end`
`end`
	避免由于清除列表变量而导致递归调用析构函数。遍历列表以断开每个节点的连接。如果不存在对节点的引用，MATLAB 会在删除节点之前调用类析构函数（请参阅 **delete** 方法）。

`methods (Access = private)`
`  function delete(node)`
`  clearList(node)`
`  end`
	类析构函数方法。MATLAB 调用 **delete** 方法来删除仍连接到列表的节点。

`  end` `end`	私有方法结束和类定义结束。

展开类代码

```
classdef dlnode < handle
  % dlnode A class to represent a doubly-linked node.
  % Link multiple dlnode objects together to create linked lists.
  properties
    Data
  end
  properties(SetAccess = private)
    Next = dlnode.empty
    Prev = dlnode.empty
  end

  methods
    function node = dlnode(Data)
      % Construct a dlnode object
      if nargin > 0
        node.Data = Data;
      end
    end

    function insertAfter(newNode, nodeBefore)
      % Insert newNode after nodeBefore.
      removeNode(newNode);
      newNode.Next = nodeBefore.Next;
      newNode.Prev = nodeBefore;
      if ~isempty(nodeBefore.Next)
        nodeBefore.Next.Prev = newNode;
      end
      nodeBefore.Next = newNode;
    end

    function insertBefore(newNode, nodeAfter)
      % Insert newNode before nodeAfter.
      removeNode(newNode);
      newNode.Next = nodeAfter;
      newNode.Prev = nodeAfter.Prev;
```

```
      if ~isempty(nodeAfter.Prev)
        nodeAfter.Prev.Next = newNode;
      end
      nodeAfter.Prev = newNode;
    end

    function removeNode(node)
      % Remove a node from a linked list.
      if ~isscalar(node)
        error('Input must be scalar')
      end
      prevNode = node.Prev;
      nextNode = node.Next;
      if ~isempty(prevNode)
        prevNode.Next = nextNode;
      end
      if ~isempty(nextNode)
        nextNode.Prev = prevNode;
      end
      node.Next = dlnode.empty;
      node.Prev = dlnode.empty;
    end

    function clearList(node)
      % Clear the list before
      % clearing list variable
      prev = node.Prev;
      next = node.Next;
      removeNode(node)
      while ~isempty(next)
        node = next;
        next = node.Next;
        removeNode(node);
      end
      while ~isempty(prev)
        node = prev;
        prev = node.Prev;
        removeNode(node)
      end
    end
  end

  methods (Access = private)
    function delete(node)
      clearList(node)
    end
  end
end
```

类属性

只有 **dlnode** 类方法可以设置 **Next** 和 **Prev** 属性，因为这些属性具有私有 set 访问权限 (**SetAccess = private**)。使用私有 set 访问权限可以防止客户端代码使用这些属性执行任何不正确的操作。**dlnode** 类方法可对这些节点执行允许的所有操作。

Data 属性具有公共 set 和 get 访问权限，允许您根据需要查询和修改 **Data** 的值。

以下代码说明 **dlnode** 类如何定义这些属性：

```
properties
  Data
end
properties(SetAccess = private)
  Next = dlnode.empty;
  Prev = dlnode.empty;
end
```

构造节点对象

要创建节点对象，请将节点数据指定为构造函数的参数：

```
function node = dlnode(Data)
  if nargin > 0
    node.Data = Data;
  end
end
```

插入节点

可以通过两种方法在列表中插入节点 - **insertAfter** 和 **insertBefore**。这些方法执行相似的操作，因此本节仅详细说明 **insertAfter**。

```
function insertAfter(newNode, nodeBefore)
  removeNode(newNode);
  newNode.Next = nodeBefore.Next;
  newNode.Prev = nodeBefore;
  if ~isempty(nodeBefore.Next)
    nodeBefore.Next.Prev = newNode;
  end
  nodeBefore.Next = newNode;
end
```

insertAfter 的工作原理

首先，**insertAfter** 调用 **removeNode** 方法，以确保新节点没有连接到任何其他节点。然后，**insertAfter** 将 **newNode Next** 和 **Prev** 属性赋给列表中 **newNode** 位置前后的节点的句柄。

例如，假设您想在包含 **n1—n2—n3** 的列表中的现有节点 **n1** 之后插入新节点 **nnew**。

首先，创建 **nnew**：

```
nnew = dlnode(rand(3));
```

接下来，调用 **insertAfter** 以将 **nnew** 插入到列表中 **n1** 的后面：

```
nnew.insertAfter(n1)
```

insertAfter 方法执行以下步骤，将 **nnew** 插入列表中的 **n1** 和 **n2** 之间：

- 将 **nnew.Next** 设置为 **n1.Next**（**n1.Next** 是 **n2**）：

  ```
  nnew.Next = n1.Next;
  ```

- 将 **nnew.Prev** 设置为 **n1**

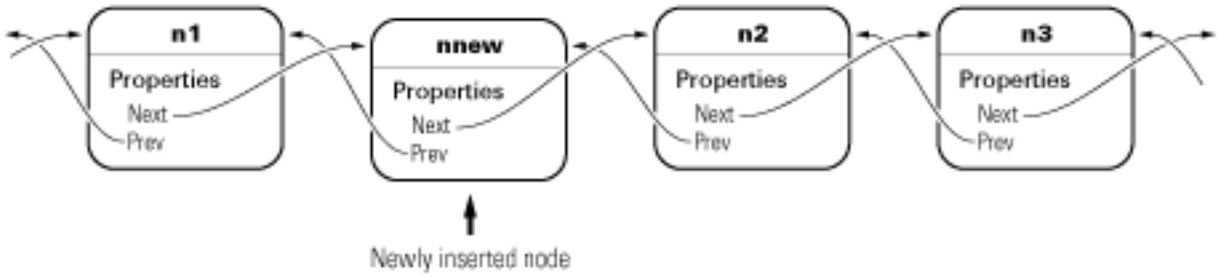
  ```
  nnew.Prev = n1;
  ```

- 如果 **n1.Next** 不为空，则 **n1.Next** 仍然是 **n2**，因此 **n1.Next.Prev** 是 **n2.Prev**，设置为 **nnew**

```
n1.Next.Prev = nnew;
```

- **n1.Next** 现在设置为 **nnew**

```
n1.Next = nnew;
```



删除节点

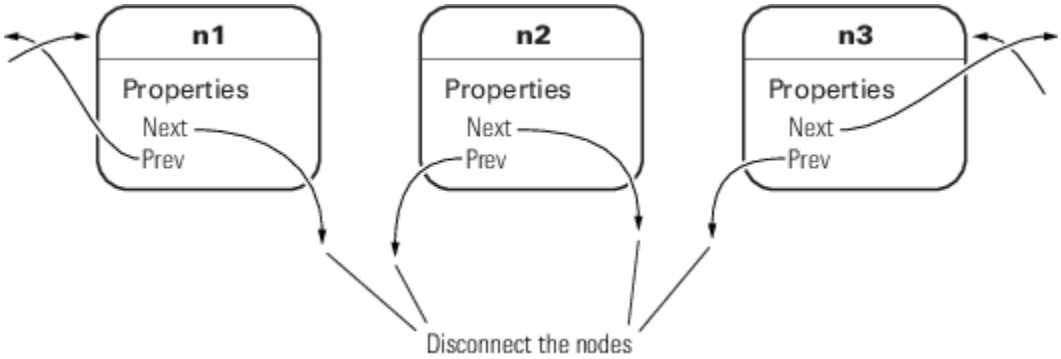
removeNode 方法从列表中删除节点，并重新连接其余节点。**insertBefore** 和 **insertAfter** 方法在尝试将要插入的节点连接到链表之前，始终会对该节点调用 **removeNode**。

调用 **removeNode** 可确保在将该节点赋给 **Next** 或 **Prev** 属性之前处于已知状态：

```
function removeNode(node)
  if ~isscalar(node)
    error('Input must be scalar')
  end
  prevNode = node.Prev;
  nextNode = node.Next;
  if ~isempty(prevNode)
    prevNode.Next = nextNode;
  end
  if ~isempty(nextNode)
    nextNode.Prev = prevNode;
  end
  node.Next = dlnode.empty;
  node.Prev = dlnode.empty;
end
```

例如，假设您从三节点列表 (**n1–n2–n3**) 中删除 **n2**：

```
n2.removeNode;
```



removeNode 从列表中删除 **n2**，并通过以下步骤重新连接其余节点：

```
n1 = n2.Prev;
n3 = n2.Next;
if n1 exists, then
  n1.Next = n3;
if n3 exists, then
  n3.Prev = n1
```

该列表会重新联接，因为 **n1** 连接到 **n3** 且 **n3** 连接到 **n1**。最后一步是确保 **n2.Next** 和 **n2.Prev** 均为空（即 **n2** 处于未连接状态）：

```
n2.Next = dlnode.empty;
n2.Prev = dlnode.empty;
```

从列表中删除节点

假设您创建一个包含 10 个节点的列表，并保存列表头部的句柄：

```
head = dlnode(1);
for i = 10:-1:2
  new = dlnode(i);
  insertAfter(new,head);
end
```

现在删除第三个节点（**Data** 属性的赋值为 **3**）：

```
removeNode(head.Next.Next)
```

现在，列表中第三个节点的数据值为 **4**：

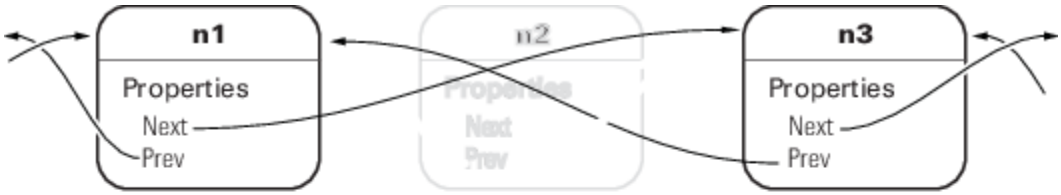
```
head.Next.Next

ans =

  dlnode with properties:

    Data: 4
    Next: [1x1 dlnode]
    Prev: [1x1 dlnode]
```

前一个节点的 **Data** 值为 **2**：

```
head.Next

ans =

  dlnode with properties:

    Data: 2
    Next: [1x1 dlnode]
    Prev: [1x1 dlnode]
```

删除节点

要删除节点，请对该节点调用 **removeNode** 方法。**removeNode** 方法会断开该处节点连接并重新连接列表，然后才允许 MATLAB 销毁删除的节点。当其他节点对某节点的引用被删除后，MATLAB 将销毁该节点，列表将重新连接。



>> removeNode(n2)

>> clear(n2)

MATLAB calls delete(n2)

删除列表

当您创建链表并分配包含某些内容（例如列表头或尾）的变量时，清除该变量会导致析构函数在整个列表中递归。如果列表足够大，清除列表变量可能会导致 MATLAB 超出其递归限制。

clearList 方法可遍历列表和断开每个节点的连接，从而避免递归并提高删除大型列表的性能。**clearList** 接受列表中任何节点的句柄，并删除其余节点。

```
function clearList(node)
  if ~isscalar(node)
    error('Input must be scalar')
  end
  prev = node.Prev;
  next = node.Next;
  removeNode(node)
  while ~isempty(next)
    node = next;
    next = node.Next;
    removeNode(node);
  end
  while ~isempty(prev)
    node = prev;
    prev = node.Prev;
    removeNode(node)
  end
end
```

例如，假设您创建一个包含多个节点的列表：

```
head = dlnode(1);
for k = 100000:-1:2
  nextNode = dlnode(k);
  insertAfter(nextNode,head)
end
```

变量 **head** 包含位于列表头部节点的句柄：

```
head
```

```
head =

  dlnode with properties:

    Data: 1
    Next: [1x1 dlnode]
    Prev: []
```

```
head.Next
```

```
ans =

  dlnode with properties:

    Data: 2
    Next: [1x1 dlnode]
    Prev: [1x1 dlnode]
```

您可以调用 **clearList** 来删除整个列表：

```
clearList(head)
```

只有那些存在显式引用的节点不会被 MATLAB 删除。在本例中，这些引用指 **head** 和 **nextNode**：

```
head
```

```
head =

  dlnode with properties:

    Data: 1
    Next: []
    Prev: []
```

```
nextNode
```

```
nextNode =

  dlnode with properties:

    Data: 2
    Next: []
    Prev: []
```

您可以通过清除变量来删除这些节点：

```
clear head nextNode
```

delete 方法

delete 方法直接调用 **clearList** 方法：

```
methods (Access = private)
  function delete(node)
    clearList(node)
  end
end
```

delete 方法具有私有访问权限，可防止用户在打算删除单个节点时调用 **delete**。当列表被销毁时，MATLAB 会隐式调用 **delete**。

要从列表中删除单个节点，请使用 **removeNode** 方法。

特化 dlnode 类

dlnode 类可实现双链表，基于该类可方便地创建更多专用链表类型。例如，假设您要创建一个列表，其中每个节点都有一个名称。

您不需要将用于实现 **dlnode** 类的代码复制一遍再进行扩展，而是可以直接从 **dlnode** 中派生一个新类（即子类 **dlnode**）。您可以创建具有 **dlnode** 的所有特性的类，还可以定义它自己的附加特性。由于 **dlnode** 是句柄类，因此，此新类也是句柄类。

NamedNode 类定义

要使用该类，请创建一个名为 **@NamedNode** 的文件夹，并将 **NamedNode.m** 保存到该文件夹中。**@NamedNode** 的父文件夹必须位于 MATLAB 路径上。或者，将 **NamedNode.m** 保存到路径文件夹。

以下类定义显示如何从 **dlnode** 类中派生 **NamedNode** 类：

```
classdef NamedNode < dlnode
  properties
    Name = ''
  end
  methods
    function n = NamedNode (name,data)
      if nargin == 0
        name = '';
        data = [];
      end
      n = n@dlnode(data);
      n.Name = name;
    end
  end
end
```

NamedNode 类添加 **Name** 属性来存储节点名称。

构造函数为 **dlnode** 类调用类构造函数，然后为 **Name** 属性赋值。

使用 NamedNode 创建双链表

使用 **NamedNode** 类与使用 **dlnode** 类相似，不同之处在于您要为每个节点对象指定一个名称。例如：

```
n(1) = NamedNode('First Node',100);
n(2) = NamedNode('Second Node',200);
n(3) = NamedNode('Third Node',300);
```

现在，使用从 **dlnode** 继承的插入方法来构建列表：

```
n(2).insertAfter(n(1))
n(3).insertAfter(n(2))
```

查询单个节点的属性时，会显示其名称和数据：

```
n(1).Next

ans =

  NamedNode with properties:

    Name: 'Second Node'
    Data: 200
    Next: [1x1 NamedNode]
    Prev: [1x1 NamedNode]

n(1).Next.Next

ans =

  NamedNode with properties:

    Name: 'Third Node'
    Data: 300
    Next: []
    Prev: [1x1 NamedNode]

n(3).Prev.Prev

ans =

  NamedNode with properties:

    Name: 'First Node'
    Data: 100
    Next: [1x1 NamedNode]
    Prev: []
```

另请参阅

详细信息

- “The Handle Superclass”

静态数据

静态数据

本节内容
“什么是静态数据” （第 4-2 页）
“静态变量” （第 4-2 页）
“静态数据对象” （第 4-3 页）
“常量数据” （第 4-4 页）

什么是静态数据

静态数据是指类的所有对象共享的数据，您可以在创建类实例后修改这些数据。使用静态数据可定义类实例使用的计数器，或在类的所有对象之间共享的其他数据。与实例数据不同，静态数据不会因对象而异。MATLAB 提供了几种定义静态数据的方式来满足您的要求。

静态变量

类可以使用持久变量来存储静态数据。您可以定义一个静态方法或局部函数，以在其中创建持久变量。该方法或函数提供对此变量的访问。当您要存储一个或两个变量时，请使用这种编程方式。

保存定义持久变量的类对象不会保存与该类相关联的静态数据。要将静态数据保存在对象中，或者定义更广泛的数据，请使用静态数据对象编程方式 “静态数据对象” （第 4-3 页）

实现

StoreData 类定义一个静态方法，该方法声明持久变量 **Var**。通过 **setgetVar** 方法，可以对持久变量中的数据进行 set 和 get 访问。由于 **setgetVar** 方法具有公共访问权限，因此您可以以全局方式对存储在持久变量中的数据进行 set 和 get 访问。通过设置方法的 **Access** 特性来控制对方法的访问权限。

```
classdef StoreData
  methods (Static)
    function out = setgetVar(data)
      persistent Var;
      if nargin
        Var = data;
      end
      out = Var;
    end
  end
end
```

通过带输入参数调用 **setgetVar** 来设置变量值。该方法将输入值赋给持久变量：

```
StoreData.setgetVar(10);
```

通过不带输入参数的方式调用 **setgetVar** 来获取变量值：

```
a = StoreData.setgetVar
```

```
a =    10
```

通过对 **StoreData** 类调用 **clear** 清除持久变量：

```
clear StoreData
a = StoreData.setgetVar
```

```
a =

    []
```

将 setgetVar 等方法添加到您需要静态属性行为的任何类中。

静态数据对象

要存储更广泛的数据，请定义具有公共属性的句柄类。将类的对象赋给使用静态数据的类的常量属性。如果您要进行以下操作，这种编程方式非常有用：

- 添加更多修改数据的属性或方法。
- 保存数据类的对象并重新加载静态数据。

实现

SharedData 类是句柄类，它允许您从多个句柄变量引用相同的对象数据：

```
classdef SharedData < handle
  properties
    Data1
    Data2
  end
end
```

UseData 类是使用存储在 SharedData 类中的数据的类的桩件。UseData 类将 SharedData 对象的句柄存储在常量属性中。

```
classdef UseData
  properties (Constant)
    Data = SharedData
  end
  % Class code here
end
```

Data 属性包含 SharedData 对象的句柄。加载 UseData 类时，MATLAB 会构造 SharedData 对象。所有随后创建的 UseData 类实例都引用同一个 SharedData 对象。

要初始化 SharedData 对象属性，请通过引用常量属性加载 UseData 类。

```
h = UseData.Data
```

```
h =

  SharedData with properties:

    Data1: []
    Data2: []
```

使用 SharedData 对象的句柄将数据赋给属性值：

```
h.Data1 = 'MyData1';
h.Data2 = 'MyData2';
```

UseData 类的每个实例都引用同一个句柄对象：

```
a1 = UseData;
a2 = UseData;
```

使用对象变量引用数据：

```
a1.Data.Data1
```

```
ans =

MyData1
```

将一个新值赋给 **SharedData** 对象中的属性：

```
a1.Data.Data1 = rand(3);
```

UseData 类的所有新对象和现有对象共享同一个 **SharedData** 对象。**a2** 现在包含 **rand(3)** 数据，这些数据在之前的步骤中赋给了 **a1**：

```
a2.Data.Data1
```

```
ans =

    0.8147    0.9134    0.2785
    0.9058    0.6324    0.5469
    0.1270    0.0975    0.9575
```

要重新初始化常量属性，请清除 **UseData** 类的所有实例，然后清除该类：

```
clear a1 a2
clear UseData
```

常量数据

要存储不变的常量值，请将数据赋给常量属性。类的所有实例共享该属性的相同值。通过设置属性的 **Access** 特性，控制对常量属性的访问权限。

更改常量属性值的唯一方法是更改类定义。使用常量属性，如 Java® 中的公共最终静态字段。

另请参阅

persistent | **clear**

相关示例

- “定义具有常量值的类属性” (第 15-2 页)
- “静态方法” (第 9-19 页)

详细信息

- “方法特性” (第 9-4 页)
- “属性特性” (第 8-7 页)
- “静态属性” (第 5-15 页)

类定义 - 语法参考

类组件

本节内容

类构建块

MATLAB 将类定义代码组织成模块化代码块，以关键字分隔。所有关键字都有相关联的 **end** 语句：

- **classdef...end** - 所有类组件的定义
- **properties...end** - 属性名称声明、属性特性设定、默认值赋值
- **methods...end** - 方法签名、方法属性和函数代码的声明
- **events...end** - 事件名称和属性的声明
- **enumeration...end** - 枚举类的枚举成员和枚举值的声明。

properties、**methods**、**events** 和 **enumeration** 只是 **classdef** 代码块内的关键字。

类定义代码块

classdef 代码块包含文件中的类定义，以 **classdef** 关键字开头，以 **end** 关键字结尾。

```
classdef (ClassAttributes) ClassName < SuperClass
  ...
end
```

例如，以下 **classdef** 定义名为 **MyClass** 的类，该类是 **handle** 类的子类。此类也定义为密封类，因此无法从此类继承。

```
classdef (Sealed) MyClass < handle
  ...
end
```

有关更多语法信息，请参阅 **classdef**。

属性代码块

properties 代码块包含属性定义，包括可选的初始值。对每组唯一属性设定使用一个单独的代码块。每个属性代码块以 **properties** 关键字开始，以 **end** 关键字结束。

```
properties (PropertyAttributes)
  PropertyName size class {validators} = DefaultValue
end
```

例如，此类使用默认值定义类型为 **double** 的私有属性 **Prop1**。

```
classdef MyClass
  properties (SetAccess = private)
    Prop1 double = 12
  end
  ...
end
```

有关详细信息，请参阅“Initialize Property Values”。

方法代码块

methods 代码块包含类方法的函数定义。对每组唯一属性设定使用一个单独的代码块。每个方法代码块以 **methods** 关键字开始，以 **end** 关键字结束。

```
methods (MethodAttributes)
  function obj = MethodName(arg1,...)
     ...
  end
```

例如，此类定义受保护方法 **MyMethod**。

```
classdef MyClass
  methods (Access = protected)
    function obj = myMethod(obj,arg1)
      ...
    end
  end
end
```

有关详细信息，请参阅“方法语法” （第 5-6 页）。

MATLAB 与 C++ 和 Java 等语言的不同之处在于您必须将类的对象显式传递给方法。

使用 **MyClass** 示例，使用类的对象 **obj** 以及函数或圆点语法调用 **MyMethod**：

```
obj = MyClass;
r = MyMethod(obj,arg1);
r = obj.MyMethod(arg1);
```

有关详细信息，请参阅“方法调用” （第 9-9 页）。

事件代码块

events 代码块（每个唯一的属性设定集对应一个）包含该类声明的事件的名称。**events** 代码块以 **events** 关键字开始，以 **end** 关键字结束。

```
classdef ClassName
  events (EventAttributes)
    EventName
  end
  ...
end
```

例如，以下类定义名为 **StateChange** 的事件，**ListenAccess** 设置为 **protected**。

```
classdef EventSource
  events (ListenAccess = protected)
    StateChanged
  end
  ...
end
```

有关详细信息，请参阅“事件”。

特性设定

特性语法

特性可修改类和类组件（属性、方法和事件）的行为。特性使您能够定义有用的行为，而无需编写复杂的代码。例如，您可以通过将属性的 **SetAccess** 特性设置为私有，但将其 **GetAccess** 特性保留为公共来创建只读属性。

```
properties (SetAccess = private)
  ScreenSize = getScreenSize
end
```

所有类定义代码块（**classdef**、**properties**、**methods** 和 **events**）都支持特定特性。所有特性都有默认值。请仅在希望更改默认值的情况下指定特性值。

注意 在任何组件代码块中，都请仅指定一次特定特性值。

特性说明

有关受支持的特性的列表，请参阅：

- “类属性” （第 6-5 页）
- “属性特性” （第 8-7 页）
- “方法特性” （第 9-4 页）
- “Event Attributes”

特性值

指定特性值时，这些值会影响在定义代码块中定义的所有组件。用不同的特性设置定义属性需要多个属性代码块。在以逗号分隔的列表中指定多个属性。

```
properties (SetObservable = true)
  AccountBalance
end

properties (SetAccess = private, Hidden = true)
  SSNumber
  CreditCardNumber
end
```

true/false 属性的更简单语法

对于值为 **true** 或 **false** 的属性，可以使用更简单的语法。属性名称本身意味着 true，在名称中添加逻辑非运算符 (~) 意味着 false。例如，以下两种定义静态方法代码块的方式是等效的。

```
methods (Static)
  ...
end

methods (Static = true)
  ...
end
```

同样，以下三种定义非静态方法代码块的方式也是等效的。所有采用逻辑值的属性的默认值都为 **false**，因此您可以通过省略该属性来获取默认行为。

```
methods
  ...
end

methods (~Static)
  ...
end

methods (Static = false)
  ...
end
```

枚举类

枚举类是一种特化类，用于定义代表单个类型值的固定名称集。枚举类使用 **enumeration** 代码块来包含该类所定义的枚举成员。

枚举代码块以 **enumeration** 关键字开始，以 **end** 关键字结束。

```
classdef ClassName < SuperClass
  enumeration
    EnumerationMember
  end
  ...
end
```

例如，以下类定义两个枚举成员，表示逻辑值 **false** 和 **true**。

```
classdef Boolean < logical
  enumeration
    No  (0)
    Yes (1)
  end
end
```

有关详细信息，请参阅“定义枚举类” （第 14-2 页）。

相关信息

“创建简单类” （第 2-2 页）

“包含类定义的文件夹” （第 6-8 页）

方法语法

本节内容

本主题描述如何使用 **methods...end** 块在 MATLAB 中定义类方法，并介绍方法参数验证。本主题重点介绍非静态的具体方法，也称为普通方法。对于其他类型的方法，请参阅：

- “类构造函数方法” （第 9-12 页）
- “句柄类析构函数” （第 7-8 页）
- “静态方法” （第 9-19 页）
- “属性 set 方法” （第 8-27 页）
- “抽象类和类成员” （第 12-16 页）

方法定义块

methods 和 **end** 关键字定义一个或多个具有相同特性设置的类方法。方法本身是使用 MATLAB 函数模块定义的。定义普通方法块的语法是：

```
methods (attributes)
  function method1(obj,arg1,...)
    ...
  end
  function method2(obj,arg1,...)
    ...
  end
  ...
end
```

除了静态方法之外，必须将类的对象显式传递给 MATLAB 方法。

例如，此类定义一个公共属性和两个公共方法。每个方法都接受两个输入参数：对象本身和用户提供的参数 **inputArg**。这些方法计算类属性 **Property1** 的值和输入参数的乘积与商。

```
classdef methodDemo
  properties
    Property1
  end
  methods
    function prod = propMultiply(obj,inputArg)
      prod = obj.Property1*inputArg;
    end
    function quotient = propDivide(obj,inputArg)
      quotient = obj.Property1/inputArg;
    end
  end
end
```

您还可以定义多个具有不同特性的方法块。在此示例中，第一个方法是受保护方法，第二个方法是私有方法。有关详细信息，请参阅“方法特性” （第 9-4 页）。

```
classdef attributeDemo
  methods (Access = protected)
    function out = method1(obj,inputArg)
      ...
    end
  end
  methods (Access = private)
    function out = method2(obj,inputArg)
      ...
    end
  end
end
```

方法参数验证

您可以为方法输入和输出参数定义限制。要验证方法参数，请像对待函数一样，将 **arguments** 块添加到方法。有关详细信息，请参阅 **arguments**。

输入参数验证

输入参数验证使您能够限制方法输入参数的大小、类和其他特征。输入参数验证的语法是：

```
arguments
  argName1 (dimensions) class {validators} = defaultValue
  ...
end
```

- **(dimensions)** - 输入大小，指定为包含两个或多个数值的以逗号分隔的列表，如 **(1,2)**。
- **class** - 类，指定为类名称，如 **double** 或用户定义的类的名称。
- **{validators}** - 以逗号分隔的验证函数（如 **mustBeNumeric** 和 **mustBeScalarOrEmpty**）列表，用花括号括起来。有关验证函数的列表，请参阅 “参数验证函数”。
- **defaultValue** - 默认值必须符合指定的大小、类型和验证规则。

有关输入参数验证语法中元素的完整描述，请参阅 **arguments**。

输入参数验证对于具有公共访问权限的方法非常有用。对方法的调用方所允许的参数值类型进行限制可以保证在执行方法体时不出错。例如，**Rectangle** 类表示坐标平面中的一个矩形，属性指定其位置（**X** 和 **Y**）及其宽度和高度。

```
classdef Rectangle
  properties
    X (1,1) double {mustBeReal} = 0
    Y (1,1) double {mustBeReal} = 0
    Width (1,1) double {mustBeReal} = 0
    Height (1,1) double {mustBeReal} = 0
  end

  methods
    function R = enlarge(R,x,y)
      arguments (Input)
        R (1,1) Rectangle
        x (1,1) {mustBeNonnegative}
        y (1,1) {mustBeNonnegative}
      end
      arguments (Output)
```

```
            R (1,1) Rectangle
        end
        R.Width = R.Width + x;
        R.Height = R.Height + y;
    end
  end
end
```

enlarge 方法通过将用户输入 **x** 和 **y** 分别加到 **Width** 和 **Height** 中来增加矩形的高度和宽度。由于该方法旨在放大一个矩形，因此验证将输入参数限制为标量值，大小限制为 **(1,1)**，非负数值限制为 **mustBeNonnegative**。

实例化该类，调用 **enlarge**，将 **5** 和 **-1** 作为 **enlarge** 的输入。参数验证返回错误。

```
rect1 = Rectangle;
rect1.enlarge(5,-1))
```

```
Error using Rectangle/enlarge
 rect1.enlarge(5,-1)
               ↑
Invalid argument at position 2. Value must be nonnegative.
```

提示 输入参数块的 **(Input)** 特性是可选的，但当在一个方法中同时定义输入和输出参数块时，推荐使用特性以提高可读性。MATLAB 将没有任何特性的参数块解释为输入参数块。

输出参数验证

输出参数验证的语法与输入验证的语法相同，不同之处在于必须指定 **(Output)** 作为参数块的特性，并且无法设置默认值。

```
arguments (Output)
  argName1 (dimensions) class {validators}
  ...
end
```

有关输出参数验证语法的完整描述，请参阅 **arguments**。

输出参数验证使类编写者能够记录方法返回什么类型的输出，并作为保障措施来防止将来对可能更改输出类型的代码进行更改。例如，**Rectangle** 的 **enlarge** 方法使用输出参数验证来确保对象方法返回 **Rectangle** 的标量实例。

```
arguments (Output)
  R (1,1) Rectangle
end
```

例如，如果 **enlarge** 后来修改为仅返回 **Rectangle** 的维度，而不是对象本身，则输出验证有助于捕获此潜在错误。

方法验证的特殊考虑事项

类方法的参数验证工作方式很像函数，但参数验证的某些方面是方法所独有的。

- 如果 **classdef** 文件包含在单独文件中定义的方法的方法原型，则您要为这些方法定义的任何 **arguments** 块都必须在单独文件中定义。有关在单独的文件中定义方法的详细信息，请参阅“在单独文件中定义方法” （第 9-6 页）。

- 子类方法不继承参数验证。要在覆盖超类方法的子类方法中保留参数验证，请在子类方法中重复超类方法中的参数验证。
- 抽象方法不支持参数验证，因为它们无法定义 **arguments** 块。有关详细信息，请参阅“抽象类和类成员” （第 12-16 页）。
- 句柄类析构函数方法无法使用参数验证。在句柄类中，当句柄对象变得不可达或通过调用 **delete** 被显式删除时，由 MATLAB 会调用名为 **delete** 的析构函数方法。一个析构函数只能有一个输入，没有输出，也没有参数验证。MATLAB 将以任何其他方式定义的方法视为普通方法，不会将其用作析构函数。有关详细信息，请参阅“句柄类析构函数” （第 7-8 页）。

另请参阅

arguments

相关示例

- “函数参数验证”

对子类对象调用超类方法

本节内容

超类与子类的关系

子类可以覆盖超类方法，从而支持由子类定义更大程度的特化。由于有子类对象本身是超类对象这种关系，在执行自定义子类代码之前调用方法的超类版本通常很有用。

如何调用超类方法

如果子类方法与超类方法同名，则子类方法可以调用超类方法。在子类中，用 @ 符号引用方法名称和超类名。

以下是调用由 **MySuperClass** 定义的 **superMethod** 的语法：

```
superMethod@MySuperClass(obj,superMethodArguments)
```

例如，子类可以调用超类 **disp** 方法来实现对象的超类部分的显示。然后，子类添加代码来显示对象的子类部分：

```
classdef MySub < MySuperClass
  methods
    function disp(obj)
      disp@MySuperClass(obj)
        ...
    end
  end
end
```

如何调用超类构造函数

如果您创建子类对象，MATLAB 会调用超类构造函数来初始化子类对象的超类部分。默认情况下，MATLAB 不带参数调用超类构造函数。如果希望用特定参数调用超类构造函数，请从子类构造函数显式调用超类构造函数。在对该对象进行任何其他引用之前，必须先调用超类构造函数。

调用超类构造函数的语法使用 @ 符号：

```
obj = obj@MySuperClass(SuperClassArguments)
```

在此类中，**MySub** 对象由 **MySuperClass** 构造函数初始化。超类构造函数使用指定的参数构造对象的 **MySuperClass** 部分。

```
classdef MySub < MySuperClass
  methods
    function obj = MySub(arg1,arg2,...)
      obj = obj@MySuperClass(SuperClassArguments);
        ...
    end
```

```
  end
end
```

有关详细信息，请参阅“子类构造函数” （第 9-15 页）。

另请参阅

相关示例

- “修改继承的方法” （第 12-7 页）

MATLAB 和其他面向对象语言的比较

本节内容
“与 C++ 和 Java 代码的一些差异” （第 5-12 页）
“对象修改” （第 5-13 页）
“静态属性” （第 5-15 页）
“常见的面向对象编程方式” （第 5-16 页）

与 C++ 和 Java 代码的一些差异

在某些重要的方面，MATLAB 编程语言与 C++ 或 Java 等其他面向对象的语言有所不同。

公共属性

与 C++ 或 Java 语言中的字段不同，您可以使用 MATLAB 属性定义独立于数据存储实现的公共接口。您可以提供对属性的公共访问，因为您可以定义 set 和 get 访问方法，这些方法会在分配或查询属性值时自动执行。例如，以下语句：

```
myobj.Material = 'plastic';
```

将 **char** 向量 **plastic** 赋给 **myobj** 的 **Material** 属性。在进行实际赋值之前，**myobj** 执行名为 **set.Material** 的方法（假设 **myobj** 的类定义了该方法），该方法可以执行任何必要的操作。有关属性访问方法的详细信息，请参阅“属性 set 方法” （第 8-27 页）。

您还可以通过设置特性来控制对属性的访问，包括公共、受保护和私有三种访问权限。有关属性特性的完整列表，请参阅“属性特性” （第 8-7 页）。

没有隐式参数

在某些语言中，方法的对象参数始终是隐式的。在 MATLAB 中，方法所作用的对象参数是显式的。

调度

在 MATLAB 类中，方法调度不像在 C++ 和 Java 代码中那样基于方法签名。当参数列表包含同等优先级的对象时，MATLAB 使用最左边的对象选择要调用的方法。

但是，如果某个参数的类优先于其他参数的类，MATLAB 将调度优先参数的方法，而不管它位于参数列表中的什么位置。

有关详细信息，请参阅 “Class Precedence”。

调用超类方法

- 在 C++ 中，您可以使用作用域运算符调用超类方法：**superclass::method**
- 在 Java 代码中，您使用的是：**superclass.method**

等效的 MATLAB 操作是 **method@superclass**。

其他差异

在 MATLAB 类中，没有 C++ 模板或 Java 泛型的等效项。但是，MATLAB 不强调类型，因此您可以编写能够处理不同类型数据的函数和类。

MATLAB 类不支持对同一函数名称使用不同的签名来重载函数。

对象修改

MATLAB 类可以定义公共属性，要修改这些属性，您可以在类的给定实例中为这些属性显式赋值。但是，只有从 **handle** 类派生的类才会表现出引用行为。如果在值类（即并非从 **handle** 派生的类）的实例中修改属性值，则只会在修改所处的上下文中更改值。

后续章节将更详细地说明此行为。

传递给函数的对象

MATLAB 按值传递所有变量。当您将对象传递给函数时，MATLAB 会将值从调用方复制到被调用函数的参数变量中。

但是，MATLAB 支持两种在复制时有不同行为的类：

- 句柄类 - 句柄类实例变量引用一个对象。句柄类实例变量的副本引用与原始变量相同的对象。如果函数修改作为输入参数传递的句柄对象，则此修改会影响这一同时由原始句柄和复制句柄引用的对象。
- 值类 - 值类实例中的属性数据独立于该实例副本中的属性数据（尽管值类属性可能包含句柄）。函数可以修改作为输入参数传递的值对象，但这种修改不会影响原始对象。

有关这两种类的行为和用法的详细信息，请参阅“句柄类和值类的比较” （第 7-2 页）。

传递值对象

当您将值对象传递给函数时，函数会创建参数变量的本地副本。该函数只能修改副本。如果要修改原始对象，请返回修改后的对象，并将其赋给原始变量名称。例如，假设有值类 **SimpleClass**：

```
classdef SimpleClass
  properties
    Color
  end
  methods
    function obj = SimpleClass(c)
      if nargin > 0
        obj.Color = c;
      end
    end
  end
end
```

创建 **SimpleClass** 的一个实例，将值 **red** 赋给其 **Color** 属性：

```
obj = SimpleClass('red');
```

将对象传递给函数 **g**，该函数将 **blue** 赋给 **Color** 属性：

```
function y = g(x)
  x.Color = 'blue';
  y = x;
end
```

```
y = g(obj);
```

