

MATLAB® Coder™

参考



MATLAB®

R2022b



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

MATLAB® Coder™ 参考

© COPYRIGHT 2011–2022 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

商标

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See www.mathworks.com/trademarks for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

专利

MathWorks products are protected by one or more U.S. patents. Please see www.mathworks.com/patents for more information.

修订历史记录

2011 年 4 月	仅限在线版本	版本 2 (版本 2011a) 中的新增内容
2011 年 9 月	仅限在线版本	版本 2.1 (版本 2011b) 中的修订内容
2012 年 3 月	仅限在线版本	版本 2.2 (版本 2012a) 中的修订内容
2012 年 9 月	仅限在线版本	版本 2.3 (版本 2012b) 中的修订内容
2013 年 3 月	仅限在线版本	版本 2.4 (版本 2013a) 中的修订内容
2013 年 9 月	仅限在线版本	版本 2.5 (版本 2013b) 中的修订内容
2014 年 3 月	仅限在线版本	版本 2.6 (版本 2014a) 中的修订内容
2014 年 10 月	仅限在线版本	版本 2.7 (版本 2014b) 中的修订内容
2015 年 3 月	仅限在线版本	版本 2.8 (版本 2015a) 中的修订内容
2015 年 9 月	仅限在线版本	版本 3.0 (版本 2015b) 中的修订内容
2015 年 10 月	仅限在线版本	版本 2.8.1 (版本 2015aSP1) 中的再发布内容
2016 年 3 月	仅限在线版本	版本 3.1 (版本 2016a) 中的修订内容
2016 年 9 月	仅限在线版本	版本 3.2 (版本 2016b) 中的修订内容
2017 年 3 月	仅限在线版本	版本 3.3 (版本 2017a) 中的修订内容
2017 年 9 月	仅限在线版本	版本 3.4 (版本 2017b) 中的修订内容
2018 年 3 月	仅限在线版本	版本 4.0 (版本 2018a) 中的修订内容
2018 年 9 月	仅限在线版本	版本 4.1 (版本 2018b) 中的修订内容
2019 年 3 月	仅限在线版本	版本 4.2 (版本 2019a) 中的修订内容
2019 年 9 月	仅限在线版本	版本 4.3 (版本 2019b) 中的修订内容
2020 年 3 月	仅限在线版本	版本 5.0 (版本 2020a) 中的修订内容
2020 年 9 月	仅限在线版本	版本 5.1 (版本 2020b) 中的修订内容
2021 年 3 月	仅限在线版本	版本 5.2 (版本 2021a) 中的修订内容
2021 年 9 月	仅限在线版本	版本 5.3 (版本 2021b) 中的修订内容
2022 年 3 月	仅限在线版本	版本 5.4 (R2022a) 中的修订内容
2022 年 9 月	仅限在线版本	版本 5.5 (版本 2022b) 中的修订内容

查看 Bug 报告以确定并解决问题

Software is inherently complex and is not free of errors. The output of a code generator might contain bugs, some of which are not detected by a compiler. MathWorks reports critical known bugs brought to its attention on its Bug Report system at www.mathworks.com/support/bugreports/. In the search bar, type the phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers. To save a search, click Save Search.

The bug reports are an integral part of the documentation for each release. Examine periodically all bug reports for a release, as such reports may identify inconsistencies between the actual behavior of a release you are using and the behavior described in this documentation.

In addition to reviewing bug reports, you should implement a verification and validation strategy to identify potential bugs in your design, code, and tools.

1	App
2	函数
3	类
4	对象
5	工具

App

MATLAB Coder

从 MATLAB 代码生成 C 代码或 MEX 函数

说明

MATLAB Coder App 从 MATLAB 代码生成 C 或 C++ 代码。您可以生成：

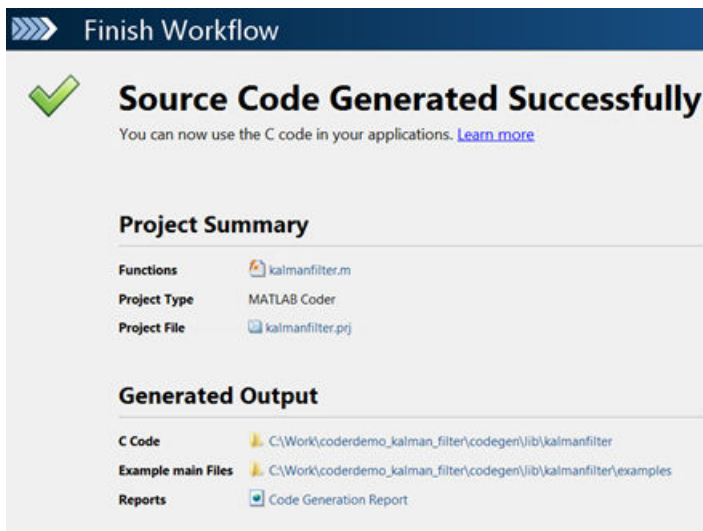
- C 或 C++ 源代码、静态库、动态链接库和可执行文件，您可以将它们集成到 MATLAB 外的现有 C 或 C++ 应用程序中。
- MEX 函数，适用于 MATLAB 函数的加速版本。

基于工作流的用户界面将逐步引导您完成代码生成。使用该 App 可以执行以下操作：

- 创建一个工程或打开一个现有工程。该工程指定输入文件、入口函数输入类型和编译配置。
- 检查代码生成就绪问题，包括不支持的函数。
- 检查您的 MATLAB 函数是否存在运行时问题。
- 使用集成的编辑器修复 MATLAB 代码中的问题。
- 将浮点 MATLAB 代码转换为定点 C 代码（需要 Fixed-Point Designer™ 许可证）。
- 将双精度 MATLAB 代码转换为单精度 C 代码（需要 Fixed-Point Designer 许可证）。
- 通过注释从 MATLAB 代码跟踪到生成的 C 或 C++ 源代码。
- 查看静态代码指标（需要 Embedded Coder® 许可证）。
- 使用软件在环和处理器在环执行验证生成代码的数值行为（需要 Embedded Coder 许可证）。
- 以 MATLAB 脚本的形式导出工程设置。
- 访问生成的文件。
- 将生成的文件打包为单个 zip 文件，以便在 MATLAB 之外部署。

当 App 创建工程时，如果安装了 Embedded Coder 产品，App 将启用 Embedded Coder 功能。如果启用了 Embedded Coder 功能，代码生成需要 Embedded Coder 许可证。要禁用 Embedded Coder 功能，请在工程编译设置中的**所有设置**选项卡上，在**高级**下，将**使用 Embedded Coder 功能**设置为“否”。

MATLAB Online™ 不支持 **MATLAB Coder** App。



打开 MATLAB Coder App

- MATLAB 工具条：在 **App** 选项卡上，点击**代码生成** 下此 App 的图标。
- MATLAB 命令提示符：输入 **coder**。

示例

- “使用 MATLAB Coder App 生成 C 代码”

编程用途

coder

另请参阅

App
定点转换器

函数
codegen

主题
“使用 MATLAB Coder App 生成 C 代码”

仿真数据检查器

检查并比较数据和仿真结果，以验证和迭代模型设计

说明

仿真数据检查器能够可视化并比较多种类型的数据。

使用仿真数据检查器，您可以在工作流的多个阶段检查和比较时间序列数据。以下示例工作流展示了仿真数据检查器如何支持设计周期的所有阶段：

1 “在仿真数据检查器中查看数据” (Simulink)

在配置为将数据记录到仿真数据检查器的模型中运行仿真，或从工作区或 MAT 文件导入数据。在以迭代方式修改模型图、参数值或模型配置时，您可以查看和验证模型输入数据或检查记录的仿真数据。

2 “Inspect Simulation Data” (Simulink)

在多个子图上绘制信号，在指定的绘图坐标区上放大和缩小，并使用数据游标来了解详情和计算数据。您还可以通过 “Create Plots Using the Simulation Data Inspector” (Simulink) 来展示您的分析。

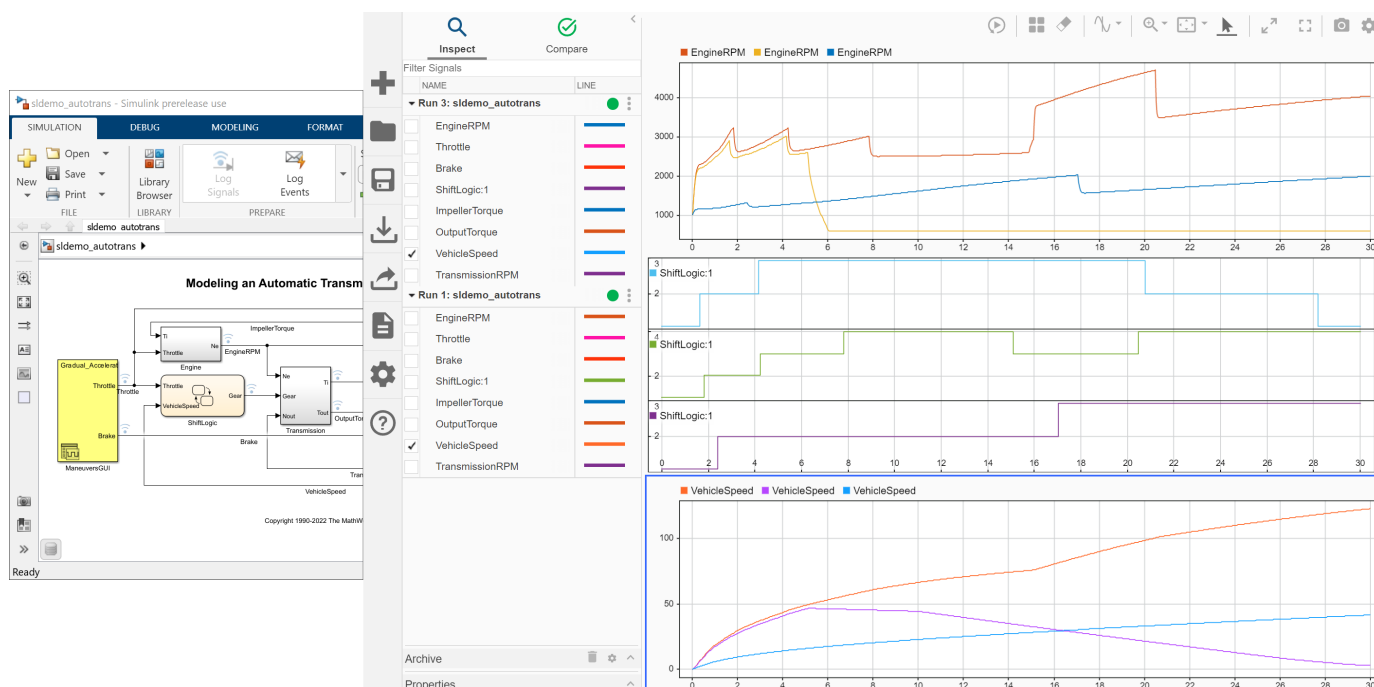
3 “Compare Simulation Data” (Simulink)

比较单个信号或仿真运行，并使用相对、绝对和时间容差分析比较结果。仿真数据检查器中的比较工具有助于迭代设计，并允许您突出显示不符合容差要求的信号。有关比较操作的详细信息，请参阅 “仿真数据检查器如何比较数据” (Simulink)。

4 “保存和共享仿真数据检查器数据和视图” (Simulink)

通过保存仿真数据检查器数据和视图与他人共享您的发现。

您还可以从命令行利用仿真数据检查器的功能。有关详细信息，请参阅 “以编程方式检查和比较数据” (Simulink)。



打开 仿真数据检查器

- Simulink® 工具条：在**仿真**选项卡的**查看结果**下，点击**数据检查器**。
- 点击信号上的流式标记以打开仿真数据检查器并绘制信号。
- MATLAB 命令提示符：输入 `Simulink.sdi.view`。

示例

对多次运行中的信号应用容差

您可以使用仿真数据检查器编程接口修改多次运行中的同一信号的参数。此示例在数据的所有四次运行中为信号增加 **0.1** 的绝对容差。

首先，清空工作区，并加载包含这些数据的仿真数据检查器会话。该会话包括从飞机纵向控制器的 Simulink® 模型的四次仿真中记录的数据。

```
Simulink.sdi.clear
```

```
Simulink.sdi.load('AircraftExample.mldatx');
```

使用 `Simulink.sdi.getRunCount` 函数在仿真数据检查器中获取运行次数。您可以将此数字用作每次运行时执行的 `for` 循环的索引。

```
count = Simulink.sdi.getRunCount;
```

然后，使用一个 `for` 循环对每次运行中的第一个信号赋予 **0.1** 的绝对容差。

```
for a = 1:count
    runID = Simulink.sdi.getRunIDByIndex(a);
```

```
aircraftRun = Simulink.sdi.getRun(runID);  
sig = getSignalByIndex(aircraftRun,1);  
sig.AbsTol = 0.1;  
end
```

- “在仿真数据检查器中查看数据” (Simulink)
- “Inspect Simulation Data” (Simulink)
- “Compare Simulation Data” (Simulink)
- “Iterate Model Design Using the Simulation Data Inspector” (Simulink)

编程用途

`Simulink.sdi.view` 从 MATLAB 命令行打开仿真数据检查器。

版本历史记录

在 R2010b 中推出

另请参阅

函数

`Simulink.sdi.clear` | `Simulink.sdi.clearPreferences` | `Simulink.sdi.snapshot`

主题

- “在仿真数据检查器中查看数据” (Simulink)
- “Inspect Simulation Data” (Simulink)
- “Compare Simulation Data” (Simulink)
- “Iterate Model Design Using the Simulation Data Inspector” (Simulink)

函数

codegen

从 MATLAB 代码生成 C/C++ 代码。

语法

```
codegen options function -args {func_inputs}
codegen options files function -args {func_inputs}
codegen options files function -args {func_inputs} -nargout number_args
codegen options files function1 -args {func1_inputs} ... functionN -args {funcN_inputs}
codegen options files function -args {func_inputs1} ... -args {func_inputsN}
```

codegen project

说明

codegen options function -args {func_inputs} 使用带 **func_inputs** 类型输入的 MATLAB 函数生成 C 或 C++ 代码，并编译生成的代码。使用 **options** 参数指定代码生成配置对象等设置。配置对象控制编译类型（MEX、lib、dll 或 exe）和代码生成参数。有关创建和使用配置对象的信息，请参阅“配置编译设置”、**coder.config**，以及配置对象参考页：**coder.CodeConfig**、**coder.MexCodeConfig** 和 **coder.EmbeddedCodeConfig**。

如果函数没有输入，请省略函数特定的 **-args {func_inputs}** 选项。

codegen options files function -args {func_inputs} 从使用在外部 **files** 中指定的自定义源代码的 MATLAB 函数生成 C/C++ 代码。有关详细信息，请参阅“从生成的代码中调用自定义 C/C++ 代码”和“Configure Build for External C/C++ Code”。

codegen options files function -args {func_inputs} -nargout number_args 生成 C/C++ 代码，并控制从 MATLAB 函数生成的 C/C++ 函数代码的输出参数个数。文件和选项参数是可选的。如果并不需要 MATLAB 函数的所有输出，则请使用 **-nargout** 选项。有关详细信息，请参阅“Specify Number of Entry-Point Function Input or Output Arguments to Generate”。

codegen options files function1 -args {func1_inputs} ... functionN -args {funcN_inputs} 从多个 MATLAB 函数生成 C/C++ 代码。请在每个函数名称的后面分别写入函数的输入参数。还可以对每个函数使用 **-nargout** 选项。从其中生成代码的函数称为入口函数。有关详细信息，请参阅“Generate Code for Multiple Entry-Point Functions”。

codegen options files function -args {func_inputs1} ... -args {func_inputsN} 从 MATLAB 函数生成多签名 MEX 函数。为同一个入口函数的输入参数提供多个 **-args** 设定。使用 **options** 参数指定代码生成配置对象和参数等设置。您必须将编译类型指定为 MEX 函数。不支持其他编译类型（lib、dll 和 exe）。有关详细信息，请参阅“Generate One MEX Function for Multiple Signatures”。

codegen project 从 MATLAB Coder 工程文件（例如 **test.prj**）生成代码。

示例

从 MATLAB 函数生成一个 MEX 函数。

编写一个 MATLAB 函数 **mcadd**，它返回两个值的和。


```
function y = mcadd(u,v) %#codegen
% The directive %#codegen indicates that the function
% is intended for code generation
y = u + v;
end
```

在 MATLAB 命令行中，运行以下 **codegen** 命令。

```
codegen mcadd -args {[0 0 0 0],0}
```

代码生成器在当前工作文件夹中生成 MEX 文件 **mcadd_mex**。

- 如果您没有指定编译目标，则代码生成默认为 MEX 代码生成。默认情况下，代码生成器将生成的 MEX 函数命名为 **mcadd_mex**。
- 要允许生成具有特定类型的 MEX 或 C/C++ 代码，您必须指定 MATLAB 入口函数的所有输入变量的属性（类、大小和复/实性）。在此示例中，需要使用 **-args** 选项为输入提供示例值。代码生成器使用这些示例值来确定第一个输入是由 **double** 类型的实数值组成的 1×4 数组，第二个输入是 **double** 类型的实数标量。

这些示例输入的实际值与代码生成无关。任何具有相同属性（类、大小和复/实性）的其他值对组都会生成相同的代码。请参阅“Specify Properties of Entry-Point Function Inputs”。

在命令行中，调用生成的 MEX 函数 **mcadd_mex**。确保您传递给 **mcadd_mex** 的值的类、大小和复/实性与您在 **codegen** 命令中指定的输入属性相匹配。

```
mcadd_mex([1 1 1 1],5)
```

```
ans =
```

```
6 6 6 6
```

运行具有这些输入值的 MATLAB 函数 **mcadd** 会产生相同的输出。此测试用例验证 **mcadd** 和 **mcadd_mex** 具有相同的行为。

从具有多个签名的 MATLAB 函数生成 MEX 函数

编写一个 MATLAB 函数 **myAdd**，它返回两个值的和。

```
function y = myAdd(u,v) %#codegen
y = u + v;
end
```

在 MATLAB 命令行中，运行以下 **codegen** 命令。

```
codegen -config:mex myAdd.m -args {1,2} -args {int8(2),int8(3)} -args {1:10,1:10} -report
```

代码生成器为 **codegen** 命令中指定的多个签名创建一个 MEX 函数 **myAdd_mex**。有关详细信息，请参阅“Generate One MEX Function for Multiple Signatures”。

在自定义文件夹中生成 C 静态库文件

编写一个 MATLAB 函数 **mcadd**，它返回两个值的和。

```
function y = mcadd(u,v) %#codegen
y = u + v;
```

使用 `-config:lib` 选项在自定义文件夹 `mcaddlib` 中生成 C 库文件。将第一个输入类型指定为由无符号 16 位整数组成的 1×4 向量。将第二个输入指定为双精度标量。

```
codegen -d mcaddlib -config:lib mcadd -args {zeros(1,4,'uint16'),0}
```

生成可执行文件

编写一个 MATLAB 函数 `coderRand`，该函数在开区间 (0,1) 上基于标准均匀分布生成一个随机标量值。

```
function r = coderRand() %#codegen
r = rand();
```

编写一个 C 主函数 `c:\myfiles\main.c`，它调用 `coderRand`。

```
/*
** main.c
*/
#include <stdio.h>
#include <stdlib.h>
#include "coderRand.h"
#include "coderRand_initialize.h"
#include "coderRand_terminate.h"
int main()
{
    coderRand_initialize();
    printf("coderRand=%g\n", coderRand());
    coderRand_terminate();

    puts("Press enter to quit:");
    getchar();

    return 0;
}
```

将代码生成参数配置为包含 C 主函数，然后生成 C 可执行文件。

```
cfg = coder.config('exe')
cfg.CustomSource = 'main.c'
cfg.CustomInclude = 'c:\myfiles'
codegen -config cfg coderRand
```

`codegen` 在当前文件夹中生成 C 可执行文件 `coderRand.exe`，在默认文件夹 `codegen\exe\coderRand` 中生成支持文件。

此示例说明如何在配置对象 `coder.CodeConfig` 中将主函数指定为参数。您也可以在命令行中单独指定包含 `main()` 的文件。您可以使用源、对象或库文件。

有关更详细的示例，请参阅“在应用程序中使用示例 C 主函数”。

生成使用可变大小输入的代码

编写一个接受单个输入的 MATLAB 函数。

```
function y = halfValue(vector) %codegen
    y = 0.5 * vector;
end
```

使用 `coder.typeof` 将输入类型定义为由双精度值组成的行向量，其最大大小为 1×16 ，第二个维度具有可变大小。

```
vectorType = coder.typeof(1, [1 16], [false true]);
```

生成 C 静态库。

```
codegen -config:lib halfValue -args {vectorType}
```

使用全局数据生成代码

编写一个 MATLAB 函数 `use_globals`，该函数接受一个输入参数 `u` 并使用两个全局变量 `AR` 和 `B`。

```
function y = use_globals(u)
    %#codegen
    % Turn off inlining to make
    % generated code easier to read
    coder.inline('never');
    global AR;
    global B;
    AR(1) = u(1) + B(1);
    y = AR * 2;
```

生成 MEX 函数。默认情况下，`codegen` 在当前文件夹中生成名为 `use_globals_mex` 的 MEX 函数。使用 `-globals` 选项在命令行指定全局变量的属性。通过使用 `-args` 选项，指定输入 `u` 为双精度实数标量。

```
codegen -globals {'AR', ones(4), 'B', [1 2 3 4]} use_globals -args {0}
```

您也可以在 MATLAB 工作区中初始化全局数据。在 MATLAB 提示符下，输入：

```
global AR B;
AR = ones(4);
B = [1 2 3];
```

生成 MEX 函数。

```
codegen use_globals -args {0}
```

生成接受枚举类型输入的代码

编写一个函数 `displayState`，它使用枚举数据根据设备的状态来激活 LED 显示。如果绿色 LED 显示亮起，则指示 ON 状态。如果红色 LED 显示亮起，则指示 OFF 状态。

```
function led = displayState(state)
    %#codegen

    if state == sysMode.ON
        led = LEDcolor.GREEN;
    else
        led = LEDcolor.RED;
    end
```

定义一个 LEDColor 枚举。在 MATLAB 路径上，创建一个名为 'LEDColor' 的文件，其中包含：

```
classdef LEDcolor < int32
    enumeration
        GREEN(1),
        RED(2),
    end
end
```

使用现有 MATLAB 枚举中的值创建一个 coder.EnumType 对象。

定义一个 sysMode 枚举。在 MATLAB 路径上，创建一个名为 'sysMode' 的文件，其中包含：

```
classdef sysMode < int32
    enumeration
        OFF(0)
        ON(1)
    end
end
```

从这个枚举创建一个 coder.EnumType 对象。

```
t = coder.typeof(sysMode.OFF);
```

为 displayState 生成一个 MEX 函数。

```
codegen displayState -args {t}
```

生成接受定点输入的静态库

编写一个 MATLAB 语言函数 mcsqrtfi，用于计算定点输入的平方根。

```
function y = mcsqrtfi(x) %#codegen
y = sqrt(x);
```

为定点输入 x 定义 numerictype 和 fimath 属性，并使用 -config:lib 选项为 mcsqrtfi 生成 C 库代码。

```
T = numerictype('WordLength',32, ...
    'FractionLength',23, ...
    'Signed',true)
F = fimath('SumMode','SpecifyPrecision', ...
    'SumWordLength',32, ...
    'SumFractionLength',23, ...
    'ProductMode','SpecifyPrecision', ...
    'ProductWordLength',32, ...
    'ProductFractionLength',23)
% Define a fixed-point variable with these
% numerictype and fimath properties
myfiprops = {fi(4.0,T,F)}
codegen -config:lib mcsqrtfi -args myfiprops
```

codegen 在默认文件夹 codegen/lib/mcsqrtfi 中生成 C 库和支持文件。

生成接受半精度输入的静态 C++ 库

您可以为 MATLAB 代码生成接受半精度输入的代码。有关详细信息，请参阅 [half](#)。

编写一个 MATLAB 函数 `foo`，它返回两个值的和。

```
function y = foo(a,b)
y = a + b;
end
```

在 MATLAB 命令行中，运行以下 `codegen` 命令。

```
codegen -lang:c++ -config:lib foo -args {half(0),half(0)} -report
```

Code generation successful: [View report](#)

代码生成器在 `work\codegen\lib\foo` 中生成静态 C++ 库，其中 `work` 是当前工作文件夹。

要查看代码生成报告，请点击 [View report](#)。在报告查看器中，检查文件 `foo.cpp` 中生成的 C++ 源代码。

```
real16_T foo(real16_T a, real16_T b)
{
    return a + b;
}
```

生成的函数 `foo` 接受并返回半精度值。C++ 半精度类型 `real16_T` 在生成的头文件 `rtwhalf.h` 中定义。检查 `real16_T` 类的 `+` 运算符的定义。

此示例中生成的代码将半精度输入转换为单精度，以单精度执行加法运算，并将结果转换回半精度。

将浮点 MATLAB 代码转换为定点 C 代码。

此示例要求具有 Fixed-Point Designer。

编写一个 MATLAB 函数 `myadd`，它返回两个值的和。

```
function y = myadd(u,v) %#codegen
y = u + v;
end
```

编写一个 MATLAB 函数 `myadd_test` 以测试 `myadd`。

```
function y = myadd_test %#codegen
y = myadd(10,20);
end
```

使用默认设置创建一个 `coder.FixptConfig` 对象 `fixptcfg`。

```
fixptcfg = coder.config('fixpt');
```

设置测试平台名称。

```
fixptcfg.TestBenchName = 'myadd_test';
```

创建一个代码生成配置对象来生成独立的 C 静态库。

```
cfg = coder.config('lib');
```

使用 `-float2fixed` 选项生成代码。

```
codegen -float2fixed fixptcfg -config cfg myadd
```

将 codegen 命令转换为等效的 MATLAB Coder 工程

定义一个 MATLAB 函数 `myadd`，它返回两个值的和。

```
function y = myadd(u,v) %#codegen
y = u + v;
end
```