函数 **g** 修改输入对象的副本并返回该副本，但不更改原始对象。

```
y.Color

ans =

   blue

obj.Color

ans =

    red
```

如果函数 **g** 未返回值，则对象 **Color** 属性的修改只会发生在函数工作区内的 **obj** 的副本上。当函数执行结束时，此副本将超出作用域。

覆盖原始变量实际上会将其替换为新对象：

```
obj = g(obj);
```

传递句柄对象

当您传递函数的句柄时，函数会创建句柄变量的副本，就像传递值对象一样。但是，由于句柄对象的副本引用与原始句柄相同的对象，因此函数可以修改对象，而不必返回修改后的对象。

例如，假设您修改 **SimpleClass** 类定义，使其成为 **handle** 类的派生类：

```
classdef SimpleHandleClass < handle
  properties
    Color
  end
  methods
    function obj = SimpleHandleClass(c)
      if nargin > 0
        obj.Color = c;
      end
    end
  end
end
```

创建 **SimpleHandleClass** 的一个实例，将值 **red** 赋给其 **Color** 属性：

```
obj = SimpleHandleClass('red');
```

将对象传递给函数 **g**，该函数将 **blue** 赋给 **Color** 属性：

```
y = g(obj);
```

函数 **g** 会设置这一由返回的句柄和原始句柄同时引用的对象的 **Color** 属性：

```
y.Color

ans =

blue

obj.Color

ans =

blue
```

变量 y 和 obj 引用同一个对象：

```
y.Color = 'yellow';
obj.Color
```

```
ans =

yellow
```

函数 g 修改了输入参数引用的对象 (obj)，并在 y 中返回该对象的句柄。

MATLAB 按值传递句柄

句柄变量是对一个对象的引用。MATLAB 按值传递该引用。

句柄的行为与 C++ 中的引用不同。如果您将对象句柄传递给函数，而该函数将另一对象赋给该句柄变量，则调用方中的变量不受影响。例如，假设您定义函数 g2：

```
function y = g2(x)
  x = SimpleHandleClass('green');
  y = x;
end
```

将句柄对象传递给 g2：

```
obj = SimpleHandleClass('red');
y = g2(obj);
y.Color
```

```
ans =

green
```

```
obj.Color
```

```
ans =

red
```

该函数覆盖作为参数传入的句柄，但不覆盖句柄引用的对象。原始句柄 obj 仍然引用原始对象。

静态属性

在 MATLAB 中，类可以定义常量属性，但不能定义 C++ 等其他语言中所说的“静态”属性。您不能将常量属性更改为类定义中指定的初始值以外的值。

一直以来，MATLAB 规定变量始终优先于函数和类的名称。赋值语句会在没有变量的情况下引入变量。

以下形式的表达式

```
A.B = C
```

引入新变量 A，它是一个 struct，其中包含值为 C 的字段 B。如果 A.B = C 可以表示类 A 的静态属性，则类 A 将优先于变量 A。

此行为与 MATLAB 的早期版本明显不兼容。例如，如果在 MATLAB 路径上引入名为 A 的类，则可能会更改 .m 代码文件中类似 A.B = C 的赋值语句的含义。

在其他语言中，类很少使用静态数据，除非作为类中的私有数据或公共常量。在 MATLAB 中，您可以像在 Java 中使用 **public**、**final** 和 **static** 字段一样使用常量属性。要使用 MATLAB 中某个类的内部数据，请使用私有方法、受保护的方法或该类私有的局部函数来创建持久变量。

避免在 MATLAB 中使用静态数据。如果类有静态数据，在多个应用程序中使用同一个类会导致应用程序之间发生冲突。在其他一些语言中，冲突不是什么大问题。这些语言将应用程序编译成在不同进程中运行的可执行文件。每个进程都有自己的类静态数据副本。MATLAB 经常在同一进程和环境中运行许多不同应用程序，每个类只有一个副本。

有关在 MATLAB 中定义和使用静态数据的方法，请参阅“静态数据” （第 4-2 页）。

常见的面向对象编程方式

下表提供一些章节的链接，这些章节讨论其他面向对象语言常用的面向对象编程方式。

编程方式	在 MATLAB 中如何运用
运算符重载	“运算符重载” （第 17-5 页）
多重继承	“Subclassing Multiple Classes”
子类化	“设计子类构造函数” （第 12-4 页）
析构函数	“句柄类析构函数” （第 7-8 页）
限定数据成员的作用域	“属性特性” （第 8-7 页）
包（限定类的作用域）	“包命名空间” （第 6-13 页）
具名常量	请参阅“定义具有常量值的类属性” （第 15-2 页）和“Named Values”
枚举	“定义枚举类” （第 14-2 页）
静态方法	“静态方法” （第 9-19 页）
静态属性	不支持。请参阅 **persistent** 变量。如需 Java 中 **static** 和 **final** 或 C++ 中 **static** 和 **const** 属性的等效项，请使用 **Constant** 属性。请参阅“定义具有常量值的类属性” （第 15-2 页） 有关可变静态数据，请参阅“静态数据” （第 4-2 页）
构造函数	“类构造函数方法” （第 9-12 页）
复制构造函数	没有直接等效项
引用/引用类	“句柄类和值类的比较” （第 7-2 页）
抽象类/接口	“抽象类和类成员” （第 12-16 页）
垃圾回收	“对象生命周期” （第 7-11 页）
实例属性	“动态属性 - 向实例添加属性” （第 8-35 页）
导入类	“导入类” （第 6-17 页）
事件和侦听程序	“事件和侦听程序概念” （第 11-5 页）

定义和组织类

用户定义的类

本节内容
“什么是类定义” （第 6-2 页）
“类成员的属性” （第 6-2 页）
“类的种类” （第 6-2 页）
“构造对象” （第 6-2 页）
“类的层次结构” （第 6-3 页）
“classdef 语法” （第 6-3 页）
“类代码” （第 6-3 页）

什么是类定义

MATLAB 类定义是一个模板，用于描述该类所有实例共有的所有元素。类成员包括定义该类的属性、方法和事件。

使用代码块定义 MATLAB 类，并使用子代码块界定各个类成员的定义。有关这些代码块的语法信息，请参阅“类组件” （第 5-2 页）。

类成员的属性

属性可以修改类定义代码块中定义的类行为和成员行为。例如，您可以指定方法是静态的或者属性是私有的。以下各节说明这些属性：

- “类属性” （第 6-5 页）
- “方法特性” （第 9-4 页）
- “属性特性” （第 8-7 页）
- “Event Attributes”

类定义可以提供信息，例如继承关系或类成员的名称，而无需实际构造类。请参阅“Class Metadata”。

有关属性语法的详细信息，请参阅“指定属性” （第 6-6 页）。

类的种类

有两个种类的 MATLAB 类 - 句柄类和值类。

- 值类代表独立的值。值对象包含对象数据，并且不与对象的副本共享该数据。MATLAB 数值类型是值类。值对象在传递给函数并经其修改后，必须将修改后的对象返回给调用方。
- 句柄类创建引用对象数据的对象。实例变量的各个副本引用同一个对象。句柄对象在传递给函数并经其修改后，可直接影响调用方工作区中的对象而无需返回对象。

有关详细信息，请参阅“句柄类和值类的比较” （第 7-2 页）。

构造对象

有关类构造函数的信息，请参阅“类构造函数方法” （第 9-12 页）。

有关创建对象数组的信息，请参阅“构造对象数组” （第 10-2 页）。

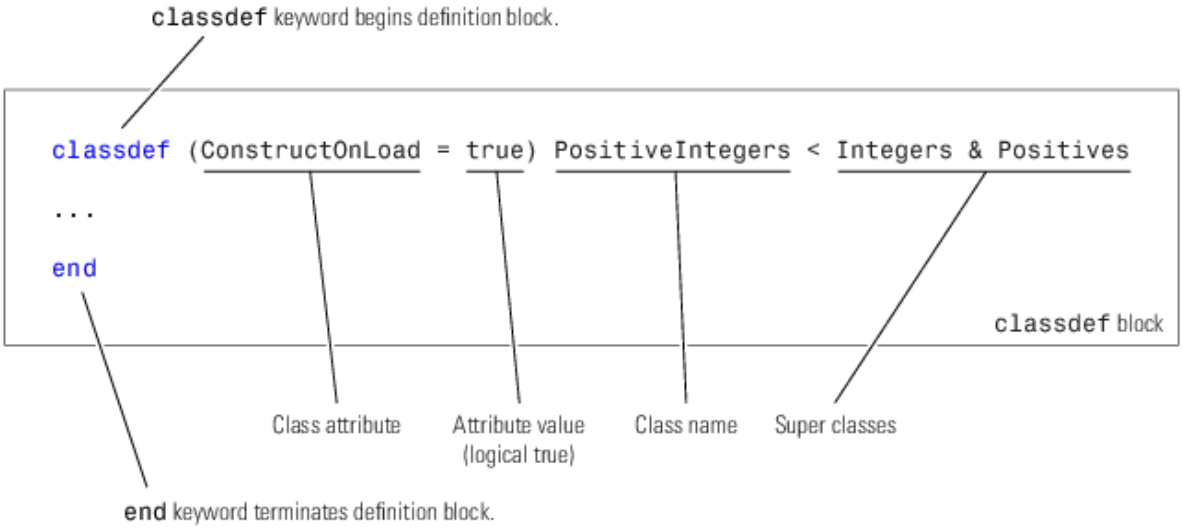
类的层次结构

有关如何定义类的层次结构的详细信息，请参阅“Hierarchies of Classes — Concepts”。

classdef 语法

类定义是由位于开头的 **classdef** 关键字和位于结尾的 **end** 关键字界定的代码块。文件只能包含一个类定义。

下图显示了 **classdef** 代码块的语法。**classdef** 关键字前面只能有注释和空行。



类代码

下面是一个简单的类定义，其中包含一个属性和一个构造函数方法，该方法在提供输入参数时设置属性的值。

```
classdef MyClass
  properties
    Prop
  end
  methods
    function obj = MyClass(val)
      if nargin > 0
        obj.Prop = val;
      end
    end
  end
end
```

要创建 **MyClass** 的对象，请将类定义保存在与该类同名的 **.m** 文件中，并使用必要的参数调用构造函数：

```
d = datestr(now);
o = MyClass(d);
```

使用圆点表示法访问属性值：

```
o.Prop
```

```
ans =

10-Nov-2005 10:38:14
```

构造函数应该支持无参数语法，以便 MATLAB 可以创建默认对象。有关详细信息，请参阅“构造函数不要求输入参数的情况” （第 9-15 页）。

有关类定义的各部分的详细信息，请参阅“类组件” （第 5-2 页）

另请参阅

相关示例

- “创建简单类” （第 2-2 页）
- “开发协同工作的类” （第 3-5 页）
- “用类表示结构化数据” （第 3-12 页）

类属性

本节内容

指定类属性

所有类都支持下表中列出的属性。属性使您能够修改类的行为。属性值应用于 **classdef** 代码块中定义的类。

```
classdef (Attribute1 = value1, Attribute2 = value2,...) ClassName
  ...
end
```

类属性

特性名称	类	描述
Abstract	**logical** （默认值为 **false**）	如果指定为 **true**，则此类为抽象类（无法实例化）。 有关详细信息，请参阅“抽象类和类成员” （第 12-16 页）。
AllowedSubclasses	**meta.class** 对象或 **meta.class** 对象的元胞数组	列出可以从该类派生的子类。采用以下形式将子类指定为 **meta.class** 对象： • 单个 **meta.class** 对象 • **meta.class** 对象元胞数组。空元胞数组 **{}** 与 **Sealed** 类相同（没有子类）。 仅使用 **?ClassName** 语法指定 **meta.class** 对象。 有关详细信息，请参阅“Specify Allowed Subclasses”。
ConstructOnLoad	**logical** （默认值为 **false**）	如果为 **true**，MATLAB 将在从 MAT 文件加载对象时自动调用类构造函数。用此属性定义的类必须有一个不带参数的构造函数。 有关详细信息，请参阅“Initialize Objects When Loading”。
HandleCompatible	**logical** （默认值为 **false**）（对于值类）	如果指定为 **true**，此类可以用作句柄类的超类。根据定义，所有句柄类均为 **HandleCompatible**。有关详细信息，请参阅“Handle Compatible Classes”。
Hidden	**logical** （默认值为 **false**）	如果为 **true**，此类不会出现在 **superclasses** 或 **help** 函数的输出中。
InferiorClasses	**meta.class** 对象或 **meta.class** 对象的元胞数组	使用此属性在类之间建立优先关系。使用 **?** 运算符指定 **meta.class** 对象的元胞数组。 基础类的优先级始终低于用户定义的类，并且不会出现在此列表中。 请参阅“Class Precedence”。
Sealed	**logical** （默认值为 **false**）	如果为 **true**，则无法对此类进行子类化。
框架特性	使用特定框架基类的类具有特定于框架的特性。有关这些特性的信息，请参阅您正在使用的特定基类的文档。	

指定属性

类成员的属性由 **classdef**、**properties**、**methods** 和 **events** 定义代码块指定。具体的属性设置应用于该特定代码块中定义的所有成员。您可以使用多个 **properties**、**methods** 和 **events** 定义代码块，对不同类成员应用不同属性设置。

超类属性值不会被继承

类属性设置不会被继承，因此超类属性值不影响子类。

特性语法

在圆括号中指定类属性值，用逗号分隔每个属性名称/属性值对组。属性列表始终跟随在 **classdef** 或类成员关键字之后，如下所示：

```
classdef (attribute-name = expression, ...) ClassName

  properties (attribute-name = expression, ...)
    ...
  end
  methods (attribute-name = expression, ...)
    ...
  end
  events (attribute-name = expression, ...)
    ...
  end
end
```

特定于类的属性

一些 MATLAB 类定义附加属性，这些属性只能用于定义这些属性的类层次结构。有关这些类支持的附加属性的信息，请参阅您正在使用的类的相关文档。

另请参阅

详细信息

- “Expressions in Attribute Specifications”

包含类定义的文件夹

本节内容

类定义位于路径上

要调用类方法，类定义必须位于 MATLAB 路径上，如下面各节所述。

类和路径文件夹

有两种类型的文件夹可以包含类定义文件。

- 路径文件夹 - 文件夹位于 MATLAB 路径上，文件夹名称不以 @ 字符开头。当您要在一个文件夹中包含多个类和函数时，请使用这种类型的文件夹。一个类的完整定义必须包含在一个文件中。
- 类文件夹 - 文件夹名称以 @ 字符开头，后跟类名。该文件夹不在 MATLAB 路径上，但其父文件夹在该路径上。当您要使用多个文件定义一个类时，请使用此类型的文件夹。

有关 MATLAB 路径的信息，请参阅 **path** 函数。

使用路径文件夹

包含类定义文件的文件夹位于 MATLAB 路径中。因此，放置在路径文件夹中的类定义在优先级方面的行为和任何普通函数一样 - 在 MATLAB 路径上第一次出现的名称优先于随后出现的所有相同名称。

每个类定义文件的名称必须与用 **classdef** 关键字指定的类的名称相匹配。使用路径文件夹就无需为每个类创建单独的类文件夹。但是，一个类的完整定义，包括所有方法，必须包含在单个文件中。

假设您在单个文件夹中定义了三个类：

```
.../path_folder/MyClass1.m
.../path_folder/MyClass2.m
.../path_folder/MyClass3.m
```

要使用这些类，请将 **path_folder** 添加到您的 MATLAB 路径：

```
addpath path_folder
```

使用类文件夹

类文件夹的名称始终以 @ 字符开头，后跟类名称。类文件夹必须包含在路径文件夹中，但类文件夹不在 MATLAB 路径上。将类定义文件放在类文件夹中，该文件夹也可以包含单独的方法文件。类定义文件必须与类文件夹同名（除 @ 字符外）。

.../parent_folder/@MyClass/MyClass.m
.../parent_folder/@MyClass/myMethod1.m
.../parent_folder/@MyClass/myMethod2.m

每个文件夹只能定义一个类。所有文件的扩展名均为 **.m** 或 **.p**。对于 MATLAB 版本 R2018a 及更高版本，独立方法可以是扩展名为 **.mlx** 的实时函数。

当您要使用多个文件定义一个类时，请使用类文件夹。MATLAB 将类文件夹中的任何函数文件视为类的一个方法。函数文件可以是 MATLAB 代码 (**.m**)、实时代码文件格式 (**.mlx**)、MEX 函数（平台相关的扩展名）和 P 代码文件 (**.p**)。

MATLAB 将类文件夹中的任何文件显式标识为该类的方法。这使您能够使用更模块化的方法来编写您的类方法。

每个文件的基本名称必须为有效的 MATLAB 函数名称。有效的函数名称以字母字符开头，并且可以包含字母、数字或下划线。有关详细信息，请参阅“在单独文件中定义方法” （第 9-6 页）。

类文件夹内的私有文件夹中的函数

私有文件夹中的函数只能从 **private** 文件夹紧邻的父级文件夹中定义的函数访问。在类文件夹中的 **private** 文件夹中定义的任何函数只能从类的方法中调用。函数可以访问类的私有成员，但它们本身不是方法。它们不要求将对象作为输入传递，且只能使用函数表示法来调用。当您需要可通过类的多个方法调用的辅助函数时，请使用 **private** 文件夹中的函数。

如果类文件夹包含 **private** 文件夹，则只有类文件夹中定义的类才能访问 **private** 文件夹中定义的函数。子类无法访问超类私有函数。有关私有文件夹的详细信息，请参阅“私有函数”。

如果您需要子类能够访问超类的私有函数，请将这些函数定义为超类的受保护方法。指定这些方法时，将 **Access** 属性设置为 **protected**。

调度私有文件夹中的方法

如果类在类文件夹的 **private** 文件夹中定义了函数，则 MATLAB 在决定调度私有函数还是 **classdef** 文件的方法时，将遵循以下优先级规则：

- 使用圆点表示法 (**obj.methodName**) 时，**private** 文件夹中的函数优先于 **classdef** 文件中定义的方法。
- 使用函数表示法 (**methodName(obj)**) 时，**classdef** 文件中定义的方法优先于 **private** 文件夹中的函数。

私有文件夹中不应出现类定义

您不能将类定义（**classdef** 文件）放入私有文件夹中，因为这样做不符合类或路径文件夹的要求。

类优先级和 MATLAB 路径

存在多个同名的类定义时，MATLAB 路径上的文件位置决定优先级。文件夹中位于 MATLAB 路径前面的类定义始终优先于该路径中后面的任何类，无论这些定义是否包含在类文件夹中。

如果函数位于路径中较早出现的文件夹中，则与路径文件夹中的类同名的函数优先于该类。但是，在类文件夹（@ 文件夹）中定义的类优先于同名的函数，即使该函数是在路径中较早出现的文件夹中定义的也是如此。

例如，假设有包含以下文件夹和文件的路径。

在路径上的顺序	文件夹和文件	文件定义的内容
1	**fldr1/Foo.m**	类 **Foo**
2	**fldr2/Foo.m**	函数 **Foo**
3	**fldr3/@Foo/Foo.m**	类 **Foo**
4	**fldr4/@Foo/bar.m**	方法 **bar**
5	**fldr5/Foo.m**	类 **Foo**

MATLAB 应用以下逻辑来确定调用哪个版本的 **Foo**：

类 **fldr1/Foo.m** 优先于类 **fldr3/@Foo**，因为：

- 在路径中，**fldr1** 位于 **fldr3** 之前，且 **fldr1/Foo.m** 是类。

类 **fldr3/@Foo** 优先于 **fldr2/Foo.m** 函数，因为：

- **fldr3/@Foo** 是类文件夹中的类。
- **fldr2/Foo.m** 不是类。
- 类文件夹中的类优先于函数。

函数 **fldr2/Foo.m** 优先于类 **fldr5/Foo.m**，因为：

- 在路径中，**fldr2** 位于 **fldr5** 类之前。
- **fldr5/Foo.m** 不在类文件夹中。
- 未在类文件夹中定义的类遵循关于函数的路径顺序。

类 **fldr3/@Foo** 优先于 **fldr4/@Foo**，因为：

- 在路径中，**fldr3** 位于 **fldr4** 之前。

如果 **fldr3/@Foo/Foo.m** 包含在 7.6 版之前创建的 MATLAB 类（即，该类不使用 **classdef** 关键字），则 **fldr4/@Foo/bar.m** 成为在 **fldr3/@Foo** 中定义的 **Foo** 类的方法。

早期版本中在类文件夹中定义的类的行为

在 MATLAB 版本 5 至 7 中，类文件夹不会遮蔽那些与之同名、但路径顺序靠后的文件夹中的类文件夹。该类将结合使用所有同名类文件夹中的方法来定义该类。此行为不再受到支持。

为了向后兼容，类文件夹中定义的类始终优先于同名的函数和脚本。此优先级也适用于路径上先于这些类出现的函数和脚本。

更改路径以更新类定义

MATLAB 只能将类的一个定义识别为当前定义。更改您的 MATLAB 路径可以更改某个类的定义文件（请参阅 **path**）。如果不存在旧定义的实例（即不再位于路径中靠前位置的定义），MATLAB 会立即将新文件夹识别为当前定义。但是，如果在更改路径之前已有该类的实例，则 MATLAB 是否使用新文件夹中的定义取决于新类的定义方式。如果新定义是在类文件夹中定义的，则 MATLAB 会立即将新文件夹识别为当前类定义。但是，对于在路径文件夹中定义的类（即，不在类 @ 文件夹中），您必须在 MATLAB 将新文件夹识别为当前类定义之前清除该类。

类文件夹中的类定义

假设您在两个文件夹 **fldA** 和 **fldB** 中定义名为 **Foo** 的类的两个版本。

fldA/@Foo/Foo.m
fldB/@Foo/Foo.m

将文件夹 **fldA** 添加到路径的顶端。

```
addpath fldA
```

创建类 **Foo** 的实例。MATLAB 使用 **fldA/@Foo/Foo.m** 作为类定义。

```
a = Foo;
```

将当前文件夹更改为 **fldB**。

```
cd fldB
```

当前文件夹始终是路径上的第一个文件夹。因此，MATLAB 将 **fldB/@Foo/Foo.m** 视为 **Foo** 类的定义。

```
b = Foo;
```

MATLAB 自动更新现有实例 **a**，以使用 **fldB** 中新的类定义。

路径文件夹中的类定义

假设您在两个文件夹 **fldA** 和 **fldB** 中定义名为 **Foo** 的类的两个版本，但不使用类文件夹。

fldA/Foo.m
fldB/Foo.m

将文件夹 **fldA** 添加到路径的顶端。

```
addpath fldA
```

创建类 **Foo** 的实例。MATLAB 使用 **fldA/Foo.m** 作为类定义。

```
a = Foo;
```

将当前文件夹更改为 **fldB**。

```
cd fldB
```

当前文件夹实际上位于路径的顶端。但是，MATLAB 没有将 **fldB/Foo.m** 识别为 **Foo** 类的定义。MATLAB 继续使用原始的类定义，直到您清除该类。

要使用 **foldB** 中 **Foo** 的定义，请清除 **Foo**。

```
clear Foo
```

MATLAB 自动更新现有对象，以符合 **fldB** 中的类定义。通常，清除实例变量是不必要的。

另请参阅

详细信息

- “包命名空间” （第 6-13 页）
- “Automatic Updates for Modified Classes”
- “实时代码文件格式 (.mlx)”
- “调用 MEX 函数”
- “Using MEX Functions for MATLAB Class Methods”
- “保护源代码的安全考虑事项”

包命名空间

本节内容

包文件夹

包是一种特殊文件夹，可以包含类文件夹、函数和类定义文件及其他包。类和函数的名称的作用域限于包文件夹内。包也是一种命名空间，其中的名称必须唯一。函数和类名在包中必须唯一。您可以使用包来组织类和函数。您还可以使用包在不同包中重用类和函数的名称。

注意 在 MATLAB 7.6 版之前创建的类（即未使用 **classdef** 的类）不支持包。

包文件夹始终以 + 字符开头。例如，

```
+mypack
+mypack/pkfcn.m  % a package function
+mypack/@myClass % class folder in a package
```

顶层包文件夹的父文件夹必须位于 MATLAB 路径上。

列出包的内容

使用 **help** 命令列出包的内容：

```
help event
```

```
Contents of event:

EventData               - event.EVENTDATA    Base class for event data
PropertyEvent             - event.PROPERTYEVENT    Event data for object property events
listener              - event.LISTENER    Listener object
proplistener            - event.PROPLISTENER    Listener object for property events
```

您也可以使用 **what** 命令：

```
what event
```

```
Classes in directory Y:xxx\matlab\toolbox\matlab\lang\+event

EventData     PropertyEvent  listener      proplistener
```

内部包

MathWorks® 将名为 **internal** 的包保留给 MATLAB 内部代码所用的工具函数使用。属于 **internal** 包的函数仅供 MathWorks 使用。不建议使用属于 **internal** 包的函数或类。这些函数和类不能保证在两个版本之间以一致的方式工作。这些函数和类可能会在任何后续版本中从 MATLAB 软件中删除，而不另行通知，也不在产品发行说明中予以记录。

引用包中的包成员

除非导入包，否则对包中的包、函数和类的所有引用都必须使用包名称前缀。（请参阅 “导入类” （第 6-17 页）。）例如，要调用以下包函数：

+mypack/pkfcn.m

请使用以下语法：

```
z = mypack.pkfcn(x,y);
```

定义不使用包前缀。例如，**pkfcn.m** 函数的函数定义行只包括函数名称：

```
function z = pkfcn(x,y)
```

仅使用类名定义包类：

```
classdef myClass
```

但是，使用包前缀来调用它：

```
obj = mypack.myClass(arg1,arg2,...);
```

调用类方法不需要包名称，因为您有类的对象。您可以使用圆点表示法或函数表示法：

```
obj.myMethod(arg)
myMethod(obj,arg)
```

静态方法需要完整的类名，其中包括包名称：

```
mypack.myClass.stMethod(arg)
```

从包外部引用包成员

包中所含函数、类和其他包的作用域限于该包内。要引用任何包成员，请在成员名称前面加上包名称，并用圆点符隔开。例如，以下语句创建 **MyClass** 的一个实例，该实例包含在 **mypack** 包中。

```
obj = mypack.MyClass;
```

访问类成员 - 多种场景

本节向您说明如何从包外部访问各种包成员。假设您有包含以下内容的 **mypack** 包：

```
+mypack
+mypack/myFcn.m
+mypack/@MyFirstClass
+mypack/@MyFirstClass/myFcn.m
+mypack/@MyFirstClass/otherFcn.m
+mypack/@MyFirstClass/MyFirstClass.m
+mypack/@MySecondClass
+mypack/@MySecondClass/MySecondClass.m
+mypack/+mysubpack
+mypack/+mysubpack/myFcn.m
```

调用 **mypack** 中的 **myFcn** 函数：

```
mypack.myFcn(arg)
```

创建 **mypack** 中每个类的实例：

```
obj1 = mypack.MyFirstClass;
obj2 = mypack.MySecondClass(arg);
```

调用包 **mysubpack** 中的 **myFcn** 函数：

```
mypack.mysubpack.myFcn(arg1,arg2);
```

如果 **mypack.MyFirstClass** 有名为 **myFcn** 的方法，请以调用对象方法的方式调用它：

```
obj = mypack.MyFirstClass;
myFcn(obj,arg);
```

如果 **mypack.MyFirstClass** 有名为 **MyProp** 的属性，请使用对象结合圆点表示法对其进行赋值：

```
obj = mypack.MyFirstClass;
obj.MyProp = x;
```

包和 MATLAB 路径

您不能将包文件夹添加到 MATLAB 路径，但是，您必须将包父文件夹添加到 MATLAB 路径。如果包父文件夹不在 MATLAB 路径上，则包成员不可访问，即使包文件夹是当前文件夹也是如此。将包文件夹设为当前文件夹不会将包父文件夹添加到路径中。

包成员的作用域限于该包内。始终使用包名称引用包成员。或者，将包导入调用包成员的函数中，请参阅“导入类” （第 6-17 页）。

包文件夹不会遮蔽路径上位于其后的包文件夹，但类会遮蔽其他类。如果两个或多个包具有相同的名称，MATLAB 会将它们视为一个包。如果不同路径文件夹中冗余命名的包定义了相同的函数名称，则 MATLAB 只能找到其中一个函数。

解析冗余名称

假设包和类同名。例如：

```
fldr_1/+foo
fldr_2/@foo/foo.m
```

调用 **which foo** 将返回可执行类构造函数的路径：

```
>> which foo
fldr_2/@foo/foo.m
```

函数和包可以有相同的名称。但是，包名称本身不是标识符。因此，如果某个冗余名称单独出现，它标识的将是函数。单独执行包名称会返回错误。

包函数与静态方法

在包和类同名的情况下，包函数优先于静态方法。例如，路径文件夹 **fldrA** 包含包函数，路径文件夹 **fldrB** 包含类静态方法：

```
fldrA/+foo/bar.m  % bar is a function in package foo
fldrB/@foo/bar.m  % bar is a static method of class foo
```

对 **which foo.bar** 的调用返回包函数的路径：

```
which foo.bar
```

```
fldrA\+foo\bar.m  % package function
```

如果同一路径文件夹包含同名的包和类文件夹，则包函数类优先于静态方法。

```
fldr/@foo/bar.m  % bar is a static method of class foo
fldr/+foo/bar.m  % bar is a function in package foo
```

对 **which foo.bar** 的调用返回包函数的路径：

```
which foo.bar
```

```
fldr/+foo/bar.m
```

假设路径文件夹 **fldr** 包含定义静态方法 **bar** 的 **classdef** 文件 **foo**，同时还包含包 **+foo**（该包包含包函数 **bar**）。

```
fldr/foo.m      % bar is a static method of class foo
fldr/+foo/bar.m % bar is a function in package foo
```

对 **which foo.bar** 的调用返回包函数的路径：

```
which foo.bar
```

```
fldr/+foo/bar.m
```

另请参阅

详细信息

- “包含类定义的文件夹” （第 6-8 页）
- “Class Precedence”

导入类

本节内容

导入类的语法

将类导入函数中以简化对类成员的访问。例如，假设有一个包含若干类的包，您将在函数中只使用其中一个类或一个静态方法。使用 **import** 命令简化代码。一旦导入了该类，就不需要引用包名称：

```
function myFunc
  import pkg.MyClass
  obj = MyClass(arg,...);              % call MyClass constructor
  obj.Prop = MyClass.staticMethod(arg,...); % call MyClass static method
end
```

使用语法 **pkg.*** 导入包中的所有类：

```
function myFunc
  import pkg.*
  obj1 = MyClass1(arg,...); % call pkg.MyClass1 constructor
  obj2 = MyClass2(arg,...); % call pkg.MyClass2 constructor
  a = pkgFunction();  % call package function named pkgFunction
end
```

导入静态方法

使用 **import** 导入静态方法后，无需使用类名即可调用此方法。用完整的类名（包括任何包）和静态方法名称调用 **import**。

```
function myFunc
  import pkg.MyClass.MyStaticMethod
  MyStaticMethod(arg,...); % call static method
end
```

导入包函数

使用 **import** 导入包函数后，无需使用包名称即可调用这些函数。用包和函数名称调用 **import**。

```
function myFunc
  import pkg.pkgFunction
  pkgFunction(arg,...); % call imported package function
end
```

包函数和类方法名称冲突

避免使用 * 通配符语法导入整个包，否则会将一组未指定的名称导入局部作用域中。例如，假设您有以下文件夹组织：

```
+pkg/timedata.m          % package function
+pkg/@MyClass/MyClass.m  % class definition file
+pkg/@MyClass/timedata.m  % class method
```

导入该包，并对 **MyClass** 的实例调用 **timedata**：

```
import pkg.*
myobj = pkg.MyClass;
timedata(myobj)
```

调用 **timedata** 会找到包函数，而不是类方法，因为 MATLAB 应用 **import** 并首先找到 **pkg.timedata**。如果存在名称冲突且计划导入包，请不要使用包。

清除导入列表

您无法从函数工作区中清除导入列表。要仅清除基础工作区，请使用：

```
clear import
```

另请参阅

import

详细信息

- “包命名空间” (第 6-13 页)

值类或句柄类 - 如何选择

句柄类和值类的比较

本节内容

基本差异

值类构造函数返回一个与其赋值变量相关联的对象。如果对此变量重新赋值，MATLAB 会创建原始对象的独立副本。如果将此变量传递给函数以修改它，函数必须将修改后的对象以输出参数形式返回。有关值类行为的信息，请参阅“避免不必要的数据副本”。

句柄类构造函数返回句柄对象，该对象是对所创建对象的引用。您可以将句柄对象赋给多个变量或将它传递给函数，而不会导致 MATLAB 创建原始对象的副本。函数对作为输入参数传递的句柄对象进行修改后，不必返回该对象。

所有句柄类都派生自抽象的 **handle** 类。

创建值类

默认情况下，MATLAB 类是值类。以下定义创建名为 **MyValueClass** 的值类：

```
classdef MyValueClass
  ...
end
```

创建句柄类

要创建 **handle** 类，请从 **handle** 类派生该类。

```
classdef MyHandleClass < handle
  ...
end
```

MATLAB 内置类的行为

MATLAB 基础类是值类（数值、**logical**、**char**、**cell**、**struct** 以及函数句柄）。例如，如果创建 **int32** 类的对象并复制该对象，结果会是两个独立的对象。更改 **a** 的值不会导致 **b** 的值发生更改。此行为是表示值的类的典型行为。

```
a = int32(7);
b = a;
a = a^4;

b
  7
```

MATLAB 图形对象作为句柄对象实现，因为它们表示视觉元素。例如，创建一个图形线条对象，并将它的句柄复制到另一个变量。这两个变量引用同一个线条对象。

```
x = 1:10; y = sin(x);
l1 = line(x,y);
l2 = l1;
```

使用句柄的任一副本设置线条对象的属性。

```
set(l2,'Color','red')
set(l1,'Color','green')

get(l2,'Color')

ans =

     0     1     0
```

对 l2 句柄调用 **delete** 函数会销毁该线条对象。如果您尝试在 l1 行设置 **Color** 属性，**set** 函数将返回错误。

```
delete(l2)
set(l1,'Color','blue')

Error using matlab.graphics.primitive.Line/set
Invalid or deleted object.
```

如果通过删除任一现有句柄来删除对象，则所有副本都将变为无效，因为您删除了被所有句柄引用的单个对象。

删除句柄对象不同于清除句柄变量。在图形对象层次结构中，对象的父级保存对该对象的引用。例如，父坐标区包含对 l1 和 l2 引用的线条对象的引用。如果从工作区中清除这两个变量，该对象仍然存在。

有关句柄对象行为的详细信息，请参阅“句柄对象行为” （第 1-7 页）。

用户定义的值类

MATLAB 将值类的对象与该对象的赋值变量相关联。当您将值对象复制到另一个变量或将值对象传递给函数时，MATLAB 会创建该对象和该对象包含的所有数据的独立副本。对原始对象的更改不会影响到新对象。值对象的行为类似于 MATLAB 数值类和 **struct** 类。每个属性的行为本质上都类似于 MATLAB 数组。

值对象始终与一个工作区或临时变量相关联。当值对象的变量超出作用域或被清除时，值对象会超出作用域。此处没有对值对象的引用，只有作为独立对象的副本。

值对象行为

以下是一个值类，它在其 **Number** 属性中存储一个值。默认属性值是数字 1。

```
classdef NumValue
  properties
    Number = 1
  end
end
```

创建一个为变量 **a** 赋值的 **NumValue** 对象。

```
a = NumValue

a =

  NumValue with properties:

    Number: 1
```

将 a 的值赋给另一个变量 b。

```
b = a

b =

  NumValue with properties:

    Number: 1
```

变量 a 和 b 是独立的。更改 a 的 Number 属性的值不会影响 b 的 Number 属性。

```
a.Number = 7

a =

  NumValue with properties:

    Number: 7

b

b =

  NumValue with properties:

    Number: 1
```

修改函数中的值对象

将值对象传递给函数时，MATLAB 会在函数工作区中创建该对象的副本。由于值对象的副本是独立的，因此函数不会修改调用方工作区中的对象。因此，修改值对象的函数必须返回修改后的对象，用以在调用方工作区中重新赋值。

有关详细信息，请参阅“对象修改” （第 5-13 页）。

用户定义的句柄类

从 handle 类派生的类的实例是对底层对象数据的引用。复制句柄对象时，MATLAB 会复制句柄，但不会复制存储在对象属性中的数据。副本引用与原始句柄相同的对象。如果更改原始对象的属性值，复制的句柄将引用相同的更改。

句柄对象行为

以下是一个句柄类，它在 Number 属性中存储一个值。默认属性值是数字 1。

```
classdef NumHandle < handle
  properties
    Number = 1
```

```
  end
end
```

创建一个为变量 **a** 赋值的 **NumHandle** 对象。

```
a = NumHandle
```

```
a =

  NumHandle with properties:

    Number: 1
```

将 **a** 的值赋给另一个变量 **b**。

```
b = a
```

```
b =

  NumHandle with properties:

    Number: 1
```

变量 **a** 和 **b** 引用同一底层对象。更改 **a** 的 **Number** 属性的值也将更改 **b** 的 **Number** 属性的值。即，**a** 和 **b** 引用同一个对象。

```
a.Number = 7
```

```
a =

  NumHandle with properties:

    Number: 7
```

```
b
```

```
b =

  NumHandle with properties:

    Number: 7
```

修改函数中的句柄对象

当您将句柄对象传递给函数时，MATLAB 会在函数工作区中创建句柄的副本。由于句柄的副本引用相同的底层对象，因此修改句柄对象的函数实际上也会修改调用方工作区中的对象。因此，函数对作为输入参数传递的句柄对象进行修改后，不必将修改后的对象返回给调用方。

有关详细信息，请参阅“对象修改” （第 5-13 页）。

删除句柄

您可以通过显式调用句柄的 **delete** 方法来销毁句柄对象。删除句柄类对象的句柄会使所有句柄无效。例如：

```
a = NumHandle;
b = a;
delete(a)
b.Number
```

```
Invalid or deleted object.
```

对句柄对象调用 **delete** 会调用该对象的析构函数。有关详细信息，请参阅 “句柄类析构函数” （第 7-8 页）。

初始化属性以包含句柄对象

有关在属性代码块中将属性初始化为默认值和在构造函数中初始化属性之间区别的信息，请参阅 “Initialize Property Values”和“Initialize Arrays of Handle Objects”。

确定对象的相等性

值对象的相等意味着这些对象属于同一个类并且具有相同的状态。

句柄对象的相等意味着句柄变量引用同一个对象。您还可以识别引用同一类且具有相同状态的不同对象的句柄变量。

值对象的相等性

要确定值对象是否大小相同且内容等值，请使用 **isequal**。例如，使用先前定义的 **NumValue** 类创建两个实例并测试相等性：

```
a = NumValue;
b = NumValue;
isequal(a,b)
```

```
ans =

   1
```

a 和 **b** 是独立的，因此不是同一个对象。然而，二者都表示相同的值。

如果更改由一个值对象表示的值，这些对象将不再相等。

```
a = NumValue;
b = NumValue;
b.Number = 7;
isequal(a,b)
```

```
ans =

   0
```

值类没有默认的 **eq** 方法来实现 == 运算。

句柄对象的相等性

句柄对象从 **handle** 基类继承 **eq** 方法。您可以使用 == 和 **isequal** 测试句柄对象之间的两种不同关系：

- 句柄引用同一个对象：== 和 **isequal** 返回 **true**。
- 句柄引用同一类且具有相同值的对象，但它们不是同一对象 - 只有 **isequal** 返回 **true**。

使用先前定义的 **NumHandle** 类创建对象并复制句柄。

```
a = NumHandle;
b = a;
```

使用 == 和 **isequal** 测试相等性。

```
a == b
```

```
ans =

   1
```

```
isequal(a,b)
```

```
ans =

   1
```

使用默认值创建 **NumHandle** 类的两个实例。

```
a = NumHandle;
b = NumHandle;
```

确定 **a** 和 **b** 是否引用同一个对象。

```
a == b
```

```
ans =

   0
```

确定 **a** 和 **b** 是否具有相同的值。

```
isequal(a,b)
```

```
ans =

   1
```

句柄类支持的功能

从 **handle** 类派生的类能够：

- 继承多个有用的方法（“Handle Class Methods”）
- 定义事件和侦听程序（“事件和侦听程序语法” （第 11-9 页））
- 定义动态属性（“动态属性 - 向实例添加属性” （第 8-35 页））
- 实现 set 和 get 方法（“为属性实现 set/get 接口” （第 7-15 页））
- 自定义复制行为（“Implement Copy for Handle Classes”）

有关句柄类及其方法的详细信息，请参阅“The Handle Superclass”。

另请参阅

相关示例

- “Which Kind of Class to Use”
- “Implement Copy for Handle Classes”
- “句柄对象行为” （第 1-7 页）

句柄类析构函数

本节内容

基础知识

类析构函数 - 一个名为 **delete** 的方法，MATLAB 在销毁句柄类的对象之前隐式调用该方法。此外，用户定义的代码可以显式调用 **delete** 来销毁对象。

非析构函数 - 一个名为 **delete** 的方法，但它不符合有效析构函数的语法要求。因此，在销毁句柄对象时，MATLAB 不会隐式调用此方法。值类中名为 **delete** 的方法不是析构函数。值类中名为 **delete**、将 **HandleCompatible** 属性设置为 **true** 的方法不是析构函数。

“对象生命周期” （第 7-11 页）

“方法特性” （第 9-4 页）

句柄类析构函数方法的语法

MATLAB 在销毁句柄类的对象时调用该类的析构函数。仅当 **delete** 被定义为具有适当语法的普通方法时，MATLAB 才会将名为 **delete** 的方法识别为类析构函数。

要成为有效的类析构函数，**delete** 方法：

- 必须定义一个标量输入参数，它是类的对象。
- 不能定义输出参数
- 不能为 **Sealed**、**Static** 或 **Abstract**
- 无法使用 **arguments** 模块进行输入参数验证。

此外，**delete** 方法不应：

- 引发错误，即使对象无效也是如此。
- 为将被销毁的对象创建新句柄
- 调用子类的方法或访问子类的属性

销毁类的对象时，MATLAB 不调用不符合条件的 **delete** 方法。不符合要求的 **delete** 方法会遮蔽 **handle** 类的 delete 方法，阻止对象被销毁。

由与句柄兼容的值类定义的 **delete** 方法不是析构函数，即便此 **delete** 方法由句柄子类继承也是如此。有关与句柄兼容的类的信息，请参阅“Handle Compatible Classes”。

将 **delete** 声明为普通方法：

```
methods
  function delete(obj)
    % obj is always scalar
  ...
  end
end
```

delete 对数组进行按元素调用

MATLAB 对数组中的每个元素单独调用 **delete** 方法。因此，**delete** 方法在每次调用中只传递一个标量参数。

对已删除的句柄调用 **delete** 应该不会生成错误，但会不起作用。这种设计使 **delete** 能够处理同时包含有效和无效对象的对象数组。

delete 方法执行期间的句柄对象

对一个对象调用 **delete** 方法始终会导致对象销毁。当在 MATLAB 代码中显式调用 **delete** 时，或者当 MATLAB 调用该方法时，对象即被销毁，因为从任何工作区都无法再访问该对象。一旦调用了 **delete** 方法，该方法就无法中止或阻止对象被销毁。

delete 方法可以访问将被删除的对象的属性。MATLAB 直到该对象的类和所有超类的 **delete** 方法完成执行后才会销毁这些属性。

如果 **delete** 方法创建的新变量包含将被删除的对象的句柄，则这些句柄无效。在 **delete** 方法完成执行后，被删除对象在任何工作区内任何变量中的句柄都无效。

isvalid 方法为 **delete** 方法中的句柄对象返回 **false**，因为对象销毁在调用该方法时开始。

MATLAB 以与构造时相反的顺序调用 **delete** 方法。也就是说，MATLAB 先调用子类 **delete** 方法，再调用超类 **delete** 方法。

如果超类期望某个属性由子类管理，则超类不应该在其 **delete** 方法中访问该属性。例如，如果子类使用继承的抽象属性来存储对象句柄，则子类应在其 **delete** 方法中销毁该对象，但超类不应在其 **delete** 方法中访问该属性。

支持销毁部分构造的对象

如果在构造对象时发生错误，可能导致在对象完成创建之前调用 **delete**。因此，类 **delete** 方法必须能够处理部分构造的对象。

例如，**PartialObject** 类的 **delete** 方法在访问 **Data** 属性包含的数据之前，会确定该属性是否为空。如果将构造函数参数赋给 **Name** 属性时出错，MATLAB 会传递部分构造的对象并将其删除。

```
classdef PartialObject < handle
  properties
```

```
    % Restrict the Name property
    % to a cell array
    Name cell
    Data
  end
  methods
    function h = PartialObject(name)
      if nargin > 0
        h.Name = name;
        h.Data.a = rand(10,1);
      end
    end
    function delete(h)
      % Protect against accessing properties
      % of partially constructed objects
      if ~isempty(h.Data)
        t = h.Data.a;
        disp(t)
      else
        disp('Data is empty')
      end
    end
  end
end
```

如果使用 **char** 向量而不是规定的元胞数组调用构造函数，则会出现错误：

```
obj = PartialObject('Test')
```

MATLAB 将部分构造的对象传递给 **delete** 方法。构造函数没有设置 **Data** 属性的值，因为设置 **Name** 属性时出错。

```
Data is empty
Error setting 'Name' property of 'PartialObject' class:
...
```

何时定义析构函数方法

在 MATLAB 销毁对象之前，请使用 **delete** 方法执行清理操作。MATLAB 对 **delete** 方法的调用是可靠的，即使执行因 Ctrl-c 或错误而被中断也是如此。

如果在构造句柄类的过程中出现错误，MATLAB 将对该对象调用类析构函数，并对属性包含的所有对象以及所有初始化基类调用析构函数。

例如，假设某方法打开一个文件以用于写入，并且您要在 **delete** 方法中关闭该文件。**delete** 方法可以对该对象存储在 **FileID** 属性中的文件标识符调用 **fclose**：

```
function delete(obj)
  fclose(obj.FileID);
end
```

类层次结构中的析构函数

如果您创建类的层次结构，每个类都可以定义它自己的 **delete** 方法。在销毁对象时，MATLAB 调用层次结构中每个类的 **delete** 方法。在 **handle** 子类中定义 **delete** 方法不会覆盖 **handle** 类的 **delete** 方法。子类 **delete** 方法扩充超类 **delete** 方法。

继承 Sealed delete 方法

类无法定义密封 (**Sealed**) 有效析构函数。当您尝试将一个定义了 **Sealed delete** 方法的类实例化时，MATLAB 将返回错误。

通常，将方法声明为 **Sealed** 会阻止子类覆盖该方法。但是，名为 **delete** 的 **Sealed** 方法并非有效的析构函数，因此无法阻止子类定义它自己的析构函数。

例如，如果超类定义名为 **delete** 的方法，该方法不是有效的析构函数，而是 **Sealed**，则子类：

- 可以定义有效的析构函数（始终命名为 **delete**）。
- 无法定义名为 **delete** 但非有效析构函数的方法。

异构层次结构中的析构函数

异构类层次结构要求异构数组必须传递给密封的方法。但是，该规则不适用于类析构函数方法。由于析构函数方法无法密封，您可以在异构层次结构中定义非密封的有效析构函数，但它确实起到析构函数的作用。

有关异构层次结构的信息，请参阅“Designing Heterogeneous Class Hierarchies”

对象生命周期

当对象生命周期结束时，MATLAB 调用 **delete** 方法。对象的生命周期在以下情况下结束：

- 任何位置都不再引用该对象
- 通过对句柄调用 **delete** 显式删除该对象

函数内部

局部变量或输入参数所引用对象的生命周期从变量赋值开始，到变量重新赋值、清除或不再于该函数或任何句柄数组中引用为止。

当显式清除变量或变量所在的函数结束时，变量会超出作用域。当变量超出作用域且其值属于定义了 **delete** 方法的句柄类时，MATLAB 调用该方法。MATLAB 不定义函数中变量之间的顺序。当同一个函数包含多个值时，不要假设 MATLAB 会按某个固定顺序先后销毁各个值。

句柄对象销毁期间的顺序

在销毁对象时，MATLAB 按以下顺序调用 **delete** 方法：

1. 对象的类的 **delete** 方法
2. 每个超类的 **delete** 方法，从最近的超类开始，沿层次结构向上直到最通用的超类

对于层次结构中位于同一级别的超类，MATLAB 按照类定义中指定的顺序调用其 **delete** 方法。例如，以下类定义先指定 **supclass1**，再指定 **supclass2**。MATLAB 先调用 **supclass1** 的 **delete** 方法，再调用 **supclass2** 的 **delete** 方法。

```
classdef myClass < supclass1 & supclass2
```

每次调用 **delete** 方法后，MATLAB 会销毁仅属于被调用方法所在类的属性值。如果销毁的属性值包含其他句柄对象，且这些对象没有其他引用，则也会调用这些对象的 **delete** 方法。

超类 **delete** 方法无法调用子类的方法或访问子类的属性。

销毁具有循环引用的对象

在一组对象中，如果有对象引用该组中的其他对象，则构成循环引用。在这种情况下，MATLAB 进行如下处理：

- 如果对象仅在该循环内被引用，则销毁它们
- 只要有循环外的 MATLAB 变量对循环内的任一对象进行外部引用，就不会销毁这些对象

MATLAB 按照与构造对象时相反的顺序销毁对象。有关详细信息，请参阅“delete 方法执行期间的句柄对象” （第 7-9 页）。

限制对对象 delete 方法的访问

要销毁句柄对象，可对其显式调用 **delete**：

```
delete(obj)
```

类可以通过将其 **delete** 方法的 **Access** 属性设置为 **private** 来防止显式销毁对象。但是，该类的方法可以调用 **private delete** 方法。

如果类的 **delete** 方法的 **Access** 属性为 **protected**，则只有该类及其子类的方法才能显式删除该类的对象。

但是，当对象生命周期结束时，无论该方法的 **Access** 属性是什么，MATLAB 都会在销毁对象时调用对象的 **delete** 方法。

继承的 private delete 方法

类析构函数的行为不同于被覆盖方法的正常行为。MATLAB 在销毁时执行每个超类的每个 **delete** 方法，即使该 **delete** 方法不是 **public** 也是如此。

当您显式调用对象的 **delete** 方法时，MATLAB 会检查定义该对象的类中 **delete** 方法的 **Access** 属性，但不会检查该对象的超类中的相应属性。具有私有 (**private**) **delete** 方法的超类无法阻止销毁子类对象。

声明私有 delete 方法对密封类最有意义。在类未密封的情况下，子类可以将自身 delete 方法的访问权限定义为公共。显式调用公共子类的 **delete** 方法会导致 MATLAB 调用私有超类的 **delete** 方法。

非析构函数 delete 方法

类可以实现名为 **delete** 但非有效类析构函数的方法。MATLAB 在销毁对象时不会隐式调用此方法。在这种情况下，**delete** 的行为就像普通的方法。

例如，如果超类实现名为 **delete** 的 **Sealed** 方法，但该方法不是有效的析构函数，则 MATLAB 不允许子类覆盖此方法。

由值类定义的 **delete** 方法不能作为类析构函数。

对 MATLAB 对象的外部引用

对象的生命周期如果涉及到外部语言执行自己的生命周期管理（也称为垃圾回收）时，MATLAB 将不会对其进行管理。MATLAB 无法检测何时可以安全地销毁在循环引用中使用的对象，因为外部环境在外部引用被销毁时不会通知 MATLAB。

如果无法避免对 MATLAB 对象的外部引用，则可通过在 MATLAB 中销毁对象来显式打破循环引用。

下一节说明在使用引用了 MATLAB 对象的 Java 对象时如何管理这种情况。

Java 引用会阻止析构函数执行

Java 不支持 MATLAB 对象使用的对象析构函数。因此，对于一个同时包含 Java 和 MATLAB 对象的应用程序，管理其中所有对象的生命周期是很重要的。

如果 Java 对象引用了 MATLAB 对象，则会阻止该 MATLAB 对象被删除。在这些情况下，MATLAB 不会调用句柄对象 **delete** 方法，即使没有句柄变量引用该对象也是如此。为了确保您的 **delete** 方法得以执行，请在句柄变量超出作用域之前对此对象显式调用 **delete**。

当您为引用 MATLAB 对象的 Java 对象定义回调时，可能会出现问题。

例如，**CallbackWithJava** 类创建一个 Java **com.mathworks.jmi.Callback** 对象，并将一个类方法赋给它作为回调函数。结果是一个 Java 对象，它通过函数句柄回调引用句柄对象。

```
classdef CallbackWithJava < handle
  methods
    function obj = CallbackWithJava
      jo = com.mathworks.jmi.Callback;
      set(jo,'DelayedCallback',@obj.cbFunc); % Assign method as callback
      jo.postCallback
    end
    function cbFunc(obj,varargin)
      c = class(obj);
      disp(['Java object callback on class ',c])
    end
    function delete(obj)
      c = class(obj);
      disp(['ML object destructor called for class ',c])
    end
  end
end
```

假设您从函数中创建 **CallbackWithJava** 对象：

```
function testDestructor
  cwj = CallbackWithJava
  ...
end
```

创建 **CallbackWithJava** 类的实例会创建 **com.mathworks.jmi.Callback** 对象并执行回调函数：

```
testDestructor

cwj =

  CallbackWithJava with no properties.

Java object callback on class CallbackWithJava
```

句柄变量 **cwj** 只存在于函数工作区中。但是，当函数结束时，MATLAB 不调用类的 **delete** 方法。**com.mathworks.jmi.Callback** 对象仍然存在，并且引用了 **CallbackWithJava** 类的对象，这会阻止 MATLAB 对象被销毁。

```
clear classes

Warning: Objects of 'CallbackWithJava' class exist.  Cannot clear this class or
any of its superclasses.
```

为避免产生无法访问的对象，请在失去 MATLAB 对象的句柄之前显式调用 **delete**。

```
function testDestructor
  cwj = CallbackWithJava
  ...
  delete(cwj)
end
```

管理应用程序中的对象生命周期

使用 Java 或其他外部语言对象的 MATLAB 应用程序应对所涉对象的生命周期进行管理。一个典型的用户界面应用程序从 MATLAB 对象引用 Java 对象，并对引用 MATLAB 对象的 Java 对象创建回调。

您可以通过各种方式打破这些循环引用：

- 当不再需要 MATLAB 对象时，对该对象显式调用 **delete**
- 注销引用 MATLAB 对象的 Java 对象回调
- 使用同时引用 Java 回调和 MATLAB 对象的中间句柄对象。

另请参阅

详细信息

- “句柄对象行为” (第 1-7 页)

为属性实现 set/get 接口

本节内容

标准 set/get 接口

一些 MATLAB 对象（如图形对象）实现基于 **set** 和 **get** 函数的接口。这些函数支持在单个函数调用中访问对象数组的多个属性。

您可以通过从以下类之一派生，将 **set** 和 **get** 功能添加到您的类中：

- **matlab.mixin.SetGet** - 当您要支持不区分大小写的属性名称部分匹配时使用。即使属性派生自 **matlab.mixin.SetGet**，使用圆点表示法引用属性时，仍需提供确切的属性名称。
- **matlab.mixin.SetGetExactNames** - 当您只想支持区分大小写的属性名称完全匹配时使用。

注意 本节中提到的 **set** 和 **get** 方法不同于属性的 set 和 get 访问方法。有关属性访问方法的信息，请参阅“属性 set 方法” （第 8-27 页）。

子类语法

使用抽象类 **matlab.mixin.SetGet** 或 **matlab.mixin.SetGetExactNames** 作为超类：

```
classdef MyClass < matlab.mixin.SetGet
  ...
end
```

由于 **matlab.mixin.SetGet** 和 **matlab.mixin.SetGetExactNames** 派生自 **handle** 类，因此您的子类也是 **handle** 类。

get 方法语法

get 方法使用对象句柄和属性名称返回对象属性值。例如，假设 **H** 是对象的句柄：

```
v = get(H,'PropertyName');
```

如果您指定一个属性名称和一个句柄数组，**get** 会以值的元胞数组形式返回每个对象的属性值：

```
CV = get(H,'PropertyName');
```

无论 **H** 的形状如何，**CV** 数组始终是一列。

如果指定一个属性名称的 **char** 向量元胞数组和一个句柄数组，**get** 将返回属性值的元胞数组。元胞中的每行对应于句柄数组中的一个对象。元胞中的每列都对应一个属性名称。

```
props = {'PropertyName1','PropertyName2'};
CV = get(H,props);
```

get 返回 m×n 元胞数组，其中 **m = length(H)** 且 **n = length(props)**。

如果指定句柄数组，但未指定属性名称，则 **get** 将返回 **struct** 类型的数组，数组中的每个结构体对应于 **H** 中的一个对象。每个结构体中的每个字段对应于由 **H** 的类定义的一个属性。每个字段的值就是对应属性的值。

```
SV = get(H);
```

如果不指定输出变量，则 **H** 必须为标量。

有关示例，请参阅“对句柄数组使用 get” （第 7-18 页）。

set 方法语法

set 方法为具有句柄 **H** 的对象的指定属性赋予指定值。如果 **H** 是句柄数组，MATLAB 会为数组 **H** 中每个对象的属性赋值。

```
set(H,'PropertyName',PropertyValue)
```

您可以将属性名称的元胞数组和属性值的元胞数组传递给 **set**：

```
props = {'PropertyName1','PropertyName2'};
vals = {Property1Value,Property2Value};
set(H,props,vals)
```

如果 **length(H)** 大于 1，属性值元胞数组 (**vals**) 可以为每个对象中的每个属性设置值。例如，假设 **length(H)** 是 2（两个对象句柄）。您要为每个对象赋予两个属性值：

```
props = {'PropertyName1','PropertyName2'};
vals = {Property11Value,Property12Value;Property21Value,Property22Value};
set(H,props,vals))
```

上述语句等效于以下两个语句：

```
set(H(1),'PropertyName1',Property11Value,'PropertyName2',Property12Value)
set(H(2),'PropertyName1',Property21Value,'PropertyName2',Property22Value)
```

如果指定标量句柄，但没有属性名称，则 **set** 将返回 **struct**，**H** 的类中的每个属性都对应该结构体中的一个字段。每个字段包含一个空元胞数组。

```
SV = set(h);
```

提示 可以在一次 **set** 调用中使用属性名称/属性值元胞数组、结构体数组（字段名称作为属性名称，字段值作为属性值）和元胞数组的任意组合。

派生自 matlab.mixin.SetGet 的类

此示例类定义一个 set/get 接口，并演示继承的方法的行为：

```
classdef LineType < matlab.mixin.SetGet
  properties
    Style = '-'
    Marker = 'o'
```

```
  end
  properties (SetAccess = protected)
    Units = 'points'
  end
  methods
    function obj = LineType(s,m)
      if nargin > 0
        obj.Style = s;
        obj.Marker = m;
      end
    end
    function set.Style(obj,val)
      if ~(strcmpi(val,'-') || ...
           strcmpi(val,'--') || ...
           strcmpi(val,'..'))
        error('Invalid line style ')
      end
      obj.Style = val;
    end
    function set.Marker(obj,val)
      if ~isstrprop(val,'graphic')
        error('Marker must be a visible character')
      end
      obj.Marker = val;
    end
  end
end
```

创建类的实例并保存其句柄：

```
h = LineType('--','*');
```

使用继承的 get 方法查询对象属性的值：

```
get(h,'Marker')
```

```
ans =

'*'
```

使用继承的 set 方法设置属性的值：

```
set(h,'Marker','Q')
```

用 set 和 get 调用属性访问方法

使用 set 和 get 方法时，MATLAB 会调用属性访问方法（LineType 类中的 set.Style 或 set.Marker）。

```
set(h,'Style','-.-')
```

```
Error using LineType/set.Style (line 20)
Invalid line style
```

有关属性访问方法的详细信息，请参阅 “属性 set 方法” （第 8-27 页）

列出所有属性

使用 get 返回包含对象属性及其当前值的 struct：

```
h = LineType('--','*');
SV = get(h)

SV =

  struct with fields:

    Style: '--'
   Marker: '*'
    Units: 'points'
```

使用 **set** 返回包含具有 **public SetAccess** 的属性的 **struct**：

```
S = set(h)

S =

  struct with fields:

    Style: {}
   Marker: {}
```

LineType 类使用 **SetAccess = protected** 定义 **Units** 属性。因此，**S = set(h)** 不为 **S** 中的 **Units** 创建字段。

set 无法返回具有非公共 set 访问权限的属性的可能值。

对句柄数组使用 get

假设您创建了一个 **LineType** 对象数组：

```
H = [LineType('..','z'),LineType('--','q')]

H =

  1x2 LineType with properties:

    Style
    Marker
    Units
```

当 **H** 是句柄数组时，**get** 返回属性值的 (**length(H)**×1) 元胞数组：

```
CV = get(H,'Style')

CV =

  2×1 cell array

    {'..'}
    {'--'}
```

当 **H** 是句柄数组而您没有指定属性名称时，**get** 返回 **struct** 数组，其中的字段具有与属性名称对应的名称。当 **H** 不是标量时，将 **get** 的输出赋给变量。

```
SV = get(H)

SV =
```

```
2x1 struct array with fields:
    Style
    Marker
    Units
```

从 SV 结构体数组的第二个数组元素中获取 Marker 属性的值：

```
SV(2).Marker

ans =

'q'
```

句柄、名称和值的数组

您可以将句柄数组、属性名称元胞数组和属性值元胞数组传递给 set。对于 H 中的每个对象，属性值元胞数组必须有一行与之对应的属性值。对于属性名称数组中的每个属性，上述每行都必须有一个与之对应的值：

```
H = [LineType('..','z'),LineType('--','q')];
set(H,{'Style','Marker'},{'..','o';'--','x'})
```

对 set 的此调用的结果是：

```
H(1)

ans =

  LineType with properties:

    Style: '..'
   Marker: 'o'
    Units: 'points

H(2)

ans =

  LineType with properties:

    Style: '--'
   Marker: 'x'
    Units: 'points'
```

自定义属性列表

通过在子类中重新定义以下方法，自定义属性列表的显示方式：

- **setdisp** - 如果在调用 set 时只使用单个标量句柄输入且不带输出参数，set 会调用 setdisp 来确定如何显示属性列表。
- **getdisp** - 如果在调用 get 时只使用单个标量句柄输入且不带输出参数，get 会调用 getdisp 来确定如何显示属性列表。

为属性名称的部分匹配设置优先级

从 matlab.mixin.SetGet 派生的类可以使用 PartialMatchPriority 属性特性为部分名称匹配指定相对优先级。当解析匹配多个属性名称的不完整且不区分大小写的文本字符串时，MATLAB 应用此属性。

当不精确的名称字符串不具有多义性时，继承的 **set** 和 **get** 方法可以解析不精确的属性名称。当部分属性名称因与多个属性匹配而具有多义性时，**PartialMatchPriority** 属性值可以确定 MATLAB 与哪个属性匹配。

默认优先级等效于 **PartialMatchPriority = 1**。要降低属性的相对优先级，请将 **PartialMatchPriority** 设置为 2 或更大的正整数值。随着 **PartialMatchPriority** 值的增加，属性的优先权会降低。

例如，在此类中，**Verbosity** 属性的名称匹配优先级高于 **Version** 属性。

```
classdef MyClass < matlab.mixin.SetGet
   properties
      Verbosity
   end
   properties (PartialMatchPriority = 2)
      Version
   end
end
```

使用可能具有多义性的不精确名称 **Ver** 调用 **set** 方法会设置 **Verbosity** 属性，因为其相对优先级较高。如果不设置 **PartialMatchPriority** 属性，具有多义性的名称将导致错误。

```
a = MyClass;
set(a,"Ver",10)
disp(a)
```

```
 MyClass with properties:

   Verbosity: 10
    Version: []
```

相同的名称选择机制也适用于 **get** 方法。

```
v = get(a,"Ver")
```

```
v =

   10
```

大小写和名称匹配

大小写不匹配的全名匹配优先于具有较高优先级属性的部分匹配。例如，此类定义优先级为 1（默认值）的 **BaseLine** 属性和优先级为 2（低于 1）的 **Base** 属性。

```
classdef MyClass < matlab.mixin.SetGet
   properties
      BaseLine
   end
   properties (PartialMatchPriority = 2)
      Base
   end
end
```

用字符串 **base** 调用 **set** 方法会设置 **Base** 属性。**BaseLine** 具有更高的优先级，但只是大小写不一样的全名匹配更优先。

```
a = MyClass;
set(a,"base",-2)
disp(a)

 MyClass with properties:

   BaseLine: []
      Base: -2
```

添加新属性时减少不兼容问题

您可以使用 **PartialMatchPriority** 属性来避免在添加新属性时引入代码不兼容问题。例如，以下类使 **set** 和 **get** 方法能够使用字符串 **Dis** 引用 **Distance** 属性，因为 **DiscreteSamples** 属性的优先级较低。

```
classdef Planet < matlab.mixin.SetGet
% Version 1.0
   properties
      Distance
   end
   properties(PartialMatchPriority = 2)
      DiscreteSamples
   end
end
```

该类的 2.0 版引入了名为 **Discontinuities** 的属性。为了防止在现有代码中导致具有多义性的部分属性名称，请使用 **PartialMatchPriority** 将 **Discontinuities** 的优先级设置为低于以前存在的属性的优先级。

```
classdef Planet < matlab.mixin.SetGet
% Version 2.0
   properties
      Diameter;
      NumMoons = 0
      ApparentMagnitude;
      DistanceFromSun;
   end
   properties(PartialMatchPriority = 2)
      DiscreteSamples;
   end
   properties(PartialMatchPriority = 3)
      Discontinuities = false;
   end
end
```

对于 **Planet** 类的 1.0 版，对 **set** 方法的以下调用并不具有多义性。

```
p = Planet;
set(p,"Disc",true)
```

然而，随着 **Discontinuities** 属性的引入，字符串 **Disc** 将具有多义性。通过降低 **Discontinuities** 属性的优先级，字符串 **Disc** 将继续匹配 **DiscreteSamples** 属性。

注意 在编写可重用代码时，使用完整的、区分大小写的属性名称可以避免多义性，防止与后续软件版本不兼容，并生成更可读的代码。

另请参阅

set | get | matlab.mixin.SetGet | matlab.mixin.SetGetExactNames

详细信息

- “使用属性的方式” (第 8-2 页)

属性 - 存储类数据

使用属性的方式

本节内容

什么是属性

属性对属于类实例的数据进行封装。属性中包含的数据可以是公共的、受保护的或私有的。这些数据可以是一组固定的常量值，也可以依赖于其他值，并且仅在查询时计算。要控制属性行为的以上方面，您可以设置属性特性，定义特定于属性的访问方法。

对象属性的灵活性

在某些方面，属性就像 **struct** 对象的字段。然而，在对象属性中存储数据提供了更大的灵活性。属性可以：

- 定义在类定义之外无法更改的常量值。请参阅“定义具有常量值的类属性” （第 15-2 页）。
- 基于其他数据的当前值计算它自己的值。请参阅“从属属性的特征” （第 8-3 页）。
- 执行函数来确定所尝试的赋值是否满足特定条件。请参阅“属性 set 方法” （第 8-27 页）。
- 当尝试获取或设置属性值时，触发事件通知。请参阅“属性 set 和查询事件” （第 11-7 页）。
- 通过代码控制对属性值的访问。请参阅 **SetAccess** 和 **GetAccess** 属性“属性特性” （第 8-7 页）。
- 控制属性值是否与对象一起保存在 MAT 文件中。请参阅“保存和加载对象” （第 13-2 页）。

有关定义和使用类的类示例，请参阅“创建简单类” （第 2-2 页）。

属性的类型

属性有两种类型：

- 存储属性 - 占用内存，并且是对象的一部分
- 从属属性 - 不占用内存，get 访问方法会在查询时计算其属性值

存储属性的特征

- 当您将对象保存到 MAT 文件时，属性值也被存储
- 可以在类定义中为其指定默认值
- 可以将属性值限制为特定的类和大小
- 可以执行验证函数来控制允许的属性值（默认值和所赋的值）
- 设置时可以使用 set 访问方法来控制可能的值

何时使用存储属性

- 需要将属性值保存在 MAT 文件中
- 属性值不依赖于其他属性值

从属属性的特征

从属属性节省内存，因为依赖于其他值的属性值仅在需要时才计算。

何时使用从属属性

如果您要执行以下操作，请将属性定义为从属属性：

- 根据其他值计算属性值（例如，您可以根据 **Width** 和 **Height** 属性计算面积）。
- 提供一个格式随其他值变化的值。例如，普通按钮的大小值由其 **Units** 属性的当前设置决定。
- 提供一个标准接口，根据其他值决定是否使用其中某一属性。例如，视计算机平台不同，工具栏上可以具有不同组件。

有关使用从属属性的类的示例，请参阅“按需计算数据” （第 3-14 页）和“A Class Hierarchy for Heterogeneous Arrays”。

另请参阅

相关示例

- “属性特性” （第 8-7 页）
- “验证属性值” （第 8-12 页）
- “属性 set 方法” （第 8-27 页）
- “静态属性” （第 5-15 页）

属性语法

本节内容

本主题描述如何使用 **properties...end** 模块在 MATLAB 中定义类属性，并介绍属性验证语法和概念。它还涵盖了从类实例中获取和设置属性值的基础知识。

属性定义代码块

properties 和 **end** 关键字定义一个或多个具有相同特性设置的类属性。以下是定义属性块的一般语法：

```
properties (attributes)
  propName1
  ...
  propNameN
end
```

注意 属性不能与类或该类定义的任何其他成员同名。

例如，下面的属性块定义了两个属性，其中 **SetAccess** 特性设置为 **private**。此特性设置意味着属性值只能由 **PrivateProps** 类的成员设置。

```
classdef PrivateProps
  properties (SetAccess = private)
    Property1
    Property2
  end
end
```

还可以为具有不同特性的属性定义多个属性块。在此示例中，一个属性块用私有 **SetAccess** 定义属性，第二个属性块定义一个抽象属性。具有不同特性的属性块可以任何顺序出现在类定义中。

```
classdef MultiplePropBlocks
  properties (SetAccess = private)
    Property1
    Property2
  end
  properties (Abstract)
    Property3
  end
end
```

有关属性特性的完整列表，请参阅“属性特性” （第 8-7 页）。

属性验证语法

在属性块中，可以使用属性验证。属性验证使您能够对每个属性值（包括大小和类）设置一个或多个限制。您还可以为每个属性定义默认值。属性验证的一般语法是：

```
properties (attributes)
  propName1 (dimensions) class {validators} = defaultValue
  ...
end
```

- **(dimensions)** - 属性值的大小，指定为圆括号中包含两个或更多数字的以逗号分隔的列表，如 **(1,2)** 或 **(1,:)**。冒号表示该维度可以包含任意长度。值的维度必须与 **(dimensions)** 完全匹配或兼容。请参阅“基本运算的兼容数组大小”了解详细信息。**(dimensions)** 不能包含表达式。
- **class** - 属性值的类，指定为类的名称，例如 **double**。该值必须为指定的类或可以转换的类。例如，指定 **double** 的属性接受 **single** 类型的值并将它们转换为 **double**。除了 MATLAB 中的可用类，您还可以使用自己的类作为属性验证器。对于用户定义的类，属性验证允许指定的 **class** 的子类通过而不报错，但不会将该子类转换为超类。
- **{validators}** - 验证函数，指定为用花括号括起来的以逗号分隔的列表，如 **mustBePositive** 和 **mustBeScalarOrEmpty**。与 **class** 不同，验证函数不修改属性值。当属性值与其条件不匹配时，验证函数会报错。有关验证函数的列表，请参阅“属性验证函数” （第 8-17 页）。您也可以定义自己的验证函数。
- **defaultValue** - 默认属性值必须符合指定的大小、类和验证规则。默认值也可以是表达式。有关 MATLAB 如何计算默认值表达式的详细信息，请参阅 “Define Properties with Default Values”。

下面的类定义一个属性。属性块未定义显式特性，这等效于定义一个公共属性块。**MyPublicData** 还必须为由正双精度值组成的向量，其默认值为 **[1 1 1]**。

```
classdef ValidationExample
  properties
    MyPublicData (1,:) double {mustBePositive} = [1 1 1]
  end
end
```

并非所有验证选项都必须同时使用，同一属性块中的不同属性可以使用不同验证器组合。在此示例中，**RestrictedByClass** 属性仅使用类验证，而 **RestrictedByFunction** 使用验证函数并赋予默认值。

```
classdef DifferentValidation
  properties
    RestrictedByClass uint32
    RestrictedByFunction {mustBeInteger} = 0
  end
end
```

有关详细信息，请参阅 “Property Class and Size Validation” 和 “属性验证函数” （第 8-17 页）。

属性访问语法

属性访问语法类似于 MATLAB 结构体字段语法。例如，如果 **obj** 是类的对象，则可以通过引用属性名称来获取属性值。

```
val = obj.PropertyName
```

通过将属性引用放在等号的左侧，为属性赋值。

```
obj.PropertyName = val
```

例如，实例化 **ValidationExample** 类并读取 **MyPublicData** 的值。

```
classdef ValidationExample
  properties
```

```
    MyPublicData (1,:) double {mustBePositive} = [1 1 1]
  end
end

x = ValidationExample;
x.MyPublicData

ans =

   1   1   1
```

为满足为其定义的验证器的属性赋予新值。

```
x.MyPublicData = [2 3 5 7];
```

您可以选择定义在您使用此结构体字段语法时 MATLAB 自动调用的 get 和 set 方法。有关详细信息，请参阅“属性 set 方法” （第 8-27 页）。

使用变量引用属性

MATLAB 可以使用以下形式的表达式从 **string** 或 **char** 变量解析属性名称：

```
object.(PropertyNameVar)
```

PropertyNameVar 是包含有效对象属性名称的变量。将属性名称作为参数传递时，请使用以下语法。例如，**getPropValue** 函数返回 **KeyType** 属性的值。

```
PropName = "KeyType";
function o = getPropValue(obj,PropName)
  o = obj.(PropName);
end
```

另请参阅

相关示例

- “属性特性” （第 8-7 页）
- “验证属性值” （第 8-12 页）
- “Initialize Property Values”