创建一个用于生成静态库的 `coder.CodeConfig` 对象。将 `TargetLang` 设置为 'C++'。

```
cfg = coder.config('lib');
cfg.TargetLang = 'C++';
```

在 MATLAB 命令行中，创建并运行 `codegen` 命令。将 `myadd` 指定为入口函数。将 `myadd` 的输入指定为维度无界的 `double` 类型可变大矩阵。将 `cfg` 指定为代码配置对象。包括 `-topproject` 选项，以将 `codegen` 命令转换为名称为 `myadd_project.prj` 的等效 MATLAB Coder 工程文件。

```
codegen -config cfg myadd -args {coder.typeof(1,[Inf,Inf]),coder.typeof(1,[Inf,Inf])} -topproject myadd_project.prj
```

```
Project file 'myadd_project.prj' was successfully created.
Open Project
```

代码生成器在当前工作文件夹中创建工程文件 `myadd_project.prj`。使用 `-topproject` 选项运行 `codegen` 不会生成代码。它只创建工程文件。

使用另一个 `codegen` 命令，从 `myadd_project.prj` 生成代码。

```
codegen myadd_project.prj
```

代码生成器在 `work\codegen\lib\myadd` 文件夹中生成 C++ 静态库函数 `myadd`，其中 `work` 是您的当前工作目录。

输入参数

options — 代码生成选项

选项值 | 以空格分隔的选项值列表

在使用 `codegen` 命令时，单个命令行选项的优先级高于配置对象指定的选项。如果各命令行选项之间发生冲突，则最右边的选项优先。选项和其他语法元素的顺序是可互换的。

指定为以下一个或多个值：

<code>-c</code>	生成 C/C++ 代码，但不调用 <code>make</code> 命令。
<code>-config:dll</code>	使用默认配置参数生成动态 C/C++ 库。
<code>-config:exe</code>	使用默认配置参数生成静态 C/C++ 可执行文件。
<code>-config:lib</code>	使用默认配置参数生成静态 C/C++ 库。

-config:mex

使用默认配置参数生成 MEX 函数。

-config:single

使用默认配置参数生成单精度 MATLAB 代码。

-config config_object

需要 Fixed-Point Designer。

指定包含代码生成参数的配置对象。**config_object** 是以下配置对象之一：

- **coder.CodeConfig** - 用于生成独立 C/C++ 库或可执行文件的参数（如果 Embedded Coder 不可用）。
- ```
% Configuration object for a dynamic linked library
cfg = coder.config('dll')
% Configuration object for an executable
cfg = coder.config('exe')
% Configuration object for a static standalone library
cfg = coder.config('lib')
```
- **coder.EmbeddedCodeConfig** - 用于生成独立 C/C++ 库或可执行文件的参数（如果 Embedded Coder 可用）。
- ```
% Configuration object for a dynamic linked library
ec_cfg = coder.config('dll')
% Configuration object for an executable
ec_cfg = coder.config('exe')
% Configuration object for a static standalone library
ec_cfg = coder.config('lib')
```
- **coder.MexCodeConfig** - 用于生成 MEX 代码的参数。
- ```
mex_cfg = coder.config
% or
mex_cfg = coder.config('mex')
```

有关详细信息，请参阅“配置编译设置”。

**-d out\_folder**

将生成的文件存储在 **out\_folder** 指定的绝对或相对路径中。**out\_folder** 不能包含：

- 空格，因为在某些操作系统配置中，空格可能导致代码生成失败。
- 非 7 位 ASCII 字符，如日语字符。

如果 **out\_folder** 指定的文件夹不存在，**codegen** 会创建它。

如果不指定文件夹位置，**codegen** 会在默认文件夹中生成文件：

**codegen/target/fcn\_name.**

**target** 可以是：

- **mex**（对于 MEX 函数）
- **exe**（对于可嵌入的 C/C++ 可执行文件）
- **lib**（对于可嵌入的 C/C++ 库）
- **dll**（对于 C/C++ 动态库）

**fcn\_name** 是命令行中第一个 MATLAB 函数（按字母顺序排列）的名称。

该函数不支持在文件夹名称中使用以下字符：星号 (\*)、问号 (?)、美元符号 (\$) 和镑符号 (#)。

---

**注意** 每次 **codegen** 为相同的代码生成相同类型的输出时，都会删除上一次编译生成的文件。如果要保留以前的某次编译生成的文件，请在开始新的编译之前将这些文件复制到其他位置。

---



**-double2single double2single\_cfg\_name**

使用 `coder.SingleConfig` 对象 `double2single_cfg_name` 指定的设置生成单精度 MATLAB 代码。`codegen` 在文件夹 `codegen/fcn_name/single` 中生成文件。

`fcn_name` 是入口函数的名称。

当与 `-config` 选项结合使用时，还可生成单精度 C/C++ 代码。`codegen` 在文件夹 `codegen/target/folder_name` 中生成单精度文件。

`target` 可以是：

- `mex`（对于 MEX 函数）
- `exe`（对于可嵌入的 C/C++ 可执行文件）
- `lib`（对于可嵌入的 C/C++ 库）
- `dll`（对于 C/C++ 动态库）

`folder_name` 是 `fcn_name` 和 `singlesuffix` 的串联。

`singlesuffix` 是 `coder.SingleConfig` 属性 `OutputFileNameSuffix` 指定的后缀。此文件夹中的单精度文件也有此后缀。

有关详细信息，请参阅“Generate Single-Precision MATLAB Code”。您必须有 Fixed-Point Designer 才能使用此选项。

**-float2fixed float2fixed\_cfg\_name**

当与 **-config** 选项结合使用时，将使用浮点到定点转换配置对象 **float2fixed\_cfg\_name** 指定的设置生成定点 C/C++ 代码。

**codegen** 在文件夹 **codegen/target/fcn\_name\_fixpt** 中生成文件。**target** 可以是：

- **mex**（对于 MEX 函数）
- **exe**（对于可嵌入的 C/C++ 可执行文件）
- **lib**（对于可嵌入的 C/C++ 库）
- **dll**（对于 C/C++ 动态库）

**fcn\_name** 是入口函数的名称。

如果不使用 **-config** 选项，则使用浮点到定点转换配置对象 **float2fixed\_cfg\_name** 指定的设置生成定点 MATLAB 代码。**codegen** 在文件夹 **codegen/fcn\_name/fixpt** 中生成文件。

您必须将 **TestBenchName** 属性设置为 **float2fixed\_cfg\_name**。例如：

```
fixptcfg.TestBenchName = 'myadd_test';
```

此命令指定 **myadd\_test** 是浮点到定点配置对象 **fixptcfg** 的测试文件。

有关详细信息，请参阅“Convert MATLAB Code to Fixed-Point C Code”。您必须有 Fixed-Point Designer 才能使用此选项。

**-g**

指定是否对 C 编译器使用调试选项。如果启用调试模式，C 编译器会禁用一些优化。编译速度会更快，但执行速度会变慢。

**-globals global\_values**

在 MATLAB 文件中指定全局变量的名称和初始值。

**global\_values** 是全局变量名称和初始值组成的元胞数组。**global\_values** 的格式是：

```
{g1, init1, g2, init2, ..., gn, initn}
```

**gn** 是指定为字符向量的全局变量的名称。**initn** 是初始值。例如：

```
-globals {'g', 5}
```

也可以使用以下格式：

```
-globals {global_var, {type, initial_value}}
```

**type** 是类型对象。要创建类型对象，请使用 **coder.typeof**。对于全局元胞数组变量，必须使用此格式。

在使用 **codegen** 生成代码之前，需要初始化全局变量。如果您没有使用 **-globals** 选项为全局变量提供初始值，**codegen** 会检查 MATLAB 全局工作区中的变量。如果不提供初始值，**codegen** 会产生错误。

MATLAB Coder 和 MATLAB 各有自己的全局数据副本。为了保持一致性，只要两者有交互，就请同步其全局数据。如果不同步数据，其全局变量可能会不同。

要为全局变量指定常量值，请使用 **coder.Constant**。例如：

```
-globals {'g', coder.Constant(v)}
```

指定 **g** 为具有常量值 **v** 的全局变量。

有关详细信息，请参阅“Generate Code for Global Data”。

**-I include\_path**

将 **include\_path** 添加到代码生成路径的开头。当 **codegen** 搜索 MATLAB 函数和自定义 C/C++ 文件时，它首先搜索代码生成路径。它不会搜索代码生成路径上的类。类必须位于 MATLAB 搜索路径上。有关详细信息，请参阅“Paths and File Infrastructure Setup”。

如果路径包含非 7 位 ASCII 字符（如日语字符），则 **codegen** 可能在此路径上找不到文件。

如果您的 **include\_path** 包含的路径含有空格，请用双引号将每个实例引起来，例如：

```
'C:\Project "C:\Custom Files"'
```

|                                  |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
|----------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>-jit</code>                | 使用即时 (JIT) 编译来生成 MEX 函数。JIT 编译可以加速 MEX 函数的生成。此选项仅适用于 MEX 函数生成。此选项与某些代码生成功能或选项不兼容，例如自定义代码或使用 OpenMP 库。                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
| <code>-lang:c</code>             | 将生成代码中要使用的语言指定为 C。<br><br>如果没有指定任何目标语言，代码生成器将生成 C 代码。                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
| <code>-lang:c++</code>           | 将生成代码中要使用的语言指定为 C++。                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| <code>-launchreport</code>       | 生成并打开一个代码生成报告。如果未指定此选项，则仅当出现错误或警告消息或者您指定了 <code>-report</code> 选项时， <b>codegen</b> 才会生成报告。                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| <code>-o output_file_name</code> | 使用基本名称 <b>output_file_name</b> 和以下扩展名之一生成 MEX 函数、C/C++ 库或 C/C++ 可执行文件： <ul style="list-style-type: none"><li>• <b>.a</b> 或 <b>.lib</b>（适用于 C/C++ 静态库）</li><li>• <b>.exe</b> 或无扩展名（适用于 C/C++ 可执行文件）</li><li>• <b>.dll</b>（适用于 Microsoft® Windows® 系统上的 C/C++ 动态库）</li><li>• <b>.so</b>（适用于 Linux® 系统上的 C/C++ 动态库）</li><li>• <b>.dylib</b>（适用于 Mac 系统上的 C/C++ 动态库）</li><li>• 生成的 MEX 函数的与平台相关的扩展名</li></ul><br><b>output_file_name</b> 可以是文件名，也可以包含现有路径。 <b>output_file_name</b> 不能包含空格，因为在某些操作系统配置中空格可能导致代码生成失败。<br><br>对于 MEX 函数， <b>output_file_name</b> 必须为有效的 MATLAB 函数名称。<br><br>如果没有为库和可执行文件指定输出文件名，则基本名称是 <b>fcn_1</b> 。 <b>fcn_1</b> 是命令行中指定的第一个 MATLAB 函数的名称。对于 MEX 函数，基本名称是 <b>fcn_1_mex</b> 。您可以运行原始 MATLAB 函数和 MEX 函数，并比较结果。 |

**-O optimization\_option**

根据 **optimization\_option** 的值优化生成的代码：

- **enable:inline** - 启用函数内联。
- **disable:inline** - 禁用函数内联。要了解有关函数内联的详细信息，请参阅“Control Inlining to Fine-Tune Performance and Readability of Generated Code”。
- **enable:openmp** - 使用 OpenMP 库（如果可用）。使用 **codegen** 为 **parfor** 循环生成的 OpenMP 库、MEX 函数或 C/C++ 代码可以在多个线程上运行。
- **disable:openmp** - 禁用 OpenMP 库。禁用 OpenMP 后，**codegen** 将 **parfor** 循环视为 **for** 循环，并生成在单线程上运行的 MEX 函数或 C/C++ 代码。请参阅“Control Compilation of parfor-Loops”。

在命令行上为每次优化指定一次 **-O**。

如果没有指定，则 **codegen** 使用内联和 OpenMP 进行优化。

**-package zip\_file\_name**

将生成的独立代码及其依存关系打包到名为 **zip\_file\_name** 的压缩 ZIP 文件中。然后，您可以使用该 ZIP 文件以转移到另一个开发环境中进行解包并重新编译代码文件。

**packNGo** 函数也提供此打包功能。

**-preservearraydims**

生成使用 N 维索引的代码。有关详细信息，请参阅“Generate Code That Uses N-Dimensional Indexing”。

**-profile**

使用 MATLAB 探查器启用对生成的 MEX 函数的探查。有关详细信息，请参阅“使用 MATLAB 探查器探查 MEX 函数”。

**-report**

生成代码生成报告。如果未指定此选项，则仅当出现错误或警告消息或者您指定了 **-launchreport** 选项时，**codegen** 才会生成报告。

如果您有 Embedded Coder，则此选项还支持生成代码替换报告。

**-reportinfo info**

将有关代码生成的信息导出到 MATLAB 基础工作区的变量 **info** 中。请参阅“Access Code Generation Report Information Programmatically”。

**-rowmajor**

生成使用行优先数组布局的代码。默认为列优先布局。有关详细信息，请参阅“Generate Code That Uses Row-Major Array Layout”。

**-silent**

如果代码生成成功且没有警告，则隐藏所有消息，包括生成报告时。

将显示警告和错误消息。

|                                           |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
|-------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>-singleC</code>                     | 生成单精度 C/C++ 代码。有关详细信息，请参阅“Generate Single-Precision C Code at the Command Line”。                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| <code>-std:c89/c90</code>                 | 您必须有 Fixed-Point Designer 才能使用此选项。<br>为生成的代码使用 C89/90 (ANSI) 语言标准。                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| <code>-std:c99</code>                     | 为生成的代码使用 C99 (ISO) 语言标准。                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| <code>-std:c++03</code>                   | 为生成的代码使用 C++03 (ISO) 语言标准。仅在生成 C++ 代码时才能使用此库。                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| <code>-std:c++11</code>                   | 为生成的代码使用 C++11 (ISO) 语言标准。仅在生成 C++ 代码时才能使用此库。                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| <code>-test test_file</code>              | 运行 <code>test_file</code> ，将调用原始 MATLAB 函数替换为调用 MEX 函数。使用此选项等效于运行 <code>coder.runTest</code> 。<br><br>仅当生成 MEX 函数或使用将 <code>VerificationMode</code> 设置为 'SIL' 或 'PIL' 的配置对象时，才支持此选项。创建具有 <code>VerificationMode</code> 参数的配置对象需要 Embedded Coder 产品。                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| <code>-toproject project_file_name</code> | 将 <code>codegen</code> 命令转换为名为 <code>project_file_name</code> 的等效 MATLAB Coder 工程文件。然后，您可以使用另一个 <code>codegen</code> 命令或 MATLAB Coder 从工程文件中生成代码。<br><br>您还可以使用 <code>-toproject</code> 选项将不完整的 <code>codegen</code> 命令转换为工程文件。例如，要创建仅包含配置对象 <code>cfg</code> 中存储的代码生成参数的工程文件 <code>myProjectTemplate.prj</code> ，请运行以下命令：<br><br><code>codegen -config cfg -toproject myProjectTemplate.prj</code><br><br>在本例中， <code>myProjectTemplate.prj</code> 不包含入口函数或输入类型的设定。因此，您无法从这个工程文件生成代码。您可以在 MATLAB Coder 中打开 <code>myProjectTemplate.prj</code> ，并将其作为模板来创建可用于生成代码的完整工程文件。<br><br>使用 <code>-toproject project_file_name</code> 选项运行 <code>codegen</code> 不会生成代码。它只创建工程文件。<br><br>请参阅“Convert codegen Command to Equivalent MATLAB Coder Project”。 |
| <code>-v</code>                           | 启用详细模式以显示代码生成状态和目标编译日志消息。                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| <code>-?</code>                           | 显示 <code>codegen</code> 命令的帮助。                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |

### function — 从其生成代码的 MATLAB 函数的名称

函数名称

指定为当前工作文件夹中或路径上存在的函数。如果 MATLAB 文件位于包含非 7 位 ASCII 字符（如日语字符）的路径上，`codegen` 命令可能找不到该文件。

如果您正在使用 LCC 编译器，请不要将入口函数命名为 `main`。

示例: `codegen myAddFunction`

### **func\_inputs — MATLAB 函数输入的示例值**

表达式 | 可变 | 字面值 | `coder.Type` 对象

定义前面的 MATLAB 函数输入的大小、类和复/实性的示例值。元胞数组中输入的位置必须对应于 MATLAB 函数定义中输入参数的位置。您也可以提供 `coder.Type` 对象，而不是示例值。要创建 `coder.Type` 对象，请使用 `coder.typeof`。

要生成一个函数，且该函数的输入参数个数少于函数定义的参数个数，请省略不需要的参数示例值。

有关详细信息，请参阅“Specify Properties of Entry-Point Function Inputs”。

示例: `codegen foo -args {1}`

示例: `codegen foo2 -args {1, ones(3,5)}`

示例: `codegen foo3 -args {1, ones(3,5), coder.typeof("hello")}`

### **files — 自定义源文件的名称**

文件名 | 以空格分隔的文件名列表

要包含在生成代码中的、以空格分隔的自定义文件列表。选项、外部文件和函数设定的顺序可以互换。您可以包括以下类型的文件：

- C 文件 (.c)
- C++ 文件 (.cpp)
- 头文件 (.h)
- 目标文件 (.o 或 .obj)
- 库 (.a、.so、.dylib 或 .lib)
- 模板联编文件 (.tmf)

---

**注意** 在以后的版本中将删除对模板联编文件 (TMF) 的支持。请改用工具链方法来编译生成的代码。

---

如果这些文件位于包含非 7 位 ASCII 字符（如日语字符）的路径上，`codegen` 命令可能找不到这些文件。

示例: `codegen foo myLib.lib`

### **number\_args — 生成的入口函数中输出参数的数量**

整数

为前面的 MATLAB 函数生成的 C/C++ 入口函数中的输出参数个数。代码生成器按照输出参数在 MATLAB 函数定义中出现的顺序生成指定数量的输出参数。

示例: `codegen myMLfnWithThreeOuts -nargout 2`

### **project — 工程文件名**

文件名

从 MATLAB Code 创建的工程文件。代码生成器使用该工程文件来设置入口函数、输入类型定义和其他选项。要打开 App 并创建或修改工程文件，请使用 `coder` 函数。

示例: `codegen foo.prj`

## 限制

- 您不能为 MATLAB 脚本生成代码。请将脚本重写为函数来生成代码。
- 不支持在当前文件夹是私有文件夹或 @ 文件夹时生成代码，因为这些文件夹在 MATLAB 中有特殊含义。您可以生成代码来调用 @ 文件夹中的方法和私有文件夹中的函数。

## 提示

- 默认情况下，代码是在文件夹 `codegen/target/function` 中生成的。MEX 函数和可执行文件被复制到当前工作文件夹。
- 为了简化代码生成过程，您可以在单独的脚本中编写代码生成命令。您在该脚本中定义您的函数输入类型和代码生成选项。要生成代码，请调用该脚本。
- 每次 `codegen` 为相同的代码或工程生成相同类型的输出时，都会删除上一次编译生成的文件。如果要保留以前的某次编译生成的文件，请在开始新的编译之前将这些文件复制到其他位置。
- 使用 `coder` 函数打开 MATLAB Coder，并创建一个 MATLAB Coder 工程。该 App 提供了用户界面，便于您添加 MATLAB 文件、定义输入参数和指定编译参数。
- 您可以使用函数语法调用 `codegen`。将 `codegen` 参数指定为字符向量或字符串标量。例如：

```
codegen('myfunction','-args',{2 3},'-report')
```

- 要提供字符串标量作为输入或将 `codegen` 参数指定为字符串标量，请使用函数语法。例如：

```
codegen('myfunction','-args',"mystring",-report')
codegen("myfunction",-args,"mystring",-report')
```

向 `codegen` 的命令形式提供字符串标量输入可能会产生意外的结果。请参阅“选择命令语法或函数语法”。

- 要以编程方式调用 `codegen`，请使用函数语法。例如：

```
A = {'myfunction','-args',{2 3}};
codegen(A{:})
```

## 版本历史记录

在 R2011a 中推出

## 另请参阅

`coder` | `coder.typeof` | `fimath` | `numericType` | `mex` | `fi` | `coder.EnumType` | `coder.runTest` | `parfor` | `coder.FixptConfig` | `coder.config` | `packNGo`

## 主题

“通过命令行生成 C 代码”

“使用 MATLAB Coder App 生成 C 代码”

“Build Process Customization”




# coder

打开 MATLAB Coder

## 语法

```
coder
coder projectname
coder -open projectname
coder -build projectname
coder -new projectname
coder -ecoder false -new projectname
coder -tocode projectname -script scriptname
coder -tocode projectname
coder -toconfig projectname
cfg = coder('-toconfig','projectname')
coder -typeEditor
```

## 说明

`coder` 会打开 MATLAB Coder。要创建工程，请在[选择源文件](#)页上提供入口函数文件名。App 会创建一个默认名称为第一个入口函数文件名的工程。要打开一个现有工程，请在 App 工具栏上，点击 ，然后点击[打开现有工程](#)。

如果安装了 Embedded Coder 产品，该 App 将创建启用了 Embedded Coder 功能的工程。如果启用了 Embedded Coder 功能，代码生成需要 Embedded Coder 许可证。要禁用 Embedded Coder 功能，请在工程编译设置中的[所有设置](#)选项卡上，在[高级](#)下，将[使用 Embedded Coder 功能](#)设置为“否”。

`coder projectname` 使用现有的名为 `projectname.prj` 的工程打开 MATLAB Coder。

`coder -open projectname` 使用现有的名为 `projectname.prj` 的工程打开 MATLAB Coder。

`coder -build projectname` 编译名为 `projectname.prj` 的现有工程。

`coder -new projectname` 打开 MATLAB Coder，创建名为 `projectname.prj` 的工程。如果安装了 Embedded Coder 产品，该 App 将创建启用了 Embedded Coder 功能的工程。要禁用这些功能，请在工程编译设置中的[所有设置](#)选项卡上，在[高级](#)下，将[使用 Embedded Coder 功能](#)设置为“否”。

`coder -ecoder false -new projectname` 打开 MATLAB Coder，创建名为 `projectname.prj` 的工程。App 创建禁用了 Embedded Coder 功能的工程，即使安装了 Embedded Coder 产品也是如此。

`coder -tocode projectname -script scriptname` 将名为 `projectname.prj` 的现有工程转换为等效的 MATLAB 命令脚本。该脚本名为 `scriptname`。

- 如果 `scriptname` 存在，`coder` 会覆盖它。
- 脚本在配置对象中重新生成工程编译配置，并编译工程。脚本：
  - 创建名为 `cfg` 的配置对象。
  - 为函数输入类型定义变量 `ARGS`。

- 为全局数据初始值定义变量 **GLOBALS**。
- 运行 **codegen** 命令。运行脚本时，作为 **codegen** 的参数入口函数必须位于搜索路径上。
- 仅在运行脚本后，**cfg**、**ARGS** 和 **GLOBALS** 才会出现在基础工作区中。

如果工程包括自动定点转换，**coder** 会生成两个脚本：

- 一个是 **scriptname** 脚本，其中包含用于执行以下操作的 MATLAB 命令：
  - 创建与工程具有相同设置的代码配置对象。
  - 运行 **codegen** 命令以将定点 MATLAB 函数转换为定点 C 函数。
- 另一个脚本，其文件名由两部分串联而成：一部分是 **scriptname** 指定的名称，另一部分是工程文件指定的生成的定点文件名后缀。如果 **scriptname** 指定了文件扩展名，则脚本文件名也包括文件扩展名。例如，如果 **scriptname** 是 **myscript.m**，后缀是默认值 **\_fixpt**，则脚本名称是 **myscript\_fixpt.m**。

此脚本包含用于执行以下操作的 MATLAB 命令：

- 创建一个浮点到定点转换配置对象，该对象具有与工程相同的定点转换设置。
- 运行 **codegen** 命令以将浮点 MATLAB 函数转换为定点 MATLAB 函数。

对于包含定点转换的工程，在将工程转换为脚本之前，请完成定点转换过程的**测试数值**步骤。

**coder -tocode projectname** 将名为 **projectname.prj** 的现有工程转换为等效的 MATLAB 命令脚本。它将脚本写入命令行窗口。

**coder -toconfig projectname** 将存储在 MATLAB Coder 工程文件中的代码配置设置导出到代码配置对象。执行此命令将返回对应于 **projectname** 的代码配置对象。有关不同工程文件设置下相应返回的代码配置对象的详细信息，请参阅“Share Build Configuration Settings”。

**cfg = coder('-toconfig','projectname')** 返回一个配置对象，该对象包含存储在某一 MATLAB Coder 工程文件中的代码配置设置。执行此命令将返回对应于 **projectname** 的代码配置对象 **cfg**。有关不同工程文件设置下相应返回的代码配置对象的详细信息，请参阅“Share Build Configuration Settings”。

**coder -typeEditor** 打开一个空的“代码生成器生成类型编辑器”对话框。如果对话框已打开，此命令会将它在屏幕中前置。

请参阅“Create and Edit Input Types by Using the Coder Type Editor”。

## 示例

### 打开现有 MATLAB Coder 工程

使用现有的名为 **my\_coder\_project** 的 MATLAB Coder 工程打开 MATLAB Coder。

```
coder -open my_coder_project
```

### 编译 MATLAB Coder 工程

编译名为 **my\_coder\_project** 的 MATLAB Coder 工程。

```
coder -build my_coder_project
```

## 创建 MATLAB Coder 工程

打开 MATLAB Coder 并创建名为 **my\_coder\_project** 的工程。

```
coder -new my_coder_project
```

## 将 MATLAB Coder 工程转换为 MATLAB 脚本

将名为 **my\_coder\_project.prj** 的 MATLAB Coder 工程转换为名为 **myscript.m** 的 MATLAB 脚本。

```
coder -tocode my_coder_project -script my_script.m
```

## 从 MATLAB Coder 工程创建代码配置对象

定义一个 MATLAB 函数 **myadd**，它返回两个值的总和。

```
function y = myadd(u,v) %#codegen
y = u + v;
end
```