属性特性

本节内容

属性特性的目的

您可以在类定义中指定特性以便为特定目的自定义属性的行为。通过设置特性来控制属性的特征，例如访问权限、数据存储和可见性。子类不继承超类成员特性。

指定属性特性

在 **properties** 关键字的同一行为属性特性赋值。

```
properties (Attribute1 = value1, Attribute2 = value2,...)
  ...
end
```

例如，定义一个具有 **private** 访问权限的属性 **Data**。

```
properties (Access = private)
  Data
end
```

对于值为 **true** 的属性，可以使用更简单的语法。属性名称本身意味着 true，在名称中添加逻辑非运算符 (~) 意味着 false。例如，此模块定义抽象属性。

```
properties (Abstract)
  ...
end
```

属性特性表

所有属性都支持下表中列出的特性。特性值适用于 **properties...end** 代码块中定义的所有属性，该代码块用于指定非默认值。未显式定义的属性采用其默认值。

属性特性

特性	值	其他信息
AbortSet	• **true** - 如果新值与当前值相同，则 MATLAB 不设置属性值或调用 set 方法。 • **false**（默认值）- MATLAB 设置属性值，而不考虑当前值。	对于句柄类，将 **AbortSet** 设置为 **true** 还可以防止触发属性 **PreSet** 和 **PostSet** 事件。 有关详细信息，请参阅“Assignment When Property Value Is Unchanged”。
Abstract	• **true** - 此属性没有实现，但具体的子类必须在 **Abstract** 未设置为 **true** 的情况下覆盖此属性。 • **false**（默认值）- 该属性是具体的，不需要在子类中被覆盖。	抽象属性不能定义设置或访问方法。请参阅 “属性访问方法” （第 8-23 页）。 抽象属性不能定义初始值。 密封类无法定义抽象成员。
Access	• **public**（默认值）- 可以从任何代码访问该属性。 • **protected** - 可以从定义类或其子类访问该属性。 • **private** - 该属性只能由定义类的成员访问。 • 对此属性进行访问和设置的类的列表。将类指定为单个 **meta.class** 对象或 **meta.class** 对象的元胞数组。有关详细信息，请参阅 “属性访问列表” （第 8-10 页）。	使用 **Access** 将 **SetAccess** 和 **GetAccess** 设置为相同的值。 将 **Access** 指定为空元胞数组 **{}** 与 **private** 访问权限相同。 有关详细信息，请参阅 “类成员访问” （第 12-10 页）。
Constant	• **true** - 该属性在该类的所有实例中具有相同的值。 • **false**（默认值）- 属性值可能因实例而异。	子类会继承常量属性，但不能更改常量属性。 常量属性也无法定义为从属属性。 对于常量属性，会忽略 **SetAccess** 的值。 有关详细信息，请参阅“定义具有常量值的类属性” （第 15-2 页）。
Dependent	• **true** - 属性值不存储在对象中。该值在访问属性时计算。 • **false**（默认值）- 属性值存储在对象中。	您可以为从属属性定义 set 方法，但 set 方法实际上无法设置该属性的值。它可以采取其他操作，例如设置另一个属性的值。有关示例，请参阅 “何时对从属属性使用 set 方法” （第 8-33 页）。 使用 **isequal** 测试对象相等性时，不考虑从属属性 get 方法返回的值。

特性	值	其他信息
GetAccess	• **public**（默认值）- 可以从任何代码读取该属性。 • **protected** - 可以从定义类或其子类读取该属性。 • **private** - 该属性只能由定义类的成员读取。 • 可以读取此属性的类的列表。将类指定为单个 **meta.class** 对象或 **meta.class** 对象的元胞数组。有关详细信息，请参阅“属性访问列表”（第 8-10 页）。有关详细信息，请参阅“属性访问列表”（第 8-10 页）。	将 **GetAccess** 指定为空元胞数组 **{}** 与 **private** 访问权限相同。 在命令行窗口中，MATLAB 不显示具有 **protected** 或 **private GetAccess** 特性的属性的名称和值。 对于属性的 **SetAccess** 和 **GetAccess** 特性，所有子类必须与超类指定相同的值。 有关详细信息，请参阅 “类成员访问” （第 12-10 页）。
GetObservable	• **true** - 您可以为句柄类属性创建侦听程序。每当查询属性值时，都会调用这些侦听程序。 • **false**（默认值）- 侦听程序没有访问此属性的权限。	有关详细信息，请参阅 “属性 set 和查询事件” （第 11-7 页）。
Hidden	• **true** - 该属性在属性列表中或调用 **get**、**set** 或 **properties** 函数的结果中不可见。 • **false**（默认值）- 属性可见。	在命令行窗口中，MATLAB 不显示其 **Hidden** 特性为 **true** 的属性的名称和值。但是，隐藏的属性在类图查看器中可见。
NonCopyable	• **true** - 在复制定义属性值的对象时，不会复制该属性值。 • **false**（默认值）- 当复制对象时，同时复制属性值。	您只能在句柄类中将 **NonCopyable** 设置为 **true**。 有关详细信息，请参阅“Exclude Properties from Copy”。
PartialMatchPriority	正整数 - 定义 **get** 和 **set** 方法中使用的部分属性名称匹配的相对优先级。默认值为 1。	仅适用于 **matlab.mixin.SetGet** 的子类。 有关详细信息，请参阅“为属性名称的部分匹配设置优先级” （第 7-19 页）。

特性	值	其他信息
SetAccess	• **public**（默认值）- 可以从任何代码设置该属性。 • **protected** - 可以从定义类或其子类设置该属性。 • **private** - 该属性只能由定义类的成员设置。 • **immutable** - 该属性只能由构造函数设置。 • 对此属性进行设置的类的列表。将类指定为单个 **meta.class** 对象或 **meta.class** 对象的元胞数组。有关详细信息，请参阅“属性访问列表”（第 8-10 页）。有关详细信息，请参阅“属性访问列表”（第 8-10 页）。	对于属性 **SetAccess** 和 **GetAccess** 特性，所有子类必须与超类指定相同的值。 有关详细信息，请参阅“类成员访问”（第 12-10 页）、“Properties Containing Objects”和“Mutable and Immutable Properties”。
SetObservable	• **true** - 您可以为句柄类属性创建侦听程序。每当设置属性值时，都会调用这些侦听程序。 • **false**（默认值）- 侦听程序没有访问此属性的权限。	有关详细信息，请参阅“属性 set 和查询事件”（第 11-7 页）。
Transient	• **true** - 当对象保存到文件或从 MATLAB 发送到另一个程序（如 MATLAB Engine 应用程序）时，属性值不会保存。 • **false**（默认值）- 保存对象时会同时保存属性值。	有关详细信息，请参阅“对象的保存和加载过程”（第 13-2 页）。
框架特性	使用特定框架基类的类具有特定于框架的特性。有关这些特性的信息，请参阅您正在使用的特定基类的文档。	

属性访问列表

对于 **Access**、**GetAccess** 和 **SetAccess** 特性，您可以使用 **meta.class** 实例的列表。例如，此类声明 **Prop1** 和 **Prop2** 属性的访问列表。

```
classdef PropertyAccess
  properties (GetAccess = {?ClassA, ?ClassB})
    Prop1
  end
  properties (Access = ?ClassC)
    Prop2
  end
end
```

对于 **Prop1**：

- 类 **ClassA** 和 **ClassB** 具有对 **Prop1** 的 get 访问权限。
- **ClassA** 和 **ClassB** 的所有子类都具有对 **Prop1** 的 get 访问权限。
- 访问列表不能继承，因此 **PropertyAccess** 的子类没有对 **Prop1** 的 get 访问权限，除非它们显式定义该访问权限。

对于 **Prop2**：

- **ClassC** 有对 **Prop2** 的 get 和 set 访问权限。
- **ClassC** 的所有子类都有对 **Prop2** 的 get 和 set 访问权限。
- 访问列表不能继承，因此 **PropertyAccess** 的子类没有对 **Prop2** 的访问权限，除非它们显式定义该访问权限。

另请参阅

相关示例

- “属性语法” （第 8-4 页）
- “Initialize Property Values”

验证属性值

本节内容

类定义中的属性验证

MATLAB 属性验证使您能够对属性值进行特定限制。您可以使用验证来约束属性值的类和大小。此外，您可以使用函数来建立属性值必须符合的条件。MATLAB 定义了一组验证函数，您也可以编写自己的验证函数。

在类定义中，属性验证的使用是可选的。

关于属性验证的其他信息

有关属性验证的详细信息，请参阅“Property Class and Size Validation”、“属性验证函数” （第 8-17 页）和“Metadata Interface to Property Validation”。

验证语法

下面代码中突出显示的区域显示属性验证的语法。

```
classdef MyClass
   properties
      Prop(dim1,dim2,...) ClassName {fcn1,fcn2,...} = defaultValue
   end
end
```

Size Class Functions

属性验证包括以下任一项：

- 大小 - 每个维度的长度，指定为正整数或冒号。冒号表示该维度中允许任何长度。赋给属性的值必须符合指定的大小或与指定的大小兼容。有关详细信息，请参阅“Property Size Validation”。
- 类 - 单个 MATLAB 类的名称。赋给属性的值必须属于指定的类或可转换为指定的类。使用 MATLAB 类或 MATLAB 支持的任何外部定义的类，但 Java 和 COM 类除外。有关详细信息，请参阅“Property Class Validation”。
- 函数 - 以逗号分隔的验证函数名称列表。MATLAB 在应用任何可能的类和大小转换后，将赋给属性的值传递给每个验证函数。如果验证失败，验证函数会引发错误，但不会返回值。有关详细信息，请参阅“属性验证函数” （第 8-17 页）。

  有关 MATLAB 验证函数的列表，请参阅“属性验证函数” （第 8-17 页）。

使用属性验证

使用公共属性的属性验证来控制用户代码赋给属性的值。

如果要将属性值限制为一组固定的标识符，请创建包含这些标识符的枚举类，并将属性值范围限制为该类。有关枚举类的信息，请参阅“定义枚举类” （第 14-2 页）。

MATLAB 类型转换规则适用于属性验证。例如，MATLAB 可以对数值类型进行强制转换。因此，将属性值限制为特定的数值类型（如双精度）不会阻止使用其他数值类型对属性赋值。

要确保只能将特定类型的值赋给属性，请将属性限制为仅支持所需类型转换的类型，或者使用验证函数指定属性允许的确切类，而不是指定属性类型。MATLAB 在执行任何验证函数之前先计算类型设定。有关详细信息，请参阅“验证的顺序” （第 8-14 页）。

指定有效默认值

确保赋给属性的默认值满足指定验证所施加的限制。如果未指定默认值，MATLAB 会使用指定类的空对象进行赋值来创建默认值；如果大小限制不允许使用空默认值，则调用默认构造函数来创建默认值。默认构造函数必须返回正确大小的对象。

使用属性验证的示例类

ValidateProps 类定义三个属性并进行验证。

```
classdef ValidateProps
  properties
    Location(1,3) double {mustBeReal, mustBeFinite}
    Label(1,:) char {mustBeMember(Label,{'High','Medium','Low'})} = 'Low'
    State(1,1) matlab.lang.OnOffSwitchState
  end
end
```

- **Location** 必须为 **double** 类的 1×3 数组，其值是有限实数。
- **Label** 必须为 **char** 向量，包含 **'High'**、**'Medium'** 或 **'Low'** 之一。
- **State** 必须为 **matlab.lang.OnOffSwitchState** 类的枚举成员（**off** 或 **on**）。

实例化时的验证

创建 **ValidateProps** 类的对象会对隐式和显式默认值执行验证：

```
a = ValidateProps

a =

  ValidateProps with properties:

    Location: [0 0 0]
       Label: 'Low'
       State: off
```

在创建对象时，MATLAB 会：

- 将 **Location** 属性值初始化为 **[0 0 0]**，以满足大小和类要求。
- 将 **Label** 属性设置为默认值 **'Low'**。默认值必须为允许的值集的成员。空的 **char** 隐式默认值会导致错误。
- 将 **State** 属性设置为 **matlab.lang.OnOffSwitchState** 类定义的 **off** 枚举成员。

有关 MATLAB 如何选择默认值的信息，请参阅“Default Values Per Size and Class”。

验证的顺序

对属性赋值（包括类定义中指定的默认值）时，MATLAB 会按以下顺序执行验证：

- 类验证 - 此验证可导致转换为不同类，例如将 **char** 转换为 **string**。属性赋值遵循 MATLAB 对数组的转换规则。
- 大小验证 - 此验证可导致大小转换，例如标量扩展或列向量到行向量的转换。对指定大小验证的属性进行赋值时，其行为与 MATLAB 数组的赋值行为相同。有关索引赋值的信息，请参阅“数组索引”。
- 验证函数 - MATLAB 按照从左到右的顺序将类验证和大小验证的结果传递给每个验证函数。在调用完所有验证函数之前可能会发生错误，从而结束验证过程。
- set 方法 - MATLAB 在调用属性 set 方法（如果为属性定义了该方法）之前执行属性验证。通过属性 set 或 get 方法为属性赋值时不会再次应用验证。通常，您可以使用属性验证来替换属性 set 方法。

属性验证错误

ValueProp 类使用大小、类和函数验证来确保对 **Value** 属性的赋值是非负的双精度标量。

```
classdef ValueProp
  properties
    Value(1,1) double {mustBeNonnegative} = 0
  end
end
```

以下语句尝试将元胞数组赋给属性。此赋值违反类验证。

```
a.Value = {10,20};
```

```
Error setting property 'Value' of class 'ValueProp':
Invalid data type. Value must be double or be convertible to double.
```

以下语句尝试将 1×2 双精度数组赋给属性。此赋值违反大小验证。

```
a.Value = [10 20];
```

```
Error setting property 'Value' of class 'ValueProp':
Size of value must be scalar.
```

以下语句尝试将双精度标量赋给属性。此赋值未通过函数验证，该验证要求使用非负数。

```
a.Value = -10;
```

```
Error setting property 'Value' of class 'ValueProp':
Value must be nonnegative.
```

验证过程在遇到第一个错误时结束。

抽象属性验证

您可以为抽象属性定义属性验证。验证会应用于实现该属性的所有子类。但是，子类无法对其自身属性的实现使用任何验证。从多个类继承属性验证时，只有一个超类中的单一 Abstract 属性可以定义验证。任何超类都不能将属性定义为具体属性。

更改验证时不更新对象

如果在类的对象存在时更改属性验证，MATLAB 不会尝试将新验证应用于现有属性值。只有在您对现有对象的属性进行赋值时，MATLAB 才会应用新验证。

加载操作期间的验证

将对象保存到 **MAT** 文件时，MATLAB 会将所有非默认属性值与对象一起保存。在加载对象时，MATLAB 会在新创建的对象中还原这些属性值。

如果类定义更改了属性验证，使得加载的属性值不再有效，则 MATLAB 会将其值替换为当前定义的该属性的默认值。但是，在将当前类定义的默认值赋给属性之前，**load** 函数会阻止验证错误的显示。因此，在加载操作期间，验证错误会以静默方式被忽略。

为了说明此行为，以下示例创建、保存并加载了 **MonthTemp** 类的一个对象。此类将 **AveTemp** 属性限制为元胞数组。

```
classdef MonthTemp
  properties
    AveTemp cell
  end
end
```

创建一个 **MonthTemp** 对象，并为 **AveTemp** 属性赋值。

```
a = MonthTemp;
a.AveTemp = {'May',70};
```

使用 **save** 保存对象。

```
save TemperatureFile a
```

编辑属性定义，以将 **AveTemp** 属性的验证类从元胞数组更改为 **containers.Map**。

```
classdef MonthTemp
  properties
    AveTemp containers.Map
  end
end
```

在 MATLAB 路径上加载具有新类定义的已保存对象。MATLAB 无法将保存的值赋给 **AveTemp** 属性，因为元胞数组 **{'May',70}** 与当前要求属性值为 **containers.Map** 的对象不兼容。MATLAB 无法将元胞数组转换为 **containers.Map**。

为了解决不兼容问题，MATLAB 将所加载对象的 **AveTemp** 属性设置为当前默认值，即空的 **containers.Map** 对象。

```
load TemperatureFile a
 a.AveTemp

ans =

  Map with properties:

      Count: 0
    KeyType: char
  ValueType: any
```

现在系统为所加载对象的 **AveTemp** 属性赋了一个不同的值，因为保存的值现在无效。但是，加载过程会抑制验证错误的显示。

为了防止在更改类定义和重新加载对象时缺失数据，请实现 **loadobj** 方法或类转换器方法，使保存的值满足当前属性验证。

有关保存和加载对象的详细信息，请参阅“对象的保存和加载过程” （第 13-2 页）。

另请参阅

相关示例

- “Property Class and Size Validation”
- “属性验证函数” （第 8-17 页）

属性验证函数

本节内容

MATLAB 验证函数

MATLAB 定义了用于属性验证的函数。这些函数支持验证的常用模式，并提供描述性错误消息。下表对 MATLAB 验证函数进行了分类并描述了它们的用法。

数值属性

名称	含义	对输入调用的函数
mustBePositive(value)	**value > 0**	**gt, isreal, isnumeric, islogical**
mustBeNonpositive(value)	**value <= 0**	**ge, isreal, isnumeric, islogical**
mustBeNonnegative(value)	**value >= 0**	**ge, isreal, isnumeric, islogical**
mustBeNegative(value)	**value < 0**	**lt, isreal, isnumeric, islogical**
mustBeFinite(value)	**value** 中不含 **NaN** 和 **Inf** 元素。	**isfinite**
mustBeNonNan(value)	**value** 中不含 **NaN** 元素。	**isnan**
mustBeNonzero(value)	**value ~= 0**	**eq, isnumeric, islogical**
mustBeNonsparse(value)	**value** 中不含稀疏元素。	**issparse**
mustBeReal(value)	**value** 没有虚部。	**isreal**
mustBeInteger(value)	**value == floor(value)**	**isreal, isfinite, floor, isnumeric, islogical**
mustBeNonmissing(value)	**value** 不能包含缺失值。	**ismissing**

与其他值的比较

名称	含义	对输入调用的函数
mustBeGreaterThan(value,c)	**value > c**	**gt, isreal, isnumeric, islogical**
mustBeLessThan(value,c)	**value < c**	**lt, isreal, isnumeric, islogical**
mustBeGreaterThanOrEqual(value,c)	**value >= c**	**ge, isreal, isnumeric, islogical**
mustBeLessThanOrEqual(value,c)	**value <= c**	**le, isreal, isnumeric, islogical**

数据类型

名称	含义	对输入调用的函数
mustBeA(value,classnames)	**value** 必须为特定类。	使用类定义关系
mustBeNumeric(value)	**value** 必须为数值。	**isnumeric**
mustBeNumericOrLogical(value)	**value** 必须为数值或逻辑值。	**isnumeric, islogical**
mustBeFloat(value)	**value** 必须为浮点数组。	**isfloat**
mustBeUnderlyingType(value,typename)	**value** 必须具有指定的基础类型。	**isUnderlyingType**

大小

名称	含义	对输入调用的函数
mustBeNonempty(value)	**value** 不为空。	**isempty**
mustBeScalarOrEmpty(value)	**value** 必须为标量或为空。	**isscalar, isempty**
mustBeVector(value)	**value** 必须为向量。	**isvector**

成员关系和范围

名称	含义	对输入调用的函数
mustBeMember(value,S)	**value** 的所有成员都可在 **S** 中找到。	**ismember**
mustBeInRange(value,lower,upper,boundflags)	**value** 必须在范围内。	**gt, ge, lt, le**

文本

名称	含义	对输入调用的函数
mustBeFile(path)	**path** 必须指向文件。	**isfile**
mustBeFolder(folder)	**path** 必须指向文件夹。	**isfolder**
mustBeNonzeroLengthText(value)	**value** 必须为一段具有非零长度的文本。	不适用
mustBeText(value)	**value** 必须为字符串数组、字符向量或字符向量元胞数组。	不适用
mustBeTextScalar(value)	**value** 必须为一段文本。	不适用
mustBeValidVariableName(varname)	**varname** 必须为有效的变量名称。	**isvarname**

使用函数验证属性

使用类定义中的属性验证函数对属性值施加特定限制。验证函数接受潜在的属性值作为参数，如果该值不满足函数所施加的特定要求，则发出错误。

在验证过程中，MATLAB 将值传递给类定义中列出的每个验证函数。MATLAB 从左到右调用每个函数，并在遇到第一个错误时发出错误警告。传递给验证函数的值是在设定类和大小时应用任意转换之后的结果。有关类和大小验证的详细信息，请参阅“Property Class and Size Validation”。

有关 MATLAB 验证函数的列表，请参阅“MATLAB 验证函数” （第 8-17 页）。

验证函数语法

将验证函数指定为一组以逗号分隔的函数名称或含参函数调用（括在花括号内）。

```
classdef MyClass
  properties
    Prop {fcn1,fcn2,...} = defaultValue
  end
end
```

MATLAB 将可能的属性值隐式传递给验证函数。但是，如果验证函数除可能的属性值之外还需要输入参数，则必须包括属性和附加参数。附加参数必须为字面值，不能引用变量。字面值采用非符号表示形式，如数字和文本。

以函数 **mustBeGreaterThan** 为例。它需要一个界限值作为输入参数。此验证函数要求属性值必须大于此界值。

将属性作为第一个参数传递。使用属性名称，但不要用引号将名称括起来。此属性定义将 **Prop** 限制为大于 **10** 的值。

```
properties
  Prop {mustBeGreaterThan(Prop,10)}
end
```

使用验证函数

以下类为每个属性指定验证函数。

- **Data** 必须为有限数值。
- **Interp** 必须为列出的三个选项之一。为此属性指定一个默认值以满足此要求。

```
classdef ValidatorFunction
  properties
    Data {mustBeNumeric, mustBeFinite}
    Interp {mustBeMember(Interp,{'linear','cubic','spline'})} = 'linear'
  end
end
```

创建类的默认对象会显示初始值。

```
a = ValidatorFunction
```

```
a =

  ValidatorFunction with properties:

      Data: []
    Interp: 'linear'
```

为属性赋值会调用验证函数。

```
a.Data = 'cubic'
```

```
Error setting property 'Data' of class 'ValidatorFunction':
Value must be numeric.
```

由于 **Data** 属性验证不包括数值类，因此 **char** 向量不会转换为数值。如果您更改 **Data** 属性的验证以将类指定为 **double**，MATLAB 会将 **char** 向量转换为 **double** 数组。

```
properties
  Data double {mustBeNumeric, mustBeFinite}
end
```

对 **char** 向量的赋值不会产生错误，因为 MATLAB 会将 **char** 向量转换为 **double** 类。

```
a.Data = 'cubic'
```

```
a =

  ValidatorFunction with properties:

      Data: [99 117 98 105 99]
    Interp: 'linear'
```

对 **Interp** 属性的赋值需要完全匹配。

```
a = ValidatorFunction;
a.Interp = 'cu'
```

```
Error setting property 'Interp' of class 'ValidatorFunction':
Value must be a member of this set
    linear
    cubic
    spline
```

使用枚举类，它支持不精确匹配且不区分大小写。

通过枚举类实现不精确匹配

使用枚举类进行属性验证具有以下优点：

- 对于明确字符向量或字符串标量，可进行不精确、不区分大小写的匹配
- 将不精确匹配转换为正确值

例如，假设您为 **Interp** 属性验证定义了 **InterpMethod** 枚举类。

```
classdef InterpMethod
  enumeration
    linear
    cubic
    spline
  end
end
```

更改 **Interp** 属性验证以使用 **InterpMethod** 类。

```
classdef ValidatorFunction
  properties
    Data {mustBeNumeric, mustBeFinite}
    Interp InterpMethod
  end
end
```

为其赋一个与 **'cubic'** 的前几个字母匹配的值。

```
a = ValidatorFunction;
a.Interp = 'cu'
```

```
a =

  ValidatorFunction with properties:

    Data: []
  Interp: cubic
```

定义验证函数

验证函数是专门为验证属性值和函数参数值而设计的普通 MATLAB 函数。用于验证属性的函数：

- 接受可能的属性值作为输入参数
- 不返回值
- 如果验证失败，将引发错误

当使用 MATLAB 验证函数无法提供所需的特定验证时，您可以创建自己的验证函数，这非常有用。您可以在类文件中创建局部函数，或将函数放在 MATLAB 路径上，以便在任何类中使用。

例如，下面的 **ImgData** 类使用局部函数定义了一个验证函数，该验证函数将 **Data** 属性限制为仅 **uint8** 或 **uint16** 值，从而排除子类并且不允许从其他数值类进行转换。预定义的验证函数 **mustBeInRange** 则限制了允许的值范围。

```
classdef ImgData
  properties
    Data {mustBeImData(Data), mustBeInRange(Data,0,255)} = uint8(0)
  end
```

```
end
function mustBeImData(a)
   % Check for specific class
   if ~(isa(a,'uint8') || isa(a,'uint16'))
      eidType = 'ImData:notUint8OrUint16';
      msgType = 'Values assigned to Data property must be uint8 or uint16 data.';
      throwAsCaller(MException(eidType,msgType))
   end
end
```

当您创建 **ImgData** 类的实例时，MATLAB 会验证默认值是否为 **0...255** 范围内的 **uint8** 或 **uint16** 值，且不为空。请注意，默认值必须与赋给该属性的任何其他值一样，满足验证要求。

```
a = ImgData
```

```
a =

  ImgData with properties:

    Data: 0
```

属性赋值按从左到右的顺序调用验证函数。将 **char** 向量赋给 **Data** 属性会导致 **mustBeImData** 引发错误。

```
a.Data = 'red';
```

```
Error setting property 'Data' of class 'ImgData'. Value assigned to
Data property is not uint8 or uint16 data.
```

赋值时如果值超出范围，则会导致 **mustBeInRange** 引发错误。

```
a.Data = uint16(312);
```

```
Error setting property 'Data' of class 'ImgData'. Value must be greater
than or equal to 0, and less than or equal to 255.
```

要了解相关函数，请参阅 **mustBeInteger**、**mustBeNumeric** 和 **mustBePositive**。

另请参阅

相关示例

- “验证属性值” (第 8-12 页)
- “Property Class and Size Validation”

属性访问方法

本节内容

属性提供对类数据的访问

在 MATLAB 中，属性可以具有公共访问权限。因此，属性可以提供在类的设计中向用户公开的数据的访问权限。

可使用属性访问方法来提供错误检查功能，或实现属性访问所产生的附带效果。访问方法的示例包括可以在设置属性时更新其他属性值的函数，或是可以在返回值之前转换属性值格式的函数。

有关访问方法语法的具体信息，请参阅“属性 get 方法” （第 8-30 页）和“属性 set 方法” （第 8-27 页）。

您可以使用属性验证来限制属性值的大小、类和其他方面。有关属性验证的信息，请参阅“验证属性值” （第 8-12 页）。

访问方法的性能注意事项

无论何时访问属性值，属性访问方法都会增加函数调用的开销。如果发生在类方法内部的属性访问对性能有重大影响，请定义私有属性来存储值。在方法内部使用这些值，而不进行任何错误检查。对于来自类外部的较代频率的访问，请定义使用访问方法进行错误检查的公共 `Dependent` 属性。

有关 `Dependent` 属性的访问方法的信息，请参阅“从属属性的 set 和 get 方法” （第 8-32 页）。

属性 set 和 get 方法

每当查询属性值或对属性赋值时，属性访问方法都会执行特定代码。这些方法使您能够执行各种操作：

- 在对属性赋值之前执行代码来执行某些操作，例如：
  - 施加值范围限制（“验证属性值” （第 8-12 页））
  - 检查类型和维度是否正确
  - 提供错误处理
- 在返回属性的当前值之前执行代码以执行某些操作，例如：
  - 计算不存储值的属性的值（请参阅“按需计算数据” （第 3-14 页））
  - 更改其他属性的值
  - 触发事件（请参阅“事件和侦听程序概述” （第 11-2 页））

要控制哪些代码可以访问属性，请参阅“属性特性” （第 8-7 页）。

当 MATLAB 调用访问方法时

每当您从访问方法外部设置或查询对应的属性值时，属性访问方法都会自动执行。MATLAB 不以递归方式调用访问方法。也就是说，当在属性的 set 方法内部设置属性时，不论正在修改的是类的什么实例，MATLAB 都不会调用 set 方法。同样，当在属性的 get 方法内部查询属性值时，MATLAB 也不会调用该属性的 get 方法。

注意 您不能直接调用属性访问方法。MATLAB 会在您访问属性值时调用这些方法。

从属性 **meta.property** 对象获取 set 和 get 访问方法的函数句柄。**meta.property SetMethod** 和 **GetMethod** 属性包含引用这些方法的函数句柄。

对访问方法的限制

定义属性访问方法时应新遵循以下限制：

- 仅为具体属性（即非抽象属性）定义
- 仅在定义属性的类中定义（除非属性在该类中是抽象的，此时具体子类必须定义访问方法）。

MATLAB 没有默认 set 或 get 属性访问方法。因此，如果您不定义属性访问方法，MATLAB 软件在对属性赋值或返回属性值之前不会调用任何方法。

定义后，只有 set 和 get 方法可以设置和查询实际属性值。有关 MATLAB 在哪些情形下不会调用属性 set 方法的信息，请参阅“调用 set 方法时” （第 8-28 页）。

注意 属性的 set 和 get 访问方法与用户可调用的 **set** 和 **get** 方法（用于通过类实例设置和查询属性值）并不等同。有关用户可调用的 **set** 和 **get** 方法的信息，请参阅“为属性实现 set/get 接口” （第 7-15 页）。

访问方法无法调用函数来访问属性

您只能从属性 set 或 get 访问方法内设置和获取属性值。您无法从 set 或 get 方法调用另一个函数并尝试从该函数访问属性值。

例如，调用另一个函数来执行实际工作的匿名函数无法访问属性值。同样，访问函数无法调用另一个函数来访问属性值。

定义访问方法

访问方法有带属性名称的特殊名称。因此，每当引用 **PropertyName** 时，就会执行 **get.PropertyName**；每当对 **PropertyName** 赋值时，就会执行 **set.PropertyName**。

在不指定特性的方法代码块中定义属性访问方法。您无法直接调用这些方法。MATLAB 只会在有代码访问属性时，才会调用这些方法。

属性访问方法不会出现在 **methods** 命令返回的类方法列表中，也会不包括在 **meta.class** 对象 **Methods** 属性中。

访问方法函数句柄

属性 **meta.property** 对象包含属性 set 和 get 方法的函数句柄。**SetMethod** 包含 set 方法的函数句柄。**GetMethod** 包含 get 方法的函数句柄。

从 **meta.property** 对象获取这些句柄：

```
mc = ?ClassName;
mp = findobj(mc.PropertyList,'Name','PropertyName');
fh = mp.GetMethod;
```

例如，如果类 **MyClass** 为其 **Text** 属性定义了 get 方法，则可以从 **meta.class** 对象获取此函数的函数句柄：

```
mc = ?MyClass;
mp = findobj(mc.PropertyList,'Name','Text');
fh = mp.GetMethod;
```

返回值 **fh** 包含为指定类的指定属性名称定义的 get 方法的函数句柄。

有关定义函数句柄的信息，请参阅“创建函数句柄”

set 和 get 方法的执行和属性事件

MATLAB 软件在 set 和 get 操作前后生成事件。您可以使用这些事件来通知侦听程序已引用属性值或已为属性赋值。生成事件的时间点如下所示：

- **PreGet** - 在调用属性 get 方法之前触发
- **PostGet** - 在属性 get 方法返回其值之后触发

如果类计算属性值 (**Dependent = true**)，则其 set 事件的行为就像 get 事件一样：

- **PreSet** - 在调用属性 set 方法之前触发
- **PostSet** - 在调用属性 set 方法之后触发

如果不计算属性值（默认为 **Dependent = false**），则使用 set 方法的赋值语句会生成事件：

- **PreSet** - 在 set 方法中新赋属性值之前触发
- **PostSet** - 在 set 方法中新赋属性值之后触发

有关使用属性事件的信息，请参阅“创建属性侦听程序” （第 11-13 页）。

访问方法和包含数组的属性

您可以对包含数组的属性使用数组索引，而无需直接调用属性的 set 和 get 方法。

对于索引引用：

```
val = obj.PropName(n);
```

MATLAB 调用 get 方法以获取引用的值。

对于索引赋值：

```
obj.PropName(n) = val;
```

MATLAB：

- 调用 get 方法以获取属性值
- 对返回的属性执行索引赋值

- 将新属性值传递给 set 方法

访问方法和对象数组

当引用或赋值发生在对象数组上时，MATLAB 在循环中调用 set 和 get 方法。在此循环中，MATLAB 始终将标量对象传递给 set 和 get 方法。

使用访问方法修改属性值

如果您要在为属性赋值或返回属性值之前执行一些附加步骤，则属性访问方法非常有用。例如，**Testpoint** 类使用属性 set 方法检查值的范围。如果在特定范围内，则应用缩放，如果不在，则将其设置为 **NaN**。

属性 get 方法在返回当前值之前应用缩放因子：

```
classdef Testpoint
  properties
    expectedResult = []
  end
  properties(Constant)
    scalingFactor = 0.001
  end
  methods
    function obj = set.expectedResult(obj,erIn)
      if erIn >= 0 && erIn <= 100
        erIn = erIn.*obj.scalingFactor;
        obj.expectedResult = erIn;
      else
        obj.expectedResult = NaN;
      end
    end
    function er = get.expectedResult(obj)
      er = obj.expectedResult/obj.scalingFactor;
    end
  end
end
```

另请参阅

详细信息

- “属性 set 方法” （第 8-27 页）
- “属性 get 方法” （第 8-30 页）
- “Properties Containing Objects”

属性 set 方法

本节内容

属性访问方法概述

有关属性访问方法的概述，请参阅“属性访问方法” （第 8-23 页）

属性 set 方法语法

每当给属性赋值时，MATLAB 都会调用属性的 set 方法。

注意 您不能直接调用属性访问方法。MATLAB 会在您访问属性值时调用这些方法。

属性 set 方法具有以下语法，其中 **PropertyName** 是属性的名称。

对于值类：

```
methods
  function obj = set.PropertyName(obj,value)
    ...
end
```

- **obj** - 其属性被赋值的对象
- **value** - 要赋给属性的新值

值类 set 函数必须将修改后的对象返回给调用函数。句柄类不需要返回修改后的对象。

对于句柄类：

```
methods
  function set.PropertyName(obj,value)
    ...
end
```

使用属性 set 方法的默认方法特性。定义 set 方法的方法代码块无法指定属性。

验证属性设置值

使用属性 set 方法验证即将分配给属性的值。属性 set 方法可以在采取存储新属性值所需的任何操作之前，对输入值执行错误检查等操作。

```
classdef MyClass
  properties
    Prop1
  end
```

```
  methods
    function obj = set.Prop1(obj,value)
      if (value > 0)
        obj.Prop1 = value;
      else
        error('Property value must be positive')
      end
    end
  end
end
```

有关属性 set 方法的示例，请参阅“将属性限制为特定值” （第 3-13 页）。

调用 set 方法时

如果存在属性 set 方法，则只要给该属性赋值，MATLAB 就会调用该方法。但是，在以下情况下，MATLAB 不会调用属性 set 方法：

- 从它自己的属性 set 方法中为属性赋值时不会调用，这样可以防止出现对 set 方法的递归调用。但是，由 set 方法调用的函数在进行属性赋值时会调用 set 方法。
- MATLAB 在调用对象构造函数之前，在对象初始化期间为属性赋值时。
- 当 MATLAB 复制值对象（任何不是 **handle** 的对象）时，MATLAB 在将属性值从一个对象复制到另一个对象时，不会调用 set 或 get 方法。
- 当属性的 **AbortSet** 特性为 **true** 时，如果为属性赋的值与其当前值相同，也不会调用 set 方法。有关此属性的详细信息，请参阅“Assignment When Property Value Is Unchanged”。

在构造函数中设置属性值

在构造函数中设置属性值会引发对属性 set 方法的调用。例如，**PropertySetMethod** 类定义 **Prop1** 属性的属性 set 方法。

```
classdef PropertySetMethod

  properties
    Prop1 = "Default String"
  end

  methods
    function obj = PropertySetMethod( str )
      if nargin > 0
        obj.Prop1 = str;
      end
    end

    function obj = set.Prop1(obj,str)
      obj.Prop1 = str;
      fprintf( 'set.Prop1 method called. Prop1 = %s\n', obj.Prop1 );
    end
  end
end
```

如果调用不带输入参数的类构造函数，MATLAB 则不会调用 **set.Prop1** 方法。

```
>> o = PropertySetMethod
```

```
o =

  PropertySetMethod with properties:

    Prop1: "Default String"
```

在构造函数中设置属性值会引发对属性 set 方法的调用。

```
>> o = PropertySetMethod("New string")

set.Prop1 method called. Prop1 = New string

o =

  PropertySetMethod with properties:

    Prop1: "New string"
```

如果将对象复制到另一个变量，MATLAB 不会调用属性 set 方法，即使赋值中的右侧对象使用属性的非默认值也是如此：

```
a = o;
a.Prop1

a.Prop1

ans =

    "New String"
```

另请参阅

相关示例

- “属性 get 方法” （第 8-30 页）
- “验证属性值” （第 8-12 页）

属性 get 方法

本节内容

属性访问方法概述

有关属性访问方法的概述，请参阅“属性访问方法” （第 8-23 页）。

属性 get 方法语法

每当查询属性值时，MATLAB 都会调用属性的 get 方法。

注意 您不能直接调用属性访问方法。MATLAB 会在您访问属性值时调用这些方法。

属性 get 方法使用以下语法，其中 **PropertyName** 是属性的名称。函数必须返回属性值。

```
methods
  function value = get.PropertyName(obj)
    ...
end
```

计算从属属性的值

SquareArea 类定义了一个从属属性 **Area**。MATLAB 不存储从属属性 **Area** 的值。当您查询 **Area** 属性的值时，MATLAB 调用 **get.Area** 方法，根据 **Width** 和 **Height** 属性来计算它的值。

```
classdef SquareArea
  properties
    Width
    Height
  end
  properties (Dependent)
    Area
  end
  methods
    function a = get.Area(obj)
      a = obj.Width * obj.Height;
    end
  end
end
```

get 方法不返回错误

MATLAB 默认对象显示会禁止显示从属性 get 方法返回的错误消息。MATLAB 不允许属性 get 方法发出的错误阻止整个对象的显示。

请使用属性 set 方法来验证属性值。在设置属性时对其值进行验证可确保对象处于有效状态。仅在要返回经 set 方法验证过的值时再使用属性 get 方法。

get 方法行为

在以下情况下，MATLAB 不调用属性 get 方法：

- 从属性自己的 get 方法中获取属性值。这样可避免对 get 方法的递归调用
- 复制值对象（即，不从 **handle** 类派生）时。将属性值从一个对象复制到另一个对象时，既不调用 set 方法也不调用 get 方法。

另请参阅

相关示例

- “从属属性的 set 和 get 方法” (第 8-32 页)

从属属性的 set 和 get 方法

本节内容

从属属性不存储数据。从属属性的值取决于一些其他值，例如非从属属性的值。

从属属性必须定义 get 访问方法 (**get.PropertyName**)，以便在查询属性时确定属性的值。

使用 **isequal** 和 **isequaln** 测试对象相等性时，不考虑从属属性 get 方法返回的值。

要能够设置从属属性的值，该属性必须定义 set 访问方法 (**set.PropertyName**)。属性 set 访问方法通常将值赋给另一个非从属属性来存储值。

例如，**Account** 类返回从属属性 **Balance** 的值，该值取决于 **Currency** 属性的值。在计算 **Balance** 属性的值之前，**get.Balance** 方法会先查询 **Currency** 属性。

在查询 **Balance** 属性时，MATLAB 才会调用 **get.Balance** 方法。您无法显式调用 **get.Balance**。

下面列出了显示从属属性及其 get 方法的类的部分代码：

```
classdef Account
  properties
    Currency
    DollarAmount
  end
  properties (Dependent)
    Balance
  end
  ...
  methods
    function value = get.Balance(obj)
      c = obj.Currency;
      switch c
        case 'E'
          v = obj.DollarAmount / 1.1;
        case 'P'
          v = obj.DollarAmount / 1.5;
        otherwise
          v = obj.DollarAmount;
      end
      format bank
      value = v;
    end
  end
end
```

计算从属属性值

属性 get 方法有一个用处是仅在需要时才确定属性的值，可以避免存储属性值。要使用这种方法，请将属性 **Dependent** 特性设置为 **true**：

```
properties (Dependent = true)
  Prop
end
```

Prop 属性的 get 方法确定该属性的值，并将其从方法中赋给对象：

```
function value = get.Prop(obj)
  value = calculateValue;
  ...
end
```

此 **get** 方法调用名为 **calculateValue** 的函数或静态方法来计算属性值，并返回 **value** 作为结果。属性 get 方法可以在方法中采取任何必要的操作来生成输出值。

有关属性 get 方法的示例，请参阅“按需计算数据” （第 3-14 页）。

何时对从属属性使用 set 方法

虽然从属属性不存储其值，但是，可以为从属属性定义 set 方法，以使代码能够设置属性。

例如，**propertyChange** 是一个值类，它将属性的名称从 **OldPropName** 更改为 **NewPropName**。您可以继续允许使用旧名称，而不将其公开给新用户。要支持旧属性名称，请使用 set 和 get 方法将 **OldPropName** 定义为从属属性。与非从属属性一样，值类中的 set 方法必须返回修改后的对象。

```
classdef propertyChange
   properties
      NewPropName
   end
   properties (Dependent, Hidden)
      OldPropName
   end

   methods
      function obj = set.OldPropName(obj,val)
         obj.NewPropName = val;
      end
      function value = get.OldPropName(obj)
         value = obj.NewPropName;
      end
   end
end
```

这样，同时存储新旧属性值便不会浪费内存。访问 **OldPropName** 的代码继续按预期工作。设置 **OldPropName** 的 **Hidden** 特性会阻止新用户查看该属性。

属性 set 方法进行的赋值运算会使系统执行为要设置的属性定义的所有 set 方法。有关示例，请参阅“按需计算数据” （第 3-14 页）。

具有从属属性的私有 set 访问权限

如果仅使用从属属性返回值，则不要为从属属性定义 set 访问方法。此时，请将从属属性的 **SetAccess** 特性设置为 **private**。以 **MaxValue** 属性的以下 get 方法为例：

```
methods
  function mval = get.MaxValue(obj)
    mval = max(obj.BigArray(:));
```

```
  end
end
```

此示例使用 **MaxValue** 属性返回仅在查询时才计算的值。对于此用法，请将 **MaxValue** 属性定义为 dependent 和 private：

```
properties (Dependent, SetAccess = private)
  MaxValue
end
```

另请参阅

相关示例

- “属性特性” (第 8-7 页)

动态属性 - 向实例添加属性

本节内容
“什么是动态属性” （第 8-35 页）
“定义动态属性” （第 8-35 页）
“列出对象动态属性” （第 8-37 页）

什么是动态属性

您可以将属性添加到从 **dynamicprops** 类派生的类的实例中。这些动态属性有时称为实例属性。您可以使用动态属性将临时数据附加到对象，或用它指定要与某个类实例（而非该类的所有对象）相关联的数据。

多个程序可以对同一个对象定义动态属性。在这些情况下，请避免名称冲突。动态属性名称必须为有效的 MATLAB 标识符（请参阅“变量名称”），并且不能与类的方法同名。

动态属性的特征

一旦定义，动态属性的行为就与类定义的属性相似：

- 使用圆点表示法设置和查询动态属性的值。（请参阅“将数据赋给动态属性” （第 8-36 页）。）
- MATLAB 在您保存和加载动态属性所附加到的对象时保存和加载这些动态属性。（请参阅“Dynamic Properties and ConstructOnLoad”。）
- 定义动态属性的特性。（请参阅“设置动态属性特性” （第 8-36 页）。）
- 默认情况下，动态属性将其 **NonCopyable** 特性设置为 **true**。如果复制包含动态属性的对象，则不会复制动态属性。（请参阅“Objects with Dynamic Properties”。）
- 添加属性 set 和 get 访问方法。（请参阅“Set and Get Methods for Dynamic Properties”。）
- 侦听动态属性事件。（请参阅“Dynamic Property Events”。）
- 使用受限语法从对象数组访问动态属性值。（请参阅“Accessing Dynamic Properties in Arrays”。）
- 在比较具有动态属性的对象时，即使属性具有相同的名称和值，**isequal** 函数也始终返回 **false**。要比较包含动态属性的对象，请为您的类重载 **isequal**。

定义动态属性

作为 **dynamicprops** 类（它本身也是 **handle** 类的子类）的子类的任何类都可以使用 **addprop** 方法定义动态属性。语法为：

```
P = addprop(H,'PropertyName')
```

其中：

P 是 **meta.DynamicProperty** 对象数组

H 是句柄数组

PropertyName 是您要添加到每个对象的动态属性的名称

命名动态属性

在命名动态属性时，请仅使用有效的名称（请参阅“变量名称”）。此外，不要使用以下名称：

- 与类方法的名称相同的名称
- 与类事件的名称相同的名称
- 包含句点 (.) 的名称
- 支持数组功能的函数的名称：**empty**、**transpose**、**ctranspose**、**permute**、**reshape**、**display**、**disp**、**details** 或 **sort**。

设置动态属性特性

要设置属性特性，请使用与动态属性关联的 **meta.DynamicProperty** 对象。例如，如果 **P** 是 **addprop** 返回的对象，则以下语句会将该属性的 **Hidden** 特性设置为 **true**：

```
P.Hidden = true;
```

属性特性 **Constant** 和 **Abstract** 对动态属性没有任何意义。将这些属性的值设置为 **true** 将不起作用。

删除动态属性

通过删除动态属性的 **meta.DynamicProperty** 对象来删除动态属性：

```
delete(P);
```

将数据赋给动态属性

假设您正在使用一组预定义的用户界面小组件类（按钮、滑块、复选框等）。您要存储小组件类的每个实例的位置。假设小组件类不是为存储特定布局方案的位置数据而设计的。您不想创建映射或哈希表来单独维护这些信息。

假设 **button** 类是 **dynamicprops** 的子类，通过添加动态属性来存储布局数据。以下是一个用于创建 **uicontrol** 按钮的简单类：

```
classdef button < dynamicprops
  properties
    UiHandle
  end
  methods
    function obj = button(pos)
      if nargin > 0
        if length(pos) == 4
          obj.UiHandle = uicontrol('Position',pos,...
            'Style','pushbutton');
        else
          error('Improper position')
        end
      end
    end
  end
end
```

创建 **button** 类的实例，添加动态属性，并设置其值：

```
b1 = button([20 40 80 20]);
b1.addprop('myCoord');
b1.myCoord = [2,3];
```

像访问任何其他属性一样访问动态属性，但只需访问您定义的对象即可：

```
b1.myCoord
```

```
ans =

   2    3
```

动态属性的 Access 特性

不推荐对动态属性使用非公共 Access 特性，因为这些属性大多属于在类方法之外创建的特定实例。动态属性的 Access 特性应用于包含该动态属性的实例所在的类。动态属性 Access 特性不一定应用于添加动态属性时所用方法所属的类。

例如，如果一个基类方法添加了一个对某实例具有私有访问权限的动态属性，则私有访问权限将仅应用于该实例所在的类。

有关动态属性特性的详细信息，请参阅 **meta.DynamicProperty**。使用句柄 **findprop** 方法获取 **meta.DynamicProperty** 对象。

列出对象动态属性

您可以使用句柄 **findprop** 方法来列出对象的动态属性。步骤如下：

- 使用 **properties** 函数获取对象属性的名称。
- 使用 **findprop** 方法获取每个属性的元数据对象。
- 使用 **isa** 函数确定元数据对象是否为 **meta.DynamicProperty** 对象。如果是，则该属性为动态属性。

getDynamicPropNames 函数展示了如何显示为输入 **obj** 定义的任何动态属性的名称。

```
function getDynamicPropNames(obj)
   % Find dynamic properties
   allprops = properties(obj);
   for i=1:numel(allprops)
      m = findprop(obj,allprops{i});
      if isa(m,'meta.DynamicProperty')
         disp(m.Name)
      end
   end
end
```

另请参阅

相关示例

- “Set and Get Methods for Dynamic Properties”
- “Dynamic Property Events”
- “Dynamic Properties and ConstructOnLoad”

方法 - 定义类操作

类设计中的方法

本节内容
“类方法” （第 9-2 页）
“示例和语法” （第 9-2 页）
“方法的种类” （第 9-2 页）
“方法命名” （第 9-3 页）

类方法

方法是一些函数，用于实现对类的对象执行一些操作。方法及其他类成员均支持封装的概念 - 通过类实例在属性中包含数据，再通过类方法对这些数据进行操作。这种设计允许对类外部的代码隐藏类的内部工作，这样，对类实现做任何更改便不会影响到类外部的代码。

方法可以访问其类的私有成员，包括其他方法和属性。这种封装使您能够隐藏数据并创建一些特殊接口，专门用来访问存储在对象中的数据。

示例和语法

有关如何编写类的入门示例，请参阅“创建简单类” （第 2-2 页）。

有关示例代码和语法，请参阅“方法语法” （第 5-6 页）。

有关如何创建修改标准 MATLAB 行为的类的讨论，请参阅 “Methods That Modify Default Behavior”。

有关使用 @ 和路径目录以及包来组织类文件的信息，请参阅“类文件组织”。

有关在多个文件中定义类时使用的语法，请参阅“在单独文件中定义方法” （第 9-6 页）。

方法的种类

方法可分为几种专门的类别，它们可以执行特定功能或具有特定的行为：

- 普通方法：作用于一个或多个对象并返回一些新对象或一些计算值的函数。这些方法就像普通的 MATLAB 函数一样，无法修改输入参数。普通方法使类能够实现算术运算符和计算函数。这些方法需要依托类对象来执行操作。请参阅“Ordinary Methods”。
- 构造函数方法：创建类的对象的特化方法。构造函数方法必须与类同名，并且通常使用从输入参数获取的数据初始化属性值。类构造函数方法必须声明至少一个输出参数，即正在构造的对象。第一个输出始终是正在构造的对象。请参阅“类构造函数方法” （第 9-12 页）。
- 析构函数方法：当对象被销毁时（例如，如果您调用 **delete(object)**，或者不再有对该对象的任何引用），会自动调用此方法。请参阅“句柄类析构函数” （第 7-8 页）。
- 属性访问方法：使类能够定义在查询或设置属性值时要执行的代码的方法。请参阅“属性 set 方法” （第 8-27 页）。
- 静态方法：与类相关联，但不一定对类对象进行操作的函数。这些方法不要求在调用方法期间引用类的实例，但通常以特定于类的方式执行操作。请参阅“静态方法” （第 9-19 页）。

- 转换方法：来自其他类的重载构造函数方法，使您的类能够将其自己的对象转换为重载构造函数的类。例如，如果您的类实现了一个 **double** 方法，则可调用此方法（而不是 double 类构造函数）来将您的类对象转换为 MATLAB double 对象。有关详细信息，请参阅“Object Converters”。
- 抽象方法：用来定义不能通过自身进行实例化，但可用来定义公共接口，供许多子类使用的类。包含抽象方法的类通常称为接口。有关详细信息和示例，请参阅“抽象类和类成员” （第 12-16 页）。

方法命名

实现方法的函数的名称可以包含圆点（例如，**set.PropertyName**），但仅限以下方法：

- 属性 set/get 访问方法（请参阅“属性 set 方法” （第 8-27 页））
- 转换为包限定类的转换方法，这需要使用包名称（请参阅“包命名空间” （第 6-13 页））

您不能将属性访问或转换方法定义为局部函数、嵌套函数，也不能在它们自己的文件中单独定义。类构造函数和以包为作用域的函数必须在函数定义中使用非限定名称；不要在函数定义语句中包含包名称。

另请参阅

相关示例

- “方法特性” （第 9-4 页）
- “避免冲突的命名规则” （第 9-22 页）

方法特性

本节内容

方法特性的目的

在类定义中指定特性使您能够为特定目的自定义方法的行为。通过设置方法特性来控制访问、可见性和实现等特征。子类不继承超类成员特性。

指定方法特性

在 **methods** 关键字所在的同一行上指定方法特性：

```
methods (Attribute1 = value1, Attribute2 = value2,...)
  ...
end
```

方法特性表

特性使您能够修改方法的行为。所有方法都支持下表中列出的特性。

特性值会应用于 **methods...end** 代码块中定义的所有指定了非默认值的方法。

方法特性

特性名称	类	描述
Abstract	**logical** 默认值为 **false**	如果设置为 **true**，则此方法没有实现。此方法的语法行中可以包含参数，子类将在实现方法时使用这些参数： • 子类不必定义与父类相同数量的输入参数和输出参数。不过，子类在实现其方法版本时通常使用相同的签名。 • 此方法不包含 **function** 或 **end** 关键字，仅包含函数语法（例如 **[a,b] = myMethod(x,y)**）。 • 该方法可以在签名行后包括注释。

Access	• 枚举，默认值为 **public**
	• **meta.class** 对象
	• **meta.class** 对象的元胞数组
		确定哪些代码可以调用此方法：
		• **public** - 不受限制的访问
		• **protected** - 从类或子类中的方法进行访问
		• **private** - 仅通过类方法（而非从子类）进行访问
		• 列出有权限访问此方法的类。采用以下格式将类指定为 **meta.class** 对象：
		  • 单个 **meta.class** 对象
		  • **meta.class** 对象元胞数组。空元胞数组 **{}** 与 **private** 访问权限相同。
		请参阅“类成员访问” （第 12-10 页）

Hidden	**logical** 默认值为 **false**	设置为 **false** 时，方法名称将显示在使用 **methods** 或 **methodsview** 命令显示的方法列表中。如果设置为 **true**，则方法名称不会包含在这些列表中且 **ismethod** 不会为方法名称返回 **true**。
Sealed	**logical** 默认值为 **false**	如果设置为 **true**，则无法在子类中重新定义方法。尝试在子类中定义同名的方法将会导致错误。
Static	**logical** 默认值为 **false**	指定为 **true** 可定义不依赖于类对象并且不需要对象参数的方法。使用类名调用方法：**classname.methodname**，或使用类实例调用方法：**obj.methodname** “静态方法” （第 9-19 页）提供了更多信息。
框架特性	使用特定框架基类的类具有特定于框架的特性。有关这些特性的信息，请参阅您正在使用的特定基类的文档。	

另请参阅

metaclass | **meta.method**

详细信息

- “方法”

在单独文件中定义方法

本节内容
“类文件夹” （第 9-6 页）
“在函数文件中定义方法” （第 9-6 页）
“在 classdef 文件中指定方法特性” （第 9-7 页）
“必须在 classdef 文件中定义的方法” （第 9-7 页）

类文件夹

您可以在不同于类定义文件的文件中定义类方法，但有某些例外（请参阅“必须在 classdef 文件中定义的方法” （第 9-7 页））。

要为类定义使用多个文件，请将类文件放在文件夹中，该文件夹的名称以 @ 字符开头，后跟类的名称（这称为类文件夹）。确保类文件夹的父文件夹位于 MATLAB 路径上。

如果类文件夹包含在一个或多个包文件夹中，则顶层包文件夹必须位于 MATLAB 路径上。

例如，文件夹 **@MyClass** 必须包含文件 **MyClass.m**（其中包含 **classdef** 代码块），还包含其他方法和函数（定义在扩展名为 **.m** 的文件中）。文件夹 **@MyClass** 可以包含多个文件：

```
@MyClass/MyClass.m
@MyClass/subsref.m
@MyClass/subsasgn.m
@MyClass/horzcat.m
@MyClass/vertcat.m
@MyClass/myFunc.m
```

方法文件的类型

MATLAB 将类文件夹中的任何函数文件均视为类的方法。函数文件可以是 MATLAB 代码 (**.m**)、实时代码文件格式 (**.mlx**)、MEX 函数（平台相关的扩展名）和 P 代码文件 (**.p**)。文件的基本名称必须为有效的 MATLAB 函数名称。有效的函数名称以字母字符开头，并且可以包含字母、数字或下划线。

有关将方法定义为 C++ MEX 函数的信息，请参阅“Using MEX Functions for MATLAB Class Methods”。

在函数文件中定义方法

要在类文件夹的单独文件中定义方法，请在一个文件中创建函数。不要在该文件中使用 **method-end** 关键字。像任何函数一样，用函数名称命名文件。

在 **myFunc.m** 文件中，实现以下方法：

```
function output = myFunc(obj,arg1,arg2)
  ...% code here
end
```

在方法代码块中声明 **classdef** 文件中的函数签名是一个不错的做法：

```
classdef MyClass
  methods
```

```
    output = myFunc(obj,arg1,arg2)
  end
  ...
end
```

在 classdef 文件中指定方法特性

如果为在单独的函数文件中定义的方法指定方法特性，请将方法签名包含在 **classdef** 文件的 **methods** 代码块中。此方法代码块指定应用于方法的特性。

例如，以下代码显示在 **methods** 代码块中将 **Access** 设置为 **private** 的方法。方法实现位于单独的文件中。不要在 **methods** 代码块中包含 **function** 或 **end** 关键字。仅包括显示输入和输出参数的函数签名。

```
classdef MyClass
  methods (Access = private)
    output = myFunc(obj,arg1,arg2)
  end
end
```

在 **@MyClass** 文件夹中名为 **myFunc.m** 的文件中定义函数：

```
function output = myFunc(obj,arg1,arg2)
  ...
end
```

在单独文件中定义静态方法

要创建静态方法，请将方法 **Static** 属性设置为 **true**，并在 **classdef** 文件的静态方法代码块中列出函数签名。在函数名称中包含输入和输出参数。例如：

```
classdef MyClass
...
  methods (Static)
    output = staticFunc1(arg1,arg2)
    staticFunc2
  end
  ...
end
```

使用相同的函数签名在单独的文件中定义函数。例如，在文件 **@MyClass/staticFunc1.m** 中：

```
function output = staticFunc1(arg1,arg2)
  ...
end
```

在 **@Myclass/staticFunc2.m** 中：

```
function staticFunc2
  ...
end
```

必须在 classdef 文件中定义的方法

在 **classdef** 文件中定义以下方法。您不能在单独文件中定义这些方法：

- 类构造函数

- 所有在其名称中使用圆点的函数，包括：
  - 转换器方法，这些方法必须使用包名称作为类名的一部分，因为类包含在包中
  - 属性 set 和 get 访问方法

相关信息

- “Converters for Package Classes”
- “属性 set 方法” （第 8-27 页）

另请参阅

相关示例

- “包含类定义的文件夹” （第 6-8 页）
- “实时代码文件格式 (.mlx)”
- “调用 MEX 函数”
- “Using MEX Functions for MATLAB Class Methods”
- “保护源代码的安全考虑事项”

方法调用

本节内容

MATLAB 类支持圆点语法和函数语法来调用方法。本主题演示这两种语法，并描述 MATLAB 如何确定要调用的方法。

圆点语法和函数语法

要使用一个参数 **arg** 调用非静态方法，其中 **obj** 是定义该方法的类的对象，请使用圆点语法或函数语法。

```
obj.methodName(arg)
methodName(obj,arg)
```

例如，**dataSetSummary** 存储一组数值数据以及该数据的均值、中位数和范围。该类定义两个方法：**showDataSet** 显示存储在 data 属性中的当前数据，而 **newDataSet** 替换 **data** 的当前值并计算该数据的均值、中位数和范围。

```
classdef dataSetSummary
   properties (SetAccess=private)
      data {mustBeNumeric}
      dataMean
      dataMedian
      dataRange
   end

   methods
      function showDataSet(obj)
         disp(obj.data)
      end
      function obj = newDataSet(obj,inputData)
         obj.data = inputData;
         obj.dataMean = mean(inputData);
         obj.dataMedian = median(inputData);
         obj.dataRange = range(inputData);
      end
   end
end
```

创建 **dataSetSummary** 的一个实例，并调用 **newDataSet** 来向对象添加数据。使用圆点语法调用 **newDataSet**。由于 **dataSetSummary** 是值类，请将结果赋回原始变量以保留更改。

```
a = dataSetSummary;
a = a.newDataSet([1 2 3 4])
```

```
a =

  dataSetSummary with properties:

       data: [1 2 3 4]
   dataMean: 2.5000
 dataMedian: 2.5000
  dataRange: 3
```

调用 **showDataSet** 方法，但对此调用使用函数语法。

```
showDataSet(a)
```

```
    1    2    3    4
```

使用表达式引用方法名称

通过将表达式括在括号中，您可以动态调用类方法。

```
obj.(expression)
```

表达式的计算结果必须为作为方法名称的字符向量或字符串。例如，对于属于类 **dataSetSummary** 的对象 **a**，以下两条语句是等效的。

```
a.showDataSet
a.("showDataSet")
```

当与函数语法结合使用时，这种方法不起作用。

对方法调用的结果进行索引

对于任何方法，如果针对其返回值定义了点索引方法，例如对象属性或结构体字段名称，那么就可对该方法的结果进行点索引。例如，向 **dataSetSummary** 类添加新方法 **returnSummary**，该方法返回结构体中的所有存储数据。

```
function outStruct = returnSummary(obj)
  outStruct = struct("Data",obj.data,...
                     "Mean",obj.dataMean,...
                     "Median",obj.dataMedian,...
                     "Range",obj.dataRange);
end
```

调用 **returnSummary** 并使用点索引来返回数据集的中位数。

```
a.returnSummary.Median
```

```
ans =

    2.5000
```

有关对函数调用结果进行索引的详细信息，请参阅“对函数调用结果进行索引”。

确定调用哪个方法

当使用圆点语法调用方法时，MATLAB 调用由圆点左侧的对象的类定义的方法。例如，如果 **classA** 和 **classB** 都定义了名为 **plus** 的方法，则以下代码始终调用由 **classA** 定义的 **plus** 方法。

```
A = classA;
B = classB;
A.plus(B)
```

不考虑其他参数。从不调用其他参数的方法，也不调用函数。

在其他语法中，MATLAB 必须确定在给定情况下调用运算符或函数的多个版本中的哪个版本。默认行为是调用与最左侧的参数相关联的方法。在以下两个语句中，都调用由 **classA** 定义的 **plus** 方法。

```
objA + objB
plus(objA,objB)
```

不过，当一个对象优先于另一个对象时，此默认行为可以更改。

对象优先级

根据类的定义方式，在方法调度时，这些类的对象可以优先于其他对象：

- 用 **classdef** 语法定义的类优先于以下 MATLAB 类：

  double、**single**、**int64**、**uint64**、**int32**、**uint32**、**int16**、**uint16**、**int8**、**uint8**、**char**、**string**、**logical**、**cell**、**struct** 和 **function_handle**。
- 用 **classdef** 语法定义的类可以使用 **InferiorClasses** 属性指定它们相对于其他类的优先级。

在“Representing Polynomials with Classes”中，**DocPolynom** 类定义 **plus** 方法，该方法对 **DocPolynom** 对象执行加法运算。构造一个 **DocPolynom** 实例。

```
p = DocPolynom([1 0 -2 -5])

p =
   x^3 - 2*x - 5
```

下面的语句向 **DocPolynom** 实例加上一个双精度值。尽管 **double** 是 **1 + p** 中最左侧的参数，但 **DocPolynom** 类优先于内置的 **double** 类。以下代码调用 **DocPolynom plus** 方法来加上多项式。

```
1 + p

ans =
   x^3 - 2*x - 4
```

您还可以通过在类属性中列出次优先级的类来指定用 **classdef** 语法定义的类的相对优先级。**InferiorClasses** 属性赋予所定义的类比作为该属性的参数列出的类更高的优先级。在 **classdef** 语句中定义 **InferiorClasses** 属性：

```
classdef (InferiorClasses = {?class1,?class2}) myClass
```

此属性建立所定义的类的相对优先级，并按照列出的类的顺序排列。有关详细信息，请参阅“Class Precedence”。

另请参阅

详细信息

- “类属性” （第 6-5 页）
- “方法特性” （第 9-4 页）
- “Class Precedence”
- “对子类对象调用超类方法” （第 5-10 页）

类构造函数方法

本节内容

类构造函数方法的目的

构造函数方法是一个创建类实例的特殊函数。通常，构造函数方法接受输入参数来指定存储在属性中的数据，并返回初始化的对象。

有关基本示例，请参阅“创建简单类” （第 2-2 页）。

未显式定义任何类构造函数的 MATLAB 类有默认的构造函数方法。此方法返回在没有输入参数的情况下创建的类的对象。类可以定义覆盖默认构造函数的构造函数方法。显式定义的构造函数可以接受输入参数、初始化属性值、调用其他方法以及执行创建类对象所需的其他操作。

构造函数方法的基本结构

构造函数方法的结构包含三个基本部分：

- 初始化前 - 计算超类构造函数的参数。
- 对象初始化 - 调用超类构造函数。
- 初始化后 - 执行任何与子类相关的操作，包括对象的引用和赋值、调用类方法、将对象传递给函数等。

以下代码说明在每个部分执行的基本操作：

```
classdef ConstructorDesign < BaseClass1
  properties
    ComputedValue
  end
  methods
    function obj = ConstructorDesign(a,b,c)

      %% Pre Initialization %%
      % Any code not using output argument (obj)
      if nargin == 0
```

```
        % Provide values for superclass constructor
        % and initialize other inputs
        a = someDefaultValue;
        args{1} = someDefaultValue;
        args{2} = someDefaultValue;
      else
        % When nargin ~= 0, assign to cell array,
        % which is passed to supclass constructor
        args{1} = b;
        args{2} = c;
      end
      compvalue = myClass.staticMethod(a);

      %% Object Initialization %%
      % Call superclass constructor before accessing object
      % You cannot conditionalize this statement
      obj = obj@BaseClass1(args{:});

      %% Post Initialization %%
      % Any code, including access to object
      obj.classMethod(arg);
      obj.ComputedValue = compvalue;
      ...
    end
  ...
  end
...
end
```

像调用任何函数一样调用构造函数，传递参数并返回类的对象。

```
obj = ConstructorDesign(a,b,c);
```

构造函数的指导原则

- 构造函数与类同名。
- 构造函数可以返回多个参数，但第一个输出必须为创建的对象。
- 如果不想对输出参数赋值，可以在构造函数中清除对象变量（请参阅“禁止显示输出对象” （第 9-18 页））。
- 如果创建类构造函数，请确保可以在没有输入参数的情况下调用它。请参阅“构造函数不要求输入参数的情况” （第 9-15 页）。
- 如果构造函数对超类构造函数进行显式调用，则该调用必须发生在对构造对象的任何其他引用之前，并且不能发生在 `return` 语句后。
- 对超类构造函数的调用不能为条件调用。您不能在循环、条件、switch、try/catch 或嵌套函数中放置超类构造调用。有关详细信息，请参阅“对超类构造函数的无条件调用” （第 9-16 页）。

默认构造函数

如果类没有定义构造函数，则 MATLAB 提供默认构造函数，它不接受任何参数，并返回一个标量对象，该对象的属性初始化为属性的默认值。由 MATLAB 提供的默认构造函数也调用所有超类构造函数，其中可以不将任何参数传递给默认子类构造函数，也可以将任何参数传递给默认子类构造函数。

当子类没有定义构造函数时，默认构造函数将其输入传递给直接超类构造函数。当子类不需要定义构造函数而超类构造函数需要输入参数时，此行为很有用。

何时定义构造函数

当需要执行默认构造函数无法执行的对象初始化时，可以定义构造函数方法。例如，当创建类的对象需要：

- 输入参数时
- 初始化类的每个实例的对象状态（例如属性值）时
- 用子类构造函数确定的值调用超类构造函数时

相关信息

有关构造枚举的特定信息，请参阅“枚举类构造函数调用顺序” （第 14-4 页）。

有关在构造函数中创建对象数组的信息，请参阅“构造对象数组” （第 10-2 页）。

如果正在创建的类是子类，则 MATLAB 会调用每个超类的构造函数来初始化对象。对超类构造函数的隐式调用不带参数。如果超类构造函数需要参数，请从子类构造函数显式调用它们。请参阅“Control Sequence of Constructor Calls”

初始化构造函数中的对象

构造函数方法将初始化的对象以输出参数形式返回。输出参数是在构造函数执行时，在执行第一行代码之前创建的。

例如，以下构造函数可以将为对象的属性 **A** 的赋值作为第一个语句，因为已将对象 **obj** 指定给 **MyClass** 的一个实例。

```
function obj = MyClass(a,b,c)
  obj.A = a;
  ...
end
```

您可以从构造函数调用其他类方法，因为对象已初始化。

构造函数还会创建一个对象，该对象的属性具有默认值 - 空值 (**[]**) 或属性定义代码块中指定的默认值。

例如，以下构造函数对输入参数进行运算，以为 **Value** 属性赋值。

```
function obj = MyClass(a,b,c)
  obj.Value = (a + b) / c;
  ...
end
```

在构造函数中引用对象

在初始化对象（例如，通过给属性赋值）时，使用输出参数的名称来引用构造函数中的对象。例如，在以下代码中，输出参数是 **obj**，而对象也被引用为 **obj**：

```
% obj is the object being constructed
function obj = MyClass(arg)
  obj.propert1 = arg*10;
  obj.method1;
  ...
end
```

有关定义默认属性值的详细信息，请参阅“Define Properties with Default Values”。

构造函数不要求输入参数的情况

在某些情况下，必须能够在没有输入参数的情况下调用构造函数：

- 将对象加载到工作区时，如果将类的 **ConstructOnLoad** 属性设置为 **true**，则 **load** 函数将不带参数调用类构造函数。
- 在创建或扩展对象数组时，如果并非所有元素都有特定值，则将以不带参数的方式调用类构造函数来填充未指定的元素（例如，**x(10,1) = MyClass(a,b,c);**）。在这种情况下，将不带参数调用构造函数一次，以用此对象的副本填充空数组元素 (**x(1:9,1)**)。

如果没有输入参数，构造函数仅使用默认属性值创建对象。最好在类构造函数中添加零参数检查，以防止在出现这两种情况时报错：

```
function obj = MyClass(a,b,c)
  if  nargin > 0
    obj.A = a;
    obj.B = b;
    obj.C = c;
    ...
  end
end
```

有关处理超类构造函数的方法，请参阅“构造函数方法的基本结构” （第 9-12 页）。

子类构造函数

子类构造函数可以显式调用超类构造函数来将参数传递给超类构造函数。子类构造函数必须在对超类构造函数的调用中指定这些参数，并且必须使用构造函数输出参数来进行调用。语法如下：

```
classdef MyClass < SuperClass
  methods
    function obj = MyClass(a,b,c,d)
      obj@SuperClass(a,b);
      ...
    end
  end
end
```

子类构造函数对超类构造函数的所有调用都必须发生在对 (**obj**) 对象的任何其他引用之前。此限制包括为属性赋值或调用普通类方法。此外，子类构造函数只能调用超类构造函数一次。

仅引用指定的超类

如果 **classdef** 没有将类指定为超类，则构造函数将无法使用此语法来调用超类构造函数。也就是说，子类构造函数只能调用 **classdef** 行中列出的直接超类构造函数。

```
classdef MyClass < SuperClass1 & SuperClass2
```

MATLAB 按从左到右的顺序调用任何未调用的构造函数，这些构造函数在 **classdef** 行中指定。MATLAB 对这些调用不传递任何参数。

对超类构造函数的无条件调用

对超类构造函数的调用必须为无条件调用。对给定超类只能调用一次。请在使用对象之前调用超类构造函数，以初始化对象的超类部分（例如，为属性赋值或调用类方法）。

要使用依赖于某些条件的不同参数调用超类构造函数，请构建参数的元胞数组，并提供对该构造函数的一次调用。

例如，当不带参数调用 **Cube** 构造函数时，**Cube** 类构造函数将使用默认值调用超类 **Shape** 构造函数。如果使用四个输入参数调用 **Cube** 构造函数，则会将 **upvector** 和 **viewangle** 传递给超类构造函数：

```
classdef Cube < Shape
  properties
    SideLength = 0
    Color = [0 0 0]
  end
  methods
    function cubeObj = Cube(length,color,upvector,viewangle)
      % Assemble superclass constructor arguments
      if nargin == 0
        super_args{1} = [0 0 1];
        super_args{2} = 10;
      elseif nargin == 4
        super_args{1} = upvector;
        super_args{2} = viewangle;
      else
        error('Wrong number of input arguments')
      end

      % Call superclass constructor
      cubeObj@Shape(super_args{:});

      % Assign property values if provided
      if nargin > 0
        cubeObj.SideLength = length;
        cubeObj.Color = color;
      end
      ...
    end
  ...
  end
end
```

零个或多个超类参数

要支持不带参数调用超类构造函数的语法，请显式提供以下语法。

假设在 **Cube** 类示例中，**Shape** 超类和 **Cube** 子类中的所有属性值都在类定义中指定了默认值。然后，您可以创建 **Cube** 的实例，而无需为超类或子类构造函数指定任何参数。

下面说明您如何在 **Cube** 构造函数中实现此行为：

```
methods
  function cubeObj = Cube(length,color,upvector,viewangle)
    % Assemble superclass constructor arguments
    if nargin == 0
```

```
      super_args = {};
    elseif nargin == 4
      super_args{1} = upvector;
      super_args{2} = viewangle;
    else
      error('Wrong number of input arguments')
    end

    % Call superclass constructor
    cubeObj@Shape(super_args{:});

    % Assign property values if provided
    if nargin > 0
      cubeObj.SideLength = length;
      cubeObj.Color = color;
    end
  ...
  end
end
```

有关子类的详细信息

有关创建子类的信息，请参阅“设计子类构造函数” （第 12-4 页）。

对继承的构造函数的隐式调用

MATLAB 将参数从默认子类构造函数隐式传递给超类构造函数。这样，就无需仅为了将参数传递给超类构造函数，而为子类实现一个构造函数方法。

例如，以下类构造函数需要输入参数（**datetime** 对象），该参数由构造函数指定给 **CurrentDate** 属性。

```
classdef BaseClassWithConstr
  properties
    CurrentDate datetime
  end
  methods
    function obj = BaseClassWithConstr(dt)
      obj.CurrentDate = dt;
    end
  end
end
```

假设您创建了 **BaseClassWithConstr** 的一个子类，但是，您的子类不需要显式构造函数方法。

```
classdef SubclassDefaultConstr < BaseClassWithConstr
  ...
end
```