创建 MATLAB Coder 工程 **myadd.prj**：

- 打开 MATLAB Coder。将 **myadd** 指定为入口函数。
- 在**定义输入类型**页中，将 **u** 和 **v** 指定为双精度标量。
- 在**生成代码**页上，将**编译类型**设置为 “MEX”。对于其他工程文件设置，保留其默认值。

在 MATLAB 命令行中，运行以下命令：

```
cfg = coder('-toconfig','myadd.prj');
```

代码生成器创建 **coder.MexCodeConfig** 对象 **cfg**，其中包含存储在 **myadd.prj** 中的代码配置设置。

检查 **cfg** 的属性。

```
cfg =
```

```
 Description: 'class MexCodeConfig: MEX configuration objects with C code.'
 Name: 'MexCodeConfig'
```

```
----- Report -----
```

```
 GenerateReport: true
 LaunchReport: false
 ReportInfoVarName: "
 ReportPotentialDifferences: false
```

```
----- Debugging -----
```

```
 EchoExpressions: true
 EnableDebugging: false
```

EnableMexProfiling: false

----- Code Generation -----

ConstantInputs: 'CheckValues'  
EnableJIT: false  
FilePartitionMethod: 'MapMFileToCFile'  
GenCodeOnly: false  
HighlightPotentialRowMajorIssues: true  
PostCodeGenCommand: "  
PreserveArrayDimensions: false  
RowMajor: false  
TargetLang: 'C'

----- Language And Semantics -----

CompileTimeRecursionLimit: 50  
ConstantFoldingTimeout: 40000  
EnableDynamicMemoryAllocation: true  
DynamicMemoryAllocationThreshold: 65536  
EnableAutoExtrinsicCalls: true  
EnableRuntimeRecursion: true  
EnableVariableSizing: true  
ExtrinsicCalls: true  
GlobalDataSyncMethod: 'SyncAlways'  
InitFltsAndDblsToZero: true  
PreserveVariableNames: 'None'  
SaturateOnIntegerOverflow: true

----- C++ Language Features -----

CppNamespace: "

----- Safety (disable for faster execution) -----

IntegrityChecks: true  
ResponsivenessChecks: true

----- Function Inlining and Stack Allocation -----

InlineStackLimit: 4000  
InlineThreshold: 10  
InlineThresholdMax: 200  
StackUsageMax: 200000

----- Optimizations -----

EnableMemcpy: true  
EnableOpenMP: true  
MemcpyThreshold: 64

----- Comments -----

GenerateComments: true  
MATLABSourceComments: false

----- Custom Code -----

```
CustomHeaderCode: "
CustomInclude: "
CustomInitializer: "
CustomLibrary: "
CustomSource: "
CustomSourceCode: "
CustomTerminator: "
ReservedNameArray: "
```

## 输入参数

### projectname — MATLAB Coder 工程名称

工程名称

您要创建、打开或编译的 MATLAB Coder 工程的名称。工程名称不能包含空格。

### scriptname — 脚本文件的名称

脚本名称

通过将 `-tocode` 选项与 `-script` 选项结合使用来创建的脚本的名称。脚本名称不能包含空格。

## 输出参数

### cfg — 代码配置对象

`coder.MexCodeConfig` | `coder.CodeConfig` | `coder.EmbeddedCodeConfig`

包含存储在 MATLAB Coder 工程文件中的配置设置的代码配置对象。

## 限制

- MATLAB Online 不支持 `coder` 函数。

## 提示

- 如果您是共享 Embedded Coder 许可证，请使用 `coder -ecoder false -new projectname` 创建一个不需要此许可证的工程。如果安装了 Embedded Coder 产品，App 将创建禁用了 Embedded Coder 功能的工程。禁用这些功能后，代码生成不需要 Embedded Coder 许可证。要启用 Embedded Coder 功能，请在工程编译设置中的**所有设置**选项卡上，在**高级**下，将**使用 Embedded Coder 功能**设置为“是”。
- 创建工程或打开现有工程会导致其他 MATLAB Coder 或定点转换器工程关闭。
- 如果您的安装不包括 Embedded Coder 产品，则不会显示 Embedded Coder 设置。但是，这些设置的值会保存在工程文件中。如果您在包含 Embedded Coder 产品的安装中打开该工程，您会看到这些设置。
- 定点转换器工程在定点转换器中打开。要将工程转换为 MATLAB Coder 工程，请在定点转换器中执行以下操作：

- 1 单击 ，然后选择**将工程重新打开为**。
- 2 选择“MATLAB Coder”。

### 替代方法

- 在 **App** 选项卡上的**代码生成**部分中，点击 **MATLAB Coder**。
- 在命令行中使用 `codegen` 函数生成代码。

### 版本历史记录

在 R2011a 中推出

### 另请参阅

**MATLAB Coder** | `codegen` | `coder.MexCodeConfig` | `coder.CodeConfig` | `coder.EmbeddedCodeConfig`

### 主题

“使用 MATLAB Coder App 生成 C 代码”  
“Convert MATLAB Coder Project to MATLAB Script”  
“Convert Fixed-Point Conversion Project to MATLAB Scripts”  
“Share Build Configuration Settings”  
“Convert MATLAB Code to Fixed-Point C Code”  
“Create and Edit Input Types by Using the Coder Type Editor”

# coder.allowpcode

包: coder

从 P 代码文件控制代码生成

## 语法

```
coder.allowpcode('plain')
```

## 说明

**coder.allowpcode('plain')** 允许您生成 P 代码文件，然后您可以将其编译为优化的 MEX 函数或可嵌入的 C/C++ 代码。此函数不会混淆处理生成的 MEX 函数或可嵌入的 C/C++ 代码。

使用此功能，您可以将算法作为提供代码生成优化的受保护的 P 代码文件进行分发。

在顶层函数的控制流语句（如 **if**、**while**、**switch**）和函数调用之前调用此函数。

MATLAB 函数可以调用 P 代码。当一个文件的 **.m** 和 **.p** 版本存在于同一文件夹中时，P 代码文件优先。

在代码生成期间外，系统会忽略 **coder.allowpcode**。

## 示例

### 从 P 代码文件生成优化的可嵌入代码

编写函数 **p\_abs**，它返回其输入的绝对值：

```
function out = p_abs(in) %#codegen
% The directive %#codegen indicates that the function
% is intended for code generation
coder.allowpcode('plain');
out = abs(in);
```

生成 P 代码文件。在 MATLAB 命令行窗口中，输入：

```
pcode p_abs
```

P 代码文件 **p\_abs.p** 会出现在当前文件夹中。

通过使用 **-args** 选项指定输入参数的大小、类和复/实性（需要 MATLAB Coder 许可证），为 **p\_abs.p** 生成 MEX 函数。

```
codegen p_abs -args { int32(0) }
```

**codegen** 在当前文件夹中生成一个 MEX 函数。

如果您有 MATLAB Coder，则可为 **p\_abs.p** 生成可嵌入的 C 代码。

```
codegen p_abs -config:lib -args { int32(0) };
```

`codegen` 在 `codegen\lib\p_abs` 文件夹中生成 C 库代码。

## 版本历史记录

在 R2011a 中推出

### 扩展功能

#### C/C++ 代码生成

使用 MATLAB® Coder™ 生成 C 代码和 C++ 代码。

#### GPU 代码生成

使用 GPU Coder™ 为 NVIDIA® GPU 生成 CUDA® 代码。

### 另请参阅

`pcode` | `codegen`

#### 主题

"编译指令 %#codegen"



# coder.ceval

调用外部 C/C++ 函数

## 语法

```
coder.ceval(cfun_name)
coder.ceval(cfun_name,cfun_arguments)

coder.ceval('-global',cfun_name)
coder.ceval('-global',cfun_name,cfun_arguments)

coder.ceval('-gpudevicefcn',devicefun_name,devicefun_arguments)

coder.ceval('-layout:rowMajor',cfun_name,cfun_arguments)
coder.ceval('-layout:columnMajor',cfun_name,cfun_arguments)
coder.ceval('-layout:any',cfun_name,cfun_arguments)

cfun_return = coder.ceval(__)
```

## 说明

`coder.ceval(cfun_name)` 执行 `cfun_name` 指定的外部 C/C++ 函数。在外部 C/C++ 源文件或库中定义 `cfun_name`。将外部源代码、库和头文件提供给代码生成器。

`coder.ceval(cfun_name,cfun_arguments)` 使用参数 `cfun_arguments` 执行 `cfun_name`。`cfun_arguments` 是按照 `cfun_name` 要求的顺序排列的逗号分隔的输入参数列表。

默认情况下，只要 C/C++ 支持按值传递参数，`coder.ceval` 就会按值向 C/C++ 函数传递参数。要使 `coder.ceval` 按引用传递参数，请使用 `coder.ref`、`coder.rref` 和 `coder.wref` 构造。如果 C/C++ 不支持按值传递参数，例如说参数是一个数组，则 `coder.ceval` 将按引用传递参数。如果您不使用 `coder.ref`、`coder.rref` 或 `coder.wref`，则生成的代码中可能会出现参数的副本，以强制执行用于数组的 MATLAB 语义。

`coder.ceval('-global',cfun_name)` 执行 `cfun_name`，并指示 `cfun_name` 使用一个或多个 MATLAB 全局变量。然后，代码生成器可以产生与此全局变量用法一致的代码。

---

**注意** 只有代码生成中支持 `-global` 标志。在 MATLAB Function 模块中调用 `coder.ceval` 时，不能包含此标志。

---

`coder.ceval('-global',cfun_name,cfun_arguments)` 使用参数 `cfun_arguments` 执行 `cfun_name`，并指示 `cfun_name` 使用一个或多个 MATLAB 全局变量。

`coder.ceval('-gpudevicefcn',devicefun_name,devicefun_arguments)` 允许您从核内调用 CUDA® GPU `__device__` 函数。'-gpudevicefcn' 向 `coder.ceval` 指示目标函数在 GPU 设备上。`devicefun_name` 是 `__device__` 函数的名称，`devicefun_arguments` 是按 `devicefun_name` 要求的顺序以逗号分隔的输入参数列表。此选项需要 GPU Coder™ 产品。

`coder.ceval('-layout:rowMajor',cfun_name,cfun_arguments)` 带参数 `cfun_arguments` 执行 `cfun_name`，并传递以行优先布局存储的数据。当从使用列优先布局的函数调用时，代码生成器将输入转换为行优先布局，并将输出转换回列优先布局。要获得更短的语法，请使用 `coder.ceval('-row',...)`。

`coder.ceval('-layout:columnMajor',cfun_name,cfun_arguments)` 带参数 `cfun_arguments` 执行 `cfun_name`，并传递以列优先布局存储的数据。当从使用行优先布局的函数调用时，代码生成器将输入转换为列优先布局，并将输出转换回行优先布局。要获得更短的语法，请使用 `coder.ceval('-col',...)`。

`coder.ceval('-layout:any',cfun_name,cfun_arguments)` 带参数 `cfun_arguments` 执行 `cfun_name`，并使用当前数组布局传递数据，即使数组布局不匹配也是如此。代码生成器不转换输入或输出数据的数组布局。

`cfun_return = coder.ceval( __ )` 执行 `cfun_name` 并返回单个标量值 `cfun_return`，它对应于 C/C++ 函数在 `return` 语句中返回的值。为了与 C/C++ 保持一致，`coder.ceval` 只能返回标量值。它不能返回数组。可以将此选项与前面语法中的任何输入参数组合一起使用。

## 示例

### 调用外部 C 函数

从要其生成 C 代码的 MATLAB 函数调用 C 函数 `foo(u)`。

为函数 `foo` 创建 C 头文件 `foo.h`，该函数接受两个 `double` 类型的输入参数，并返回 `double` 类型的值。

```
double foo(double in1, double in2);
```

编写 C 函数 `foo.c`。

```
#include <stdio.h>
#include <stdlib.h>
#include "foo.h"

double foo(double in1, double in2)
{
 return in1 + in2;
}
```

编写使用 `coder.ceval` 调用 `foo` 的函数 `callfoo`。将源代码和头文件提供给函数中的代码生成器。

```
function y = callfoo %#codegen
y = 0.0;
if coder.target('MATLAB')
 % Executing in MATLAB, call MATLAB equivalent of
 % C function foo
 y = 10 + 20;
else
 % Executing in generated code, call C function foo
 coder.updateBuildInfo('addSourceFiles','foo.c');
 coder.cinclude('foo.h');
 y = coder.ceval('foo', 10, 20);
end
end
```

为函数 `callfoo` 生成 C 库代码。`codegen` 函数在 `\codegen\lib\callfoo` 子文件夹中生成 C 代码。

```
codegen -config:lib callfoo -report
```

## 调用 C 库函数

从 MATLAB 代码中调用 C 库函数：

编写 MATLAB 函数 `myabsval`。

```
function y = myabsval(u)
%#codegen
y = abs(u);
```

通过使用 `-args` 选项指定输入参数的大小、类型和复/实性，为 `myabsval` 生成 C 静态库。

```
codegen -config:lib myabsval -args {0.0}
```

`codegen` 函数在文件夹 `\codegen\lib\myabsval` 中创建库文件 `myabsval.lib` 和头文件 `myabsval.h`。（库文件扩展名可以根据您的平台而有所不同。）它在同一个文件夹中生成函数 `myabsval_initialize` 和 `myabsval_terminate`。

编写一个 MATLAB 函数，以使用 `coder.ceval` 调用生成的 C 库函数。

```
function y = callmyabsval(y)
%#codegen
% Check the target. Do not use coder.ceval if callmyabsval is
% executing in MATLAB
if coder.target('MATLAB')
 % Executing in MATLAB, call function myabsval
 y = myabsval(y);
else
 % add the required include statements to generated function code
 coder.updateBuildInfo('addIncludePaths','$(START_DIR)\codegen\lib\myabsval');
 coder.cinclude('myabsval_initialize.h');
 coder.cinclude('myabsval.h');
 coder.cinclude('myabsval_terminate.h');

 % Executing in the generated code.
 % Call the initialize function before calling the
 % C function for the first time
 coder.ceval('myabsval_initialize');

 % Call the generated C library function myabsval
 y = coder.ceval('myabsval',y);

 % Call the terminate function after
 % calling the C function for the last time
 coder.ceval('myabsval_terminate');
end
```

生成 MEX 函数 `callmyabsval_mex`。在命令行中提供生成的库文件。

```
codegen -config:mex callmyabsval codegen\lib\myabsval\myabsval.lib -args {-2.75}
```

您可以使用 `coder.updateBuildInfo` 在函数中指定该库，而不是在命令行中提供库。使用此选项可预配置编译。将以下行添加到 `else` 模块中：

```
coder.updateBuildInfo('addLinkObjects','myabsval.lib','$(START_DIR)\codegen\lib\myabsval',100,true,true);
```

---

**注意** 只有使用 MATLAB Coder 生成代码时，才支持 `START_DIR` 宏。

---

运行调用库函数 `myabsval` 的 MEX 函数 `callmyabsval_mex`。

```
callmyabsval_mex(-2.75)
```

```
ans =
```

```
2.7500
```

调用 MATLAB 函数 `callmyabsval`。

```
callmyabsval(-2.75)
```

```
ans =
```

```
2.7500
```

`callmyabsval` 函数在 MATLAB 和代码生成中的执行均展示出了预期的行为。

### 调用使用全局变量的 C 函数

调用修改全局变量的 C 函数时，请使用 `'-global'` 标志。

编写调用 C 函数 `addGlobal` 的 MATLAB 函数 `useGlobal`。使用 `'-global'` 标志向代码生成器指示 C 函数使用全局变量。

```
function y = useGlobal()
global g;
t = g;
% compare execution with/without '-global' flag
coder.ceval('-global','addGlobal');
y = t;
end
```

为函数 `addGlobal` 创建 C 头文件 `addGlobal.h`。

```
void addGlobal(void);
```

在文件 `addGlobal.c` 中编写 C 函数 `addGlobal`。此函数包含代码生成器在您为函数 `useGlobal` 生成代码时创建的头文件 `useGlobal_data.h`。此头文件包含 `g` 的全局变量声明。

```
#include "addGlobal.h"
#include "useGlobal_data.h"
void addGlobal(void) {
 g++;
}
```

为 `useGlobal` 生成 MEX 函数。要定义代码生成器的输入，请在工作区中声明全局变量。

```
global g;
g = 1;
codegen useGlobal -report addGlobal.h addGlobal.c
y = useGlobal_mex();
```

使用 `'-global'` 标志时，MEX 函数产生结果  $y = 1$ 。 `'-global'` 标志向代码生成器指示 C 函数可能修改全局变量。对于 `useGlobal`，代码生成器产生以下代码：

```
real_T useGlobal(const emlrtStack *sp)
{
 real_T y;
 (void)sp;
 y = g;
 addGlobal();
 return y;
}
```

如果没有 `'-global'` 标志，MEX 函数将产生  $y = 2$ 。因为没有关于 C 函数修改 `g` 的指示，代码生成器假定 `y` 和 `g` 是相同的。将生成以下 C 代码：

```
real_T useGlobal(const emlrtStack *sp)
{
 (void)sp;
 addGlobal();
 return g;
}
```

### 调用使用不同数组布局的 C 函数

假设您有设计为使用行优先布局的 C 函数 `testRM`。您要将此函数集成到对数组执行运算的 MATLAB `bar` 函数中。函数 `bar` 设计为使用列优先布局，采用 `coder.columnMajor` 指令。

```
function out = bar(in)
%#codegen
coder.columnMajor;
coder.ceval('-layout:rowMajor','testRM', ...
 coder.rref(in),coder.wref(out));
end
```

在生成的代码中，代码生成器会先在变量 `in` 上插入从列优先布局到行优先布局的布局转换，然后再将其传递给 `testRM`。在输出变量 `out` 上，代码生成器会插入一个转换回列优先布局的布局转换。

通常，如果您没有为 `coder.ceval` 指定 `layout` 选项，则外部函数参数假定使用列优先布局。

### 调用接受复数输入的 C 函数

假设您有一个调用自定义 C 代码的 MATLAB 函数，该函数接受复数输入。您必须定义您的 C 代码输入参数，以便来自 MATLAB 函数的复数输入可以映射到您的 C 代码。

在生成代码中，复数定义为 `struct`，它有两个字段 `re` 和 `im`，分别是复数的实部和虚部。此 `struct` 是在头文件 `rtwtypes.h` 中定义的，您可以在当前路径的 `codegen\lib\functionName` 文件夹中找到此头文件。`struct` 的定义如下：

```
typedef struct {
 real32_T re; /*Real Component*/
 real32_T im; /*Imaginary Component*/
} creal_T;
```

有关详细信息，请参阅“Mapping MATLAB Types to Types in Generated Code”。

要集成的 C 代码必须包含 `rtwtypes.h` 头文件。C 代码示例 `foo.c` 如下所示：

```
#include "foo.h"
#include <stdio.h>
#include <stdlib.h>
#include "rtwtypes.h"

double foo(creal_T x) {
 double z = 0.0;
 z = x.re*x.re + x.im*x.im;
 return (z);
}
```

`struct` 命名为 `creal_T`。头文件 `foo.h` 也必须定义为：

```
#include "rtwtypes.h"
double foo(creal_T x);
```

MATLAB 代码通过使用具有复数输入的 `coder.ceval` 函数来执行 `foo.c`：

```
function y = complexCeval %#codegen
y = 0.0;
coder.updateBuildInfo('addSourceFiles','foo.c');
coder.cinclude('foo.h');
y = coder.ceval('foo', 10+20i);
end
```

`coder.ceval` 命令接受复数输入。代码生成器将复数映射到 `struct creal_T` 变量 `x` 及其字段 `re` 和 `im`。

通过运行以下命令，为函数 `complexCeval` 生成代码：

```
codegen -config:lib -report complexCeval
```

## 输入参数

**cfun\_name** — C/C++ 函数名称

字符向量 | 字符串标量

要调用的外部 C/C++ 函数的名称。

示例： `coder.ceval('foo')`

数据类型： `char` | `string`

**cfun\_arguments** — C/C++ 函数参数

标量变量 | 数组 | 数组的元素 | 结构体 | 结构体字段 | 对象属性

按照 **cfun\_name** 要求的顺序排列的逗号分隔的输入参数列表。

示例： `coder.ceval('foo', 10, 20);`

示例： `coder.ceval('myFunction', coder.ref(x));`

数据类型： `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `struct`

复数支持： 是

## 限制

- 对于使用 `coder.extrinsic` 声明为外部函数的函数，不能使用 `coder.ceval`。
- 当 LCC 编译器创建库时，它会向库函数名称添加一个前导下划线。如果库的编译器是 LCC，而您的代码生成编译器不是 LCC，则您必须向函数名称添加前导下划线，例如 `coder.ceval('_mylibfun')`。如果库的编译器不是 LCC，而 MATLAB 代码从该库中调用函数，则您不能使用 LCC 从该代码中生成代码。这些库函数名称没有 LCC 编译器所要求的前导下划线。
- 如果某属性具有 `get` 方法、`set` 方法或验证程序，或者是具有某些特性的 System object™ 属性，则您不能按引用将该属性传递给外部函数。请参阅“Passing By Reference Not Supported for Some Properties”。
- 行优先代码生成不支持将可变大小矩阵作为入口函数参数。

## 提示

- 对于代码生成，您必须在调用 `coder.ceval` 之前指定返回值和输出参数的类型、大小和复/实性数据类型。
- 要将 `coder.ceval` 应用于一个函数，而该函数接受或返回 MATLAB 代码中不存在的变量，如指针、用于文件 I/O 的 `FILE` 类型和 C/C++ 宏，请使用 `coder.opaque` 函数。
- `coder.ceval` 只能在 MATLAB 中用于代码生成。在未编译的 MATLAB 代码中，`coder.ceval` 将生成错误。要确定 MATLAB 函数是否在 MATLAB 中执行，请使用 `coder.target`。如果函数在 MATLAB 中执行，则调用 C/C++ 函数的 MATLAB 版本。
- 使用 `coder.ceval` 调用的外部代码和生成的代码在同一个进程中运行，并共享内存。如果外部代码错误地写入包含生成的代码使用的数据结构体的内存，可能导致进程出现意外行为或崩溃。例如，如果外部代码尝试在其结束后将数据写入数组，进程可能会出现意外行为或崩溃。
- MATLAB 在 Windows 平台上使用 UTF-8 作为其系统编码。因此，从生成的 MEX 函数中进行的系统调用接受并返回 UTF-8 编码的字符串。相反，由 MATLAB Coder 生成的代码使用 Windows 区域设置指定的编码对文本数据进行编码。因此，如果您的 MATLAB 入口函数使用 `coder.ceval` 来调用接受不同系统编码的外部 C/C++ 函数，则生成的 MEX 函数可能会产生乱码文本。如果发生这种情况，您必须更新外部 C/C++ 函数来处理这种情况。

## 版本历史记录

在 R2011a 中推出

## 扩展功能

### C/C++ 代码生成

使用 MATLAB® Coder™ 生成 C 代码和 C++ 代码。

### GPU 代码生成

使用 GPU Coder™ 为 NVIDIA® GPU 生成 CUDA® 代码。

## 另请参阅

`coder.ref` | `coder.rref` | `coder.wref` | `coder.target` | `codegen` | `coder.extrinsic` | `coder.opaque` | `coder.columnMajor` | `coder.rowMajor` | `coder.updateBuildInfo` | `coder.ExternalDependency` | `coder.reservedName`

### 主题

“外部代码集成”

"Generate Code That Uses Row-Major Array Layout"

"Interface with Row-Major Data in MATLAB Function Block" (Simulink)

"Unknown Output Type for coder.ceval"



# coder.cinclude

在生成的代码中包含头文件

## 语法

```
coder.cinclude(headerfile)
coder.cinclude(headerfile,'InAllSourceFiles',allfiles)
```

## 说明

**coder.cinclude(headerfile)** 在生成的 C/C++ 源代码中包含头文件。

MATLAB Coder 在从包含 **coder.cinclude** 调用的 MATLAB 代码生成的 C/C++ 源文件中生成 include 语句。

在 Simulink 模型中，当 MATLAB Function 模块中出现 **coder.cinclude** 调用时，代码生成器会将 include 语句放在模型头文件中。

---

**注意** 尽可能靠近需要头文件的 **coder.ceval** 调用来放置 **coder.cinclude** 调用。

---

**coder.cinclude(headerfile,'InAllSourceFiles',allfiles)** 使用 **allfiles** 选项确定是否在几乎所有 C/C++ 源文件中包含头文件。

如果 **allfiles** 为 **true**，则 MATLAB Coder 在几乎所有 C/C++ 源文件中生成 include 语句，但一些实用工具文件除外。此行为是 R2016a 及更早版本中的 **coder.cinclude** 行为。这些附加文件中如果存在 include 语句，会使编译时间增加并使生成的代码的可读性降低。仅当代码依赖于旧行为时，才使用此选项。如果 **allfiles** 为 **false**，则行为与 **coder.cinclude(headerfile)** 的行为相同。

在 MATLAB Function 模块中，**coder.cinclude(headerfile,'InAllSourceFiles', allfiles)** 与 **coder.cinclude(headerfile)** 相同。

## 示例

### 在使用 MATLAB Coder codegen 命令生成的 C/C++ 代码中包含头文件

从调用外部 C 函数的 MATLAB 函数中生成代码。使用 **coder.cinclude** 将所需头文件包含在生成的 C 代码中。

在可写文件夹中，创建子文件夹 **mycfiles**。

编写一个使输入翻倍的 C 函数 **myMult2.c**。将其保存在 **mycfiles** 中。

```
#include "myMult2.h"
double myMult2(double u)
{
 return 2 * u;
}
```

编写头文件 `myMult2.h`。将其保存在 `mycfiles` 中。

```
#if !defined(MYMULT2)
#define MYMULT2
extern double myMult2(double);
#endif
```