您可以通过用超类参数调用默认构造函数来构造 **SubclassDefaultConstr** 的对象：

```
obj = SubclassDefaultConstr(datetime);
```

有关子类构造函数的信息，请参阅“子类构造函数” （第 9-15 页）和“默认构造函数” （第 9-13 页）。

类构造过程中的错误

对于句柄类，在下列情况下出现错误时，MATLAB 会调用 **delete** 方法：

- 在错误发生前，代码中存在对对象的引用。
- 在错误发生前，代码中存在较早的 **return** 语句。

MATLAB 对对象调用 **delete** 方法，对属性中包含的任何对象调用 **delete** 方法，对任何初始化的基类调用 **delete** 方法。

根据错误发生的时间，MATLAB 可以在对象完全构造之前调用类析构函数。因此，类 **delete** 方法必须能够对部分构造的对象进行操作，这些对象可能没有所有属性的值。有关详细信息，请参阅“支持销毁部分构造的对象” （第 7-9 页）。

有关如何销毁对象的信息，请参阅“句柄类析构函数” （第 7-8 页）。

禁止显示输出对象

当在对构造函数的调用中没有指定输出变量时，可以禁止将类实例赋给 **ans** 变量。这种编程方式对于创建图形界面窗口的 App 非常有用，这些窗口会保留构造的对象。这些 App 不需要返回对象。

使用 **nargout** 来确定是否带输出参数调用了构造函数。例如，如果调用时未指定输出，则 **MyApp** 类的类构造函数会清除对象变量 **obj**：

```
classdef MyApp
  methods
    function obj = MyApp
      ...
      if nargout == 0
        clear obj
      end
    end
    ...
  end
end
```

当类构造函数不返回对象时，MATLAB 不会触发 **meta.class InstanceCreated** 事件。

另请参阅

相关示例

- “用构造函数简化接口” （第 3-14 页）
- “子类构造函数实现” （第 12-5 页）

静态方法

本节内容

什么是静态方法

静态方法与类相关联，但不与该类的特定实例相关联。这些方法不需要类的对象作为输入参数。因此，您可以在不创建类对象的情况下调用静态方法。

为什么定义静态方法

当您不想在执行某些代码之前创建类的实例时，静态方法非常有用。例如，假设您要设置 MATLAB 环境，或者使用静态方法计算创建类实例所需的数据。

假设类需要根据特定容差计算的 pi 值。该类可以定义自己版本的内置 **pi** 函数，以便在该类中使用。这种方式可以保持类内部工作的封装，但不需要类的实例返回值。

定义静态方法

要将方法定义为静态，请将方法代码块 **Static** 属性设置为 **true**。例如：

```
classdef MyClass
  methods(Static)
    function p = pi(tol)
      [n d] = rat(pi,tol);
      p = n/d;
    end
  end
end
```

调用静态方法

调用静态方法时，使用类的名称、后跟圆点 (.)，然后是方法的名称：

```
classname.staticMethodName(args,...)
```

调用上一节中 **MyClass** 的 **pi** 方法需要以下语句：

```
value = MyClass.pi(.001);
```

您也可以像调用其他任何方法一样，使用类的实例调用静态方法：

```
obj = MyClass;
value = obj.pi(.001);
```

继承静态方法

子类可以重新定义静态方法，除非该方法的 **Sealed** 属性在超类中也设置为 **true**。

另请参阅

相关示例

- “实现 AccountManager 类” (第 3-9 页)

在类定义中重载函数

本节内容

为什么重载函数

类可以通过实现同名的方法来重新定义 MATLAB 函数。在定义行为与现有 MATLAB 类型相似的特化类型时，重载非常有用。例如，您可以实现关系运算、绘图函数和其他常用的 MATLAB 函数来处理类中的对象。

您还可以通过实现控制默认行为的特定函数来修改这些行为。有关修改默认行为的函数的详细信息，请参阅“Methods That Modify Default Behavior”。

实现重载的 MATLAB 函数

类方法可以提供仅对类实例运行的 MATLAB 函数的实现。这种限制是允许的，因为 MATLAB 可以始终识别对象属于哪个类。

MATLAB 使用主导参数来确定要调用函数的哪个版本。如果主导参数是对象，则 MATLAB 调用由该对象的类定义的方法（如果存在的话）。

如果类定义了一个与全局函数同名的方法，则该类对该函数的实现称为重载原始全局实现。

要重载 MATLAB 函数，请执行以下操作：

- 定义与要重载的函数同名的方法。
- 确保方法参数列表接受类的对象，MATLAB 使用该对象来确定调用哪个版本。
- 执行方法中的必要步骤来实现该函数。例如，访问用于操作数据的对象属性。

通常，重载函数的方法会产生类似于 MATLAB 函数的结果。但是，关于如何实现重载方法不做要求。重载方法不需要与重载函数的签名相匹配。

注意 MATLAB 不支持在同一类中定义多个具有相同名称但具有不同签名的方法。

重载 bar 函数

重载常用函数来处理类的对象可为我们提供方便。例如，假设有一个类定义了一个属性，用来存储您经常绘制的数据。**MyData** 类覆盖 **bar** 函数，并在图中添加标题：

```
classdef MyData
  properties
    Data
  end
  methods
    function obj = MyData(d)
      if nargin > 0
```

```
        obj.Data = d;
      end
    end
    function bar(obj)
      y = obj.Data;
      bar(y,'EdgeColor','r');
      title('My Data Graph')
    end
  end
end
```

MyData bar 方法与 MATLAB **bar** 函数同名。但是，**MyData bar** 方法需要 **MyData** 对象作为输入。由于该方法针对 **MyData** 对象而特化，因此它可以从 **Data** 属性中提取数据并创建特化图。

要使用 **bar** 方法，请创建一个对象：

```
y = rand(1,10);
md = MyData(y);
```

使用该对象调用方法：

```
bar(md)
```

您也可以使用圆点表示法：

```
md.bar
```

实现 MATLAB 运算符

设计用来实现新 MATLAB 数据类型的类通常会定义一些运算符，如表示加法、减法或相等性的运算符。

例如，标准 MATLAB 加法 (+) 无法将两个多项式相加，因为该运算不是由简单加法定义的。但是，**polynomial** 类可以定义自己的 **plus** 方法，MATLAB 语言调用该方法来使用 + 符号执行 **polynomial** 对象的相加：

```
p1 + p2
```

有关重载运算符的信息，请参阅“运算符重载” （第 17-5 页）。

避免冲突的命名规则

方法、属性和事件的名称的作用域限于类内。因此，请遵守以下规则以避免名称冲突：

- 您可以重用在不相关类中使用的名称。
- 如果成员没有公共或受保护访问权限，则可以在子类中重用名称。然后，这些名称便可以引用完全不同的方法、属性和事件，而不会影响超类定义
- 在一个类中，所有名称都存在于相同的命名空间中，并且必须唯一。一个类无法定义两个同名的方法，而且一个类无法定义与方法同名的局部函数。
- 静态方法的名称被认为没有其类前缀。因此，没有类前缀的静态方法名称不能与任何其他方法的名称相同。

另请参阅

相关示例

- “Class Support for Array-Creation Functions”

对象数组

构造对象数组

本节内容

在构造函数中构建数组

类构造函数可以通过构建数组并将其作为输出参数返回来创建数组。

例如，**ObjectArray** 类创建与输入数组大小相同的对象数组。然后，它将每个对象的 **Value** 属性初始化为对应的输入数组值。

```
classdef ObjectArray
  properties
    Value
  end
  methods
    function obj = ObjectArray(F)
      if nargin ~= 0
        m = size(F,1);
        n = size(F,2);
        obj(m,n) = obj;
        for i = 1:m
          for j = 1:n
            obj(i,j).Value = F(i,j);
          end
        end
      end
    end
  end
end
```

要预分配对象数组，请首先指定数组的最后一个元素。MATLAB 用 **ObjectArray** 对象填充第一个到倒数第二个数组元素。

预分配数组后，为每个对象的 **Value** 属性指定输入数组 **F** 中的对应值。要使用类，请执行以下操作：

- 创建 5×5 幻方数组数字
- 创建 5×5 对象数组

```
F = magic(5);
A = ObjectArray(F);
whos
```

```
  Name      Size             Bytes  Class          Attributes

  A         5x5                304  ObjectArray
  F         5x5                200  double
```

引用对象数组中的属性值

给定一个对象数组 **objArray**，其中每个对象都有属性 **PropName**：

- 使用数组索引引用特定对象的属性值：

  objArray(ix).PropName
- 使用圆点表示法引用对象数组中相同属性的所有值。MATLAB 返回以逗号分隔的属性值列表。

  objArray.PropName
- 要将以逗号分隔的列表赋给变量，请将右侧表达式括在方括号中：

  values = [objArray.PropName]

例如，给定 **ObjProp** 类：

```
classdef ObjProp
  properties
    RegProp
  end
  methods
    function obj = ObjProp
      obj.RegProp = randi(100);
    end
  end
end
```

创建一个 **ObjProp** 对象数组：

```
for k = 1:5
  objArray(k) = ObjProp;
end
```

使用数组索引访问对象数组第二个元素的 **RegProp** 属性：

```
objArray(2).RegProp
```

```
ans =

   91
```

将所有 **RegProp** 属性的值赋给一个数值数组：

```
propValues = [objArray.RegProp]
```

```
propValues =

   82   91   13   92   64
```

使用标准索引操作来访问该数值数组的值。有关数值数组的详细信息，请参阅 “矩阵和数组” 。

另请参阅

相关示例

- “初始化对象数组” (第 10-4 页)
- “Initialize Arrays of Handle Objects”
- “类构造函数方法” (第 9-12 页)

初始化对象数组

本节内容

调用构造函数

在创建对象数组的过程中，MATLAB 可以不带参数地调用类构造函数，即使该构造函数不构建对象数组也是如此。例如，假设您定义以下类：

```
classdef SimpleValue
  properties
    Value
  end
  methods
    function obj = SimpleValue(v)
      obj.Value = v;
    end
  end
end
```

执行以下语句创建数组：

```
a(1,7) = SimpleValue(7)
```

```
Error using SimpleValue (line 7)
Not enough input arguments.
```

出现此错误是因为 MATLAB 调用构造函数时没有参数来初始化数组中的元素 1 到 6。

您的类必须支持无输入参数的构造函数语法。一个简单的解决办法是测试 `nargin`，让 `nargin == 0` 时不执行代码但不引发错误：

```
classdef SimpleValue
  properties
    Value
  end
  methods
    function obj = SimpleValue(v)
      if nargin > 0
        obj.Value = v;
      end
    end
  end
end
```

使用修改后的类定义，前面的数组赋值语句执行时不会出错：

```
a(1,7) = SimpleValue(7)
```

```
a =

  1x7 SimpleValue array with properties:

    Value
```

指定给数组元素 **a(1,7)** 的对象使用传递给构造函数的输入参数作为赋给属性的值：

```
a(1,7)

ans =
  SimpleValue with properties:

    Value: 7
```

MATLAB 不带输入元素地创建元素 **a(1,1:6)** 中包含的对象。属性的默认值为空 **[]**。例如：

```
a(1,1)

ans =
  SimpleValue with properties:

    Value: []
```

MATLAB 调用 **SimpleValue** 构造函数一次，并将返回的对象复制到数组的各个元素。

对象属性的初始值

当 MATLAB 不带参数调用构造函数来初始化对象数组时，会发生以下赋值情况之一：

- 如果属性定义指定了默认值，则 MATLAB 会指定这些值。
- 如果构造函数在没有输入参数的情况下赋值，则 MATLAB 会指定这些值。
- 如果上述两种情况都不适用，则 MATLAB 会将空双精度值（即 **[]**）赋给该属性。

另请参阅

相关示例

- “Initialize Arrays of Handle Objects”

空数组

本节内容

创建空数组

空数组没有元素，但属于某个类。所有非抽象类都有一个名为 **empty** 的静态方法，该方法创建同一类的空数组。**empty** 方法使您能够指定输出数组的维度。不过，至少一个维数必须为 **0**。例如，定义 **SimpleValue** 类：

```
classdef SimpleValue
  properties
    Value
  end
  methods
    function obj = SimpleValue(v)
      if nargin > 0
        obj.Value = v;
      end
    end
  end
end
```

创建 **SimpleValue** 类的 5×0 空数组。

```
ary = SimpleValue.empty(5,0)
```

```
ary =

  5x0 SimpleValue array with properties:

    Value
```

以不带参数形式调用 **empty** 会返回 0×0 空数组。

为空数组赋值

空对象定义数组的类。要将非空对象指定给空数组，MATLAB 调用类构造函数为每个其他数组元素创建该类的默认实例。一旦将非空对象指定给数组，所有数组元素都必须为非空对象。

注意 默认情况下，类构造函数必须避免返回空对象。

例如，使用“初始化对象数组” （第 10-4 页）节中定义的 **SimpleValue**，创建空数组：

```
ary = SimpleValue.empty(5,0);
class(ary)
```

```
ans =

SimpleValue
```

ary 是 **SimpleValue** 类的数组。但是，它是空数组：

```
ary(1)
```

```
Index exceeds matrix dimensions.
```

如果为属性值赋值，MATLAB 会调用 **SimpleClass** 构造函数将数组增长到所需大小：

```
ary(5).Value = 7;
ary(5).Value
```

```
ans =

   7
```

```
ary(1).Value
```

```
ans =

   []
```

MATLAB 用 **SimpleValue** 对象填充数组元素 1 到 5，这些对象是通过不带参数调用类构造函数创建的。然后，MATLAB 将属性值 **7** 赋给 **ary(5)** 的对象。

另请参阅

相关示例

- “Initialize Arrays of Handle Objects”

事件 - 发送和响应消息

事件和侦听程序概述

本节内容
“为什么使用事件和侦听程序” （第 11-2 页）
“事件和侦听程序基础知识” （第 11-2 页）
“事件语法” （第 11-2 页）
“创建侦听程序” （第 11-3 页）

为什么使用事件和侦听程序

事件是对象响应发生的事情而广播的通知，例如属性值更改或用户与应用程序的交互。侦听程序在通知关注的事件发生时执行函数。使用事件来通告对对象的更改。侦听程序通过执行回调函数来响应。

有关详细信息，请参阅“事件和侦听程序概念” （第 11-5 页）。

事件和侦听程序基础知识

在使用事件和侦听程序时：

- 只有 **handle** 类可以定义事件和侦听程序。
- 在类定义的 **events** 代码块中定义事件名称（“事件和侦听程序语法” （第 11-9 页））。
- 使用事件属性指定对事件的访问权限（“Event Attributes”）。
- 调用句柄的 **notify** 方法触发事件。事件通知将指定事件广播给为此事件注册的所有侦听程序。
- 使用句柄的 **addlistener** 方法将侦听程序耦合到事件源对象。MATLAB 会在销毁事件的源时同时销毁侦听程序。
- 使用句柄的 **listener** 方法可创建与事件源对象的生命周期不耦合的侦听程序。如果您在能够独立添加、删除或修改的不同组件中定义事件源和侦听程序，这种方式非常有用。您的应用程序代码控制侦听程序对象的生命周期。
- 侦听程序回调函数必须定义至少两个输入参数 - 事件源对象句柄和事件数据（有关详细信息，请参阅“Listener Callback Syntax”）。
- 通过子类化 **event.EventData** 类修改传递给每个侦听程序回调的数据。

预定义的事件

MATLAB 定义事件用于侦听属性集和查询。有关详细信息，请参阅“侦听对属性值的更改” （第 11-13 页）。

所有句柄对象都定义名为 **ObjectBeingDestroyed** 的事件。MATLAB 在调用类析构函数之前触发此事件。

事件语法

在 **events** 代码块中定义事件名称：

```
classdef ClassName < handle
  events
    EventName
```

```
  end
end
```

例如，**MyClass** 定义名为 **StateChange** 的事件：

```
classdef MyClass < handle
  events
    StateChange
  end
end
```

使用 **handle** 类的 **notify** 方法触发事件：

```
classdef ClassName < handle
  events
    EventName
  end

  methods
    function anyMethod(obj)
      notify(obj,'EventName');
    end
  end
end
```

任何函数或方法都可以为定义事件的类的特定实例触发事件。例如，**triggerEvent** 方法调用 **notify** 来触发 **StateChange** 事件：

```
classdef MyClass < handle
  events
    StateChange
  end
  methods
    function triggerEvent(obj)
      notify(obj,'StateChange')
    end
  end
end
```

用 **triggerEvent** 方法触发 **StateChange** 事件：

```
obj = MyClass;
obj.triggerEvent
```

有关详细信息，请参阅“事件和侦听程序语法” （第 11-9 页）。

创建侦听程序

使用 **handle** 类 **addlistener** 或 **listener** 方法定义侦听程序。使用以下语法之一传递侦听程序回调函数的函数句柄：

- **addlistener(SourceOfEvent,'EventName',@functionName)** - 针对普通函数。
- **addlistener(SourceOfEvent,'EventName',@Obj.methodName)** - 针对 **Obj** 的方法。
- **addlistener(SourceOfEvent,'EventName',@ClassName.methodName)** - 针对类 **ClassName** 的静态方法。

```
ListenerObject = addlistener(SourceOfEvent,'EventName',@listenerCallback);
```

addlistener 返回侦听程序对象。输入参数为：

- **SourceOfEvent** - 定义事件的类的对象。事件在此对象上触发。
- **EventName** - **events** 类代码块中定义的事件名称。
- **@listenerCallback** - 引用响应事件而执行的函数的函数句柄。

例如，为 **StateChange** 事件创建侦听程序对象：

```
function lh = createListener(src)
  lh = addlistener(src,'StateChange',@handleStateChange)
end
```

为侦听程序定义回调函数。回调函数必须接受事件源对象和事件数据对象作为前两个参数：使用事件源参数访问触发事件的对象。使用事件数据对象查找有关事件的信息。

```
function handleStateChange(src,eventData)
  % src - handle to object that triggered the event
  % eventData - event.EventData object containing
  %            information about the event.
  ...
end
```

有关详细信息，请参阅“Listener Callback Syntax”。

另请参阅

event.EventData | **handle**

相关示例

- “Listener Lifecycle”
- “Implement Property Set Listener”

事件和侦听程序概念

本节内容

事件模型

事件表示对象内发生的变化或动作。例如，

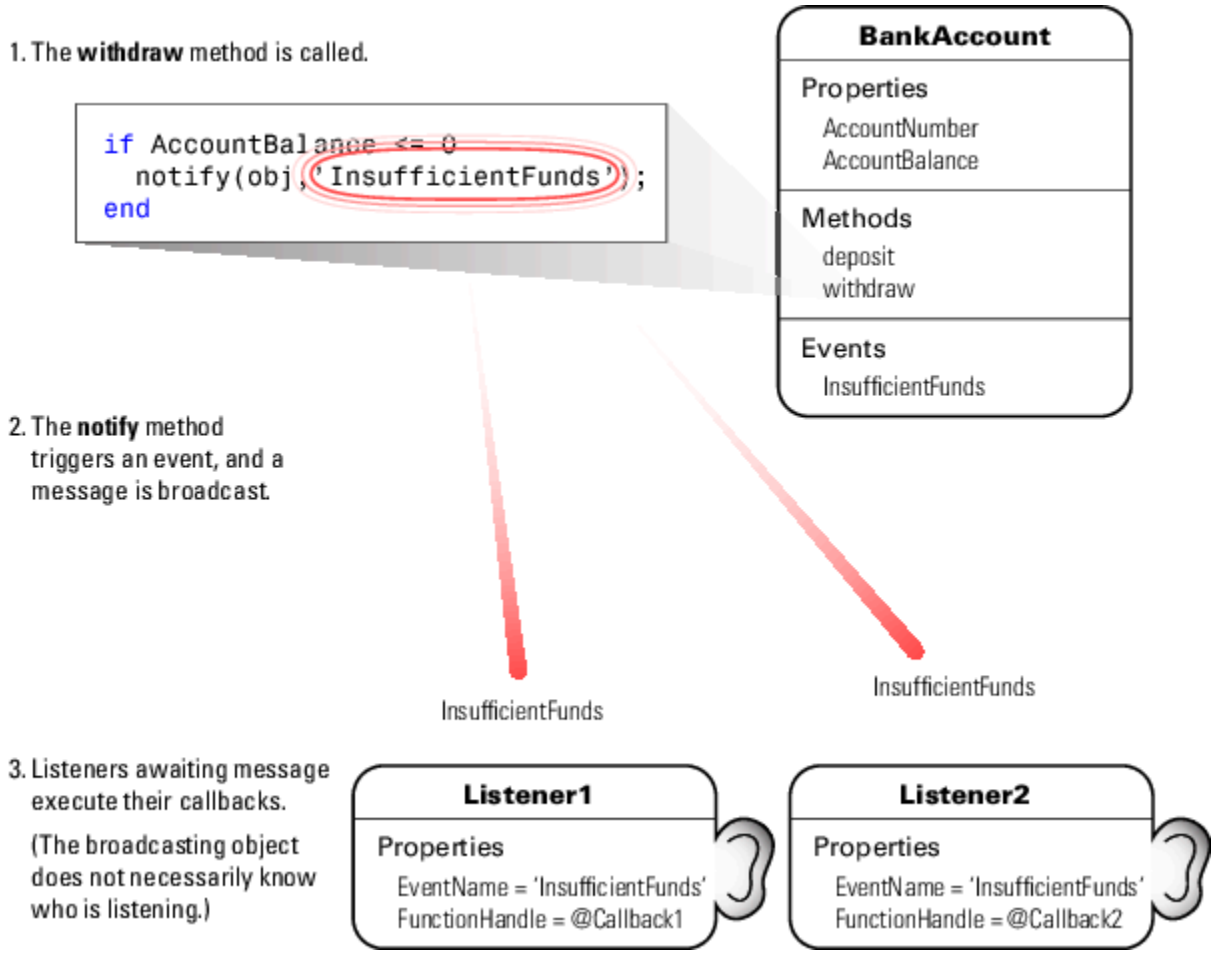
- 类数据的修改
- 方法的执行
- 查询或设置属性值
- 对象的销毁

基本上，您可以通过编程方式检测到的任何活动都可以生成事件并将信息传递给其他对象。

MATLAB 类定义过程，该过程将事件的发生通告给响应这些事件的其他对象。事件模型的工作原理如下：

- 句柄类声明用于表示事件的名称。“命名事件” （第 11-9 页）
- 创建事件声明类的对象后，将侦听程序关联到该对象。“Control Listener Lifecycle”
- 对句柄类的 **notify** 方法的调用会向侦听程序广播事件通知。类用户决定何时触发事件。“触发事件” （第 11-9 页）
- 侦听程序在得到通知事件已发生时执行回调函数。“Specifying Listener Callbacks”
- 您可以将侦听程序绑定到定义事件的对象的生命周期，或者将侦听程序的工作范围限制为侦听程序对象的存在期限和作用域内。“Control Listener Lifecycle”

下图展示了事件模型。



局限性

事件的使用有一些限制：

- 事件源无法保证触发事件时侦听程序存在。
- 一个侦听程序无法阻止其他侦听程序得到事件发生的通知。
- 各侦听程序执行的顺序没有定义。
- 侦听程序不应修改传递给侦听程序回调的事件数据对象，因为其他侦听程序传递的是同一个句柄对象。

默认事件数据

事件通过向回调函数传递事件数据参数来向侦听程序回调提供信息。默认情况下，MATLAB 向侦听程序回调传递一个 **event.EventData** 对象。该对象有两个属性：

- **EventName** - 类 **event** 代码块中定义的事件名称
- **Source** - 作为事件源的对象

MATLAB 在所需的事件数据参数中将源对象传递给侦听程序回调。使用源对象从侦听程序回调函数中访问对象的任何公共属性。

自定义事件数据

您可以创建 **event.EventData** 类的子类，为侦听程序回调函数提供附加信息。子类将定义包含附加数据的属性，并提供构造派生事件数据对象的方法，以便将其传递给 **notify** 方法。

“定义特定于事件的数据” （第 11-12 页）提供说明如何自定义这些数据的示例。

事件仅在句柄类中

您只能在句柄类中定义事件。存在此限制是因为值类仅在单个 MATLAB 工作区中可见，因此没有回调或侦听程序可以访问触发事件的对象。回调可以访问对象的副本。但是，访问副本没有用，因为回调无法访问触发事件的对象的当前状态或在该对象中进行任何更改。

“句柄类和值类的比较” （第 7-2 页）提供关于句柄类的一般信息。

“事件和侦听程序语法” （第 11-9 页）说明定义句柄类和事件的语法。

属性 set 和查询事件

有四个与属性相关的预定义事件：

- **PreSet** - 在设置属性值（调用其 set 访问方法）前即刻触发
- **PostSet** - 设置属性值后即刻触发
- **PreGet** - 在响应属性值查询（调用其 get 访问方法）前即刻触发
- **PostGet** - 返回查询的属性值后即刻触发

这些事件是预定义的，不需要在类 **events** 代码块中列出。

发生属性事件时，将向回调传递 **event.PropertyEvent** 对象。该对象有三个属性：

- **EventName** - 此数据对象描述的事件的名称
- **Source** - 源对象，该对象的类定义数据对象描述的事件
- **AffectedObject** - 事件的来源属性所属的对象，即 **AffectedObject** 包含属性被访问或修改的对象。

您可以通过子类化 **event.EventData** 类来定义自己的属性更改事件数据。**event.PropertyEvent** 类是 **event.EventData** 的密封的子类。

有关创建属性侦听程序的过程的描述，请参阅“侦听对属性值的更改” （第 11-13 页）。

有关示例，请参阅 “The PostSet Event Listener”。

有关控制对属性值的访问的方法的信息，请参阅“属性 set 方法” （第 8-27 页）。

侦听程序

侦听程序封装对事件的响应。侦听程序对象属于 **event.listener** 类，这是句柄类，它定义以下属性：

- **Source** - 生成事件的对象的句柄或句柄数组
- **EventName** - 事件的名称
- **Callback** - 当启用的侦听程序收到事件通知时执行的函数

- **Enabled** - 回调函数仅在 **Enabled** 是 **true** 时执行。有关示例，请参阅“Enable and Disable Listeners”。
- **Recursive** - 允许侦听程序再次触发之前导致回调执行的同一事件。

  默认情况下，**Recursive** 为 **false**。如果回调触发的事件正是其自身所关联的事件，则侦听程序无法递归执行。因此，如果回调必须触发自身所关联的事件，请将 **Recursive** 设置为 **true**。将 **Recursive** 属性设置为 **true** 会造成无限递归达到递归极限并触发错误的情况。

“Control Listener Lifecycle”提供了更具体的信息。

事件和侦听程序语法

本节内容

要实现的组件

事件和侦听程序的实现涉及以下各部分：

- 指定句柄类中事件的名称 - “命名事件” （第 11-9 页）。
- 当动作发生时触发事件的函数或方法 - “触发事件” （第 11-9 页）。
- 侦听程序对象执行回调函数以响应触发的事件 - “侦听事件” （第 11-10 页）。
- 事件传递给回调函数的默认或自定义事件数据 - “定义特定于事件的数据” （第 11-12 页）。

命名事件

通过在 **events** 代码块中声明事件名称来定义事件。例如，以下类创建名为 **ToggledState** 的事件：

```
classdef ToggleButton < handle
  properties
    State = false
  end
  events
    ToggledState
  end
end
```

触发事件

OnStateChange 方法调用 **notify** 来触发 **ToggledState** 事件。将作为事件源的对象的句柄和事件名称传递给 **notify**。

```
classdef ToggleButton < handle
  properties
    State = false
  end
  events
    ToggledState
  end
  methods
    function OnStateChange(obj,newState)
      if newState ~= obj.State
        obj.State = newState;
        notify(obj,'ToggledState');
      end
    end
```

```
  end
end
```

侦听事件

在调用 **notify** 而触发事件后，MATLAB 向为该事件和源对象定义的所有侦听程序广播消息。创建侦听程序有两种方式：使用句柄类 **addlistener** 或 **listener** 方法。

使用 addlistener 创建持久侦听程序

如果您希望侦听程序在通常的变量作用域外持久存在，请使用 **addlistener** 创建它。事件源对象中包含对侦听程序对象的引用。当销毁事件源对象时，MATLAB 会同时销毁对应的侦听程序。

以下代码定义 **ToggleState** 事件的侦听程序：

```
lh = addlistener(obj,'ToggleState',@RespondToToggle.handleEvnt);
```

addlistener 有以下参数：

- **obj** - 作为事件源的对象
- **ToggleState** - 作为 **char** 向量传递的事件名称
- **@RespondToToggle.handleEvnt** - 回调函数的函数句柄（请参阅以下定义“定义侦听程序” （第 11-11 页））。

使用 handle.listener 来分离侦听程序和源

如果您要管理侦听程序的生命周期，并且不希望事件源和侦听程序对象互相耦合，请使用 **listener** 方法创建侦听程序。当销毁事件源时，MATLAB 不会同时销毁由 **listener** 创建的侦听程序。但是，当使用 **listener** 创建侦听程序时，代码必须使侦听程序对象句柄保持在作用域内。

listener 方法需要的参数与 **addlistener** 相同：事件命名对象、事件名称和回调函数句柄。**listener** 返回侦听程序对象的句柄。

```
lh = listener(obj,'EventName',@callbackFunction)
```

例如，以下代码使用前面讨论过的 **ToggleState** 事件：

```
lh = listener(obj,'ToggleState',@RespondToToggle.handleEvnt)
```

回调函数

侦听程序回调函数必须接受至少两个参数，MATLAB 会自动将这两个参数传递给回调函数。以下是必需的参数：

- 事件源 - 即导致调用 **addlistener** 或 **event.listener** 的 **obj**。
- **event.EventData** 对象或 **event.EventData** 的子类，如“定义特定于事件的数据” （第 11-12 页）中所述的 **ToggleEventData** 对象。

定义回调函数以接受源对象和事件数据参数。

```
function callbackFunction(src,evtdata)
  ...
end
```

有关回调语法的详细信息，请参阅“Listener Callback Syntax”。

定义侦听程序

RespondToToggle 类定义侦听 **ToggleButton** 类中定义的 **ToggleState** 事件的对象。

```
classdef RespondToToggle < handle
  methods
    function obj = RespondToToggle(toggle_button_obj)
      addlistener(toggle_button_obj,'ToggledState',@RespondToToggle.handleEvnt);
    end
  end
  methods (Static)
    function handleEvnt(src,~)
      if src.State
        disp('ToggledState is true')
      else
        disp('ToggledState is false')
      end
    end
  end
end
```

类 **RespondToToggle** 在其构造函数中添加侦听程序。在这种情况下，类将回调 (**handleEvnt**) 定义为接受两个必需参数的静态方法：

- **src** - 触发事件的对象（即 **ToggleButton** 对象）的句柄
- **evtdata** - **event.EventData** 对象

例如，以下代码段创建两个类的对象：

```
tb = ToggleButton;
rtt = RespondToToggle(tb);
```

每当您调用 **ToggleButton** 对象的 **OnStateChange** 方法时，**notify** 都会触发事件。在本例中，回调显示 **State** 属性的值：

```
tb.OnStateChange(true)
```

```
ToggledState is true
```

```
tb.OnStateChange(false)
```

```
ToggledState is false
```

删除侦听程序

通过对侦听程序对象的句柄调用 **delete** 来删除侦听程序对象。例如，如果类 **RespondToToggle** 将侦听程序句柄保存为属性，则您可以删除该侦听程序。

```
classdef RespondToToggle < handle
  properties
    ListenerHandle % Property for listener handle
  end
  methods
    function obj = RespondToToggle(toggle_button_obj)
      hl = addlistener(toggle_button_obj,'ToggledState',@RespondToToggle.handleEvnt);
      obj.ListenerHandle = hl; % Save listener handle
    end
  end
  methods (Static)
    function handleEvnt(src,~)
      if src.State
        disp('ToggledState is true')
      else
        disp('ToggledState is false')
      end
```

```
    end
  end
end
```

通过更改此代码，您可以从 **RespondToToggle** 类的实例中删除侦听程序。例如：

```
tb = ToggleButton;
rtt = RespondToToggle(tb);
```

对象 **rtt** 正在侦听由对象 **tb** 触发的 **ToggleState** 事件。要删除侦听程序，请对包含侦听程序句柄的属性调用 **delete**。

```
delete(rtt.ListenerHandle)
```

要临时停用侦听程序，请参阅“Temporarily Deactivate Listeners”。

定义特定于事件的数据

假设您要将切换按钮的状态作为事件的结果传递给侦听程序回调。通过子类化 **event.EventData** 类并添加属性来包含相关信息，您可以向默认事件数据中添加更多数据。然后，您可以将此对象传递给 **notify** 方法。

注意 要保存和加载作为 **event.EventData** 的子类的对象，如 **ToggleEventData**，请为该子类启用 **ConstructOnLoad** 类属性。

```
classdef (ConstructOnLoad) ToggleEventData < event.EventData
  properties
    NewState
  end

  methods
    function data = ToggleEventData(newState)
      data.NewState = newState;
    end
  end
end
```

对 **notify** 的调用可以使用 **ToggleEventData** 构造函数来创建必要的参数。

```
evtdata = ToggleEventData(newState);
notify(obj,'ToggledState',evtdata);
```

另请参阅

相关示例

- “Listener Callback Syntax”
- “侦听对属性值的更改” (第 11-13 页)
- “Techniques for Using Events and Listeners”

侦听对属性值的更改

本节内容

创建属性侦听程序

对于句柄类，您可以为预声明的属性事件（命名为：**PreSet**、**PostSet**、**PreGet**、**PostGet**）定义侦听程序。要为这些指定事件创建侦听程序，请执行以下操作：

- 指定 **SetObservable** 和/或 **GetObservable** 属性特性。
- 定义回调函数
- 通过在对 **addlistener** 或 **listener** 的调用中包含属性和事件的名称来创建属性侦听程序。
- 如有必要，子类化 **event.data** 以创建特化事件数据对象来传递给回调函数。
- 如果新值与当前值相同，则阻止回调的执行（请参阅“Assignment When Property Value Is Unchanged”）。

设置属性特性以启用属性事件

在属性代码块中，启用 **SetObservable** 特性。您可以为此代码块中定义的属性定义 **PreSet** 和 **PostSet** 侦听程序：

```
properties (SetObservable)
  PropOne
  PropTwo
end
```

定义属性事件的回调函数

当 MATLAB 触发属性事件时，侦听程序执行回调函数。将回调函数定义为具有两个特定参数的函数，当侦听程序调用函数时，这两个参数会自动传递给函数：

- 事件源 - 说明作为属性事件源的对象的 **meta.property** 对象
- 事件数据 - 包含事件信息的 **event.PropertyEvent** 对象

如有必要，您可以传递附加参数。将此方法定义为 **Static** 通常很简单，因为这两个参数在其属性中包含最必要的信息。

例如，假设 **handlePropEvents** 函数是类的一个静态方法，该类为另一个类的对象的两个属性创建侦听程序：

```
        methods (Static)
  function handlePropEvents(src,evnt)
    switch src.Name
      case 'PropOne'
        % PropOne has triggered an event
      case 'PropTwo'
        % PropTwo has triggered an event
    end
  end
end
```

另一种可能的方式是在 **switch** 语句中使用 **event.PropertyEvent** 对象的 **EventName** 属性，采用事件名称作为控制关键字，本例中为 **PreSet** 或 **PostSet**。

“Class Metadata”提供有关 **meta.property** 类的详细信息。

将侦听程序添加到属性

addlistener 句柄类方法使您能够将侦听程序关联到属性，而无需将侦听程序对象存储为持久变量。对于属性事件，请使用带四个参数的 **addlistener**。

以下是对 **addlistener** 的调用：

```
addlistener(EventObject,'PropOne','PostSet',@ClassName.handlePropertyEvents);
```

参数如下：

- **EventObject** - 生成事件的对象的句柄
- **PropOne** - 要侦听的属性的名称
- **PostSet** - 要侦听的事件的名称
- **@ClassName.handlePropertyEvents** - 引用静态方法的函数句柄，这需要使用类名

如果您的侦听程序回调是普通方法，而不是静态方法，则语法为：

```
addlistener(EventObject,'PropOne','PostSet',@obj.handlePropertyEvents);
```

其中 **obj** 是定义回调方法的对象的句柄。

如果侦听程序回调是非类方法函数，则可以向该函数传递函数句柄。假设回调函数是包函数：

```
addlistener(EventObject,'PropOne','PostSet',@package.handlePropertyEvents);
```

有关将函数作为参数传递的详细信息，请参阅“创建函数句柄”。

属性事件和侦听程序类

以下两个类显示如何为两个属性（**PropOne** 和 **PropTwo**）创建 **PostSet** 属性侦听程序。

用于生成事件的类

通过指定 **SetObservable** 属性特性，**PropEvent** 类启用属性 **PreSet** 和 **PostSet** 事件触发。这些属性还启用 **AbortSet** 特性，如果属性设置为与其当前值相同的值，该属性将阻止属性事件的触发（请参阅“Assignment When Property Value Is Unchanged”）。

```
classdef PropEvent < handle
  properties (SetObservable, AbortSet)
    PropOne
    PropTwo
  end
  methods
    function obj = PropEvent(p1,p2)
      if nargin > 0
        obj.PropOne = p1;
        obj.PropTwo = p2;
      end
    end
  end
end
```