编写 MATLAB 函数 `myfunc`，该函数包含 `myMult2.h` 并且仅为代码生成调用 `myMult2`。

```
function y = myfunc
%#codegen
y = 21;
if ~coder.target('MATLAB')
 % Running in generated code
 coder.cinclude('myMult2.h');
 y = coder.ceval('myMult2', y);
else
 % Running in MATLAB
 y = y * 2;
end
end
```

为静态库创建代码配置对象。指定 `myMult2.h` 和 `myMult2.c` 的位置

```
cfg = coder.config('lib');
cfg.CustomInclude = fullfile(pwd,'mycfiles');
cfg.CustomSource = fullfile(pwd,'mycfiles','myMult2.c');
```

生成代码。

```
codegen -config cfg myfunc -report
```

文件 `myfunc.c` 包含以下语句：

```
#include "myMult2.h"
```

该 `include` 语句不会出现在任何其他文件中。

### 在从 Simulink 模型中的 MATLAB Function 模块生成的 C/C++ 代码中包含头文件

从调用外部 C 函数的 MATLAB Function 模块中生成代码。使用 `coder.cinclude` 将所需头文件包含在生成的 C 代码中。

在可写文件夹中，创建子文件夹 `mycfiles`。

编写一个使输入翻倍的 C 函数 `myMult2.c`。将其保存在 `mycfiles` 中。

```
#include "myMult2.h"
double myMult2(double u)
{
 return 2 * u;
}
```

编写头文件 `myMult2.h`。将其保存在 `mycfiles` 中。

```
#if !defined(MYMULT2)
#define MYMULT2
```

```
extern double myMult2(double);
#endif
```

创建一个包含 MATLAB Function 模块的 Simulink 模型，该模块连接到 Outport 模块。



在 MATLAB Function 模块中，添加包含 `myMult2.h` 并调用 `myMult2` 的函数 `myfunc`。

```
function y = myfunc
%#codegen
y = 21;
coder.cinclud('myMult2.h');
y = coder.ceval('myMult2', y);
% Specify the locations of myMult2.h and myMult2.c
coder.extrinsic('pwd', 'fullfile');
customDir = coder.const(fullfile(pwd, 'myfiles'));
coder.updateBuildInfo('addIncludePaths', customDir);
coder.updateBuildInfo('addSourcePaths', customDir);
coder.updateBuildInfo('addSourceFiles', 'myMult2.c');
end
```

打开“配置参数”对话框。

在求解器窗格上，选择定步长求解器。

将模型另存为 `mymodel`。

编译模型。

文件 `mymodel.h` 包含以下语句：

```
#include "myMult2.h"
```

要阅读有关在 MATLAB Function 模块中集成自定义代码的详细信息，请参阅“使用 MATLAB Function 模块集成 C 代码”（Simulink）。

## 输入参数

### headerfile — 头文件的名称

字符向量 | 字符串标量

指定为字符向量或字符串标量的头文件的名称。`headerfile` 必须为编译时常量。

使用尖括号 `<>` 将系统头文件名称括起来。为系统头文件生成的 `#include` 语句的格式为 `#include <sysheader>`。系统头文件必须位于标准位置或包含路径中。通过使用代码生成自定义代码参数来指定包含路径。

示例： `coder.cinclud('<sysheader.h>')`

对于不是系统头文件的头文件，省略尖括号。对于不是系统头文件的头文件，生成的 `#include` 语句的格式为 `#include "myHeader"`。头文件必须位于当前文件夹或包含路径中。通过使用代码生成自定义代码参数来指定包含路径。

示例: `coder.cinclude('myheader.h')`

数据类型: `char`

### **allfiles — 所有源文件选项**

`true` | `false`

在所有生成的 C/C++ 源文件中包含头文件的选项。如果 `allfiles` 为 `true`，则 MATLAB Coder 在几乎所有 C/C++ 源文件中生成 `include` 语句，但一些实用工具文件除外。如果 `allfiles` 为 `false`，则行为与 `coder.cinclude(headerfile)` 的行为相同。

在 MATLAB Function 模块中，代码生成器将忽略所有源文件选项。

数据类型: `logical`

## **限制**

- 不要在运行时条件构造（如 `if` 语句、`switch` 语句、`while` 循环和 `for` 循环）中调用 `coder.cinclude`。您可以在编译时条件语句（如 `coder.target`）中调用 `coder.cinclude`。例如：

```
...
if ~coder.target('MATLAB')
 coder.cinclude('foo.h');
 coder.ceval('foo');
end
...
```

## **提示**

- 在 `coder.ceval` 调用之前，调用 `coder.cinclude` 以包含 `coder.ceval` 调用的外部函数所需的头文件。
- 生成的 C/C++ 代码中的外部 `include` 语句会增加编译时间并降低代码可读性。为避免 MATLAB Coder 生成的代码中出现外部 `include` 语句，请遵循以下最佳做法：
  - 尽可能靠近需要头文件的 `coder.ceval` 调用来放置 `coder.cinclude` 调用。
  - 不要将 `allfiles` 设置为 `true`。

对于 MATLAB Function 模块，代码生成器在模型头文件中生成 `include` 语句。

- 在 R2016a 及更早版本中，对于任何 `coder.cinclude` 调用，MATLAB Coder 都会在几乎所有生成的 C/C++ 源文件中包含头文件，但一些实用工具文件除外。如果您的代码依赖于此旧有行为，可以使用以下语法保留旧行为：

```
coder.cinclude(headerfile,'InAllSourceFiles',true)
```

## **版本历史记录**

在 R2013a 中推出

## **扩展功能**

### **C/C++ 代码生成**

使用 MATLAB® Coder™ 生成 C 代码和 C++ 代码。

**GPU 代码生成**

使用 GPU Coder™ 为 NVIDIA® GPU 生成 CUDA® 代码。

**另请参阅**

`codegen` | `coder.ceval` | `coder.config` | `coder.target` | `coder.reservedName`

**主题**

“Configure Build for External C/C++ Code”

## coder.config

包: coder

创建 MATLAB Coder 代码生成配置对象

### 语法

```
config_obj = coder.config
config_obj = coder.config(build_type)
config_obj = coder.config(build_type,'ecoder',ecoder_flag)
config_obj = coder.config(numeric_conversion_type)
```

### 说明

`config_obj = coder.config` 创建一个 `coder.MexCodeConfig` 代码生成配置对象，该对象与 `codegen` 一起使用来生成 MEX 函数。将 `coder.MexCodeConfig` 对象与 `codegen` 命令的 `-config` 选项结合使用。

`config_obj = coder.config(build_type)` 创建一个代码生成配置对象，该对象与 `codegen` 结合使用来生成 MEX 函数或独立代码（静态库、动态链接库或可执行程序）。将该代码生成配置对象与 `codegen` 命令的 `-config` 选项结合使用。

`config_obj = coder.config(build_type,'ecoder',ecoder_flag)` 根据 `ecoder_flag` 是 `true` 还是 `false` 来创建 `coder.EmbeddedCodeConfig` 对象或 `coder.CodeConfig` 对象。`build_type` 为 `'lib'`、`'dll'` 或 `'exe'`。将标志设置为 `true` 以使用以下功能：

- 通过执行软件在环 (SIL) 和处理器在环 (PIL) 来验证代码。
- 代码追溯或双向可追溯性。
- 硬件特定优化和自定义替换库。
- 自定义生成的代码的外观。

请参阅“从 MATLAB 代码中生成代码的 Embedded Coder 功能” (Embedded Coder)。

`config_obj = coder.config(numeric_conversion_type)` 创建以下配置对象来与 `codegen` 结合使用：

- `coder.FixptConfig`（当从浮点 MATLAB 代码生成定点 MATLAB 或 C/C++ 代码时）。与 `codegen` 命令的 `-float2fixed` 选项结合使用。
- `coder.SingleConfig`（从双精度 MATLAB 代码生成单精度 MATLAB 代码时）。与 `codegen` 命令的 `-double2single` 选项结合使用。

定点转换或单精度转换需要 Fixed-Point Designer。

**注意** 使用 `coder.config` 函数创建代码配置对象后，您可以在命令行以编程方式修改其属性，也可以使用配置参数对话框以交互方式修改其属性。请参阅“Specify Configuration Parameters in Command-Line Workflow Interactively”。

## 示例

### 从 MATLAB 函数生成 MEX 函数

从 MATLAB 函数生成适合代码生成并支持代码生成报告的 MEX 函数。

编写一个 MATLAB 函数 `coderand`，该函数在开区间 (0,1) 上基于标准均匀分布生成一个随机标量值。

```
function r = coderand() %#codegen
% The directive %#codegen declares that the function
% is intended for code generation
r = rand();
```

创建一个代码生成配置对象来生成 MEX 函数。

```
cfg = coder.config % or cfg = coder.config('mex')
```

打开代码生成报告。

```
cfg.GenerateReport = true;
```

通过使用 `-config` 选项，在当前文件夹中生成指定配置对象的 MEX 函数。

```
% Generate a MEX function and code generation report
codegen -config cfg coderand
```

### 生成独立 C 静态或动态库或生成独立 C 可执行文件

为独立 C 静态库创建一个代码生成配置对象。

```
cfg = coder.config('lib')
% Returns a coder.EmbeddedCodeConfig object if the Embedded
% Coder product is installed.
% Otherwise, returns a coder.CodeConfig object.
```

创建一个代码生成配置对象，以生成独立 C 动态库。

```
cfg = coder.config('dll')
% Returns a coder.EmbeddedCodeConfig object if the Embedded
% Coder product is installed.
% Otherwise, returns a coder.CodeConfig object.
```

创建一个代码生成配置对象，以生成独立 C 可执行文件。

```
cfg = coder.config('exe')
% Returns a coder.EmbeddedCodeConfig object if the Embedded
% Coder product is installed.
% Otherwise, returns a coder.CodeConfig object.
```

### 在安装了 Embedded Coder 的情况下创建 coder.CodeConfig 对象

即使您的系统上安装了 Embedded Coder 产品，也要创建一个 `coder.CodeConfig` 对象。

```
cfg = coder.config('lib','ecoder',false)
```

在没有 Embedded Coder 的情况下创建 `coder.EmbeddedCodeConfig` 对象。

```
cfg = coder.config('lib','ecoder',true)
```

创建数值转换配置对象

创建一个 `coder.FixptConfig` 对象。

```
fixptcfg = coder.config('fixpt');
```

创建一个 `coder.SingleConfig` 对象。

```
scfg = coder.config('single');
```

输入参数

**build\_type** — 要创建的代码生成对象的类型

'mex' | 'lib' | 'dll' | 'exe'

| 配置对象类型 | 生成的代码  | 代码生成配置对象（已安装 Embedded Coder）          | 代码生成配置对象（未安装 Embedded Coder）     |
|--------|--------|---------------------------------------|----------------------------------|
| 'mex'  | MEX 函数 | <code>coder.MexCodeConfig</code>      | <code>coder.MexCodeConfig</code> |
| 'lib'  | 静态库    | <code>coder.EmbeddedCodeConfig</code> | <code>coder.CodeConfig</code>    |
| 'dll'  | 动态库    | <code>coder.EmbeddedCodeConfig</code> | <code>coder.CodeConfig</code>    |
| 'exe'  | 可执行文件  | <code>coder.EmbeddedCodeConfig</code> | <code>coder.CodeConfig</code>    |

示例： `coder.config('mex');`

数据类型： `char` | `string`

**numeric\_conversion\_type** — 数值转换对象类型

'fixpt' | 'single'

|          |                                                                                                              |
|----------|--------------------------------------------------------------------------------------------------------------|
| 'fixpt'  | 创建一个 <code>coder.FixptConfig</code> 配置对象，以便在从浮点 MATLAB 代码生成定点 MATLAB 或 C/C++ 代码时与 <code>codegen</code> 结合使用。 |
| 'single' | 创建一个 <code>coder.SingleConfig</code> 配置对象，以便在从双精度 MATLAB 代码生成单精度 MATLAB 代码时与 <code>codegen</code> 结合使用。      |

示例： `coder.config('fixpt');`

数据类型： `char` | `string`

**ecoder\_flag** — Embedded Coder 代码配置对象标志

false | true



|       |                                                                                                                                                                                                                                                                                                                                                                                                                                           |
|-------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| true  | <p>不管 Embedded Coder 是否存在，都创建 <b>coder.EmbeddedCodeConfig</b> 配置对象以允许使用以下功能：</p> <ul style="list-style-type: none"><li>• 通过执行软件在环 (SIL) 和处理器在环 (PIL) 来验证代码。</li><li>• 代码追溯或双向可追溯性。</li><li>• 硬件特定优化和自定义替换库。</li><li>• 自定义生成的代码的外观</li></ul> <p>请参阅 “从 MATLAB 代码中生成代码的 Embedded Coder 功能” (Embedded Coder)。</p> <p><b>build_type</b> 必须是 'lib'、'dll' 或 'exe'。</p> <p>但是，使用 <b>coder.EmbeddedCodeConfig</b> 对象生成代码需要 Embedded Coder 产品。</p> |
| false | <p>创建一个 <b>coder.CodeConfig</b> 配置对象，即使安装了 Embedded Coder 产品也是如此。 <b>build_type</b> 必须为 'lib'、'dll' 或 'exe'。</p>                                                                                                                                                                                                                                                                                                                          |

示例： `coder.config('lib','ecoder',false);`

数据类型： `logical`

输出参数

**config\_obj** — 代码生成配置句柄

`coder.CodeConfig` | `coder.MexCodeConfig` | `coder.EmbeddedCodeConfig` | `coder.FixptConfig` | `coder.SingleConfig`

MATLAB Coder 代码生成配置对象的句柄。

替代方法

使用 `coder` 函数打开 MATLAB Coder，并创建一个 MATLAB Coder 工程。该 App 提供了用户界面，便于您添加 MATLAB 文件、定义输入参数和指定编译参数。

版本历史记录

在 R2011a 中推出

另请参阅

`coder.CodeConfig` | `coder.EmbeddedCodeConfig` | `coder.MexCodeConfig` | `coder.FixptConfig` | `codegen` | `coder.SingleConfig`

主题

- “通过生成 MEX 函数加快 MATLAB 算法的执行速度”
- “通过命令行生成 C 代码”
- “Convert MATLAB Code to Fixed-Point C Code”
- “Generate Single-Precision C Code at the Command Line”
- “Specify Configuration Parameters in Command-Line Workflow Interactively”

## coder.const

在生成的代码中将表达式折叠为常量

### 语法

```
out = coder.const(expression)
[out1,...,outN] = coder.const(handle,arg1,...,argN)
```

### 说明

`out = coder.const(expression)` 会计算 `expression` 并在生成的代码中将 `out` 替换为计算结果。

`[out1,...,outN] = coder.const(handle,arg1,...,argN)` 计算具有句柄 `handle` 的多输出函数。然后，它会在生成的代码中将 `out1,...,outN` 替换为计算结果。

### 示例

#### 在生成的代码中指定常量

此示例说明如何使用 `coder.const` 在生成的代码中指定常量。

编写函数 `AddShift`，它接受 `Shift` 输入并将其添加到一个向量的元素。该向量由前 10 个自然数的平方组成。`AddShift` 可生成此向量。

```
function y = AddShift(Shift) %#codegen
y = (1:10).^2+Shift;
```

使用 `codegen` 命令为 `AddShift` 生成代码。打开代码生成报告。

```
codegen -config:lib -launchreport AddShift -args 0
```

代码生成器生成用于创建该向量的代码。它在向量创建过程中向向量的每个元素添加 `Shift`。在生成的代码中，`AddShift` 的定义如下：

```
void AddShift(double Shift, double y[10])
{
 int k;
 for (k = 0; k < 10; k++) {
 y[k] = (double)((1 + k) * (1 + k)) + Shift;
 }
}
```

将表达式 `(1:10).^2` 替换为 `coder.const((1:10).^2)`，然后再次使用 `codegen` 命令为 `AddShift` 生成代码。打开代码生成报告。

```
codegen -config:lib -launchreport AddShift -args 0
```

代码生成器创建包含前 10 个自然数的平方的向量。在生成的代码中，它将 `Shift` 添加到此向量的每个元素。在生成的代码中，`AddShift` 的定义如下：

```
void AddShift(double Shift, double y[10])
{
```

```

int i;
static const signed char iv[10] = { 1, 4, 9, 16, 25, 36,
 49, 64, 81, 100 };

for (i = 0; i < 10; i++) {
 y[i] = (double)iv[i] + Shift;
}
}

```

## 在生成的代码中创建查找表

此示例说明如何在生成的代码中将用户编写的函数折叠为常量。

编写函数 `getsine`，该函数接受 `index` 作为输入，并从正弦查找表中返回 `index` 引用的元素。函数 `getsine` 使用另一个函数 `gettable` 创建查找表。

```

function y = getsine(index) %#codegen
 assert(isa(index, 'int32'));
 persistent tbl;
 if isempty(tbl)
 tbl = gettable(1024);
 end
 y = tbl(index);

function y = gettable(n)
 y = zeros(1,n);
 for i = 1:n
 y(i) = sin((i-1)/(2*pi*n));
 end

```

使用 `int32` 类型的参数为 `getsine` 生成代码。打开代码生成报告。

```
codegen -config:lib -launchreport getsine -args int32(0)
```

生成的代码包含创建该查找表的说明。

将以下语句：

```
tbl = gettable(1024);
```

替换为：

```
tbl = coder.const(gettable(1024));
```

使用 `int32` 类型的参数为 `getsine` 生成代码。打开代码生成报告。

生成的代码包含查找表本身。`coder.const` 强制在代码生成期间计算表达式 `gettable(1024)`。生成的代码不包含计算的说明。生成的代码包含计算本身的结果。

## 使用多输出函数在生成的代码中指定常量

此示例说明如何在 `coder.const` 语句中使用多输出函数在生成的代码中指定常量。

编写函数 `MultiplyConst`，该函数接受 `factor` 作为输入，并将两个向量 `vec1` 和 `vec2` 的每个元素乘以 `factor`。该函数使用另一个函数 `EvalConsts` 生成 `vec1` 和 `vec2`。

```
function [y1,y2] = MultiplyConst(factor) %#codegen
 [vec1,vec2]=EvalConsts(pi.*(1./2.^(1:10)),2);
 y1=vec1.*factor;
 y2=vec2.*factor;

function [f1,f2]=EvalConsts(z,n)
 f1=z.^(2*n)/factorial(2*n);
 f2=z.^(2*n+1)/factorial(2*n+1);
```

使用 `codegen` 命令为 `MultiplyConst` 生成代码。打开代码生成报告。

```
codegen -config:lib -launchreport MultiplyConst -args 0
```

代码生成器生成用于创建向量的代码。

将以下语句：

```
[vec1,vec2]=EvalConsts(pi.*(1./2.^(1:10)),2);
```

替换为：

```
[vec1,vec2]=coder.const(@EvalConsts,pi.*(1./2.^(1:10)),2);
```

使用 `codegen` 命令为 `MultiplyConst` 生成代码。打开代码生成报告。

```
codegen -config:lib -launchreport MultiplyConst -args 0
```

代码生成器不生成用于创建向量的代码。在这种情况下，它会计算向量并在生成的代码中指定计算的向量。

### 通过处理 XML 文件读取常量

此示例说明如何使用 `coder.const` 调用外部函数。

编写包含以下语句的 XML 文件 `MyParams.xml`：

```
<params>
 <param name="hello" value="17"/>
 <param name="world" value="42"/>
</params>
```

将 `MyParams.xml` 保存在当前文件夹中。

编写读取一个 XML 文件的 MATLAB 函数 `xml2struct`。该函数会识别在另一个标记 `params` 内的 XML 标记 `param`。

识别 `param` 后，该函数将其属性 `name` 的值赋给结构体 `s` 的字段名称。该函数还将属性 `value` 的值赋给该字段的值。

```
function s = xml2struct(file)

s = struct();
doc = xmlread(file);
```

```

els = doc.getElementsByTagName('params');
for i = 0:els.getLength-1
 it = els.item(i);
 ps = it.getElementsByTagName('param');
 for j = 0:ps.getLength-1
 param = ps.item(j);
 paramName = char(param.getAttribute('name'));
 paramValue = char(param.getAttribute('value'));
 paramValue = evalin('base', paramValue);
 s.(paramName) = paramValue;
 end
end
end

```

将 `xml2struct` 保存在当前文件夹中。

编写 MATLAB 函数 `MyFunc`，该函数使用函数 `xml2struct` 将 XML 文件 `MyParams.xml` 读入结构体 `s`。使用 `coder.extrinsic` 将 `xml2struct` 声明为外部函数，并在 `coder.const` 语句中调用它。

```

function y = MyFunc(u) %#codegen
 assert(isa(u, 'double'));
 coder.extrinsic('xml2struct');
 s = coder.const(xml2struct('MyParams.xml'));
 y = s.hello + s.world + u;

```

使用 `codegen` 命令为 `MyFunc` 生成代码。打开代码生成报告。

```
codegen -config:dll -launchreport MyFunc -args 0
```

代码生成器在代码生成期间执行对 `xml2struct` 的调用。它在生成的代码中用值 17 和 42 替换结构体字段 `s.hello` 和 `s.world`。

## 输入参数

**expression** — MATLAB 表达式或用户编写的函数

具有常量的表达式 | 具有常量参数的单输出函数

MATLAB 表达式或用户定义的单输出函数。

该表达式只能包含编译时常量。该函数只能接受常量参数。例如，以下代码会导致代码生成错误，因为 `x` 不是编译时常量。

```

function y=func(x)
 y=coder.const(log10(x));

```

要修复该错误，请将 `x` 赋给 MATLAB 代码中的常量。或者，在代码生成过程中，您可以使用 `coder.Constant` 定义输入类型，如下所示：

```
codegen -config:lib func -args coder.Constant(10)
```

示例： `2*pi`, `factorial(10)`

**handle** — 函数句柄

函数句柄

内置函数或用户编写函数的句柄。

示例： `@log`, `@sin`

数据类型: `function_handle`

**arg1,...,argN** — 具有句柄 `handle` 的函数的参数

作为常量的函数参数

具有句柄 `handle` 的函数的参数。

参数必须为编译时常量。例如，以下代码会导致代码生成错误，因为 `x` 和 `y` 不是编译时常量。

```
function y=func(x,y)
 y=coder.const(@nchoosek,x,y);
```

要修复该错误，请将 `x` 和 `y` 赋给 MATLAB 代码中的常量。或者，在代码生成过程中，您可以使用 `coder.Constant` 定义输入类型，如下所示：

```
codegen -config:lib func -args {coder.Constant(10),coder.Constant(2)}
```

## 输出参数

**out** — `expression` 的值

计算的表达式的值

`expression` 的值。在生成的代码中，MATLAB Coder 将出现的 `out` 替换为 `expression` 的值。

**out1,...,outN** — 具有句柄 `handle` 的函数的输出

具有句柄 `handle` 的函数的输出的值

具有句柄 `handle` 的函数的输出。MATLAB Coder 会计算该函数，并在生成的代码中将出现的 `out1,...,outN` 替换为常量。

## 提示

- 如果可能，代码生成器会自动对表达式进行常量折叠。通常，自动常量折叠发生在只具有标量的表达式中。当代码生成器不自行对表达式进行常量折叠时，请使用 `coder.const`。
- 当对计算密集型函数调用进行常量折叠时，为了减少代码生成时间，请执行外部函数调用。外部函数调用会导致由 MATLAB（而不是由代码生成器）来计算函数调用。例如：

```
function j = fcn(z)
 zTable = coder.const(0:0.01:100);
 jTable = coder.const(feval('besselj',3,zTable));
 j = interp1(zTable,jTable,z);
end
```

请参阅“Use `coder.const` with Extrinsic Function Calls”。

- 如果 `coder.const` 无法对函数调用进行常量折叠，请尝试执行外部函数调用来强制进行常量折叠。外部函数调用会导致由 MATLAB（而不是由代码生成器）来计算函数调用。例如：

```
function yi = fcn(xi)
 y = coder.const(feval('rand',1,100));
 yi = interp1(y,xi);
end
```

请参阅“Use `coder.const` with Extrinsic Function Calls”。

## 版本历史记录

在 R2013b 中推出

### 扩展功能

#### C/C++ 代码生成

使用 MATLAB® Coder™ 生成 C 代码和 C++ 代码。

#### GPU 代码生成

使用 GPU Coder™ 为 NVIDIA® GPU 生成 CUDA® 代码。

### 另请参阅

#### 主题

“常量折叠”

“Fold Function Calls into Constants”

“Use coder.const with Extrinsic Function Calls”

## **`coder.cstructname`**

**包：** `coder`

在生成代码中命名 C 结构体类型

### **语法**

```
coder.cstructname(var,structName)
coder.cstructname(var,structName,'extern','HeaderFile',headerfile)
coder.cstructname(var,structName,'extern','HeaderFile',headerfile,'Alignment',alignment)

outtype = coder.cstructname(intype,structName)
outtype = coder.cstructname(intype,structName,'extern','HeaderFile',headerfile)
outtype = coder.cstructname(intype,structName,'extern','HeaderFile',
headerfile,'Alignment',alignment)
```

### **说明**

**coder.cstructname** 为生成的或外部定义的、用于 MATLAB 变量（在生成的代码中表示为结构体）的 C 语言结构体类型命名。

**coder.cstructname**(var,structName) 为针对 MATLAB 变量 **var** 生成的 C 语言结构体类型命名。输入 **var** 可以是结构体或元胞数组。在为函数生成代码时，可在该函数中使用此语法。将 **coder.cstructname** 放置在定义 **var** 的语句之后，第一次使用 **var** 的语句之前。如果 **var** 是入口（顶层）函数输入参数，请将 **coder.cstructname** 放在函数的开头且在任何控制流语句之前。

**coder.cstructname**(var,structName,'extern','HeaderFile',headerfile) 指定用作 **var** 的 C 结构体类型的名称为 **structName**，且是在外部文件 **headerfileName** 中定义的。