定义侦听程序的类

PropListener 类定义两个侦听程序：

- 属性 **PropOne** 的 **PostSet** 事件
- 属性 **PropTwo** 的 **PostSet** 事件

您可以使用类似的方式为其他事件或其他属性定义侦听程序。不必为每个侦听程序使用相同的回调函数。有关传递给侦听程序回调函数的参数中包含的详细信息，请参阅 **meta.property** 和 **event.PropertyEvent** 参考页。

```
classdef PropListener < handle
  % Define property listeners
  methods
    function obj = PropListener(evtobj)
      if nargin > 0
        addlistener(evtobj,'PropOne','PostSet',@PropListener.handlePropEvents);
        addlistener(evtobj,'PropTwo','PostSet',@PropListener.handlePropEvents);
      end
    end
  end
  methods (Static)
    function handlePropEvents(src,evnt)
      switch src.Name
        case 'PropOne'
          sprintf('PropOne is %s\n',num2str(evnt.AffectedObject.PropOne))
        case 'PropTwo'
          sprintf('PropTwo is %s\n',num2str(evnt.AffectedObject.PropTwo))
      end
    end
  end
end
```

另请参阅

相关示例

- “Assignment When Property Value Is Unchanged”

如何构建类层次结构

子类语法

本节内容

子类定义语法

要定义作为另一个类的子类的类，请将超类添加到 **classdef** 行中的 < 字符后：

```
classdef ClassName < SuperClass
```

从多个类继承时，使用 **&** 字符来指示超类的组合：

```
classdef ClassName < SuperClass1 & SuperClass2
```

有关从多个超类派生的详细信息，请参阅“Class Member Compatibility”。

类属性

子类不继承超类属性。

子类化 double

假设您要定义从 **double** 派生的类，并将值限制为正数。**PositiveDouble** 类：

- 支持默认构造函数（无输入参数）。请参阅“构造函数不要求输入参数的情况” （第 9-15 页）
- 使用 **mustBePositive** 将输入限制为正值。
- 用输入值调用超类构造函数来创建双精度数值。

```
classdef PositiveDouble < double
  methods
    function obj = PositiveDouble(data)
      if nargin == 0
        data = 1;
      else
        mustBePositive(data)
      end
      obj = obj@double(data);
    end
  end
end
```

使用 1×5 数值数组创建 **PositiveDouble** 类的对象：

```
a = PositiveDouble(1:5);
```

您可以像对任何双精度值一样对该类的对象执行运算。

```
sum(a)
```

```
ans =

   15
```

PositiveDouble 类的对象必须为正值。

```
a = PositiveDouble(0:5);

Error using mustBePositive (line 19)
Value must be positive.

Error in PositiveDouble (line 7)
        mustBePositive(data)
```

另请参阅

相关示例

- “设计子类构造函数” （第 12-4 页）
- “Subclasses of MATLAB Built-In Types”

设计子类构造函数

本节内容

显式调用超类构造函数

从子类构造函数显式调用每个超类构造函数使您能够：

- 将参数传递给超类构造函数
- 控制 MATLAB 调用超类构造函数的顺序

如果您不从子类构造函数中显式调用超类构造函数，MATLAB 将不使用任何参数隐式调用这些构造函数。超类构造函数必须支持无参数语法以支持隐式调用，构造函数按它们出现在类块顶部的顺序从左到右进行调用。要更改 MATLAB 调用构造函数的顺序，或使用参数调用构造函数，请从子类构造函数显式调用超类构造函数。

如果没有定义子类构造函数，可以用超类参数调用默认构造函数。有关详细信息，请参阅“默认构造函数” （第 9-13 页）和“对继承的构造函数的隐式调用” （第 9-17 页）。

从子类调用超类构造函数

要在子类构造函数中调用每个超类的构造函数，请使用以下语法：

```
obj@SuperClass1(args,...);
...
obj@SuperclassN(args,...);
```

其中，**obj** 是子类构造函数的输出，**SuperClass...** 是超类的名称，**args** 是对应超类构造函数所需的任何参数。

例如，类定义的以下代码段显示名为 **Stocks** 的类，它是名为 **Assets** 的类的子类。

```
classdef Stocks < Assets
  methods
    function s = Stocks(asset_args,...)
      if nargin == 0
        % Assign values to asset_args
      end
      % Call asset constructor
      s@Assets(asset_args);
      ...
    end
  end
end
```

“子类构造函数” （第 9-15 页）提供关于创建子类构造函数方法的详细信息。

引用包中包含的超类

如果超类包含在包中，请包含包名称。例如，Assests 类在 finance 包中：

```
classdef Stocks < finance.Assets
  methods
    function s = Stocks(asset_args,...)
      if nargin == 0
        ...
      end
      % Call asset constructor
      s@finance.Assets(asset_args);
      ...
    end
  end
end
```

使用多个超类初始化对象

要从多个超类派生一个类，请通过调用每个超类构造函数来初始化子类对象：

```
classdef Stocks < finance.Assets & Taxable
  methods
    function s = Stocks(asset_args,tax_args,...)
      if nargin == 0
        ...
      end
      % Call asset and member class constructors
      s@finance.Assets(asset_args)
      s@Taxable(tax_args)
      ...
    end
  end
end
```

子类构造函数实现

为了确保类构造函数支持零参数语法，请在调用超类构造函数之前为输入参数变量分配默认值。您无法对超类构造函数的子类调用进行条件化。请将对超类构造函数的调用置于任何条件代码块之外。

例如，Stocks 类构造函数支持 if 语句的无参数语法，但在 if 代码块之外调用了超类构造函数。

```
classdef Stocks < finance.Assets
  properties
    NumShares
    Symbol
  end
  methods
    function s = Stocks(description,numshares,symbol)
      if nargin == 0
        description = '';
        numshares = 0;
        symbol = '';
      end
      s@finance.Assets(description);
      s.NumShares = numshares;
      s.Symbol = symbol;
```

```
    end
  end
end
```

从构造函数中仅可调用直接超类

从子类构造函数中仅可调用直接超类构造函数。例如，假设类 **B** 派生自类 **A**，类 **C** 派生自类 **B**。**C** 类的构造函数无法调用 **A** 类的构造函数来初始化属性。类 **B** 必须初始化类 **A** 属性。

类 **A**、**B** 和 **C** 的以下实现显示如何在每个类中设计这种关系。

类 **A** 定义属性 **x** 和 **y**，但只为 **x** 赋值：

```
classdef A
  properties
    x
    y
  end
  methods
    function obj = A(x)
      ...
        obj.x = x;
    end
  end
end
```

类 **B** 从类 **A** 继承属性 **x** 和 **y**。类 **B** 构造函数调用类 **A** 构造函数以初始化 **x**，然后为 **y** 赋值。

```
classdef B < A
  methods
    function obj = B(x,y)
      ...
      obj@A(x);
      obj.y = y;
    end
  end
end
```

类 **C** 接受属性 **x** 和 **y** 的值，并将这些值传递给类 **B** 构造函数，后者又调用类 **A** 构造函数：

```
classdef C < B
  methods
    function obj = C(x,y)
      ...
      obj@B(x,y);
    end
  end
end
```

另请参阅

相关示例

- “构造函数不要求输入参数的情况” （第 9-15 页）

修改继承的方法

本节内容

何时修改超类方法

类设计使您可以将子类对象传递给超类方法。由于子类对象为超类对象，因此超类方法可以正确执行。但是，子类可以实现自己的超类方法版本，供 MATLAB 在传递子类对象时调用。

当需要在子类中提供专门的行为时，子类会覆盖继承的方法（即，实现一个同名的方法）。以下是一些覆盖超类方法的模式。

- 通过从子类方法内部调用超类方法来对超类方法进行扩展。子类方法除了调用超类方法外，还可以执行子类特定的处理。
- 在超类方法中，使用受保护方法实现过程中的一系列步骤。然后，通过重新定义从公共超类方法内部调用的受保护方法，在子类方法中重新实现这些步骤。
- 在子类中重新定义同名方法，但使用不同的实现以不同的方式对子类对象执行相同的操作。
- 在子类中实现抽象超类方法。抽象超类可以定义无实现的方法，并依赖子类来提供实现。有关详细信息，请参阅“定义接口超类” （第 12-20 页）。

用于覆盖超类方法的子类方法为 **Access** 特性定义的值必须与超类方法所定义的值相同。

扩展超类方法

通过从子类方法调用同名的超类方法，您可以在不影响超类方法的情况下为子类对象扩展超类方法。

例如，假设超类和子类都定义了名为 **foo** 的方法。子类方法调用超类方法，并执行除调用超类方法外的其他步骤。子类方法可以对子类中不属于超类的特化部分进行操作。

例如，以下子类定义一个 **foo** 方法，该方法调用超类 **foo** 方法：

```
classdef Sub < Super
  methods
    function foo(obj)
      % preprocessing steps
       ...
      foo@Super(obj);
      % postprocessing steps
       ...
    end
  end
end
```

在子类中重新实现超类过程

超类方法可以定义一个执行一系列步骤的过程，每个步骤都使用一个方法（通常将步骤方法的 **Access** 特性设置为 **protected**）。这种模式（称为模板方法）使子类可以创建自己的受保护方法版本，实现该过程中的各个步骤。该过程专门用于该子类。

这种编程方式的实现如下所示：

```
classdef Super
  methods (Sealed)
    function foo(obj)
      step1(obj) % Call step1
      step2(obj) % Call step2
      step3(obj) % Call step3
    end
  end
  methods (Access = protected)
    function step1(obj)
      % Superclass version
    end
    function step2(obj)
      % Superclass version
    end
    function step3(obj)
      % Superclass version
    end
  end
end
```

子类不会覆盖 **foo** 方法。相反，它仅覆盖执行系列步骤（**step1(obj)**、**step2(obj)**、**step3(obj)**）的受保护方法。也就是说，此方法使子类可以特化每个步骤所采取的操作，但不控制过程中各步骤的顺序。当您将子类对象传递给超类 **foo** 方法时，根据调度规则，MATLAB 会调用子类步骤的方法。有关方法调度的详细信息，请参阅“方法调用” （第 9-9 页）。

```
classdef Sub < Super
  ...
  methods (Access = protected)
    function step1(obj)
      % Subclass version
    end
    function step2(obj)
      % Subclass version
    end
    function step3(obj)
      % Subclass version
    end
    ...
  end
end
```

重新定义超类方法

您可以在子类中彻底重新定义超类方法。在这种情况下，超类和子类都定义同名的方法。然而，实现会有所不同，并且子类方法不会调用超类方法。当相同的操作需要对超类和子类进行不同的实现时，可能需要创建独立版本的同名方法。

```
classdef Super
  methods
    function foo(obj)
      % Superclass implementation
    end
  end
end

classdef Sub < Super
  methods
    function foo(obj)
      % Subclass implementation
    end
  end
end
```

在子类中实现抽象方法

抽象方法没有实现。继承抽象方法的子类必须提供子类特定的实现，该子类才能成为具体类。有关详细信息，请参阅“抽象类和类成员” （第 12-16 页）。

```
classdef Super
  methods (Abstract)
    foo(obj)
      % Abstract method without implementation
  end
end

classdef Sub < Super
  methods
    function foo(obj)
      % Subclass implementation of concrete method
    end
  end
end
```

另请参阅

相关示例

- “对子类对象调用超类方法” （第 5-10 页）
- “抽象类和类成员” （第 12-16 页）
- “Modify Inherited Properties”

类成员访问

本节内容

基础知识

要学习本节的材料，需要预先了解以下概念：

术语和概念

- 类成员 - 由类定义的属性、方法和事件
- 定义类 - 该类定义要为其指定访问权限的类成员
- 获取访问权限 - 读取属性值的权限，由属性 **GetAccess** 特性控制
- 设置访问权限 - 为属性赋值的权限；由属性 **SetAccess** 特性控制
- 方法访问 - 确定哪些其他方法和函数可以调用类方法；由方法 **Access** 属性控制
- 侦听访问权限 - 定义侦听程序的权限；由事件 **ListenAccess** 属性控制
- 通知访问权限 - 触发事件的权限，由事件 **NotifyAccess** 属性控制

访问类成员的可能值

下列类成员属性可以包含类列表：

- 属性 - **Access**、**GetAccess** 和 **SetAccess**。有关所有属性特性的列表，请参阅“属性特性” （第 8-7 页）。
- 方法 - **Access**。有关所有方法属性的列表，请参阅“方法特性” （第 9-4 页）。
- 事件 - **ListenAccess** 和 **NotifyAccess**。有关所有事件属性的列表，请参阅“Event Attributes”。

这些属性接受下列值：

- **public** - 不受限制的访问
- **protected** - 可由定义类及其子类进行访问
- **private** - 仅由定义类进行访问
- 访问列表 - 一个或多个类的列表。只有定义类和列表中的类才能访问该属性所应用到的类成员。如果您指定了类列表，MATLAB 不允许任何其他类访问（即访问权限为 **private**，但列出的类除外）。

访问控制列表的应用

访问控制列表可用于控制对特定类属性、方法和事件的访问。您可使用访问控制列表来指定类列表，向列表中的类授予对这些类成员的访问权限。

这种方式在类系统的设计中提供了更大的灵活性和更强的控制。例如，使用访问控制列表定义不同的类列表，但不允许从类系统外访问类成员。

控制对类成员的访问

在成员访问属性语句中指定允许访问特定类成员的类。例如：

methods (Access = {?ClassName1,?ClassName2,...})

请使用类 **meta.class** 对象来引用访问列表中的类。要指定多个类，可使用 **meta.class** 对象的元胞数组。引用包中的类时，需要使用包名称。

注意 请显式指定 **meta.class** 对象（使用 **?** 运算符创建），不能通过函数或其他 MATLAB 表达式的返回值来指定。

MATLAB 如何解释特性值

- 将访问权限授予类列表时，仅以下类可获取访问权：
  - 定义类
  - 列表中的类
  - 列表中类的子类
- 如果访问列表中包含定义类，则会为该定义类的所有子类都提供访问权。
- 访问列表中的类必须先被加载，MATLAB 才会解析对它的引用。如果 MATLAB 找不到访问列表中包含的类，则实际上相当于删除了该类的访问权限。
- MATLAB 会用空的 **meta.class** 对象替换列表中未解析的 **meta.class** 条目。
- 空访问列表（即空元胞数组）等效于 **private** 访问。

指定元类对象

仅使用 **?** 运算符和类名生成 **meta.class** 对象。赋予特性的值不能包含任何其他 MATLAB 表达式，包括可返回允许的特性值的函数：

- **meta.class** 对象
- **meta.class** 对象构成的元胞数组
- 值 **public**、**protected** 或 **private**

需要显式指定这些值，如本节的示例代码所示。

具有访问列表的属性

这些示例类显示授予类读取权限 (**GetAccess**) 的属性的行为。**GrantAccess** 类将 **Prop1** 属性的 **GetAccess** 授予 **NeedAccess** 类：

```
classdef GrantAccess
  properties (GetAccess = ?NeedAccess)
    Prop1 = 7
  end
end
```

NeedAccess 类定义一个使用 GrantAccess 的 Prop1 值的方法。dispObj 方法定义为 Static 方法，但是，它可能是一个普通方法。

```
classdef NeedAccess
  methods (Static)
    function dispObj(GrantAccessObj)
      disp(['Prop1 is: ',num2str(GrantAccessObj.Prop1)])
    end
  end
end
```

对 Prop1 的访问权限是私有访问权，因此如果您尝试从类定义之外访问该属性，MATLAB 将返回错误。例如，从命令行：

```
a = GrantAccess;
a.Prop1
```

```
Getting the 'Prop1' property of the 'GrantAccess' class is not allowed.
```

但是，MATLAB 允许 NeedAccess 类访问 Prop1：

```
NeedAccess.dispObj(a)
```

```
Prop1 is: 7
```

具有访问列表的方法

授予类访问方法的权限后，该类可以：

- 使用定义类的实例调用方法。
- 用相同的名称定义自己的方法（如果不是子类）。
- 覆盖子类中的方法（仅当定义方法的超类在访问列表中包含自身或子类时）。

以下示例类说明从访问列表中其他类的方法调用的方法的行为。AcListSuper 类授予 AcListNonSub 类访问其 m1 方法的权限：

```
classdef AcListSuper
  methods (Access = {?AcListNonSub})
    function obj = m1(obj)
      disp ('Method m1 called')
    end
  end
end
```

由于 AcListNonSub 在 m1 的访问列表中，因此其方法可以使用 AcListSuper 的实例调用 m1：

```
classdef AcListNonSub
  methods
    function obj = nonSub1(obj,AcListSuper_Obj)
      % Call m1 on AcListSuper class
      AcListSuper_Obj.m1;
    end
    function obj = m1(obj)
      % Define a method named m1
      disp(['Method m1 defined by ',class(obj)])
    end
```

```
  end
end
```

创建两个类的对象：

```
a = AcListSuper;
b = AcListNonSub;
```

使用 **AcListNonSub** 方法调用 **AcListSuper** 和 **m1** 方法：

```
b.nonSub1(a);
```

```
Method m1 called
```

调用 **AcListNonSub** 的 **m1** 方法：

```
b.m1;
```

```
Method m1 defined by AcListNonSub
```

无访问权限的子类

如果方法的访问列表包含定义类，则从该类派生的所有子类都将被授予访问权。如果子类派生自的类的方法的访问列表不包括定义类， 则：

- 子类方法无法调用超类方法。
- 子类方法可以使用访问列表中的类实例间接调用超类方法。
- 子类无法覆盖超类方法。
- 超类方法访问列表中所含的非子类的类的方法可以调用超类方法。

例如，**AcListSub** 是 **AcListSuper** 的子类。**AcListSuper** 类定义方法 **m1** 的访问列表。但是，该列表不包括 **AcListSuper**，因此 **AcListSuper** 的子类无权访问方法 **m1**：

```
classdef AcListSub < AcListSuper
  methods
    function obj = sub1(obj,AcListSuper_Obj)
      % Access m1 via superclass object (***NOT ALLOWED***)
      AcListSuper_Obj.m1;
    end
    function obj = sub2(obj,AcListNonSub_Obj,AcListSuper_obj)
      % Access m1 via object that is in access list (is allowed)
      AcListNonSub_Obj.nonSub1(AcListSuper_Obj);
    end
  end
end
```

不能直接调用超类方法

尝试从 **sub1** 方法调用超类的 **m1** 方法会导致错误，因为子类不在 **m1** 的访问列表中：

```
a = AcListSuper;
c = AcListSub;
c.sub1(a);
```

```
Cannot access method 'm1' in class 'AcListSuper'.

Error in AcListSub/sub1 (line 4)
      AcListSuper_Obj.m1;
```

间接调用超类方法

您可以使用超类方法的访问列表中某个类的对象，从无权访问超类方法的子类调用超类方法。

AcListSub 的 **sub2** 方法调用 **m1** 访问列表中某个类 (**AcListNonSub**) 的方法。此方法 **nonSub1** 可以访问超类 **m1** 方法：

```
a = AcListSuper;
b = AcListNonSub;
c = AcListSub;
c.sub2(b,a);
```

```
Method m1 called
```

不能重新定义超类方法

当方法的访问列表中没有子类时，这些子类无法定义同名的方法。此行为与方法的 **Access** 明确声明为 **private** 的情况不同。

例如，向 **AcListSub** 类定义中添加以下方法会在您尝试实例化该类时产生错误。

```
methods (Access = {?AcListNonSub})
  function obj = m1(obj)
    disp('AcListSub m1 method')
  end
end
```

```
c = AcListSub;
```

```
Class 'AcListSub' is not allowed to override the method 'm1' because neither it nor its
superclasses have been granted access to the method by class 'AcListSuper'.
```

从列表中的非子类通过子类调用超类

AcListNonSub 类位于 **m1** 方法访问列表中。该类可以使用 **AcListSub** 类的对象定义调用 **m1** 方法的方法。虽然 **AcListSub** 不在方法 **m1** 的访问列表中，但它是 **AcListSuper** 的子类。

例如，将以下方法添加到 **AcListNonSub** 类中：

```
methods
  function obj = nonSub2(obj,AcListSub_Obj)
    disp('Call m1 via subclass object:')
    AcListSub_Obj.m1;
  end
end
```

调用 **nonSub2** 方法会导致执行超类的 **m1** 方法：

```
b = AcListNonSub;
c = AcListSub;
b.nonSub2(c);
```

```
Call m1 via subclass object:
Method m1 called
```

此行为与任何子类对象的行为是一致的，子类对象可以替代其超类的对象。

具有访问列表的抽象方法

包含声明为 **Abstract** 的方法的类是抽象类。子类负责使用在类定义中声明的函数签名实现抽象方法。

当抽象方法有访问列表时，只有访问列表中的类才能实现该方法。不在访问列表中的子类无法实现抽象方法，因此子类本身是抽象类。

另请参阅

相关示例

- “属性特性” （第 8-7 页）
- “Method Access List”
- “Event Access List”

抽象类和类成员

本节内容

抽象类

抽象类十分适用于描述一组类所共有的功能，但需要在每个类中有唯一的实现。

抽象类术语

抽象类 - 无法实例化但定义子类使用的类组件的类。

抽象成员 - 在抽象类中声明但在子类中实现的属性或方法。

具体类 - 可以实例化的类。具体类不包含抽象成员。

具体成员 - 由类完全实现的属性或方法。

接口 - 一种抽象类，它描述一组类所共有的功能，但需要在每个类中有唯一的实现。抽象类定义每个子类的接口，而不指定实际实现。

抽象类作为一组相关子类的基础（即超类）。抽象类可以定义子类实现的抽象属性和方法。每个子类都可以以支持其特定要求的方式实现具体的属性和方法。

实现具体子类

子类必须实现所有继承的抽象属性和方法才能成为具体类。否则，子类本身就是抽象类。

MATLAB 不强制子类使用相同签名或属性实现具体方法。

抽象类：

- 可以定义非抽象的属性和方法
- 通过继承传递其具体成员
- 不需要定义任何抽象成员

将类声明为抽象类

当类做出以下声明时为抽象类：

- 声明 **Abstract** 类属性
- 声明抽象方法
- 声明抽象属性

如果抽象类的子类没有为所有继承的抽象方法或属性定义具体实现，该子类也是抽象类。

抽象类

在 **classdef** 语句中将类声明为抽象类：

```
classdef (Abstract) AbsClass
  ...
end
```

对于声明 **Abstract** 类属性的类：

- 具体子类必须重新定义声明为抽象的任何属性或方法。
- 抽象类不需要定义任何抽象方法或属性。

定义任何抽象方法或属性时，MATLAB 会自动将类 **Abstract** 特性设置为 **true**。

抽象方法

定义抽象方法：

```
methods (Abstract)
  abstMethod(obj)
end
```

对于声明 **Abstract** 方法属性的方法：

- 不要使用 **function...end** 代码块来定义抽象方法，只能使用方法签名进行定义。
- 抽象方法在抽象类中没有实现。
- 具体子类不需要支持相同数量的输入和输出参数，也不需要使用相同的参数名称。不过，子类在实现其方法版本时通常使用相同的签名。
- 抽象方法无法定义 **arguments** 代码块。

抽象属性

定义抽象属性：

```
properties (Abstract)
  AbsProp
end
```

对于声明 **Abstract** 属性特性的属性：

- 具体子类必须不用 **Abstract** 特性重新定义抽象属性。
- 具体子类必须对 **SetAccess** 和 **GetAccess** 属性使用与抽象超类中使用的那些属性相同的值。
- 抽象属性无法定义访问方法，也无法指定初始值。定义具体属性的子类可以创建访问方法并指定初始值。

有关访问方法的详细信息，请参阅“属性 set 方法” （第 8-27 页）。

确定类是否为抽象类

通过查询 **meta.class** 对象的 **Abstract** 属性，确定类是否为抽象类。例如，**AbsClass** 定义两个抽象方法：

```
classdef AbsClass
  methods(Abstract)
```

```
    result = absMethodOne(obj)
    output = absMethodTwo(obj)
  end
end
```

使用 **meta.class Abstract** 属性的逻辑值来确定类是否为抽象类：

```
mc = ?AbsClass;
if ~mc.Abstract
  % not an abstract class
end
```

显示抽象成员名称

使用 **meta.abstractDetails** 函数显示抽象属性或方法的名称以及定义类的名称：

```
meta.abstractDetails('AbsClass');
```

```
Abstract methods for class AbsClass:
  absMethodTwo   % defined in AbsClass
  absMethodOne   % defined in AbsClass
```

查找继承的抽象属性和方法

meta.abstractDetails 函数返回您尚未在子类中实现的任何继承的抽象属性或方法的名称和定义类。如果您需要子类是具体类并且必须确定子类继承哪些抽象成员，请使用此函数。

例如，假设您创建在上一节中定义的 **AbsClass** 类的子类。在本例中，子类只实现 **AbsClass** 定义的抽象方法之一。

```
classdef SubAbsClass < AbsClass
% Does not implement absMethodOne
% defined as abstract in AbsClass
  methods
    function out = absMethodTwo(obj)
      ...
    end
  end
end
```

使用 **meta.abstractDetails** 确定您是否实现了所有继承的类成员：

```
meta.abstractDetails(?SubAbsClass)
```

```
Abstract methods for class SubAbsClass:
  absMethodOne   % defined in AbsClass
```

SubAbsClass 类是抽象类，因为它没有实现在 **AbsClass** 中定义的 **absMethodOne** 方法。

```
msub = ?SubAbsClass;
msub.Abstract
```

```
ans =

     1
```

如果您实现了在 **AbsClass** 中定义的两个方法，子类将变为具体类。

另请参阅

相关示例

- “定义接口超类” (第 12-20 页)

定义接口超类

本节内容

接口

类定义的属性和方法构成接口，该接口决定类用户如何与类的对象交互。创建一组相关类时，接口定义所有这些类的公共接口。接口的实际实现可能因类而异。

以一组设计用于表示各种图形类型的类为例。所有类都必须实现 **Data** 属性，以包含用于生成图形的数据。然而，对于不同类型的图形，数据的形式可能有很大不同。每个类可以通过不同方式实现 **Data** 属性。

对于方法，也存在同样的差异。所有类都可以有一个创建图形的 **draw** 方法，但是，该方法的实现会根据图形的类型而变化。

接口类的基本思想是指定每个子类必须实现的属性和方法，而不定义实际实现。这种方式使您能够对一组相关对象强制实施一致的接口。在将来添加更多的类时，接口始终保持不变。

接口类实现图

此示例为用于表示特化图形的类创建一个接口。接口是抽象类，它定义子类必须实现的属性和方法，但不指定如何实现这些组件。

这种方式强制实施一致的接口，同时提供必要的灵活性来以不同方式实现每个特化子类的内部工作。

在本示例中，包文件夹包含接口、派生的子类和工具函数：

```
+graphics/GraphInterface.m  % abstract interface class
+graphics/LineGraph.m       % concrete subclass
```

接口的属性和方法

graph 类指定以下属性，子类必须定义这些属性：

- **Primitive** - 用于实现特化图形的图形对象的句柄。类用户不需要直接访问这些对象，因此该属性具有 **protected SetAccess** 和 **GetAccess**。
- **AxesHandle** - 用于图形的坐标区的句柄。特化 **graph** 对象可以设置坐标区对象属性。该属性具有 **protected SetAccess** 和 **GetAccess**。
- **Data** - **GraphInterface** 类的所有子类都必须存储数据。数据的类型各不相同，每个子类都定义存储机制。子类用户可以更改数据值，因此该属性具有公共访问权限。

GraphInterface 类命名子类必须实现的三个抽象方法。**GraphInterface** 类还在注释中建议每个子类构造函数必须接受所有类属性的绘图数据和属性名称/属性值对组。

- 子类构造函数 - 接受数据和 P/V 对组并返回对象。
- **draw** - 用于创建绘图基元，并根据子类实现的图形类型渲染数据的图形。
- **zoom** - 通过更改坐标区 **CameraViewAngle** 属性来实现 zoom 方法。该接口推荐使用 **camzoom** 函数来实现子类之间的一致性。由 **addButtons** 静态方法创建的缩放按钮使用此方法作为回调。

- **updateGraph - set.Data** 方法调用的方法，用于在 **Data** 属性发生变化时更新绘图数据。

接口决定类设计

从 **GraphInterface** 抽象类派生的类包实现以下行为：

- 创建特化 **GraphInterface** 对象（子类对象）的实例，而不渲染绘图
- 在创建特化 **GraphInterface** 对象时，指定任一或不指定任何对象属性
- 更改任一对象属性都会自动更新当前显示的绘图
- 允许每个特化 **GraphInterface** 对象实现它需要的任何附加属性，以便让类用户控制这些特性。

定义接口

GraphInterface 类是抽象类，用于定义子类使用的方法和属性。抽象类中的注释说明预期的实现：

```
classdef GraphInterface < handle
  % Abstract class for creating data graphs
  % Subclass constructor should accept
  % the data that is to be plotted and
  % property name/property value pairs
  properties (SetAccess = protected, GetAccess = protected)
    Primitive
    AxesHandle
  end
  properties
    Data
  end
  methods (Abstract)
    draw(obj)
    % Use a line, surface,
    % or patch graphics primitive
    zoom(obj,factor)
    % Change the CameraViewAngle
    % for 2D and 3D views
    % use camzoom for consistency
    updateGraph(obj)
    % Update the Data property and
    % update the drawing primitive
  end

  methods
    function set.Data(obj,newdata)
      obj.Data = newdata;
      updateGraph(obj)
    end
    function addButtons(gobj)
      hfig = get(gobj.AxesHandle,'Parent');
      uicontrol(hfig,'Style','pushbutton','String','Zoom Out',...
        'Callback',@(src,evnt)zoom(gobj,.5));
      uicontrol(hfig,'Style','pushbutton','String','Zoom In',...
        'Callback',@(src,evnt)zoom(gobj,2),...
        'Position',[100 20 60 20]);
    end
  end
end
```

GraphInterface 类实现属性 set 方法 (set.Data) 以监控 Data 属性的更改。另一种方式是将 Data 属性定义为 Abstract，并使子类能够确定是否为此属性实现 set 访问方法。GraphInterface 类定义一个调用抽象方法（updateGraph，每个子类都必须实现该方法）的 set 访问方法。GraphInterface 接口为整个类包施加特定设计，而不会限制灵活性。

用于所有子类的方法

addButtons 方法为 zoom 方法添加普通按钮，每个子类都必须实现这些方法。通过使用方法而不是普通函数，可使 addButtons 能够访问受保护的类数据（坐标区句柄）。使用对象 zoom 方法作为普通按钮回调。

```
function addButtons(gobj)
  hfig = get(gobj.AxesHandle,'Parent');
  uicontrol(hfig,'Style','pushbutton',...
    'String','Zoom Out',...
    'Callback',@(src,evnt)zoom(gobj,.5));
  uicontrol(hfig,'Style','pushbutton',...
    'String','Zoom In',...
    'Callback',@(src,evnt)zoom(gobj,2),...
    'Position',[100 20 60 20]);
end
```

派生具体类 - LineGraph

此示例只定义一个用于表示简单线图的子类。它派生自 GraphInterface，但提供对抽象方法 draw、zoom、updateGraph 及其自己的构造函数的实现。基类 GraphInterface 和子类都包含在包 (graphics) 中，您必须使用它来引用类名：

```
classdef LineGraph < graphics.GraphInterface
```

添加属性

LineGraph 类实现在 GraphInterface 类中定义的接口，并添加两个附加属性 - LineColor 和 LineType。此类为每个属性定义初始值，因此在构造函数中指定属性值是可选的。您可以创建没有数据的 LineGraph 对象，但无法从该对象生成图形。

```
properties
  LineColor = [0 0 0];
  LineType = '-';
end
```

LineGraph 构造函数

构造函数接受 struct 及 x 和 y 坐标数据，以及属性名称/属性值对组：

```
function gobj = LineGraph(data,varargin)
  if nargin > 0
    gobj.Data = data;
    if nargin > 2
      for k=1:2:length(varargin)
        gobj.(varargin{k}) = varargin{k+1};
      end
    end
  end
end
```

实现 draw 方法

LineGraph 的 draw 方法使用属性值创建一个 line 对象。LineGraph 类将 line 句柄存储为受保护的类数据。为了支持类构造函数不使用输入参数，draw 会在继续之前检查 Data 属性以确定它是否为空：

```
function gobj = draw(gobj)
  if isempty(gobj.Data)
    error('The LineGraph object contains no data')
  end
  h = line(gobj.Data.x,gobj.Data.y,...
    'Color',gobj.LineColor,...
    'LineStyle',gobj.LineType);
  gobj.Primitive = h;
  gobj.AxesHandle = get(h,'Parent');
end
```

实现 zoom 方法

LineGraph 的 zoom 方法使用 GraphInterface 类的注释中推荐的 camzoom 函数。camzoom 提供方便的接口来支持缩放功能，并使用 addButtons 方法创建的普通按钮以正确进行操作。

定义属性 set 方法

通过属性 set 方法，可以方便地在构造函数中首次更改属性值时自动执行代码。（请参阅“属性 set 方法” （第 8-27 页）。）每当属性值更改时，linegraph 类都会使用 set 方法更新 line 原始数据（这会导致绘图重绘）。使用属性 set 方法可以快速更新数据图，而无需调用 draw 方法。draw 方法通过重置所有值为当前属性值来更新绘图。

三个属性使用 set 方法：LineColor、LineType 和 Data。LineColor 和 LineType 是由 LineGraph 类添加的属性，并且是该类使用的 line 基元特有的属性。其他子类可以定义对其特化所独有的不同属性（例如，FaceColor）。

GraphInterface 类实现 Data 属性 set 方法。但是，GraphInterface 类要求每个子类定义名为 updateGraph 的方法，该方法处理所使用的特定绘图基元的绘图数据更新。

LineGraph 类

以下是 LineGraph 类定义。

```
classdef LineGraph < graphics.GraphInterface
  properties
    LineColor = [0 0 0]
    LineType = '-'
  end

  methods
    function gobj = LineGraph(data,varargin)
      if nargin > 0
        gobj.Data = data;
        if nargin > 1
          for k=1:2:length(varargin)
            gobj.(varargin{k}) = varargin{k+1};
          end
        end
      end
    end
```

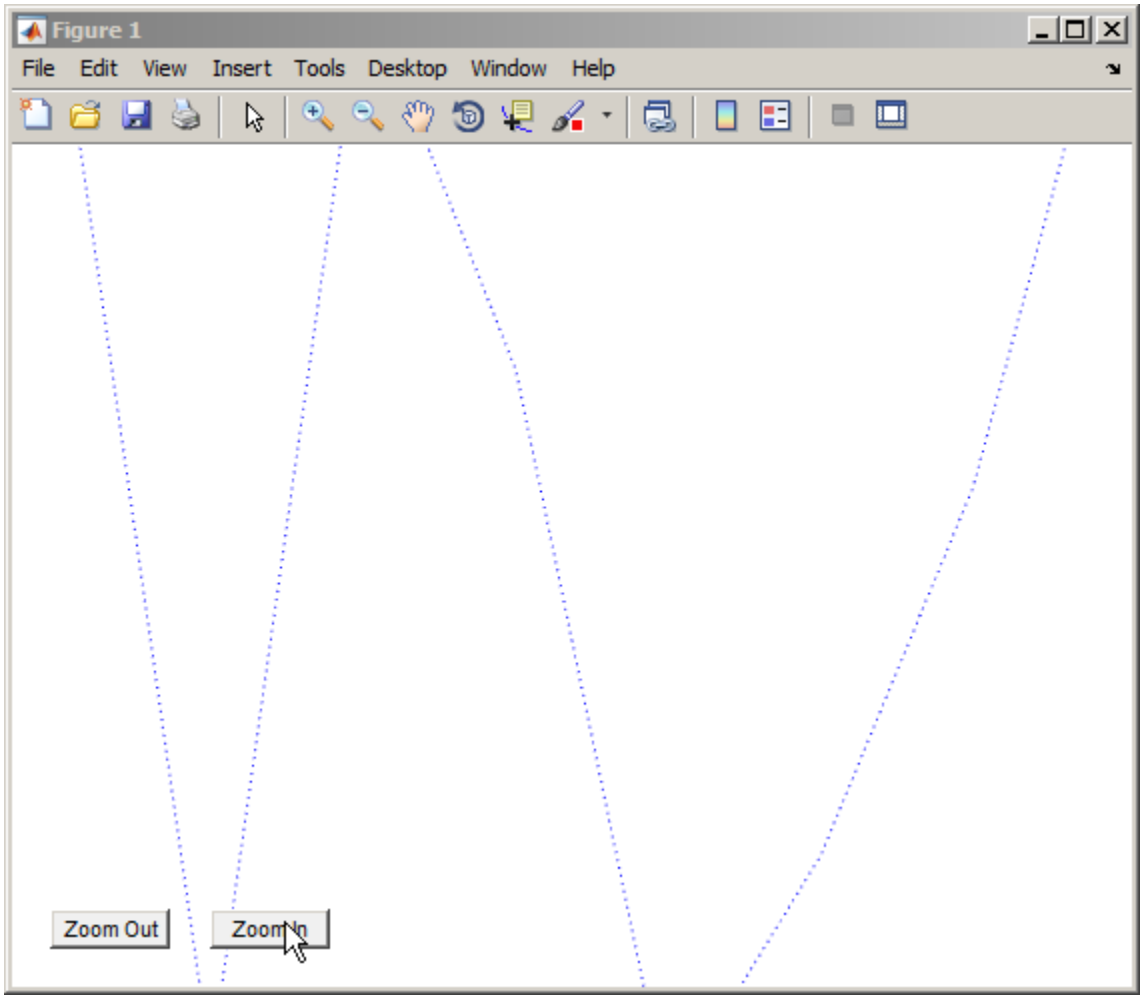
```
    function gobj = draw(gobj)
      if isempty(gobj.Data)
        error('The LineGraph object contains no data')
      end
      h = line(gobj.Data.x,gobj.Data.y,...
        'Color',gobj.LineColor,...
        'LineStyle',gobj.LineType);
      gobj.Primitive = h;
      gobj.AxesHandle = h.Parent;
    end

    function zoom(gobj,factor)
      camzoom(gobj.AxesHandle,factor)
    end

    function updateGraph(gobj)
      set(gobj.Primitive,...
        'XData',gobj.Data.x,...
        'YData',gobj.Data.y)
    end

    function set.LineColor(gobj,color)
      gobj.LineColor = color;
      set(gobj.Primitive,'Color',color)
    end

    function set.LineType(gobj,ls)
      gobj.LineType = ls;
      set(gobj.Primitive,'LineStyle',ls)
    end
  end
end
```