可以在不指定头文件的情况下使用 'extern' 选项。但是，最好指定头文件，以便代码生成器在正确的位置生成 **#include** 语句。

**coder.cstructname**(var,structName,'extern','HeaderFile',headerfile,'Alignment',alignment) 还指定外部定义的结构体类型 **structName** 的运行时内存对齐方式。如果您有 Embedded Coder 并使用自定义代码替换库 (CRL)，请指定对齐方式，以便代码生成器可以匹配要求结构体对齐的 CRL 函数。请参阅“Data Alignment for Code Replacement” (Embedded Coder)。

**outtype** = **coder.cstructname**(intype,structName) 返回一个结构体或一个元胞数组类型对象 **outtype**，它指定要生成的 C 结构体类型的名称。**coder.cstructname** 使用输入类型 **intype** 的属性创建 **outtype**。然后，它将 **Type** 属性设置为 **structName**。使用此语法创建可用于 **codegen -args** 选项的类型对象。在为函数生成代码时，不能在该函数中使用此语法。

您不能在 MATLAB Function 模块中使用此语法。

**outtype** = **coder.cstructname**(intype,structName,'extern','HeaderFile',headerfile) 返回指定外部定义的 C 结构体类型的名称和位置的类型对象 **outtype**。代码生成器对类型为 **outtype** 的变量使用外部定义的结构体类型。

您不能在 MATLAB Function 模块中使用此语法。



`outtype = coder.cstructname(intype,structName,'extern','HeaderFile',headerfile,'Alignment',alignment)` 创建一个还指定 C 结构体类型对齐方式的类型对象 `outtype`。

您不能在 MATLAB Function 模块中使用此语法。

## 示例

### 为函数中的变量指定 C 结构体类型名称

在 MATLAB 函数 `myfun` 中，为变量 `v` 的所生成的 C 结构体类型指定名称 `MyStruct`。

```
function y = myfun()
 %#codegen
 v = struct('a',1,'b',2);
 coder.cstructname(v, 'myStruct');
 y = v;
end
```

生成独立的 C 代码。例如，生成静态库。

```
codegen -config:lib myfun -report
```

要查看生成的结构体类型，请打开 `codegen/lib/myfun/myfun_types.h` 或在代码生成报告中查看 `myfun_types.h`。生成的 C 结构体类型为：

```
typedef struct {
 double a;
 double b;
} myStruct;
```

### 为子结构体所生成的 C 结构体类型指定名称

在 MATLAB 函数 `myfun1` 中，为结构体 `v` 所生成的 C 结构体类型指定名称 `MyStruct`。将名称 `mysubStruct` 指定给为子结构体 `v.b` 生成的结构体类型。

```
function y = myfun()
 %#codegen
 v = struct('a',1,'b',struct('f',3));
 coder.cstructname(v, 'myStruct');
 coder.cstructname(v.b, 'mysubStruct');
 y = v;
end
```

生成的 C 结构体类型 `mysubStruct` 为：

```
typedef struct {
 double f;
} mysubStruct;
```

生成的 C 结构体类型 `myStruct` 为：

```
typedef struct {
 double a;
```

```
mysubStruct b;
} myStruct;
```

### 为元胞数组所生成的结构体类型指定名称

在 MATLAB 函数 myfun2 中，为元胞数组 c 所生成的 C 结构体类型指定名称 myStruct。

```
function z = myfun2()
c = {1 2 3};
coder.cstructname(c,'myStruct')
z = c;
```

为 c 生成的 C 结构体类型为：

```
typedef struct {
 double f1;
 double f2;
 double f3;
} myStruct;
```

### 为外部定义的 C 结构体类型指定名称

指定传递给 C 函数的结构体具有在 C 头文件中定义的结构体类型。

为接受 mycstruct 类型的参数的 mycadd 函数创建 C 头文件 mycadd.h。在头文件中定义类型 mycstruct。

```
#ifndef MYCADD_H
#define MYCADD_H

typedef struct {
 double f1;
 double f2;
} mycstruct;

double mycadd(mycstruct *s);
#endif
```

编写 C 函数 mycadd.c。

```
#include <stdio.h>
#include <stdlib.h>

#include "mycadd.h"

double mycadd(mycstruct *s)
{
 return s->f1 + s->f2;
}
```

编写按引用 mycadd 传递结构体的 MATLAB 函数 mymAdd。使用 coder.cstructname 指定在生成的代码中，结构体具有在 mycadd.h 中定义的 C 类型 mycstruct。

```
function y = mymAdd
%#codegen
```

```
s = struct('f1', 1, 'f2', 2);
coder.cstructname(s, 'mycstruct', 'extern', 'HeaderFile', 'mycadd.h');
y = 0;
y = coder.ceval('mycadd', coder.ref(s));
```

为函数 `mymAdd` 生成 C 静态库。

```
codegen -config:lib mymAdd mycadd.c
```

生成的头文件 `mymadd_types.h` 不包含结构体 `mycstruct` 的定义，因为 `mycstruct` 是外部类型。

### 创建一个为生成的 C 结构体类型指定名称的结构体类型对象

假设入口函数 `myFunction` 接受结构体参数。要在命令行中指定输入参数的类型，请执行以下操作：

- 1 定义示例结构体 `S`。
- 2 通过使用 `coder.typeof` 从 `S` 创建类型 `T`。
- 3 使用 `coder.cstructname` 创建类型 `T1`，该类型：
  - 具有 `T` 的属性。
  - 为生成的 C 结构体类型指定名称 `myStruct`。
- 4 通过使用 `-args` 选项将类型传递给 `codegen`。

例如：

```
S = struct('a',double(0),'b',single(0));
T = coder.typeof(S);
T1 = coder.cstructname(T,'myStruct');
codegen -config:lib myFunction -args T1
```

您也可以直接从示例结构体创建结构体类型。

```
S = struct('a',double(0),'b',single(0));
T1 = coder.cstructname(S,'myStruct');
codegen -config:lib myFunction -args T1
```

## 输入参数

**var** — MATLAB 结构体或元胞数组变量

结构体 | 元胞数组

在生成的代码中表示为结构体的 MATLAB 结构体或元胞数组变量。

**structName** — C 结构体类型的名称

字符向量 | 字符串标量

生成的或外部定义的 C 结构体类型的名称，指定为字符向量或字符串标量。

**headerfile** — 包含 C 结构体类型定义的头文件

字符向量 | 字符串标量

包含 C 结构体类型定义的头文件，指定为字符向量或字符串标量。

要指定文件的路径，请执行下列操作：

- 使用 `codegen -I` 选项或 MATLAB Coder App 设置的自定义代码选项卡上的其他包括目录参数。
- 对于 MATLAB Function 模块，在仿真目标和代码生成 > 自定义代码窗格中的附加编译信息下，设置包含目录参数。

或者，结合使用 `coder.updateBuildInfo` 和 `'addIncludePaths'` 选项。

示例：'mystruct.h'

### alignment — 结构体的运行时内存对齐

-1（默认） | 2 的幂

生成的或外部定义的结构体的运行时内存对齐。

### intype — 用于创建新类型对象的类型对象或变量

`coder.StructType` | `coder.CellType` | 结构体 | 元胞数组

用于创建类型对象的结构体类型对象、元胞数组类型对象、结构体变量或元胞数组变量。

## 限制

- 您不能将 `coder.cstructname` 直接应用于全局变量。要命名用于全局变量的结构体类型，请使用 `coder.cstructname` 创建命名结构体类型的类型对象。然后，当您运行 `codegen` 时，指定全局变量具有该类型。请参阅“Name the C Structure Type to Use With a Global Structure Variable”。
- 对于元胞数组输入，外部定义的结构体的字段名称必须为 `f1`、`f2` 等，以此类推。
- 您无法将 `coder.cstructname` 直接应用于类属性。

## 提示

- 有关代码生成器如何确定结构体字段的 C/C++ 类型的信息，请参阅“Mapping MATLAB Types to Types in Generated Code”。
- 对结构体数组使用 `coder.cstructname` 将设置基元素的结构体类型的名称，而不是数组的名称。因此，您不能将 `coder.cstructname` 应用于结构体数组元素，然后将其应用于具有不同 C 结构体类型名称的数组。例如，不允许使用以下代码。第二个 `coder.cstructname` 尝试将基类型的名称设置为 `myStructArrayName`，这与以前指定的名称 `myStructName` 发生冲突。

```
% Define scalar structure with field a
myStruct = struct('a', 0);
coder.cstructname(myStruct,'myStructName');
% Define array of structure with field a
myStructArray = repmat(myStruct,4,6);
coder.cstructname(myStructArray,'myStructArrayName');
```

- 将 `coder.cstructname` 应用于结构体数组的元素会产生与将 `coder.cstructname` 应用于整个结构体数组相同的结果。如果将 `coder.cstructname` 应用于结构体数组的元素，则必须使用单一下标引用该元素。例如，您可以使用 `var(1)`，但不能使用 `var(1,1)`。将 `coder.cstructname` 应用于 `var(:)` 会产生与将 `coder.cstructname` 应用于 `var` 或 `var(n)` 相同的结果。
- 异构元胞数组在生成的代码中表示为结构体。以下是对元胞数组使用 `coder.cstructname` 的注意事项：
  - 在要生成代码的函数中，将 `coder.cstructname` 与元胞数组变量结合使用会生成异构元胞数组。因此，如果元胞数组是入口函数输入，并且其类型为永久性同构，则不能将 `coder.cstructname` 与元胞数组结合使用。

- 将 `coder.cstructname` 与同构 `coder.CellType` 对象 `intype` 结合使用会使返回的对象为异构类型。因此，不能将 `coder.cstructname` 与永久性同构 `coder.CellType` 对象结合使用。有关元胞数组何时为永久性同构的信息，请参阅“Specify Cell Array Inputs at the Command Line”。
- 当与 `coder.CellType` 对象结合使用时，`coder.cstructname` 会创建永久性异构的 `coder.CellType` 对象。
- 在具有行优先和列优先数组布局的工程中使用由 `coder.cstructname` 命名的结构体时，代码生成器会在某些情况下重命名结构体，将 `row_` 或 `col_` 附加到结构体名称的开头。此重命名为这两个数组布局中使用的类型提供唯一的类型定义。
- 下列提示仅适用于 MATLAB Function 模块：
  - MATLAB Function 模块输入和输出结构体与总线信号相关联。生成的结构体类型名称来自总线信号名称。请勿使用 `coder.cstructname` 为输入或输出信号的结构体类型命名。请参阅“在 MATLAB Function 模块中创建结构体” (Simulink)。
  - 代码生成器会根据标识符命名规则生成结构体类型名称，即使您使用 `coder.cstructname` 命名结构体类型也是如此。如果您有 Embedded Coder，则可以自定义命名规则。请参阅“Construction of Generated Identifiers” (Embedded Coder)。

## 版本历史记录

在 R2011a 中推出

## 扩展功能

### C/C++ 代码生成

使用 MATLAB® Coder™ 生成 C 代码和 C++ 代码。

### GPU 代码生成

使用 GPU Coder™ 为 NVIDIA® GPU 生成 CUDA® 代码。

## 另请参阅

`coder.ceval` | `codegen` | `coder.StructType` | `coder.CellType`

### 主题

“用于代码生成的结构体定义”

“Code Generation for Cell Arrays”

“Specify Cell Array Inputs at the Command Line”

“从生成的代码中调用自定义 C/C++ 代码”

## `coder.inline`

控制生成的代码中特定函数的内联

### 语法

```
coder.inline('always')
coder.inline('never')
coder.inline('default')
```

### 说明

`coder.inline('always')` 在生成的代码中强制对当前函数进行内联（第 2-57 页）。将 `coder.inline` 指令放在要内联的函数中。代码生成器不对入口函数和递归函数进行内联。此外，代码生成器不会将函数内联到 `parfor` 循环中，也不会内联从 `parfor` 循环调用的函数。

`coder.inline('never')` 可阻止在生成的代码中内联当前函数。当您简化 MATLAB 源代码和生成代码之间的映射时，请阻止内联。

---

**注意** 如果使用 `codegen` 或 `fiaccel` 命令，则可以使用 `-O disable:inline` 选项对所有函数禁用内联。

如果您使用 `codegen` 命令或 MATLAB Coder 生成 C/C++ 代码，则对于为您编写的函数生成的代码和为 MathWorks® 函数生成的代码，您可能有不同的速度和可读性要求。某些其他全局设置使您能够分别控制生成的代码库的这两个部分以及它们之间边界处的内联行为。请参阅“Control Inlining to Fine-Tune Performance and Readability of Generated Code”。

---

`coder.inline('default')` 指示代码生成器使用内部启发式方法来确定是否内联当前函数。通常，启发式方法会生成高度优化的代码。仅当需要微调这些优化时，才在 MATLAB 函数中显式使用 `coder.inline`。

### 示例

#### 防止函数内联

在此示例中，函数 `foo` 在生成的代码中没有内联：

```
function y = foo(x)
 coder.inline('never');
 y = x;
end
```

#### 在控制流语句中使用 `coder.inline`

您可以在控制流代码中使用 `coder.inline`。如果软件检测到矛盾的 `coder.inline` 指令，生成的代码将使用默认内联启发式方法并发出警告。

假设您要为在内存有限的系统上运行的除法函数生成代码。为了优化生成的代码中的内存使用，函数 `inline_division` 根据是执行标量除法还是向量除法来手动控制内联：

```
function y = inline_division(dividend, divisor)

% For scalar division, inlining produces smaller code
% than the function call itself.
if isscalar(dividend) && isscalar(divisor)
 coder.inline('always');
else
 % Vector division produces a for-loop.
 % Prohibit inlining to reduce code size.
 coder.inline('never');
end

if any(divisor == 0)
 error('Cannot divide by 0');
end

y = dividend / divisor;
```

## 详细信息

### 内联

内联是一种将函数调用替换为该函数的内容（主体）的方法。内联消除了函数调用的开销，但会生成更多的 C/C++ 代码。如果生成的 C/C++ 代码使用了内联，则可能有进一步优化的空间。

## 版本历史记录

在 R2011a 中推出

## 扩展功能

### C/C++ 代码生成

使用 MATLAB® Coder™ 生成 C 代码和 C++ 代码。

### GPU 代码生成

使用 GPU Coder™ 为 NVIDIA® GPU 生成 CUDA® 代码。

## 另请参阅

codegen

### 主题

“Control Inlining to Fine-Tune Performance and Readability of Generated Code”  
“优化策略”

## coder.load

从 MAT 文件或 ASCII 文件加载编译时常量

### 语法

```
S = coder.load(filename)
S = coder.load(filename,var1,...,varN)
S = coder.load(filename,'-regexp',expr1,...,exprN)
S = coder.load(filename,'-ascii')
S = coder.load(filename,'-mat')
S = coder.load(filename,'-mat',var1,...,varN)
S = coder.load(filename,'-mat','-regexp', expr1,...,exprN)
```

### 说明

`S = coder.load(filename)` 从 `filename` 加载编译时常量。

- 如果 `filename` 是 MAT 文件，则 `coder.load` 将变量从 MAT 文件加载到结构体数组中。
- 如果 `filename` 是 ASCII 文件，则 `coder.load` 将数据加载到一个双精度数组中。

`coder.load` 在代码生成时（也称为编译时）加载数据。如果您在生成代码后更改 `filename` 的内容，则更改不会反映在已生成的代码的行为中。

`S = coder.load(filename,var1,...,varN)` 仅从 MAT 文件 `filename` 中加载指定的变量。

`S = coder.load(filename,'-regexp',expr1,...,exprN)` 只加载与指定的正则表达式匹配的变量。

`S = coder.load(filename,'-ascii')` 将 `filename` 视为 ASCII 文件，而不管文件扩展名如何。

`S = coder.load(filename,'-mat')` 将 `filename` 视为 MAT 文件，而不管文件扩展名如何。

`S = coder.load(filename,'-mat',var1,...,varN)` 将 `filename` 视为 MAT 文件，并仅从该文件加载指定的变量。

`S = coder.load(filename,'-mat','-regexp', expr1,...,exprN)` 将 `filename` 视为 MAT 文件，并仅加载与指定的正则表达式匹配的变量。

### 示例

#### 从 MAT 文件加载编译时常量

为函数 `edgeDetect1` 生成代码，该函数根据给定的归一化图像返回基于阈值检测出边缘的图像。`edgeDetect1` 使用 `coder.load` 在编译时从一个 MAT 文件加载边缘检测核。

将 Sobel 边缘检测核保存在一个 MAT 文件中。

```
k = [1 2 1; 0 0 0; -1 -2 -1];
```

```
save sobel.mat k
```



编写函数 `edgeDetect1`。

```
function edgeImage = edgeDetect1(originalImage, threshold) %#codegen
assert(all(size(originalImage) <= [1024 1024]));
assert(isa(originalImage, 'double'));
assert(isa(threshold, 'double'));

S = coder.load('sobel.mat','k');
H = conv2(double(originalImage),S.k, 'same');
V = conv2(double(originalImage),S.k,'same');
E = sqrt(H.*H + V.*V);
edgeImage = uint8((E > threshold) * 255);
```

为静态库创建代码生成配置对象。

```
cfg = coder.config('lib');
```

为 `edgeDetect1` 生成一个静态库。

```
codegen -report -config cfg edgeDetect1
```

`codegen` 在 `codegen\lib\edgeDetect1` 文件夹中生成 C 代码。

## 从 ASCII 文件加载编译时常量

为函数 `edgeDetect2` 生成代码，该函数根据给定的归一化图像返回基于阈值检测出边缘的图像。`edgeDetect2` 使用 `coder.load` 在编译时从一个 ASCII 文件加载边缘检测核。

将 Sobel 边缘检测核保存在一个 ASCII 文件中。

```
k = [1 2 1; 0 0 0; -1 -2 -1];
save sobel.dat k -ascii
```

编写函数 `edgeDetect2`。

```
function edgeImage = edgeDetect2(originalImage, threshold) %#codegen
assert(all(size(originalImage) <= [1024 1024]));
assert(isa(originalImage, 'double'));
assert(isa(threshold, 'double'));

k = coder.load('sobel.dat');
H = conv2(double(originalImage),k, 'same');
V = conv2(double(originalImage),k,'same');
E = sqrt(H.*H + V.*V);
edgeImage = uint8((E > threshold) * 255);
```

为静态库创建代码生成配置对象。

```
cfg = coder.config('lib');
```

为 `edgeDetect2` 生成一个静态库。

```
codegen -report -config cfg edgeDetect2
```

`codegen` 在 `codegen\lib\edgeDetect2` 文件夹中生成 C 代码。

## 输入参数

### **filename** — 文件名

字符向量 | 字符串标量

文件名。 **filename** 必须为编译时常量。

**filename** 可包含文件扩展名以及完整或部分路径。如果 **filename** 没有扩展名，`load` 会查找名为 **filename.mat** 的文件。如果 **filename** 的扩展名不是 `.mat`，`load` 会将该文件视为 ASCII 数据。

ASCII 文件必须包含由数字组成的矩形表格，并且每行中的元素数目相等。文件分隔符（每行中的元素之间的字符）可以为空格、逗号、分号或制表符。文件可包含 MATLAB 注释（以百分比符号 `%` 开头的行）。

示例： `'myFile.mat'`

### **var1,...,varN** — 要加载的变量的名称

字符向量 | 字符串标量

变量的名称，指定为一个或多个字符向量或字符串标量。每个变量名称必须为一个编译时常量。使用 `*` 通配符来匹配模式。

示例： `coder.load('myFile.mat','A*')` 会加载文件中所有名称以 `A` 开头的变量。

### **expr1,...,exprN** — 指示要加载哪些变量的正则表达式

字符向量 | 字符串标量

指示要加载哪些变量的正则表达式，指定为一个或多个字符向量或字符串标量。每个正则表达式必须为一个编译时常量。

示例： `coder.load('myFile.mat','-regexp','^A')` 只加载其名称以 `A` 开头的变量。

## 输出参数

### **S** — 加载的变量或数据

结构体数组 |  $m \times n$  数组

如果 **filename** 是 MAT 文件，则 **S** 是结构体数组。

如果 **filename** 是 ASCII 文件，则 **S** 是 `double` 类型的  $m \times n$  数组。 $m$  是文件中的行数， $n$  是一行中的值数。

## 限制

- `coder.load` 的参数必须为编译时常量。
- 输出 **S** 必须为不带下标的结构体或数组的名称。例如，不允许使用 `S(i) = coder.load('myFile.mat')`。
- 在用于代码生成的函数中，您不能使用 `save` 将工作区数据保存到文件中。代码生成器不支持 `save` 函数。此外，您不能将 `coder.extrinsic` 与 `save` 结合使用。在生成代码之前，您可以使用 `save` 将工作区数据保存到文件中。

## 提示

- **coder.load(filename)** 在编译时（而不是在运行时）加载数据。如果您在生成代码后更改 **filename** 的内容，则更改不会反映在已生成的代码的行为中。如果您正在生成代码或者正在为 Simulink 仿真代码，您可以使用 MATLAB 函数 **load** 加载运行时值。
- 如果 MAT 文件包含不支持的构造，请使用 **coder.load(filename,var1,...,varN)** 仅加载支持的构造。
- 如果您在 MATLAB Coder 工程中生成代码，代码生成器会为 **coder.load** 函数进行增量代码生成。当 **coder.load** 使用的 MAT 文件或 ASCII 文件发生变化时，软件会重新编译代码。

## 版本历史记录

在 R2013a 中推出

## 扩展功能

### C/C++ 代码生成

使用 MATLAB® Coder™ 生成 C 代码和 C++ 代码。

### GPU 代码生成

使用 GPU Coder™ 为 NVIDIA® GPU 生成 CUDA® 代码。

## 另请参阅

[matfile](#) | [regexp](#) | [save](#)

### 主题

“正则表达式”

## coder.nullcopy

包: coder

在代码生成中声明未初始化的变量

### 语法

**X = coder.nullcopy(A)**

### 说明

**X = coder.nullcopy(A)** 将 A 的类型、大小和复/实性复制到 X，但不复制元素值。该函数为 X 预分配内存，而不会导致初始化内存的开销。在代码生成中，**coder.nullcopy** 函数声明未初始化的变量。在 MATLAB 中，**coder.nullcopy** 返回输入，满足 X 等于 A。

如果 X 是包含可变大小数组的结构体或类，则您必须指定每个数组的大小。**coder.nullcopy** 不会将数组或嵌套数组的大小从其参数复制到结果中。

---

**注意** 在函数或程序中使用 X 之前，请确保 X 中的数据已完全初始化。通过 **coder.nullcopy** 声明变量而不指定变量的所有元素会导致不确定的程序行为。有关详细信息，请参阅 “How to Eliminate Redundant Copies by Defining Uninitialized Variables”。

---

### 示例

#### 声明不进行初始化的变量

此示例说明如何声明一个数组类型变量而不初始化该数组中的任何值。

要为以下函数生成代码，必须在通过下标对输出变量 **outp** 进行索引之前，将 **outp** 完全声明为  $n \times n$  双精度实数数组。要在不初始化数组中所有值的情况下执行此声明，请使用 **coder.nullcopy**。

```
function outp = foo(n) %#codegen
```

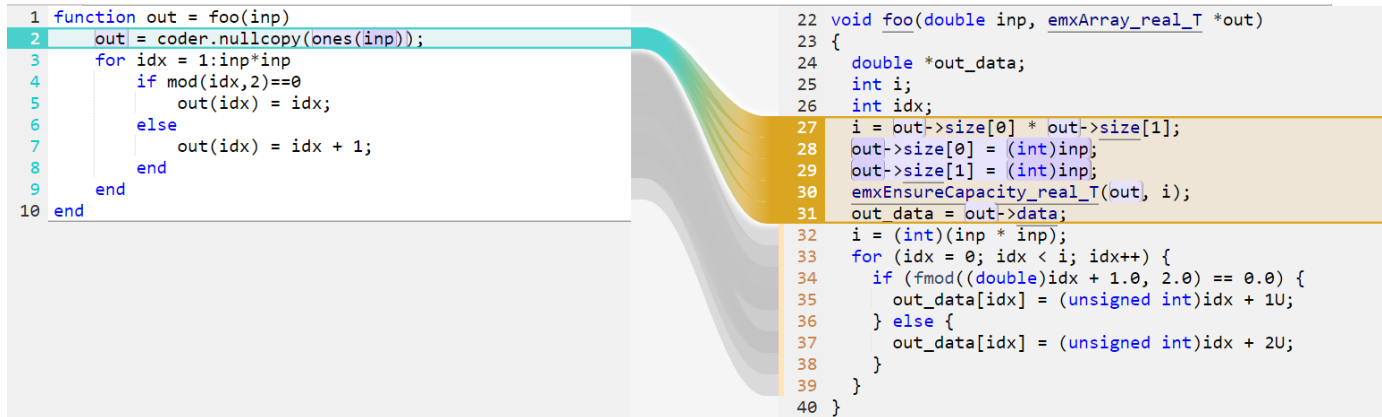
```
outp = coder.nullcopy(ones(n));
for idx = 1:n*n
 if mod(idx,2) == 0
 outp(idx) = idx;
 else
 outp(idx) = idx + 1;
 end
end
```

运行此 **codegen** 命令以生成代码并启动报告。

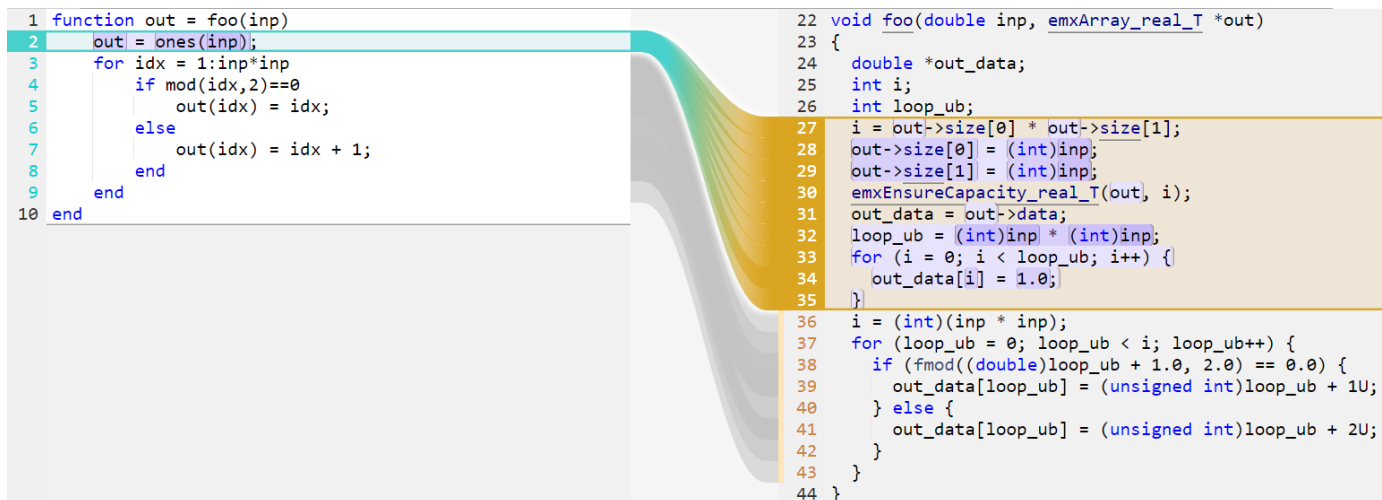
```
codegen -config:lib -c foo -args {0} -launchreport
```