使用 LineGraph 类

LineGraph 类定义由 graph 基类指定的简单 API，并实现其特化类型的图形：

```
d.x = 1:10;
d.y = rand(10,1);
lg = graphics.LineGraph(d,'LineColor','b','LineType',':');
lg.draw;
lg.addButtons;
```

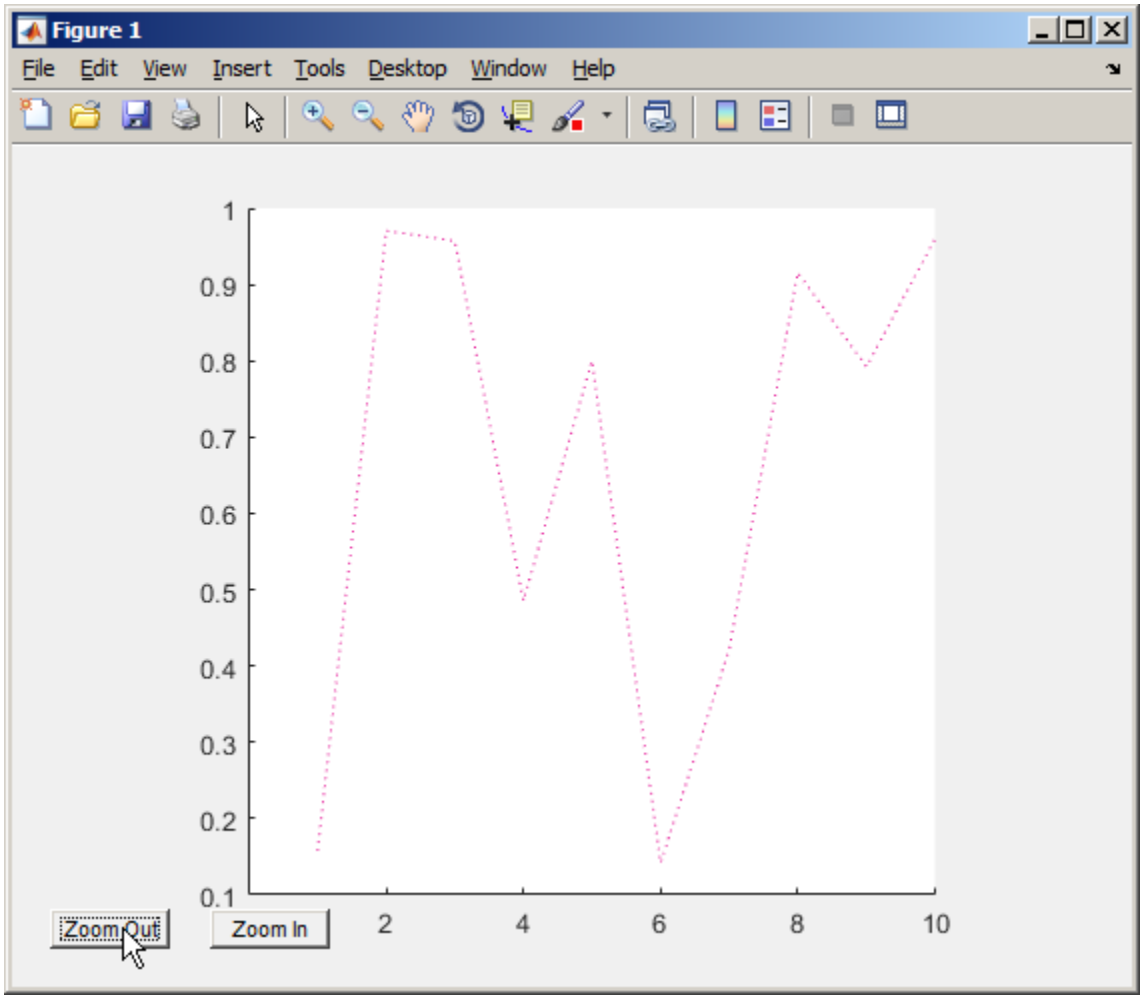
点击**放大**按钮会显示为该按钮提供回调的 zoom 方法。



更改属性会更新图形：

```
d.y = rand(10,1);
lg.Data = d;
lg.LineColor = [0.9,0.1,0.6];
```

现在点击**缩小**并查看新结果：



另请参阅

相关示例

- “抽象类和类成员” (第 12-16 页)

保存和加载对象

对象的保存和加载过程

本节内容

保存和加载对象

使用 **save** 和 **load** 来存储和重新加载对象:

```
save filename object
load filename object
```

保存哪些信息?

将对象保存在 MAT 文件中会保存:

- 对象类的全名，包括任何包限定符
- 动态属性的值
- 当类的第一个对象保存到 MAT 文件时该类定义的所有属性默认值。
- 所有属性的名称和值，但存在以下例外:
  - 如果属性的当前值与类定义中指定的默认值相同，则不会保存属性。
  - 如果属性的 **Transient**、**Constant** 或 **Dependent** 特性设置为 **true**，则不会保存属性。

有关属性特性的描述，请参阅“属性特性” (第 8-7 页)。

要保存图形对象，请参阅 **savefig**。

如何加载属性数据?

从 MAT 文件加载对象时，**load** 函数会还原对象。

- **load** 创建一个新对象。
- 如果类 **ConstructOnLoad** 属性设置为 **true**，则 **load** 调用不带参数的类构造函数。否则，**load** 不会调用类构造函数。
- **load** 将保存的属性值赋给对象属性。这些所赋的值受制于由类定义的任何属性验证。然后调用由类定义的属性 set 方法（**Dependent**、**Constant** 或 **Transient** 属性除外，这些属性不会保存或加载）。
- **load** 将保存在 MAT 文件中的默认值赋给其值未保存的属性，因为这些属性在保存时设置为默认值。这些赋值导致对类定义的属性 set 方法的调用。
- 如果正在加载的对象的属性包含对象，**load** 将创建一个相同类的新对象并将其赋给该属性。如果属性中包含的对象为句柄对象，则该属性会包含一个相同类的新句柄对象。

MATLAB 调用属性 set 方法，以确保在类定义发生变化的情况下属性值仍然有效。

有关信息，请参阅“属性 set 方法” （第 8-27 页）和“验证属性值” （第 8-12 页）。

加载过程中的错误

如果类的新版本删除、重命名或更改了属性的验证，**load** 在尝试设置已更改或删除的属性时可能生成错误。

如果在从文件加载对象时出现错误，MATLAB 会执行以下操作之一：

- 如果该类定义一个 **loadobj** 方法，MATLAB 将保存的值返回 **struct** 中的 **loadobj** 方法。
- 如果该类没有定义 **loadobj** 方法，MATLAB 会以静默方式忽略这些错误。**load** 函数用不会产生错误的属性值来重新构成对象。

在传递给 **loadobj** 方法的 **struct** 中，字段名称对应于属性名称。字段值是对应属性的保存值。

如果保存的对象派生自多个具有同名私有属性的超类，则 **struct** 仅包含最直接超类的属性值。

有关如何实现 **saveobj** 和 **loadobj** 方法的信息，请参阅“Modify the Save and Load Process”。

对属性验证的更改

如果类定义更改了属性验证，使得加载的属性值不再有效，则 MATLAB 会用当前定义的默认值代替该属性。该类可以定义 **loadobj** 方法或转换器方法，以提供类版本之间的兼容性。

有关属性验证的信息，请参阅“验证属性值” （第 8-12 页）。

另请参阅

isequal

相关示例

- “对象保存和加载”

枚举

定义枚举类

本节内容

枚举类

通过向类定义中添加 **enumeration** 代码块可创建枚举类。例如，**WeekDays** 类列举一周中的工作日。

```
classdef WeekDays
  enumeration
    Monday, Tuesday, Wednesday, Thursday, Friday
  end
end
```

为了执行以下各节中的 MATLAB 代码，请将 **WeekDays** 类定义放在您的路径上的 **.m** 文件中。

构造枚举成员

使用类名和成员名引用枚举成员：

```
ClassName.MemberName
```

例如，将枚举成员 **WeekDays.Tuesday** 赋给变量 **today**：

```
today = WeekDays.Tuesday;
```

today 是 **WeekDays** 类的变量：

```
whos
```

```
  Name      Size            Bytes  Class      Attributes

  today     1x1               104  WeekDays
```

```
today
```

```
today =

    Tuesday
```

转换为超类值

如果枚举类指定超类，您可以通过将枚举对象传递给超类构造函数，将枚举对象转换为超类。然而，超类构造函数必须能够接受自己的类作为输入，并返回超类的实例。MATLAB 的内置数值类（如 **uint32**）允许此转换。

例如，**Bearing** 类派生自 **uint32** 内置类：

```
classdef Bearing < uint32
  enumeration
    North (0)
    East  (90)
    South (180)
    West  (270)
  end
end
```

将 **Bearing.East** 成员赋给变量 **a**：

```
a = Bearing.East;
```

将 **a** 传递给超类构造函数，并返回 **uint32** 值：

```
b = uint32(a);
whos
```

```
  Name      Size             Bytes  Class     Attributes

  a         1x1                  4  Bearing
  b         1x1                  4  uint32
```

uint32 构造函数接受 **Bearing** 子类的对象，并返回 **uint32** 类的对象。

在枚举类中定义方法

在枚举类中定义方法的方式与在任何 MATLAB 类中的定义方式一样。例如，为 **WeekDays** 枚举类定义一个名为 **isMeetingDay** 的方法。用例是用户在星期二定期开会。该方法检查输入参数是否为 **WeekDays** 成员 **Tuesday** 的实例。

```
classdef WeekDays
  enumeration
    Monday, Tuesday, Wednesday, Thursday, Friday
  end
  methods
    function tf = isMeetingDay(obj)
      tf = WeekDays.Tuesday == obj;
    end
  end
end
```

使用 **WeekDays** 类的实例调用 **isMeetingDay**：

```
today = WeekDays.Tuesday;
today.isMeetingDay
```

```
ans =

   1
```

您也可以使用枚举成员作为方法的直接输入：

```
isMeetingDay(WeekDays.Wednesday)
```

```
ans =

     0
```

在枚举类中定义属性

当必须存储与枚举成员相关的数据时，可向枚举类添加属性。在类构造函数中设置属性值。例如，**SyntaxColors** 类定义三个属性。当引用类成员时，类构造函数将输入参数的值赋给对应的属性。

```
classdef SyntaxColors
  properties
    R
    G
    B
  end
  methods
    function c = SyntaxColors(r, g, b)
      c.R = r; c.G = g; c.B = b;
    end
  end
  enumeration
    Error   (1, 0, 0)
    Comment (0, 1, 0)
    Keyword (0, 0, 1)
    String  (1, 0, 1)
  end
end
```

当引用枚举成员时，构造函数会初始化属性值：

```
e = SyntaxColors.Error;
e.R
```

```
ans =

     1
```

由于 **SyntaxColors** 是值类（它不从 **handle** 派生），只有类构造函数可以设置属性值：

```
e.R = 0
```

```
You cannot set the read-only property 'R' of SyntaxColors.
```

有关定义属性的枚举类的详细信息，请参阅“Mutable Handle vs. Immutable Value Enumeration Members”。

枚举类构造函数调用顺序

枚举代码块中的每个语句均为枚举成员的名称，其后可跟参数列表。如果枚举类定义了构造函数，MATLAB 会调用该构造函数来创建枚举实例。

MATLAB 为没有显式定义构造函数的所有枚举类提供默认构造函数。默认构造函数创建枚举类的实例时：

- 如果枚举成员没有定义输入参数，则不使用输入参数
- 使用枚举类中为该成员定义的输入参数

例如，**Bool** 类的输入参数是 **0**（表示 **Bool.No**）和 **1**（表示 **Bool.Yes**）。

```
classdef Bool < logical
  enumeration
    No  (0)
    Yes (1)
  end
end
```

值 0 和 1 属于 logical 类，因为默认构造函数将参数传递给第一个超类。也就是说，以下语句：

```
n = Bool.No;
```

导致对 logical 的调用，等效于构造函数中的以下语句：

```
function obj = Bool(val)
  obj@logical(val)
end
```

MATLAB 只将成员参数传递给第一个超类。例如，假设 Bool 派生自另一个类：

```
classdef Bool < logical & MyBool
  enumeration
    No  (0)
    Yes (1)
  end
end
```

MyBool 类可以添加一些特化行为：

```
classdef MyBool
  methods
    function boolValues = testBools(obj)
      ...
    end
  end
end
```

默认的 Bool 构造函数的行为类似以下函数：

- 参数传递给第一个超类构造函数
- 没有参数传递给后续构造函数

```
function obj = Bool(val)
  obj@logical(val)
  obj@MyBool
end
```

另请参阅

相关示例

- “Refer to Enumerations”
- “枚举的运算” （第 14-6 页）

枚举的运算

本节内容

枚举支持的运算

您可以对枚举使用逻辑、集合关系和字符串比较运算。这些运算还支持在条件语句中使用枚举，例如 **switch** 和 **if** 语句。**string** 和 **char** 函数使您能够将枚举成员转换为字符串和 **char** 向量。

枚举类示例

本主题使用 **WeekDays** 类来说明如何对枚举执行运算。**WeekDays** 类定义枚举一周工作日的成员。

```
classdef WeekDays
  enumeration
    Monday, Tuesday, Wednesday, Thursday, Friday
  end
end
```

有关定义枚举的信息，请参阅“定义枚举类” （第 14-2 页）。

默认方法

枚举类有以下默认方法：

```
methods('WeekDays')
```

```
Methods for class WeekDays:

WeekDays   char      intersect  ne       setxor     strcmpi    strncmp    union
cellstr    eq        ismember   setdiff  strcmp     string     strncmpi
```

WeekDays 方法将文本格式转换为枚举。支持的格式包括字符串、**char** 向量、字符串数组和 **char** 向量元胞数组。例如：

```
f = WeekDays(["Monday" "Friday"])
```

```
f =
```

```
1×2 WeekDays enumeration array

  Monday    Friday
```

将枚举成员转换为字符串或 char 向量

枚举成员到字符串和 **char** 向量的转换对于创建包含枚举成员名称的文本非常有用。例如，使用 **string** 函数可将枚举成员转换为字符串并将其包含在一个句子中：

```
string(WeekDays.Monday) + " is our meeting day."
```

```
ans =

  "Monday is our meeting day."
```

以类似的方式使用 **char** 函数：

```
['Today is ' char(WeekDays.Friday) '.']
```

```
ans =

  'Today is Friday.'
```

将枚举数组转换为字符串数组或 char 向量元胞数组

使用 **string** 函数将枚举数组转换为字符串数组：

```
sa = [WeekDays.Tuesday WeekDays.Thursday];
string(sa)
```

```
ans =

  1×2 string array

    "Tuesday"    "Thursday"
```

使用 **cellstr** 将枚举数组转换为 **char** 向量元胞数组。

```
ca = cellstr([WeekDays.Tuesday WeekDays.Thursday]);
class(ca)
```

```
ans =

  'cell'
```

元胞数组中的两个元胞都包含 **char** 向量：

```
class([ca{1:2}])
```

```
ans =

  'char'
```

枚举、字符串和 char 向量的关系运算

您可以使用关系运算符 **eq** (==) 和 **ne** (~=) 以及 **isequal** 方法将枚举实例与 **char** 向量和字符串进行比较。

关系运算符 eq 和 ne

使用 **eq** 和 **ne** 将枚举成员与文本值进行比较。例如，您可以将枚举成员与字符串进行比较：

```
today = WeekDays.Friday;
today == "Friday"
```

```
ans =

  logical

   1
```

将枚举数组与一个 **char** 向量进行比较。返回值是一个逻辑数组，指示枚举数组的哪些成员等效于 **char** 向量：

```
wd = [WeekDays.Monday WeekDays.Wednesday WeekDays.Friday];
wd == 'Friday'
```

```
ans =

  1×3 logical array

   0   0   1
```

此示例使用 **ne** 函数比较枚举数组和等长字符串数组的对应元素：

```
sa = ["Monday" "Wednesday" "Friday"];
md = [WeekDays.Tuesday WeekDays.Thursday WeekDays.Friday];
md ~= sa
```

```
ans =

  1×3 logical array

   1   1   0
```

char 向量 **Wednesday** 等于 (==) 枚举成员 **WeekDays.Wednesday**。您可以在条件语句中使用此相等性：

```
today = 'Wednesday';
   ...
if today == WeekDays.Wednesday
   disp('Team meeting at 2:00')
end
```

isequal 方法

isequal 方法还支持枚举实例与字符串、字符向量、字符串数组和字符向量元胞数组之间的比较。

```
a = WeekDays.Monday;
isequal(a,"Monday")
```

```
ans =

    logical

     1
```

将枚举数组与单个项进行比较时，**isequal** 的行为与 **eq** 和 **ne** 略有不同。**isequal** 方法要求所比较的两个值大小相同。因此，将枚举数组与 **char** 向量或字符串标量进行比较时，**isequal** 返回 false，即使文本与数组中的一个枚举成员匹配也是如此。

```
wd = [WeekDays.Monday WeekDays.Wednesday WeekDays.Friday];
isequal(wd,"Friday")
```

```
ans =

  logical

   0
```

switch 语句中的枚举

相等 (**eq**) 和不相等 (**ne**) 函数使您能够在 **switch** 语句中使用枚举成员。例如，使用前面定义的 **WeekDays** 枚举构造一个 switch 语句：

```
function c = Reminder(day)
  % Add error checking here
  switch(day)
    case WeekDays.Monday
      c = 'No meetings';
    case WeekDays.Tuesday
      c = 'Department meeting at 10:00';
    case {WeekDays.Wednesday WeekDays.Friday}
      c = 'Team meeting at 2:00';
    case WeekDays.Thursday
      c = 'Volleyball night';
  end
end
```

将 **WeekDays** 枚举类的成员传递给 **Reminder** 函数：

```
today = WeekDays.Wednesday;
Reminder(today)
```

```
ans =

Team meeting at 2:00
```

有关详细信息，请参阅“Objects in Conditional Statements”。

代替字符串或 char 向量

您可以使用字符串或 **char** 向量来表示特定枚举成员：

```
function c = Reminder2(day)
  switch(day)
    case 'Monday'
      c = 'Department meeting at 10:00';
    case 'Tuesday'
      c = 'Meeting Free Day!';
    case {'Wednesday' 'Friday'}
      c = 'Team meeting at 2:00';
    case 'Thursday'
      c = 'Volleyball night';
```

```
  end
end
```

虽然您可以使用 **char** 向量或字符串代替显式指定枚举，但是 MATLAB 必须将文本格式转换为枚举。如果没有必要，就不需要进行这种转换。

枚举集合关系

枚举类提供用于确定集合关系的方法。

- **ismember** - 如果枚举数组的元素在集合中，则为 true
- **setdiff** - 枚举数组的差集
- **intersect** - 枚举数组的交集
- **setxor** - 枚举数组的异或集
- **union** - 枚举数组的并集

确定今天是否为您的团队开会的日期。创建一组对应于团队开会日期的枚举成员。

```
today = WeekDays.Tuesday;
teamMeetings = [WeekDays.Wednesday WeekDays.Friday];
```

使用 **ismember** 确定 **today** 是否属于 **teamMeetings** 集合：

```
ismember(today,teamMeetings)
```

```
ans =
    0
```

枚举和文本的混合集

如果将枚举成员和文本值都传递给枚举类方法，则该类会尝试将文本值转换为枚举的类。

确定 **char** 向量 **'Friday'** 是否为枚举数组的成员。

```
teamMeetings = [WeekDays.Wednesday WeekDays.Friday];
ismember('Friday',teamMeetings)
```

```
ans =

  logical

   1
```

确定枚举成员是否为字符串数组的成员。

```
ismember(WeekDays.Friday,["Wednesday" "Friday"])
```

```
ans =

  logical

   1
```

枚举的文本比较方法

枚举类提供将枚举成员与文本进行比较的方法。字符串比较方法的参数之一必须为 **char** 向量或字符串。比较两个枚举成员返回 **false**。

- **strcmp** - 比较枚举成员
- **strncmp** - 比较枚举成员的前 n 个字符
- **strcmpi** - 对枚举成员进行不区分大小写的比较
- **strncmpi** - 对枚举成员的前 n 个字符进行不区分大小写的比较

将枚举成员与字符串或 char 向量进行比较

字符串比较方法可以将枚举成员与 **char** 向量和字符串进行比较。

```
today = WeekDays.Tuesday;
strcmp(today,'Friday')
```

```
ans =

   0
```

```
strcmp(today,"Tuesday")
```

```
ans =

   1
```

获取有关枚举的信息

使用 **enumeration** 函数获取关于枚举类的信息。例如：

```
enumeration WeekDays
```

```
Enumeration members for class 'WeekDays':

   Monday
   Tuesday
   Wednesday
   Thursday
   Friday
```

有关类自检如何使用枚举的详细信息，请参阅“Metaclass EnumeratedValues Property”。

对枚举的测试

要确定某值是否为枚举，请使用 **isenum** 函数。例如：

```
today = WeekDays.Wednesday;
isenum(today)
```

```
ans =

   1
```

对于空枚举对象，**isenum** 返回 **true**：

```
noday = WeekDays.empty;
isenum(noday)
```

```
ans =

   1
```

要确定某个类是否为枚举类，请使用 **meta.class** 对象。

```
today = WeekDays.Wednesday;
mc = metaclass(today);
mc.Enumeration
```

```
ans =

   1
```

另请参阅

相关示例

- “Enumeration Class Restrictions”

常量属性

定义具有常量值的类属性

本节内容

定义命名常量

您可以通过创建定义常量属性的 MATLAB 类来定义可以按名称引用的常量。

常量属性可用于定义可以按名称访问的常量值。通过在属性代码块中声明 **Constant** 特性，创建具有常量属性的类。设置 **Constant** 特性意味着，一旦初始化为属性代码块中指定的值，就无法更改该值。

为常量属性赋值

为 **Constant** 属性赋值，包括 MATLAB 表达式。例如：

```
classdef NamedConst
  properties (Constant)
    R = pi/180
    D = 1/NamedConst.R
    AccCode = '0145968740001110202NPQ'
    RN = rand(5)
  end
end
```

MATLAB 在加载类时计算表达式。因此，MATLAB 赋给 **RN** 的值是调用 **rand** 函数后所得的结果，该值不会因随后对 **NamedConst.RN** 的引用而更改。调用 **clear classes** 会导致 MATLAB 重新加载类并重新初始化常量属性。

引用常量属性

使用类名和属性名称引用常量：

```
ClassName.PropName
```

例如，要使用在上一节中定义的 **NamedConst** 类，请引用表示从度到弧度转换的常量 **R**：

```
radi = 45*NamedConst.R
```

```
radi =

    0.7854
```

常量包

要创建可以按名称访问的常量值的库，请首先创建一个包文件夹，然后定义各种类来组织常量。例如，要实现一组用于进行天文计算的常量，请在名为 **constants** 的包中定义 **AstroConstants** 类：

```
+constants/@AstroConstants/AstroConstants.m
```

该类定义一组已经赋值的 Constant 属性：

```
classdef AstroConstants
  properties (Constant)
  C = 2.99792458e8     % m/s
  G = 6.67259          % m/kgs
  Me = 5.976e24        % Earth mass (kg)
  Re = 6.378e6         % Earth radius (m)
  end
end
```

要使用这组常量，请用完全限定的类名引用它们。例如，以下函数使用在 AstroContants 中定义的一些常量：

```
function E = energyToOrbit(m,r)
  E = constants.AstroConstants.G * constants.AstroConstants.Me * m * ...
    (1/constants.AstroConstants.Re-0.5*r);
end
```

将包导入函数中即无需重复包名称（请参阅 import）：

```
function E = energyToOrbit(m,r)
  import constants.*;
  E = AstroConstants.G * AstroConstants.Me * m * ...
    (1/AstroConstants.Re - 0.5 * r);
end
```

把句柄对象赋予常量属性

如果类使用句柄对象值来定义常量属性，则可以为句柄对象属性赋值。要访问句柄对象，请创建一个局部变量。

例如，ConstMapClass 类定义常量属性。常量属性的值是句柄对象（containers.Map 对象）。

```
classdef ConstMapClass < handle
  properties (Constant)
    ConstMapProp = containers.Map
  end
end
```

要将当前日期赋予 Date 键，请从常量属性返回句柄，然后使用赋值语句左侧的局部变量进行赋值：

```
localMap = ConstMapClass.ConstMapProp
localMap('Date') = datestr(clock);
```

您无法在赋值语句的左侧使用对常量属性的引用。例如，MATLAB 将以下语句解释为创建名为 ConstMapClass 的具有字段 ConstMapProp 的 struct：

```
ConstMapClass.ConstMapProp('Date') = datestr(clock);
```

把任何对象赋予常量属性

您可以将定义类的实例赋给常量属性。MATLAB 在加载该类时创建赋给常量属性的实例。仅当定义类是 handle 类时，才可使用这种编程方式。

MyProject 就是此类的一个示例：

```
classdef MyProject < handle
  properties (Constant)
    ProjectInfo = MyProject
  end
  properties
    Date
    Department
    ProjectNumber
  end
  methods (Access = private)
    function obj = MyProject
      obj.Date = datestr(clock);
      obj.Department = 'Engineering';
      obj.ProjectNumber = 'P29.367';
    end
  end
end
```

通过 Constant 属性引用属性数据：

```
MyProject.ProjectInfo.Date
```

```
ans =

18-Apr-2002 09:56:59
```

由于 MyProject 是句柄类，因此您可以获取赋给常量属性的实例的句柄：

```
p = MyProject.ProjectInfo;
```

使用此句柄访问 MyProject 类中的数据：

```
p.Department
```

```
ans =

Engineering
```

使用此句柄修改 MyProject 类的非常量属性：

```
p.Department = 'Quality Assurance';
```

p 是赋给 ProjectInfo 常量属性的 MyProject 实例的句柄：

```
MyProject.ProjectInfo.Department
```

```
ans =

Quality Assurance
```

清除该类会导致将 MyProject 的新实例赋给 ProjectInfo 属性。

```
clear MyProject
MyProject.ProjectInfo.Department
```

```
ans =

Engineering
```

仅当属性声明为 Constant 时，您才能将定义类的实例赋作属性的默认值。

常量属性 - 不支持 get 事件

常量属性不支持属性 **PreGet** 或 **PostGet** 事件。如果您将 **Constant** 属性的 **GetObservable** 特性设置为 **true**，则 MATLAB 会在类初始化期间发出警告。

另请参阅

相关示例

- “静态数据” (第 4-2 页)

详细信息

- “Named Values”

来自类元数据的信息

特化对象行为

自定义对象索引

本节内容

默认对象索引

默认情况下，MATLAB 类支持对象数组索引。许多类设计不要求对此行为进行修改。

数组使您能够使用下标表示法对数组元素进行引用和赋值。此表示法指定特定数组元素的索引。例如，假设您创建两个数值数组（使用 **randi** 和串联）。

创建一个由 1 到 9 的整数组成的 3×4 数组：

```
A = randi(9,3,4)
```

```
A =

   4   8   5   7
   4   2   6   3
   7   5   7   7
```

创建一个由数字 3、6、9 组成的 1×3 数组：

```
B = [3 6 9];
```

使用括号中的索引值对任一数组的元素进行引用和赋值：

```
B(2) = A(3,4);
B
```

```
B =
   3   7   9
```

MATLAB 默认行为也适用于用户定义的对象。例如，创建由同一个类的对象组成的数组：

```
for k=1:3
  objArray(k) = MyClass;
end
```

引用对象数组 **objArray** 中的第二个元素时，返回当 **k = 2** 时构造的对象：

```
D = objArray(2);
class(D)
```

```
ans =

MyClass
```

您可以将对象赋给由同一个类的对象组成的数组或赋给未初始化的变量：

```
newArray(3,4) = D;
```

对象数组的行为很像 MATLAB 中的数值数组。您不需要实现任何特殊方法即可为您的类提供标准数组行为。

有关数组索引的一般信息，请参阅“数组索引”。

使用模块化索引类自定义对象索引

自 R2021b 开始提供。基于“Code Patterns for subsref and subsasgn Methods”推荐。

要修改类的索引行为，请从一个或多个模块化索引 mixin 类继承。每个类负责一组索引操作：

- **matlab.mixin.indexing.RedefinesParen** - 圆括号引用、赋值和删除
- **matlab.mixin.indexing.RedefinesDot** - 圆点属性引用、方法调用和赋值
- **matlab.mixin.indexing.RedefinesBrace** - 花括号引用和赋值

每个类都定义抽象方法，这些方法处理该类定义的每个索引操作的细节。实现这些方法来执行您的设计所需的操作。

您可以分别从这些类继承。例如，您可以仅从 **RedefinesParen** 继承，从而仅自定义圆括号索引。这种情况下的圆点和花括号索引行为是默认 MATLAB 行为。

您也可以选择只自定义一两个级别的索引，将其他操作转移给另一个 MATLAB 对象。例如，您可以创建一个类，它自定义圆括号索引（使用 **RedefinesParen**），但使用圆点方法调用的默认行为：

```
myInstance(2,1).value
```

请参阅“Customize Parentheses Indexing”中有关此行为的示例。

另请参阅

相关示例

- “Customize Parentheses Indexing for Mapping Class”

end 作为对象索引

为对象定义 end 索引

当您在对象索引表达式中使用 **end** 时，如 **A(4:end)**，**end** 函数会返回与该维度中最后一个元素对应的索引值。

类可以重载 **end** 函数以实现特殊的行为。如果您的类定义一个 **end** 方法，MATLAB 将调用该方法来确定如何解释表达式。

end 方法使用以下调用语法：

```
ind = end(A,k,n)
```

这些参数的说明如下：

- **A** 是对象
- **k** 是使用 **end** 语法的表达式中的索引
- **n** 是表达式中的索引总数
- **ind** 是表达式中要使用的索引值

例如，假设有一个 3×5 数组 **A**。当 MATLAB 遇到以下表达式时：

```
A(end-1,:)
```

MATLAB 使用以下参数调用为对象 **A** 定义的 **end** 方法：

```
ind = end(A,1,2)
```

这些参数意味着 **end** 语句出现在第一个索引中，并且有两个索引。**end** 类方法返回第一个维度的最后一个元素的索引值（在本例中为被减去 **1** 的值）。原始表达式的计算如下：

```
A(3-1,:)
```

有关自定义索引的类中 **end** 的重载的示例，请参阅“Customize Parentheses Indexing for Mapping Class”。

另请参阅

相关示例

- “Objects in Index Expressions”

运算符重载

本节内容
“为什么重载运算符” （第 17-5 页）
“如何定义运算符” （第 17-5 页）
“示例实现 - 可相加对象” （第 17-5 页）
“MATLAB 运算符和关联的函数” （第 17-7 页）

为什么重载运算符

通过实现适合您的类的运算符，您可以将您的类的对象集成到 MATLAB 语言中。例如，包含数值数据的对象可以定义算术运算，如 +、*、-，以便在算术表达式中使用这些对象。通过实现关系运算符，您可以在条件语句（如 **switch** 和 **if** 语句）中使用对象。

如何定义运算符

您可以实现 MATLAB 运算符来处理您的类的对象。要实现运算符，请定义关联的类方法。

每个运算符都有关联的函数（例如，+ 运算符有关联的 **plus.m** 函数）。您可以通过创建具有适当名称的类方法来实现任何运算符。此方法可以执行适合将实现的运算的任何步骤。

有关运算符和关联的函数名称的列表，请参阅 “MATLAB 运算符和关联的函数” （第 17-7 页）。

运算中的对象优先级

用户定义的类优先于内置类。例如，假设 **q** 是 **double** 类的对象，**p** 是用户定义的类。以下两个表达式都会在用户定义的类（如果存在）中生成对 **plus** 方法的调用：

```
q + p
p + q
```

此方法是否可以添加 **double** 类和用户定义的类的对象取决于您如何实现该方法。

当 **p** 和 **q** 是不同类的对象时，MATLAB 应用优先级规则来确定使用哪个方法。

有关 MATLAB 如何确定调用哪个方法的详细信息，请参阅“方法调用” （第 9-9 页）。

运算符优先级

重载运算符保留运算符的原始 MATLAB 优先级。有关运算符优先级的信息，请参阅“运算符优先级”。

示例实现 - 可相加对象

Adder 类通过定义 **plus** 方法来实现此类的对象的相加。**Adder** 将对象的相加定义为 **NumericData** 属性值的相加。**plus** 方法构造并返回一个 **Adder** 对象，该对象的 **NumericData** 属性值是执行相加的结果。

Adder 类还通过定义 **lt** 方法实现小于运算符 (<)。**lt** 方法在比较每个对象的 **NumericData** 属性中的值后，返回一个逻辑值。

```
classdef Adder
  properties
    NumericData
  end
  methods
    function obj = Adder(val)
      obj.NumericData = val;
    end
    function r = plus(obj1,obj2)
      a = double(obj1);
      b = double(obj2);
      r = Adder(a + b);
    end
    function d = double(obj)
      d = obj.NumericData;
    end
    function tf = lt(obj1,obj2)
      if obj1.NumericData < obj2.NumericData
        tf = true;
      else
        tf = false;
      end
    end
  end
end
```

使用双精度转换器，您可以将数值添加到 **Adder** 对象中，并对类的对象执行相加。

```
a = Adder(1:10)
```

```
a =

  Adder with properties:

    NumericData: [1 2 3 4 5 6 7 8 9 10]
```

将两个对象相加：

```
a + a
```

```
ans =

  Adder with properties:

    NumericData: [2 4 6 8 10 12 14 16 18 20]
```

加上一个对象，该对象具有可转换为双精度的任意值：

```
b = uint8(255) + a
```

```
b =

  Adder with properties:

    NumericData: [256 257 258 259 260 261 262 263 264 265]
```

使用 < 运算符比较对象 **a** 和 **b**：

```
a < b
```

```
ans =

     1
```

确保您的类提供实现类设计所需的任何错误检查。

MATLAB 运算符和关联的函数

下表列出了 MATLAB 运算符的对应函数名称。实现运算符以处理数组（标量扩展、向量化算术运算等）时，可能还需要修改索引和串联方式。使用下表中的链接可了解关于每个函数的具体信息。

运算	要定义的方法	描述
a + b	**plus(a,b)**	二元加法
a - b	**minus(a,b)**	二元减法
-a	**uminus(a)**	一元减法
+a	**uplus(a)**	一元加法
a.*b	**times(a,b)**	按元素乘法
a*b	**mtimes(a,b)**	矩阵乘法
a./b	**rdivide(a,b)**	右按元素除法
a.\b	**ldivide(a,b)**	左按元素除法
a/b	**mrdivide(a,b)**	矩阵右除
a\b	**mldivide(a,b)**	矩阵左除
a.^b	**power(a,b)**	按元素求幂
a^b	**mpower(a,b)**	矩阵幂
a < b	**lt(a,b)**	小于
a > b	**gt(a,b)**	大于
a <= b	**le(a,b)**	小于或等于
a >= b	**ge(a,b)**	大于或等于
a ~= b	**ne(a,b)**	不等于
a == b	**eq(a,b)**	相等性
a & b	**and(a,b)**	逻辑 AND
a | b	**or(a,b)**	逻辑 OR
~a	**not(a)**	逻辑非
a:d:b	**colon(a,d,b)**	冒号运算符
a:b	**colon(a,b)**	
a'	**ctranspose(a)**	复共轭转置
a.'	**transpose(a)**	矩阵转置
[a b]	**horzcat(a,b,...)**	水平串联
[a; b]	**vertcat(a,b,...)**	垂直串联
a(s1,s2,...sn)	**subsref(a,s)**	下标引用
a(s1,...,sn) = b	**subsasgn(a,s,b)**	通过下标赋值

运算	要定义的方法	描述
b(a)	**subsindex(a)**	下标索引

另请参阅

相关示例

- “Define Arithmetic Operators”
- “Methods That Modify Default Behavior”

自定义对象显示

定义自定义数据类型

设计相关类