在代码生成报告中，点击[追溯代码](#)以查看 MATLAB 代码和生成的代码之间的映射。要使用代码可追溯性功能，您必须有 Embedded Coder。

下图显示使用和不使用 **coder.nullcopy** 生成的代码之间的比较。使用 **coder.nullcopy** 和 **ones** 可以指定数组 **outp** 的大小，而无需将每个元素初始化为 1。



如果不使用 **coder.nullcopy**，生成的代码会显式将 **outp** 中的每个元素初始化为 1（参见第 32 行到第 35 行）。



**注意** 在某些情况下，即使您没有在 MATLAB 代码中显式包含 **coder.nullcopy** 指令，代码生成器也会自动执行与 **coder.nullcopy** 对应的优化。

## 输入参数

### A — 要复制的变量

标量 | 向量 | 矩阵 | 类 | 多维数组

要复制的变量，指定为标量、向量、矩阵或多维数组。

示例： **coder.nullcopy(A);**

数据类型： **single** | **double** | **int8** | **int16** | **int32** | **int64** | **uint8** | **uint16** | **uint32** | **uint64** | **logical** | **char** | **string** | **class**

复数支持： 是

### 限制

- 您无法对稀疏矩阵使用 `coder.nullcopy`。
- 您无法将 `coder.nullcopy` 用于支持重载括号或需要索引方法来访问其数据的类，如 `table`。

### 版本历史记录

在 R2011a 中推出

### 扩展功能

#### C/C++ 代码生成

使用 MATLAB® Coder™ 生成 C 代码和 C++ 代码。

#### GPU 代码生成

使用 GPU Coder™ 为 NVIDIA® GPU 生成 CUDA® 代码。

### 另请参阅

#### 主题

“Eliminate Redundant Copies of Variables in Generated Code”

# coder.opaque

在生成的代码中声明变量

## 语法

```
y = coder.opaque(type)
y = coder.opaque(type,value)
y = coder.opaque(__,'Size',Size)
y = coder.opaque(__,'HeaderFile',HeaderFile)
```

## 说明

**y = coder.opaque(type)** 在生成的代码中使用指定的类型声明变量 **y** 且不设置初始值。

- **y** 可以是变量或结构体字段。
- MATLAB 代码不能设置或访问 **y**，但外部 C 函数可以接受 **y** 作为参数。
- **y** 可以是：
  - **coder.rref**、**coder.wref** 或 **coder.ref** 的参数
  - **coder.ceval** 的输入或输出参数
  - 用户编写的 MATLAB 函数的输入或输出参数
  - 支持代码生成的 MATLAB 部分工具箱函数的输入
- 来自 **y** 的赋值在生成的代码中使用相同的类型声明另一个变量。例如：

```
y = coder.opaque('int');
z = y;
```

在生成的代码中声明类型为 **int** 的变量 **z**。

- 您可以基于使用 **coder.opaque** 声明的另一个变量对 **y** 赋值，或基于使用 **coder.opaque** 声明的变量进行赋值。变量必须具有相同的类型。
- 您可以将 **y** 与使用 **coder.opaque** 声明的另一个变量进行比较，或与基于使用 **coder.opaque** 声明的变量赋值进行比较。变量必须具有相同的类型。

**y = coder.opaque(type,value)** 指定 **y** 的类型和初始值。

**y = coder.opaque( \_\_,'Size',Size)** 指定 **y** 的大小（以字节为单位）。您可以使用上述任何语法指定大小。

**y = coder.opaque( \_\_,'HeaderFile',HeaderFile)** 指定包含类型定义的头文件。代码生成器为生成的代码中需要 **#include** 语句的头文件生成该语句。您可以使用上述任意语法指定头文件。

## 示例

### 声明指定初始值的变量

为函数 **valtest** 生成代码，如果调用 **myfun** 成功，该函数将返回 1。此函数使用 **coder.opaque** 声明变量 **x1**，该变量具有类型 **int** 和初始值 0。赋值 **x2 = x1** 将 **x2** 声明为具有 **x1** 的类型和初始值的变量。

编写函数 `valtest`。

```
function y = valtest
%codegen
%declare x1 to be an integer with initial value '0'
x1 = coder.opaque('int','0');
%Declare x2 to have same type and initial value as x1
x2 = x1;
x2 = coder.ceval('myfun');
%test the result of call to 'myfun' by comparing to value of x1
if x2 == x1
 y = 0;
else
 y = 1;
end
end
```

### 声明指定初始值和头文件的变量

为 MATLAB 函数 `filetest` 生成代码，该函数使用 `fopen/fread/fclose` 返回它自己的源代码。此函数使用 `coder.opaque` 声明存储 `fopen/fread/fclose` 使用的文件指针的变量。对 `coder.opaque` 的调用使用类型 `FILE *`、初始值 `NULL` 和头文件 `<stdio.h>` 声明变量 `f`。

编写 MATLAB 函数 `filetest`。

```
function buffer = filetest
%#codegen

% Declare 'f' as an opaque type 'FILE *' with initial value 'NULL'
%Specify the header file that contains the type definition of 'FILE *';

f = coder.opaque('FILE *', 'NULL', 'HeaderFile', '<stdio.h>');
% Open file in binary mode
f = coder.ceval('fopen', cstring('filetest.m'), cstring('rb'));

% Read from file until end of file is reached and put
% contents into buffer
n = int32(1);
i = int32(1);
buffer = char(zeros(1,8192));
while n > 0
 % By default, MATLAB converts constant values
 % to doubles in generated code
 % so explicit type conversion to int32 is inserted.
 n = coder.ceval('fread', coder.ref(buffer(i)), int32(1), ...
 int32(numel(buffer)), f);
 i = i + n;
end
coder.ceval('fclose',f);

buffer = strip_cr(buffer);

% Put a C termination character '\0' at the end of MATLAB character vector
function y = cstring(x)
 y = [x char(0)];

% Remove all character 13 (CR) but keep character 10 (LF)
function buffer = strip_cr(buffer)
j = 1;
for i = 1:numel(buffer)
 if buffer(i) ~= char(13)
 buffer(j) = buffer(i);
 j = j + 1;
 end
end
```



```
end
buffer(i) = 0;
```

### 比较使用 coder.opaque 声明的变量

比较使用 coder.opaque 声明的变量来测试是否成功打开文件。

使用 coder.opaque 声明变量 null，该变量具有类型 FILE \* 和初始值 NULL。

```
null = coder.opaque('FILE *', 'NULL', 'HeaderFile', '<stdio.h>');
```

使用赋值声明与 null 具有相同的类型和值的另一个变量 ftmp。

```
ftmp = null;
ftmp = coder.ceval('fopen', ['testfile.txt', char(0)], ['r', char(0)]);
```

对这两个变量进行比较。

```
if ftmp == null
 %error condition
end
```

### 对使用 coder.opaque 声明的变量进行类型转换

此示例说明如何对使用 coder.opaque 声明的变量进行类型转换。函数 castopaque 调用 C 运行时函数 strncmp 以对字符串 s1 和 s2 中的最多 n 个字符进行比较。n 是较短字符串中的字符数。为了对 strncmp 输入 nsizet 生成正确的 C 类型，函数将 n 转换为 C 类型 size\_t，并将结果赋值给 nsizet。该函数使用 coder.opaque 声明 nsizet。在使用 strncmp 的输出 retval 之前，函数将 retval 转换为 MATLAB 类型 int32，并将结果存储在 y 中。

编写此 MATLAB 函数：

```
function y = castopaque(s1,s2)

% <0 - the first character that does not match has a lower value in s1 than in s2
% 0 - the contents of both strings are equal
% >0 - the first character that does not match has a greater value in s1 than in s2
%
%#codegen

coder.cinclude('<string.h>');
n = min(numel(s1), numel(s2));

% Convert the number of characters to compare to a size_t

nsizet = cast(n,'like',coder.opaque('size_t','0'));

% The return value is an int
retval = coder.opaque('int');
retval = coder.ceval('strncmp', cstr(s1), cstr(s2), nsizet);

% Convert the opaque return value to a MATLAB value
y = cast(retval, 'int32');

%-----
function sc = cstr(s)
% NULL terminate a MATLAB character vector for C
sc = [s, char(0)];
```

生成 MEX 函数。

```
codegen castopaque -args {blanks(3), blanks(3)} -report
```

使用输入 'abc' 和 'abc' 调用 MEX 函数。

```
castopaque_mex('abc','abc')
```

```
ans =
```

```
0
```

由于两个字符串相等，因此输出为 0。

使用输入 'abc' 和 'abd' 调用 MEX 函数。

```
castopaque_mex('abc','abd')
```

```
ans =
```

```
-1
```

由于第二个字符串中的第三个字符 d 大于第一个字符串中的第三个字符 c，因此输出为 -1。

使用输入 'abd' 和 'abc' 调用 MEX 函数。

```
castopaque_mex('abd','abc')
```

```
ans =
```

```
1
```

由于第一个字符串中的第三个字符 d 大于第二个字符串中的第三个字符 c，因此输出为 1。

在 MATLAB 工作区中，您可以看到 y 的类型为 int32。

### 声明指定初始值和大小的变量

将 y 声明为初始值等于 0 的 4 个字节的整数。

```
y = coder.opaque('int','0', 'Size', 4);
```

## 输入参数

### type — 变量的类型

字符向量 | 字符串标量

生成的代码中变量的类型。type 必须为编译时常量。类型必须是：

- 内置的 C 数据类型或头文件中定义的类型
- 支持通过赋值进行复制的 C 类型
- C 声明中的合法前缀

示例：'FILE \*'

### value — 变量的初始值

字符向量 | 字符串标量

生成的代码中变量的初始值。**value** 必须为编译时常量。指定一个不依赖于 MATLAB 变量或函数的 C 表达式。

如果您在 **value** 中未提供初始值，请在使用之前先初始化变量的值。要初始化使用 **coder.opaque** 声明的变量，请执行以下操作：

- 基于具有使用 **coder.opaque** 声明的相同类型的另一个变量进行赋值，或基于使用 **coder.opaque** 声明的变量进行赋值。
- 基于外部 C 函数赋值。
- 使用 **coder.wref** 将变量的地址传递给外部函数。

指定具有 **type** 指定的类型的 **value**。否则，生成的代码可能产生意外的结果。

示例：'NULL'

### Size — 变量的大小

整数

生成的代码中的变量的字节数，指定为整数。如果不指定大小，则变量的大小为 8 个字节。

数据类型： `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### HeaderFile — 头文件的名称

字符向量 | 字符串标量

包含 **type** 的定义的头文件名称。**HeaderFile** 必须为编译时常量。

对于系统头文件，使用尖括号。

示例：'<stdio.h>' 生成 `#include <stdio.h>`

对于应用程序头文件，使用双引号。

示例：'"foo.h"' 生成 `#include "foo.h"`

如果您省略尖括号或双引号，代码生成器会生成双引号。

示例：'foo.h' 生成 `#include "foo.h"`

在编译配置参数中指定包含路径。

示例：`cfg.CustomInclude = 'c:\myincludes'`

## 提示

- 指定具有 **type** 指定的类型的 **value**。否则，生成的代码可能产生意外的结果。例如，以下 **coder.opaque** 声明可能产生意外的结果。

```
y = coder.opaque('int', '0.2')
```

- **coder.opaque** 声明变量的类型。它不实例化变量。稍后，您可以通过在 MATLAB 代码中使用变量来对其进行实例化。在下面的示例中，来自 **coder.ceval** 的 **fp1** 的赋值对 **fp1** 进行实例化。

```
% Declare fp1 of type FILE *
fp1 = coder.opaque('FILE *');
% Create the variable fp1
fp1 = coder.ceval('fopen', ['testfile.txt', char(0)], ['r', char(0)]);
```

- 在 MATLAB 环境中，**coder.opaque** 返回在 **value** 中指定的值。如果未提供 **value**，它将返回空字符向量。

- 您可以对使用 `coder.opaque` 声明的变量或基于使用 `coder.opaque` 声明的变量赋值进行比较。变量必须具有相同的类型。以下示例说明如何对这些变量进行比较。“比较使用 `coder.opaque` 声明的变量”（第 2-67 页）
- 要避免在生成的代码中多次包含相同的头文件，请将头文件括入条件预处理器语句 `#ifndef` 和 `#endif` 中。例如：

```
#ifndef MyHeader_h
#define MyHeader_h
<body of header file>
#endif
```

- 您可以使用 MATLAB `cast` 函数对使用 `coder.opaque` 声明的变量进行类型转换。请仅针对数值类型将 `cast` 与 `coder.opaque` 结合使用。

要将 `coder.opaque` 声明的变量转换为 MATLAB 类型，您可以使用 `B = cast(A,type)` 语法。例如：

```
x = coder.opaque('size_t','0');
x1 = cast(x, 'int32');
```

也可以使用 `B = cast(A,'like',p)` 语法。例如：

```
x = coder.opaque('size_t','0');
x1 = cast(x, 'like', int32(0));
```

要将 MATLAB 变量转换为 `coder.opaque` 声明的变量的类型，您必须使用 `B = cast(A,'like',p)` 语法。例如：

```
x = int32(12);
x1 = coder.opaque('size_t', '0');
x2 = cast(x, 'like', x1);
```

将 `cast` 与 `coder.opaque` 结合使用来为以下项生成正确的数据类型：

- 使用 `coder.ceval` 调用的 C/C++ 函数的输入。
- 您对使用 `coder.ceval` 调用的 C/C++ 函数的输出赋予的变量。

如果没有这种类型转换，在代码生成过程中可能出现编译器警告。

## 版本历史记录

在 R2011a 中推出

## 扩展功能

### C/C++ 代码生成

使用 MATLAB® Coder™ 生成 C 代码和 C++ 代码。

### GPU 代码生成

使用 GPU Coder™ 为 NVIDIA® GPU 生成 CUDA® 代码。

## 另请参阅

`coder.ceval` | `coder.ref` | `coder.rref` | `coder.wref`

### 主题

“指定编译配置参数”

“从生成的代码中调用自定义 C/C++ 代码”

## coder.ref

指示要传引用的数据

### 语法

```
coder.ref(arg)
coder.ref(arg,'gpu')
```

### 说明

`coder.ref(arg)` 表示 `arg` 是要传引用给外部 C/C++ 函数的表达式或变量。只能在 `coder.ceval` 调用中使用 `coder.ref`。C/C++ 函数可以读取或写入按引用传递的变量。为您要按引用传递到函数的每个参数使用一个单独的 `coder.ref` 构造。

另请参阅 `coder.rref` 和 `coder.wref`。

`coder.ref(arg,'gpu')` 指示 `arg` 是 GPU 参数。此选项需要有效的 GPU Coder 许可证。如果 `coder.ceval` 调用 CUDA GPU `__device__` 函数，代码生成器将忽略 `'gpu'` 设定。

### 示例

#### 按引用传递标量变量

假设有 C 函数 `addone`，它返回输入加 1 之后的值：

```
double addone(double* p) {
 return *p + 1;
}
```

C 函数将输入变量 `p` 定义为指向一个双精度数的指针。

通过引用将输入传递给 `addone`：

```
...
y = 0;
u = 42;
y = coder.ceval('addone', coder.ref(u));
...
```

#### 按引用传递多个参数

```
...
u = 1;
v = 2;
y = coder.ceval('my_fcn', coder.ref(u), coder.ref(v));
...
```

#### 按引用传递类属性

```
...
x = myClass;
x.prop = 1;
```

```
coder.ceval('foo', coder.ref(x.prop));
...
```

### 按引用传递结构体

要指示结构体类型在 C 头文件中定义，请使用 `coder.cstructname`。

假设有一个 C 函数 `incr_struct`。此函数可读取和写入输入参数。

```
#include "MyStruct.h"

void incr_struct(struct MyStruct *my_struct)
{
 my_struct->f1 = my_struct->f1 + 1;
 my_struct->f2 = my_struct->f2 + 1;
}
```

C 头文件 `MyStruct.h` 定义名为 `MyStruct` 的结构体类型：

```
#ifndef MYSTRUCT
#define MYSTRUCT

typedef struct MyStruct
{
 double f1;
 double f2;
} MyStruct;
```

```
void incr_struct(struct MyStruct *my_struct);
```

```
#endif
```

在 MATLAB 函数中，通过引用向 `incr_struct` 传递一个结构体。要指示 `s` 的结构体类型具有在 C 头文件 `MyStruct.h` 中定义的名称 `MyStruct`，请使用 `coder.cstructname`。

```
function y = foo
%#codegen
y = 0;
coder.updateBuildInfo('addSourceFiles','incr_struct.c');

s = struct('f1',1,'f2',2);
coder.cstructname(s,'MyStruct','extern','HeaderFile','MyStruct.h');
coder.ceval('incr_struct', coder.ref(s));
```

要生成独立库代码，请输入：

```
codegen -config:lib foo -report
```

### 按引用传递结构体字段

```
...
s = struct('s1', struct('a', [0 1]));
coder.ceval('foo', coder.ref(s.s1.a));
...
```

您也可以传递结构体数组的元素：

```
...
c = repmat(struct('u',magic(2)),1,10);
```

```
b = repmat(struct('c',c),3,6);
a = struct('b',b);
coder.ceval('foo', coder.ref(a.b(3,4).c(2).u));
...
```

## 输入参数

### arg — 要按引用传递的参数

标量变量 | 数组 | 数组的元素 | 结构体 | 结构体字段 | 对象属性

要按引用传递给外部 C/C++ 函数的参数。参数不能是类、System object、元胞数组或元胞数组的索引。

数据类型: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `struct`  
 复数支持: 是

## 限制

- 不能按引用传递以下数据类型:
  - 类或 System object
  - 元胞数组或元胞数组索引
- 如果某属性具有 `get` 方法、`set` 方法或验证程序，或者是具有某些特性的 System object 属性，则您不能按引用将该属性传递给外部函数。请参阅“Passing By Reference Not Supported for Some Properties”。

## 提示

- 如果 `arg` 是数组，则 `coder.ref(arg)` 提供数组第一个元素的地址。`coder.ref(arg)` 函数不包含有关数组大小的信息。如果 C 函数需要知道数据的元素数量，请将该信息作为单独的参数进行传递。例如:
 

```
coder.ceval('myFun',coder.ref(arg),int32(numel(arg)));
```
- 当您按引用将结构体传递给外部 C/C++ 函数时，可以使用 `coder.cstructname` 提供在 C 头文件中定义的 C 结构体类型的名称。
- 在 MATLAB 中，`coder.ref` 会导致错误。要参数化您的 MATLAB 代码以使代码能够在 MATLAB 以及生成的代码中运行，请使用 `coder.target`。
- 您可以使用 `coder.opaque` 来声明传递给外部 C/C++ 函数以及从该函数传回的变量。

## 版本历史记录

在 R2011a 中推出

## 扩展功能

### C/C++ 代码生成

使用 MATLAB® Coder™ 生成 C 代码和 C++ 代码。

### GPU 代码生成

使用 GPU Coder™ 为 NVIDIA® GPU 生成 CUDA® 代码。



**另请参阅**

[coder.rref](#) | [coder.wref](#) | [coder.ceval](#) | [coder.opaque](#) | [coder.cstructname](#) | [numel](#)

**主题**

“从生成的代码中调用自定义 C/C++ 代码”

## coder.resize

包: `coder`

调整 `coder.Type` 对象的大小

### 语法

```
t_out = coder.resize(t,sz)
t_out = coder.resize(t,sz,variable_dims)
t_out = coder.resize(t,[],variable_dims)
t_out = coder.resize(t,sz,variable_dims,Name,Value)
t_out = coder.resize(t,'sizelimits',limits)
```

### 说明

`t_out = coder.resize(t,sz)` 将 `t` 的大小调整为 `sz`。

`t_out = coder.resize(t,sz,variable_dims)` 返回 `coder.Type` `t` 的修改副本，其大小（上界）为 `sz`，变量维度为 `variable_dims`。如果 `variable_dims` 或 `sz` 是标量，该函数会将这些标量应用于 `t` 的所有维度。默认情况下，`variable_dims` 不应用于 `sz` 为 0 或 1（表示固定）的维度。使用 'uniform' 选项可覆盖此特例。对于大小为 `inf` 的维度，`coder.resize` 函数会忽略 `variable_dims`。这些维度是可变大小。`t` 可以是元胞数组类型，在这种情况下，`coder.resize` 会调整元胞数组中所有元素的大小。

`t_out = coder.resize(t,[],variable_dims)` 将 `t` 更改为具有可变维度 `variable_dims`，同时保持大小不变。

`t_out = coder.resize(t,sz,variable_dims,Name,Value)` 通过使用一个或多个名称-值对组参数指定的附加选项来调整 `t` 的大小。

`t_out = coder.resize(t,'sizelimits',limits)` 根据 `limits` 向量中的阈值调整 `t` 的单个维度的大小。`limits` 向量是一个包含两个正整数元素的行向量。`t` 的每个维度都会根据 `limits` 向量中的阈值单独调整大小。

- 当一个维度的大小 `S` 小于在 `limits` 中定义的两个阈值时，该维度保持不变。
- 当一个维度的大小 `S` 大于或等于在 `limits` 中定义的第一个阈值且小于第二个阈值时，该维度将更改为具有可变大小且上界为 `S`。
- 但是，当一个维度的大小 `S` 也大于或等于在 `limits` 中定义的第二个阈值时，该维度将变为具有无界变量大小。

如果 `limits` 的值是标量，将对阈值进行标量扩展以表示两个阈值。例如，如果 `limits` 定义为 4，则会将其解释为 [4 4]。

'sizelimits' 选项允许您在生成的代码中为大型数组动态分配内存。

### 示例

#### 将固定大小数组更改为无界可变大小数组

将固定大小数组更改为无界可变大小数组。

```

t = coder.typeof(ones(3,3))

t =

coder.PrimitiveType
 3×3 double

coder.resize(t,inf)

ans =

coder.PrimitiveType
 :inf×:inf double
% ':' indicates variable-size dimensions

```

### 将固定大小数组更改为有界可变大小数组

将固定大小数组更改为有界可变大小数组。

```

t = coder.typeof(ones(3,3))

t =

coder.PrimitiveType
 3×3 double

coder.resize(t,[4 5],1)

ans =

coder.PrimitiveType
 :4×:5 double
% ':' indicates variable-size dimensions

```

### 调整结构体字段的大小

调整结构体字段的大小。

```

ts = coder.typeof(struct('a',ones(3, 3)))

ts =

coder.StructType
 1×1 struct
 a: 3×3 double

coder.resize(ts,[5, 5],'recursive',1)

ans =

coder.StructType
 5×5 struct
 a: 5×5 double

```

### 调整元胞数组的大小

调整元胞数组的大小。

```
tc = coder.typeof({1 2 3})

tc =

coder.CellType
 1×3 homogeneous cell
 base: 1×1 double

coder.resize(tc,[5, 5],'recursive',1)

ans =

coder.CellType
 5×5 homogeneous cell
 base: 1×1 double
```

### 基于有界和无界阈值将固定大小数组更改为可变大小数组

基于有界和无界阈值将固定大小数组更改为可变大小数组。

```
t = coder.typeof(ones(100,200))

t =

coder.PrimitiveType
 100×200 double

coder.resize(t,'sizelimits',[99 199])

ans =

coder.PrimitiveType
 :100×:inf double
% ':' indicates variable-size dimensions
```

## 输入参数

### limits — 定义阈值的向量

由整数值组成的行向量

由可变大小阈值组成的行向量。如果 **limits** 的值是标量，将对阈值进行标量扩展。如果 **t** 的一个维度的大小 **sz** 大于或等于在 **limits** 中定义的第一个阈值且小于第二个阈值，该维度将更改为具有可变大小且上界为 **sz**。如果 **t** 的一个维度的大小 **sz** 也大于或等于第二个阈值，该维度将变为具有无界变量大小。

但是，如果大小 **sz** 小于两个阈值，则该维度保持不变。

示例： `coder.resize(t,'sizelimits',[99 199]);`

数据类型： `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### sz — 对象类型的新大小

由整数值组成的行向量

**coder.Type** 对象 **t\_out** 的新大小

示例: `coder.resize(t,[3,4]);`

数据类型: `single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64`

**t** — 要调整大小的 **coder.Type** 对象

**coder.Type** 对象

如果 **t** 是 **coder.CellType** 对象, 则 **coder.CellType** 对象必须为同构对象。

示例: `coder.resize(t,inf);`

数据类型: `single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical | char | string | struct | table | cell | function_handle | categorical | datetime | duration | calendarDuration | fi`

复数支持: 是

**variable\_dims** — 可变或固定维度

由逻辑值组成的行向量

指定 **t\_out** 的每个维度是固定大小还是可变大小。

示例: `coder.resize(t,[4 5],1);`

数据类型: `logical`

**名称-值对组参数**

将可选的参数对组指定为 `Name1=Value1,...,NameN=ValueN`, 其中 **Name** 是参数名称, **Value** 是对应的值。名称-值参数必须出现在其他参数后, 但参数对组的顺序无关紧要。

在 R2021a 之前, 使用逗号分隔每个名称和值, 并用引号将 **Name** 引起来。

示例: `coder.resize(t,[5, 5],'recursive', 1);`

**recursive** — 调整 **t** 及其中包含的所有类型的大小

`false` (默认) | `true`

将 **recursive** 设置为 `true` 会调整 **t** 及其中包含的所有类型的大小。

数据类型: `logical`

**uniform** — 通过对大小为 1 的维度应用启发式方法来调整 **t** 的大小

`false` (默认) | `true`

将 **uniform** 设置为 `true` 会调整 **t** 的大小, 并对大小为 1 的维度应用启发式方法。

启发式方法的工作方式如下:

- 如果 **variable\_dims** 是标量 `true`, 则将所有维度的大小都调整为在 **sz** 中指定的上界变量大小。这包括大小为 1 的维度。例如:

```
t = coder.typeof(1, [1 5]);
tResize = coder.resize(t,[1 7],true,'uniform',true);
```

这将生成对象 **tResize**, 如下所示:

```
tResize =
```

```
coder.PrimitiveType
:1×7 double
```

Edit Type Object

- 如果使用 'sizelimits' 选项将 **uniform** 设置为 **true**，则会根据 'sizelimits' 启发式方法，将大小为 1 的维度也调整为可变大小。例如：

```
t = coder.typeof(1, [1 5]);
tResize = coder.resize(t,[],'sizelimits',[0 6],'uniform',true);
```

这些命令会生成一个 **tResize** 对象，如下所示：

```
tResize =

coder.PrimitiveType
:1×5 double
```

Edit Type Object

- 如果将 **variable\_dims** 指定为非标量逻辑值，则 **uniform** 设置不起作用。但是，如果 **variable\_dims** 是标量并且 **uniform** 设置为 **false**，则只会调整大于 1 的维度的大小。

数据类型： **logical**

**sizelimits** — 根据在 **limits** 向量中提供的阈值，调整 **t** 的各个维度的大小  
**limits** (默认)

将 **sizelimits** 选项与 **limits** 向量结合使用会调整 **t** 的各个维度的大小。

```
t = coder.typeof(1, [1 5]);
tResize = coder.resize(t,[],'sizelimits',[0 6],'uniform',true);
```

数据类型： **single** | **double** | **int8** | **int16** | **int32** | **int64** | **uint8** | **uint16** | **uint32** | **uint64**

## 输出参数

**t\_out** — 调整大小后的类型对象  
coder.Type 对象

调整大小后的 **coder.Type** 对象

数据类型： **single** | **double** | **int8** | **int16** | **int32** | **int64** | **uint8** | **uint16** | **uint32** | **uint64** | **logical** | **char** | **string** | **struct** | **table** | **cell** | **function\_handle** | **categorical** | **datetime** | **duration** | **calendarDuration** | **fi**  
复数支持： 是

## 限制

- 对于稀疏矩阵，**coder.resize** 删除可变大小维度的上界。

## 版本历史记录

在 R2011a 中推出

## **另请参阅**

**`coder.typeof` | `coder.newtype` | `codegen`**

## coder.rref

指示要传引用的只读数据

### 语法

```
coder.rref(arg)
coder.rref(arg,'gpu')
```

### 说明

`coder.rref(arg)` 表示 `arg` 是通过引用外部 C/C++ 函数传递的只读表达式或变量。仅在 `coder.ceval` 调用中使用 `coder.rref`。

`coder.rref` 函数支持代码生成器对生成的代码进行优化。由于假定外部函数不写入 `coder.rref(arg)`，因此代码生成器可以对 `coder.ceval` 调用前后发生的对 `arg` 赋值执行优化（如表达式折叠）。表达式折叠指将多项操作组合为一条语句，以避免使用临时变量并提高代码性能。

---

**注意** 代码生成器假定您通过 `coder.rref(arg)` 传递的内存为只读。为了避免不可预测的结果，C/C++ 函数不能写入此变量。

---

另请参阅 `coder.ref` 和 `coder.wref`。

`coder.rref(arg,'gpu')` 指示 `arg` 是 GPU 参数。此选项需要有效的 GPU Coder 许可证。如果 `coder.ceval` 调用 CUDA GPU `__device__` 函数，代码生成器将忽略 'gpu' 设定。

### 示例

#### 将标量变量作为只读引用传递

假设 C 函数 `addone`，它返回常量输入加一之后的值：

```
double addone(const double* p) {
 return *p + 1;
}
```

C 函数将输入变量 `p` 定义为指向一个双精度常量的指针。

通过引用将输入传递给 `addone`：

```
...
y = 0;
u = 42;
y = coder.ceval('addone', coder.rref(u));
...
```

#### 将多个参数作为只读引用传递

```
...
u = 1;
v = 2;
```



```
y = coder.ceval('my_fcn', coder.rref(u), coder.rref(v));
...
```

### 将类属性作为只读引用传递

```
...
x = myClass;
x.prop = 1;
y = coder.ceval('foo', coder.rref(x.prop));
...
```

### 将结构体作为只读引用传递

要指示结构体类型在 C 头文件中定义，请使用 `coder.cstructname`。

假设有一个 C 函数 `use_struct`。此函数读取但不写入输入参数。

```
#include "MyStruct.h"

double use_struct(const struct MyStruct *my_struct)
{
 return my_struct->f1 + my_struct->f2;
}
```

C 头文件 `MyStruct.h` 定义名为 `MyStruct` 的结构体类型：

```
#ifndef MYSTRUCT
#define MYSTRUCT

typedef struct MyStruct
{
 double f1;
 double f2;
} MyStruct;

double use_struct(const struct MyStruct *my_struct);

#endif
```

在 MATLAB 函数中，将结构体作为只读引用传递给 `use_struct`。要指示 `s` 的结构体类型具有在 C 头文件 `MyStruct.h` 中定义的名称 `MyStruct`，请使用 `coder.cstructname`。

```
function y = foo
%#codegen
y = 0;
coder.updateBuildInfo('addSourceFiles','use_struct.c');

s = struct('f1',1,'f2',2);
coder.cstructname(s,'MyStruct','extern','HeaderFile','MyStruct.h');
y = coder.ceval('use_struct', coder.rref(s));
```

要生成独立库代码，请输入：

```
codegen -config:lib foo -report
```

### 将结构体字段作为只读引用传递

```
...
s = struct('s1', struct('a', [0 1]));
```

```
y = coder.ceval('foo', coder.rref(s.s1.a));
...
```

您也可以传递结构体数组的元素：

```
...
c = repmat(struct('u',magic(2)),1,10);
b = repmat(struct('c',c),3,6);
a = struct('b',b);
coder.ceval('foo', coder.rref(a.b(3,4).c(2).u));
...
```

## 输入参数

### arg — 要按引用传递的参数

标量变量 | 数组 | 数组的元素 | 结构体 | 结构体字段 | 对象属性

要按引用传递给外部 C/C++ 函数的参数。参数不能是类、System object、元胞数组或元胞数组的索引。

数据类型： single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical  
| char | struct  
复数支持： 是

## 限制

- 不能按引用传递以下数据类型：
  - 类或 System object
  - 元胞数组或元胞数组索引
- 如果某属性具有 get 方法、set 方法或验证程序，或者是具有某些特性的 System object 属性，则您不能按引用将该属性传递给外部函数。请参阅“Passing By Reference Not Supported for Some Properties”。

## 提示

- 如果 arg 是数组，则 coder.rref(arg) 提供数组第一个元素的地址。coder.rref(arg) 函数不包含有关数组大小的信息。如果 C 函数需要知道数据的元素数量，请将该信息作为单独的参数进行传递。例如：

```
coder.ceval('myFun',coder.rref(arg),int32(numel(arg)));
```

- 当您按引用将结构体传递给外部 C/C++ 函数时，可以使用 coder.cstructname 提供在 C 头文件中定义的 C 结构体类型的名称。
- 在 MATLAB 中，coder.rref 会导致错误。要参数化您的 MATLAB 代码以使代码能够在 MATLAB 以及生成的代码中运行，请使用 coder.target。
- 您可以使用 coder.opaque 来声明传递给外部 C/C++ 函数以及从该函数传回的变量。

## 版本历史记录

在 R2011a 中推出

## 扩展功能

### C/C++ 代码生成

使用 MATLAB® Coder™ 生成 C 代码和 C++ 代码。

### GPU 代码生成

使用 GPU Coder™ 为 NVIDIA® GPU 生成 CUDA® 代码。

## 另请参阅

[coder.ref](#) | [coder.wref](#) | [coder.ceval](#) | [coder.opaque](#) | [coder.cstructname](#)

### 主题

“从生成的代码中调用自定义 C/C++ 代码”

## **coder.screener**

包: **coder**

确定函数是否适合代码生成

### **语法**

```
coder.screener(fcn)
coder.screener(fcn,'-gpu')
coder.screener(fcn_1,...,fcn_n)
info = coder.screener(___)
```

### **说明**

**coder.screener**(fcn) 分析 MATLAB 入口函数 **fcn** 将不支持的函数和语言功能标识为代码生成合规性问题。代码生成合规性问题显示在就绪报告中。

如果 **fcn** 直接或间接调用不是 MathWorks 函数的其他函数（MATLAB 内置函数和工具箱函数），**coder.screener** 将分析这些函数。它不分析 MathWorks 函数。

**coder.screener** 可能无法检测到所有代码生成问题。在某些情况下，**coder.screener** 可能会报告伪错误。

为了避免未检测到的代码生成问题和伪错误，在生成代码之前，请通过执行以下附加检查来验证您的 MATLAB 代码是否适合代码生成：

- 在使用 **coder.screener** 之前，请修复代码分析器标识的问题。
- 在使用 **coder.screener** 后，在生成 C/C++ 代码之前，通过生成和验证 MEX 函数，验证您的 MATLAB 代码是否适合代码生成。

**coder.screener** 函数生成的报告不涉及代码生成器视为外部函数的函数。这些函数的示例有 **plot**、**disp** 和 **figure**。请参阅“使用 MATLAB 引擎在生成的代码中执行函数调用”。

**coder.screener**(fcn,'-gpu') 分析 MATLAB 入口函数 **fcn**，以识别 GPU 代码生成不支持的函数和语言功能。

**coder.screener**(fcn\_1,...,fcn\_n) 分析多个 MATLAB 入口函数。

**info** = **coder.screener**(\_\_\_) 返回 **coder.ScreenerInfo** 对象。此对象的属性包含代码生成就绪分析结果。使用 **info** 以编程方式访问代码生成就绪结果。有关属性列表，请参阅 **coder.ScreenerInfo** Properties。

### **示例**

#### **标识不支持的函数**

**coder.screener** 函数会标识对代码生成不支持的函数的调用。它会检查入口函数 **foo1** 和 **foo1** 调用的函数 **foo2**。

编写 `foo2` 函数并将其保存在 `foo2.m` 文件中。

```
function [tf1,tf2] = foo2(source,target)
G = digraph(source,target);
tf1 = hascycles(G);
tf2 = isdag(G);
end
```

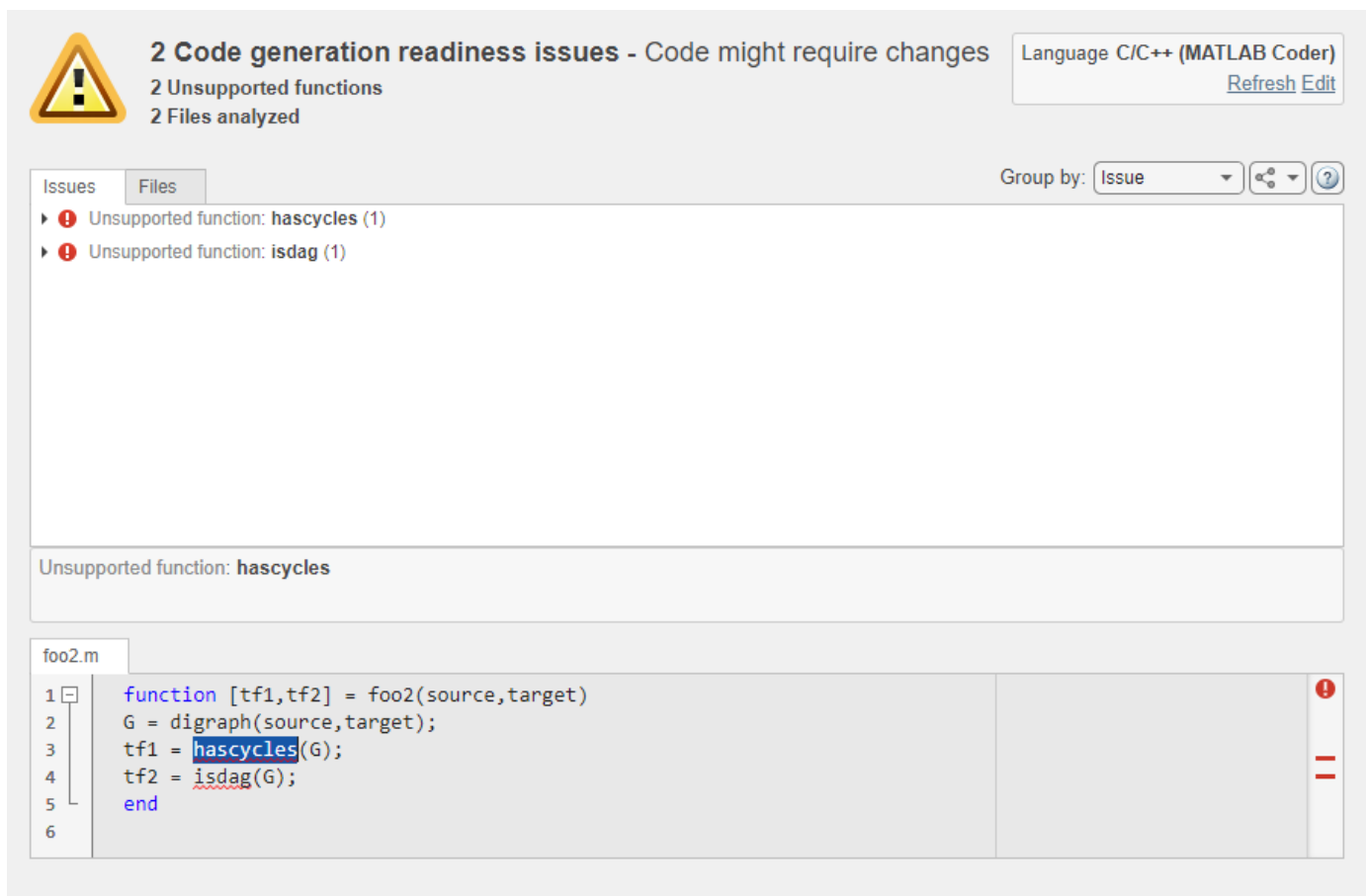
编写调用 `foo2` 的函数 `foo1`。将 `foo1` 保存在 `foo1.m` 文件中。

```
function [tf1,tf2] = foo1(source,target)
assert(numel(source)==numel(target))
[tf1,tf2] = foo2(source,target);
end
```

分析 `foo1`。

```
coder.screener('foo1')
```

代码生成就绪报告显示不支持的 MATLAB 函数调用的摘要。报告的**问题**选项卡指示 `foo2.m` 包含一个对 `isdag` 函数的调用和一个对 `hascycles` 的调用，而代码生成不支持这些函数。



The screenshot shows the **2 Code generation readiness issues - Code might require changes** window. It indicates **2 Unsupported functions** and **2 Files analyzed**. The language is set to **C/C++ (MATLAB Coder)**. The **Issues** tab is active, showing a list of unsupported functions: `hascycles` (1) and `isdag` (1). Below the list, a detailed view for the `Unsupported function: hascycles` is shown. At the bottom, the `foo2.m` file is open, displaying the MATLAB code. The code defines a function `foo2` that uses `digraph`, `hascycles`, and `isdag`. The `hascycles` and `isdag` calls are highlighted with red squiggly lines, indicating they are unsupported for code generation. A red exclamation mark icon is visible in the right margin of the code editor.

函数 `foo2` 调用两个不支持的 MATLAB 函数。要生成 MEX 函数，请修改代码以使用 `coder.extrinsic` 指令执行对 `hascycles` 和 `isdag` 外部函数的调用，然后重新运行代码生成就绪工具。

```
function [tf1,tf2] = foo2(source,target)
coder.extrinsic('hascycles','isdag');
```

```
G = digraph(source,target);
tf1 = hascycles(G);
tf2 = isdag(G);
end
```

对入口函数 `foo1` 重新运行 `coder.screener`。

```
coder.screener('foo1')
```

报告不再指示代码生成不支持 `hascycles` 和 `isdag` 函数。为 `foo1` 生成 MEX 函数时，代码生成器会将这两个函数调度给 MATLAB 来执行。

### 以编程方式访问代码生成就绪结果

您可以带可选输出参数调用 `coder.screener` 函数。如果您使用此语法，`coder.screener` 函数将返回 `coder.ScreenerInfo` 对象，其中包含您的 MATLAB 代码库的代码生成就绪分析结果。请参阅 `coder.ScreenerInfo` Properties。

此示例使用在前面的示例中定义的文件 `foo1.m` 和 `foo2.m`。调用 `coder.screener` 函数：

```
info = coder.screener('foo1.m')
```

```
info =
```

ScreenerInfo with properties:

```
Files: [2×1 coder.CodeFile]
Messages: [2×1 coder.Message]
UnsupportedCalls: [2×1 coder.CallSite]
```

[View Screener Report](#)

要访问第一个不受支持调用的相关信息，请对 `UnsupportedCalls` 属性进行索引。

```
firstCall = info.UnsupportedCalls(1)
```

```
firstCall =
```

CallSite with properties:

```
CalleeName: 'hascycles'
File: [1×1 coder.CodeFile]
StartIndex: 78
EndIndex: 86
```

不受支持的调用为 `hascycles`，查看包含该调用的文件文本。

```
firstCall.File.Text
```

```
ans =
```

```
'function [tf1,tf2] = foo2(source,target)
G = digraph(source,target);
tf1 = hascycles(G);
tf2 = isdag(G);
end
'
```

要将整个代码生成就绪报告导出为 MATLAB 字符串，请使用 `textReport` 函数。

```
reportString = textReport(info)

reportString =

'Code Generation Readiness (Text Report)
=====

2 Code generation readiness issues
2 Unsupported functions
2 Files analyzed

Configuration
=====

Language: C/C++ (MATLAB Coder)

Code Generation Issues
=====

Unsupported function: digraph (2)
- foo2.m (Line 3)
- foo2.m (Line 4)

'
```

## 识别不支持的数据类型

`coder.screener` 函数标识代码生成不支持的 MATLAB 数据类型。

编写包含 MATLAB 日历持续时间数组数据类型的函数 `myfun1`。

```
function out = myfun1(A)
out = calyears(A);
end
```

分析 `myfun1`。

```
coder.screener('myfun1');
```

代码生成就绪报告表明代码生成不支持 `calyears` 数据类型。在生成代码之前，修复报告的问题。

## 输入参数

**fcn** — 入口函数的名称

字符向量 | 字符串标量

用于分析的 MATLAB 入口函数的名称。指定为字符向量或字符串标量。

示例： `coder.screener('myfun');`

数据类型： `char` | `string`

**fcn\_1,...,fcn\_n** — 入口函数名称列表

字符向量 | 字符串标量

用于分析的 MATLAB 入口函数名称的以逗号分隔的列表。指定为字符向量或字符串标量。

示例: `coder.screener('myfun1','myfun2');`

数据类型: `char` | `string`

### 替代方法

- “Run Code Generation Readiness Tool from the Current Folder Browser”
- “Run the Code Generation Readiness Tool Using the MATLAB Coder App”

### 版本历史记录

在 R2012b 中推出

### 另请参阅

`codegen` | `coder.extrinsic`

### 主题

“支持 C/C++ 代码生成的 MATLAB 语言功能”

“C/C++ 代码生成支持的函数和对象”

“代码生成就绪工具”



# coder.target

确定代码生成目标是否为指定的目标

## 语法

```
tf = coder.target(target)
```

## 说明

如果代码生成目标是 `target`，则 `tf = coder.target(target)` 返回 `true` (1)。否则，它返回 `false` (0)。

如果为 MATLAB 类生成代码，则 MATLAB 会在代码生成之前加载类时计算类的初始值。如果您在 MATLAB 类属性初始化中使用 `coder.target`，`coder.target('MATLAB')` 将返回 `true`。

## 示例

### 使用 `coder.target` 参数化 MATLAB 函数

参数化 MATLAB 函数，以便它在 MATLAB 或生成的代码中工作。当函数在 MATLAB 中运行时，它会调用 MATLAB 函数 `myabsval`。然而，生成的代码会调用 C 库函数 `myabsval`。

编写 MATLAB 函数 `myabsval`。

```
function y = myabsval(u)
%#codegen
y = abs(u);
```

通过使用 `-args` 选项指定输入参数的大小、类型和复/实性，为 `myabsval` 生成 C 静态库。

```
codegen -config:lib myabsval -args {0.0}
```

`codegen` 函数在文件夹 `\codegen\lib\myabsval` 中创建库文件 `myabsval.lib` 和头文件 `myabsval.h`。（库文件扩展名可以根据您的平台而有所不同。）它在同一个文件夹中生成函数 `myabsval_initialize` 和 `myabsval_terminate`。

编写一个 MATLAB 函数，以使用 `coder.ceval` 调用生成的 C 库函数。

```
function y = callmyabsval(y)
%#codegen
% Check the target. Do not use coder.ceval if callmyabsval is
% executing in MATLAB
if coder.target('MATLAB')
 % Executing in MATLAB, call function myabsval
 y = myabsval(y);
else
 % add the required include statements to generated function code
 coder.updateBuildInfo('addIncludePaths','$(START_DIR)\codegen\lib\myabsval');
 coder.cinclude('myabsval_initialize.h');
 coder.cinclude('myabsval.h');
 coder.cinclude('myabsval_terminate.h');
```

```

% Executing in the generated code.
% Call the initialize function before calling the
% C function for the first time
coder.ceval('myabsval_initialize');

% Call the generated C library function myabsval
y = coder.ceval('myabsval',y);

% Call the terminate function after
% calling the C function for the last time
coder.ceval('myabsval_terminate');
end

```

生成 MEX 函数 `callmyabsval_mex`。在命令行中提供生成的库文件。

```
codegen -config:mex callmyabsval codegen\lib\myabsval\myabsval.lib -args {-2.75}
```

您可以使用 `coder.updateBuildInfo` 在函数中指定该库，而不是在命令行中提供库。使用此选项可预配置编译。将以下行添加到 `else` 模块中：

```
coder.updateBuildInfo('addLinkObjects','myabsval.lib',$(START_DIR)\codegen\lib\myabsval',100,true,true);
```

---

**注意** 只有使用 MATLAB Coder 生成代码时，才支持 `START_DIR` 宏。

---

运行调用库函数 `myabsval` 的 MEX 函数 `callmyabsval_mex`。

```
callmyabsval_mex(-2.75)
```

```
ans =
```

```
2.7500
```

调用 MATLAB 函数 `callmyabsval`。

```
callmyabsval(-2.75)
```

```
ans =
```

```
2.7500
```

`callmyabsval` 函数在 MATLAB 和代码生成中的执行均展示出了预期的行为。

## 输入参数

**target** — 代码生成目标

'MATLAB' | 'C' | 'C++' | 'CUDA' | 'OpenCL' | 'SystemC' | 'SystemVerilog' | 'Verilog' | 'VHDL' | 'MEX' | 'Sfun' | 'Rtw' | 'HDL' | 'Custom'

代码生成目标，指定为字符向量或字符串标量。指定下列目标之一。

'MATLAB'                      在 MATLAB 中运行（不生成代码）

'C', 'C++', 'CUDA', 'OpenCL' 'SystemC', 'SystemVerilog', 'Verilog', 'VHDL'	代码生成支持的目标语言
'MEX'	生成 MEX 函数
'Sfun'	仿真 Simulink 模型。也用于在加速模式下运行。
'Rtw'	生成 LIB、DLL 或 EXE 目标。也用于在 Simulink Coder 和快速加速模式下运行。
'HDL'	生成 HDL 目标
'Custom'	生成自定义目标

示例: `tf = coder.target('MATLAB')`

示例: `tf = coder.target("MATLAB")`

---

**注意** 对于 CUDA 或 SystemC 代码生成, `coder.target('C++')` 始终为 `true`。

---

## 版本历史记录

在 R2011a 中推出

### 扩展功能

#### C/C++ 代码生成

使用 MATLAB® Coder™ 生成 C 代码和 C++ 代码。

#### GPU 代码生成

使用 GPU Coder™ 为 NVIDIA® GPU 生成 CUDA® 代码。

### 另请参阅

`coder.ceval` | `coder.cinclude` | `coder.updateBuildInfo` | `coder.BuildConfig` |  
`coder.ExternalDependency`

#### 主题

“从生成的代码中调用自定义 C/C++ 代码”

## coder.varsize

包: coder

声明可变大小数据

### 语法

```
coder.varsize(varName1,...,varNameN)
coder.varsize(varName1,...,varNameN,ubounds)
coder.varsize(varName1,...,varNameN,ubounds,dims)
```

### 说明

`coder.varsize(varName1,...,varNameN)` 声明名为 `varName1,...,varNameN` 的变量具有可变大小。声明指示代码生成器允许变量在生成的代码执行期间更改大小。使用此语法，您不必指定变量维度的上界或哪些维度可以更改大小。代码生成器会计算上界。除单一维度（第 2-99 页）外，所有维度都允许更改大小。

根据以下限制和规范使用 `coder.varsize`：

- 在用于代码生成的 MATLAB 函数中使用 `coder.varsize`。
- `coder.varsize` 声明必须先于变量的首次使用。例如：

```
...
x = 1;
coder.varsize('x');
disp(size(x));
...
```

- 使用 `coder.varsize` 声明输出参数具有可变大小或解决大小不匹配错误。否则，要定义可变大小数据，请使用“为代码生成定义可变大小数据”中所述的方法。

**注意** 对于 MATLAB Function 模块，要声明可变大小的输出变量，请使用符号窗格和属性检查器。请参阅“声明可变大小的 MATLAB Function 模块变量”（Simulink）。如果您在 `coder.varsize` 声明中提供上界，则上界必须与属性检查器中的上界相匹配。

有关其他限制和规范，请参阅“限制”（第 2-97 页）和“提示”（第 2-99 页）。

`coder.varsize(varName1,...,varNameN,ubounds)` 还为变量的每个维度指定上界。所有变量必须具有相同的维数。除单一维度（第 2-99 页）外，所有维度都允许更改大小。

`coder.varsize(varName1,...,varNameN,ubounds,dims)` 还指定变量每个维度的上界，以及每个维度是固定大小还是可变大小。如果某个维度具有固定大小，则对应的 `ubound` 元素指定该维度的固定大小。所有变量都有相同的固定大小维度和可变大小维度。

### 示例

## 使用 coder.varsize 解决大小不匹配错误

使用（读取）变量后，更改该变量的大小会导致大小不匹配错误。使用 `coder.varsize` 指定变量的大小可以更改。

以下函数的代码生成会产生大小不匹配错误，因为在 `y = size(x)` 行使用 `x` 后，`x = 1:10` 会更改 `x` 的第二个维度的大小。

```
function [x,y] = usevarsize(n)
%#codegen
x = 1;
y = size(x);
if n > 10
 x = 1:10;
end
```

要声明 `x` 可以更改大小，请使用 `coder.varsize`。

```
function [x,y] = usevarsize(n)
%#codegen
x = 1;
coder.varsize('x');
y = size(x);
if n > 10
 x = 1:10;
end
```

如果您删除 `y = size(x)` 行，则不再需要 `coder.varsize` 声明，因为 `x` 在其大小更改之前不会被使用。

## 声明具有上界的可变大小数组

指定 `A` 是行向量，其第二个维度具有可变大小且上界为 20。

```
function fcn()
...
coder.varsize('A',[1 20]);
...
end
```

当您不提供 `dims` 时，除单一维度外，所有维度都具有可变大小。

## 声明兼具固定和可变维度的可变大小数组

指定 `A` 是数组，其第一个维度具有固定大小 3，第二个维度具有可变大小且上界为 20。

```
function fcn()
...
coder.varsize('A',[3 20], [0 1]);
...
end
```

### 声明可变大小结构体字段

在此函数中，语句 `coder.varsize('data.values')` 声明 `data` 的每个元素内的字段 `values` 具有可变大小。

```
function y = varsize_field()
%#codegen

d = struct('values', zeros(1,0), 'color', 0);
data = repmat(d, [3 3]);
coder.varsize('data.values');

for i = 1:numel(data)
 data(i).color = rand-0.5;
 data(i).values = 1:i;
end

y = 0;
for i = 1:numel(data)
 if data(i).color > 0
 y = y + sum(data(i).values);
 end
end
```

### 声明可变大小元胞数组

指定元胞数组 `C` 的第一个维度具有固定大小，第二个维度具有可变大小且上界为 3。`coder.varsize` 声明必须先于 `C` 的首次使用。

```
...
C = {1 [1 2]};
coder.varsize('C', [1 3], [0 1]);
y = C{1};
...
end
```

在没有 `coder.varsize` 声明的情况下，`C` 是异构元胞数组，其元素具有相同的类，但大小不同。在有 `coder.varsize` 声明的情况下，`C` 是同构元胞数组，其元素具有相同的类和最大大小。每个元素的第一个维度的大小固定为 1。每个元素的第二个维度的大小可变且上界为 3。

### 声明元胞数组具有可变大小的元素

指定元胞数组 `C` 的元素是向量，其第一个维度具有固定大小，第二个维度具有可变大小且上界为 5。

```
...
C = {1 2 3};
coder.varsize('C{:}', [1 5], [0 1]);
```

```
C = {1, 1:5, 2:3};
...
```

## 输入参数

**varName1,...,varNameN** — 要声明为具有可变大小的变量的名称

字符向量 | 字符串标量

要声明为具有可变大小的变量的名称，指定为一个或多个字符向量或字符串标量。

示例： `coder.varsize('x','y')`

**ubounds** — 数组维度的上界

[] (默认) | 由整数常量组成的向量

数组维度的上界，指定为由整数常量组成的向量。

如果没有指定 **ubounds**，代码生成器会计算每个变量的上界。如果 **ubounds** 元素对应于固定大小维度，则值是该维度的固定大小。

示例： `coder.varsize('x','y',[1 2])`

**dims** — 指示每个维度具有固定大小还是可变大小

逻辑向量

指示每个维度具有固定大小还是可变大小，指定为逻辑向量。对应于 **dims** 中的 0 或 **false** 的维度具有固定大小。对应于 1 或 **true** 的维度具有可变大小。

当没有指定 **dims** 时，除单一维度之外，其他维度都具有可变大小。

示例： `coder.varsize('x','y',[1 2], [0 1])`

## 限制

- **coder.varsize** 声明指示代码生成器允许变量的大小发生变化。它不会更改变量的大小。以如下代码为例：

```
...
x = 7;
coder.varsize('x', [1,5]);
disp(size(x));
...
```

在 **coder.varsize** 声明后，**x** 仍然是 1×1 数组。您不能为超出 **x** 的当前大小的元素赋值。例如，以下代码会产生运行时错误，因为索引 3 超出了 **x** 的维数。

```
...
x = 7;
coder.varsize('x', [1,5]);
x(3) = 1;
...
```

- 函数输入参数不支持 **coder.varsize**。在这种情况下：
  - 如果函数是入口函数，请在命令行中使用 **coder.typeof** 指定输入参数具有可变大小。或者，通过使用 App 的 **定义输入类型** 步骤，指定入口函数输入参数具有可变大小。

- 如果函数不是入口函数，请在调用方函数中使用 `coder.varsize`，变量是被调用函数的输入。
- 对于稀疏矩阵，`coder.varsize` 会删除可变大小的上限。
- 对元胞数组使用 `coder.varsize` 的限制：
  - 只有同构元胞数组才能具有可变大小的。将 `coder.varsize` 用于异构元胞数组时，代码生成器会尝试使元胞数组的结构相同。代码生成器会尝试查找适用于元胞数组所有元素的类和最大大小。以元胞数组 `c = {1, [2 3]}` 为例。两个元素都可以用双精度类型表示，其第一个维度具有固定大小 1，第二个维度具有可变大小的上界为 2。如果代码生成器找不到通用的类和最大大小，代码生成将失败。以元胞数组 `c = {'a', [2 3]}` 为例。代码生成器找不到能够同时表示这两个元素的类，因为第一个元素是 `char`，第二个元素是 `double`。
  - 如果使用 `cell` 函数定义固定大小元胞数组，则无法使用 `coder.varsize` 指定元胞数组具有可变大小的。例如，以下代码会导致代码生成错误，因为 `x = cell(1,3)` 使 `x` 成为固定大小的  $1 \times 3$  元胞数组。

```
...
x = cell(1,3);
coder.varsize('x',[1 5])
...
```

您可以将 `coder.varsize` 与使用花括号定义的元胞数组结合使用。例如：

```
...
x = {1 2 3};
coder.varsize('x',[1 5])
...
```

- 要使用 `cell` 函数创建可变大小的元胞数组，请使用以下代码模式：

```
function mycell(n)
%#codegen
x = cell(1,n);
for i = 1:n
 x{i} = i;
end
end
```

请参阅“使用 `cell` 定义可变大小的元胞数组”。

要指定元胞数组的上界，请使用 `coder.varsize`。

```
function mycell(n)
%#codegen
x = cell(1,n);
for i = 1:n
 x{i} = i;
coder.varsize('x',[1,20]);
end
end
```

- `coder.varsize` 不支持：
  - 全局变量
  - MATLAB 类或类属性
  - 字符串标量



## 详细信息

### 单一维度

满足 `size(A,dim) = 1` 的维度。

## 提示

- 在代码生成报告或 MATLAB 函数报告中，冒号 (:) 指示维度具有可变大小。例如，`1x:2` 的大小表示第一个维度具有固定大小 1，第二个维度具有可变大小且上界为 2。
- 如果使用 `coder.varsize` 指定一个维度的上界为 1，则默认情况下，该维度具有固定大小 1。要指定维度可以是 0（空数组）或 1，请将 `dims` 参数的对应元素设置为 `true`。例如，以下代码指定 `x` 的第一个维度具有固定大小 1，其他维度具有上界为 5 的可变大小。

```
coder.varsize('x',[1,5,5])
```

而以下代码指定 `x` 的第一个维度的上界为 1 且大小可变（可以是 0 或 1）。

```
coder.varsize('x',[1,5,5],[1,1,1])
```

---

**注意** 对于 MATLAB Function 模块，您不能指定大小为 1 的输出信号具有可变大小。

---

- 如果使用输入变量（或使用输入变量进行计算的结果）指定数组的大小，则在生成的代码中会将其声明为可变大小。不要对该数组重用 `coder.varsize`，除非您还要为其大小指定上界。
- 如果没有用 `coder.varsize` 声明指定上界，并且代码生成器无法确定上界，则生成的代码使用动态内存分配。动态内存分配可能会降低生成的代码的速度。要避免动态内存分配，请通过提供 `ubounds` 参数来指定上界。

## 版本历史记录

在 R2011a 中推出

### 另请参阅

`coder.typeof`

### 主题

“可变大小数组的代码生成”

“在代码生成的可变大小支持方面与 MATLAB 的不兼容性”

“Avoid Duplicate Functions in Generated Code”

## coder.wref

指示要按引用传递的只写数据

### 语法

```
coder.wref(arg)
coder.wref(arg,'gpu')
```

### 说明

`coder.wref(arg)` 表示 `arg` 是按引用传递给外部 C/C++ 函数的只写表达式或变量。仅在 `coder.ceval` 调用中使用 `coder.wref`。此函数使代码生成器能够通过忽略您的 MATLAB 代码中对 `arg` 的先前赋值来优化生成的代码，因为外部函数假定是不从数据中读取的。写入外部代码中 `arg` 的所有元素，以完全初始化内存。

---

**注意** C/C++ 函数必须完全初始化 `coder.wref(arg)` 引用的内存。通过在 C/C++ 代码中为 `arg` 的每个元素赋值来初始化内存。如果生成的代码尝试从未初始化的内存中读取，可能导致未定义的运行时行为。

---

另请参阅 `coder.ref` 和 `coder.rref`。

`coder.wref(arg,'gpu')` 指示 `arg` 是 GPU 参数。此选项需要有效的 GPU Code 许可证。如果 `coder.ceval` 调用 CUDA GPU `__device__` 函数，代码生成器将忽略 `'gpu'` 设定。

### 示例

#### 按引用将数组作为只写数组传递

假设您有一个 C 函数 `init_array`。

```
void init_array(double* array, int numel) {
 for(int i = 0; i < numel; i++) {
 array[i] = 42;
 }
}
```

C 函数将输入变量 `array` 定义为指向一个双精度数的指针。

调用 C 函数 `init_array` 以将 `y` 的所有元素初始化为 42：

```
...
Y = zeros(5, 10);
coder.ceval('init_array', coder.wref(Y), int32(numel(Y)));
...
```

#### 将多个参数作为只写引用传递

```
...
U = zeros(5, 10);
V = zeros(5, 10);
```

```
coder.ceval('my_fcn', coder.wref(U), int32(numel(U)), coder.wref(V), int32(numel(V)));
...
```

### 将类属性作为只写引用传递

```
...
x = myClass;
x.prop = 1;
coder.ceval('foo', coder.wref(x.prop));
...
```

### 将结构体作为只写引用传递

要指示结构体类型在 C 头文件中定义，请使用 `coder.cstructname`。

假设有一个 C 函数 `init_struct`。此函数写入但不读取输入参数。

```
#include "MyStruct.h"

void init_struct(struct MyStruct *my_struct)
{
 my_struct->f1 = 1;
 my_struct->f2 = 2;
}
```

C 头文件 `MyStruct.h` 定义名为 `MyStruct` 的结构体类型：

```
#ifndef MYSTRUCT
#define MYSTRUCT

typedef struct MyStruct
{
 double f1;
 double f2;
} MyStruct;
```

```
void init_struct(struct MyStruct *my_struct);
```

```
#endif
```

在 MATLAB 函数中，将结构体作为只写引用传递给 `init_struct`。使用 `coder.cstructname` 指示 `s` 的结构体类型具有在 C 头文件 `MyStruct.h` 中定义的名称 `MyStruct`。

```
function y = foo
%#codegen
y = 0;
coder.updateBuildInfo('addSourceFiles','init_struct.c');

s = struct('f1',1,'f2',2);
coder.cstructname(s,'MyStruct','extern','HeaderFile','MyStruct.h');
coder.ceval('init_struct', coder.wref(s));
```

要生成独立库代码，请输入：

```
codegen -config:lib foo -report
```

### 将结构体字段作为只写引用传递

```
...
s = struct('s1', struct('a', [0 1]));
```

```
coder.ceval('foo', coder.wref(s.s1.a));
...
```

您也可以传递结构体数组的元素：

```
...
c = repmat(struct('u',magic(2)),1,10);
b = repmat(struct('c',c),3,6);
a = struct('b',b);
coder.ceval('foo', coder.wref(a.b(3,4).c(2).u));
...
```

## 输入参数

### arg — 要按引用传递的参数

标量变量 | 数组 | 数组的元素 | 结构体 | 结构体字段 | 对象属性

要按引用传递给外部 C/C++ 函数的参数。参数不能是类、System object、元胞数组或元胞数组的索引。

数据类型： single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical  
| char | struct  
复数支持： 是

## 限制

- 不能按引用传递以下数据类型：
  - 类或 System object
  - 元胞数组或元胞数组索引
- 如果某属性具有 get 方法、set 方法或验证程序，或者是具有某些特性的 System object 属性，则您不能按引用将该属性传递给外部函数。请参阅“Passing By Reference Not Supported for Some Properties”。

## 提示

- 如果 arg 是数组，则 `coder.wref(arg)` 提供数组第一个元素的地址。`coder.wref(arg)` 函数不包含有关数组大小的信息。如果 C 函数需要知道数据的元素数量，请将该信息作为单独的参数进行传递。例如：

```
coder.ceval('myFun',coder.wref(arg),int32(numel(arg)));
```

- 当您按引用将结构体传递给外部 C/C++ 函数时，可以使用 `coder.cstructname` 提供在 C 头文件中定义的 C 结构体类型的名称。
- 在 MATLAB 中，`coder.wref` 会导致错误。要参数化您的 MATLAB 代码以使代码能够在 MATLAB 以及生成的代码中运行，请使用 `coder.target`。
- 您可以使用 `coder.opaque` 来声明传递给外部 C/C++ 函数以及从该函数传回的变量。

## 版本历史记录

在 R2011a 中推出

## 扩展功能

### C/C++ 代码生成

使用 MATLAB® Coder™ 生成 C 代码和 C++ 代码。

### GPU 代码生成

使用 GPU Coder™ 为 NVIDIA® GPU 生成 CUDA® 代码。

## 另请参阅

[coder.ref](#) | [coder.rref](#) | [coder.ceval](#) | [coder.opaque](#) | [coder.cstructname](#)

### 主题

“从生成的代码中调用自定义 C/C++ 代码”

## parfor

并行 for 循环

### 语法

```
parfor LoopVar = InitVal:EndVal; Statements; end
parfor (LoopVar = InitVal:EndVal, NumThreads); Statements; end
```

### 说明

`parfor LoopVar = InitVal:EndVal; Statements; end` 在生成的 MEX 函数中或在共享内存多核平台上并行运行的 C/C++ 代码中创建一个循环。

`parfor` 循环对 `InitVal` 和 `Endval` 之间的 `LoopVar` 值执行 `Statements`。`LoopVar` 指定由整数值组成的向量，这些值按 1 递增。

`parfor (LoopVar = InitVal:EndVal, NumThreads); Statements; end` 在创建并行 `for` 循环时，最多使用 `NumThreads` 个线程。

### 示例

#### 为 parfor 生成 MEX

为 `parfor` 循环生成一个 MEX 函数，以在最大可用核数上执行。

编写一个 MATLAB 函数 `test_parfor`，它在 `parfor` 循环中调用快速傅里叶变换函数 `fft`。由于循环迭代是并行运行的，因此该计算可以比类似的 `for` 循环更快地完成。

```
function a = test_parfor %#codegen
 a = ones(10,256);
 r = rand(10,256);
 parfor i = 1:10
 a(i,:) = real(fft(r(i)));
 end
end
```

为 `test_parfor` 生成一个 MEX 函数。在 MATLAB 命令行中输入：

```
codegen test_parfor
```

`codegen` 在当前文件夹中生成一个 MEX 函数 `test_parfor_mex`。

运行 MEX 函数。在 MATLAB 命令行中输入：

```
test_parfor_mex
```

该 MEX 函数使用可用的核运行。

## 指定 parfor 的最大线程数

为 parfor 循环生成 MEX 函数时，请指定最大线程数。

编写一个 MATLAB 函数 `specify_num_threads`，该函数使用输入 `u` 来指定 parfor 循环中的最大线程数。

```
function y = specify_num_threads(u) %#codegen
 y = ones(1,100);
 % u specifies maximum number of threads
 parfor (i = 1:100,u)
 y(i) = i;
 end
end
```

为 `specify_num_threads` 生成一个 MEX 函数。使用 `-args 0` 指定输入的类型。在 MATLAB 命令行中输入：

```
% -args 0 specifies that input u is a scalar double
% u is typecast to an integer by the code generator
codegen -report specify_num_threads -args 0
```

`codegen` 在当前文件夹中生成一个 MEX 函数 `specify_num_threads_mex`。

运行该 MEX 函数，指定它最多在四个线程上并行运行。在 MATLAB 命令行中输入：

```
specify_num_threads_mex(4)
```

生成的 MEX 函数最多在四个核上运行。如果可用核少于四个，MEX 函数将在调用时的最大可用核数上运行。

## 在不使用并行化的情况下为 parfor 生成 MEX

在为 parfor 循环生成 MEX 函数之前禁用并行化。

编写一个 MATLAB 函数 `test_parfor`，它在 parfor 循环中调用快速傅里叶变换函数 `fft`。

```
function a = test_parfor %#codegen
 a = ones(10,256);
 r = rand(10,256);
 parfor i = 1:10
 a(i,:) = real(fft(r(i)));
 end
end
```

为 `test_parfor` 生成一个 MEX 函数。禁用 OpenMP，这样，`codegen` 便不会生成可在多个线程上运行的 MEX 函数。

```
codegen -O disable:OpenMP test_parfor
```

`codegen` 在当前文件夹中生成一个 MEX 函数 `test_parfor_mex`。

运行 MEX 函数。

```
test_parfor_mex
```

MEX 函数在单一线程上运行。

如果禁用并行化，MATLAB Coder 会将 **parfor** 循环视为 **for** 循环。该软件会生成在单一线程上运行的 MEX 函数。禁用并行化以比较生成的 MEX 函数或 C/C++ 代码的串行和并行版本的性能。您还可以禁用并行化来调试并行版本的问题。

## 输入参数

### LoopVar — 循环索引

整数

循环索引变量，其初始值为 **InitVal**，结束值为 **EndVal**。

### InitVal — 循环索引的初始值

整数

循环索引变量 **Loopvar** 的初始值。此参数与 **EndVal** 一起指定 **parfor** 范围向量，形式必须为 **M:N**。

### EndVal — 循环索引的结束值

整数

循环索引变量 **LoopVar** 的结束值。此参数与 **InitVal** 一起指定 **parfor** 范围向量，形式必须为 **M:N**。

### Statements — 循环体

文本

要在 **parfor** 循环中执行的一系列 MATLAB 命令。

如果您在同一行中放置多个语句，请用分号分隔这些语句。例如：

```
parfor i = 1:10
 arr(i) = rand(); arr(i) = 2*arr(i)-1;
end
```

### NumThreads — 并行运行的最大线程数

可用的核数量（默认） | 非负整数

要使用的最大线程数。如果您指定上限，MATLAB Coder 使用的线程数不能超过此数量，即使有额外的可用核也是如此。如果您请求的线程数超过可用核数，MATLAB Coder 将使用在调用时可用的最大核数。如果循环迭代数少于线程数，一些线程将不执行任何工作。

如果 **parfor** 循环无法在多个线程上运行（例如，如果只有一个核可用或 **NumThreads** 为 0），MATLAB Coder 会以串行方式执行循环。

## 限制

- 必须使用支持 Open Multiprocessing (OpenMP) 应用程序接口的编译器。请参阅 [https://www.mathworks.com/support/compilers/current\\_release/](https://www.mathworks.com/support/compilers/current_release/)。如果使用不支持 OpenMP 的编译器，则 MATLAB Coder 会将 **parfor** 循环视为 **for** 循环。在生成的 MEX 函数或 C/C++ 代码中，循环迭代在单个线程上运行。
- OpenMP 应用程序接口与 JIT MEX 编译不兼容。请参阅 “JIT Compilation Does Not Support OpenMP”。
- 不要在 **parfor** 循环内使用以下结构：
  - 在 **parfor** 循环体中，不能使用 **coder.extrinsic** 调用外部函数。



- 不能在 **parfor** 循环内写入全局变量。
- MATLAB Coder 不支持在归约中使用 **coder.ceval**。例如，不能为以下 **parfor** 循环生成代码：

```
parfor i = 1:4
 y = coder.ceval('myCFcn',y,i);
end
```

在这种情况下，应使用 **coder.ceval** 编写一个调用 C 代码的局部函数，并在 **parfor** 循环中调用此函数。例如：

```
parfor i = 1:4
 y = callMyCFcn(y,i);
end
function y = callMyCFcn(y,i)
 y = coder.ceval('mCyFcn', y , i);
end
```

- 在 **parfor** 循环中，不能使用 **varargin** 或 **varargout**。
- 循环索引的类型必须可由目标硬件上的整数类型表示。在生成的代码中使用不需要多字类型的类型。
- 用于独立代码生成的 **parfor** 需要工具链方法来编译可执行文件或库。不要更改使代码生成器使用模板联编文件方法的设置。请参阅“Project or Configuration Is Using the Template Makefile”。
- 要在您的 MATLAB 代码中使用 **parfor**，您需要 Parallel Computing Toolbox™ 许可证。

有关限制的详尽列表，请参阅“parfor 限制”。

## 提示

- 在以下情况下，请使用 **parfor** 循环：
  - 您需要一个简单计算的多次循环迭代。**parfor** 会将这些循环迭代分为若干组，以使每个线程执行一组迭代。
  - 循环迭代需要很长时间才能执行完毕。
- 当循环中的迭代依赖于其他迭代的结果时，请不要使用 **parfor** 循环。

归约是此规则的一个例外。归约变量会将依赖于所有迭代的值累加在一起，但与迭代顺序无关。

- 输入参数 **NumThreads** 设置生成的代码中的 OpenMP **num\_threads()** 子句。OpenMP 还支持通过设置环境变量 **OMP\_NUM\_THREADS** 或使用 **omp\_set\_num\_threads()** 来全局限制 C/C++ 中的线程数。有关详细信息，请参阅 openMP 设定。 <https://www.openmp.org/specifications/>

## 版本历史记录

在 R2012b 中推出

## 另请参阅

### 函数

**codegen**

### 主题

“使用并行 for 循环 (parfor) 生成代码”

“使用并行 for 循环 (parfor) 的算法加速”

“Control Compilation of parfor-Loops”

“何时使用 parfor 循环”

“何时不使用 parfor 循环”

“Classification of Variables in parfor-Loops”

“Automatically Parallelize for Loops in Generated Code”

# getStdLibInfo

类: `coder.BuildConfig`

包: `coder`

获取标准库信息

## 语法

```
[linkLibPath,linkLibExt,execLibExt,libPrefix] = getStdLibInfo(bldecfg)
```

## 说明

`[linkLibPath,linkLibExt,execLibExt,libPrefix] = getStdLibInfo(bldecfg)` 返回与编译上下文 `bldecfg` 相关联的标准库信息。

## 输入参数

**bldecfg** — 代码生成期间的编译上下文

`coder.BuildConfig` 对象

代码生成期间的编译上下文，指定为 `coder.BuildConfig` 对象。使用 `coder.BuildConfig` 方法获取有关编译上下文的信息。

## 输出参数

**linkLibPath** — 特定于架构的库路径

字符向量

特定于标准 MATLAB 架构的库路径，以字符向量形式返回。该字符向量可以为空。

数据类型: `char`

**linkLibExt** — 链接时使用的库文件扩展名

`'lib' | '.dylib' | '.so' | ''`

链接时使用的特定于平台的库文件扩展名，以 `'lib'`、`'.dylib'`、`'.so'` 或 `''` 形式返回。

数据类型: `enumerated`

**execLibExt** — 在运行时使用的库文件扩展名

`'lib' | '.dylib' | '.so' | ''`

在运行时使用的特定于平台的库文件扩展名，以 `'lib'`、`'.dylib'`、`'.so'` 或 `''` 形式返回。

数据类型: `enumerated`

**libPrefix** — 特定于架构的库名称前缀

字符向量

特定于标准架构的库名称前缀，指定为字符向量。该字符向量可以为空。

数据类型: `char`

## 版本历史记录

在 R2013b 中推出

## 另请参阅

`coder.BuildConfig`

# addDesignRangeSpecification

在参数中添加设计范围设定

## 语法

**addDesignRangeSpecification(fcnName,paramName,designMin, designMax)**

## 说明

**addDesignRangeSpecification(fcnName,paramName,designMin, designMax)** 指定允许函数 **fcnName** 中的参数 **paramName** 使用的最小值和最大值。定点转换过程使用此设计范围信息来派生代码中的下游变量的范围。

## 输入参数

**fcnName** — 函数名称

字符串

函数名称，指定为字符串。

数据类型： **char**

**paramName** — 参数名称

字符串

参数名称，指定为字符串。

数据类型： **char**

**designMin** — 允许此参数使用的最小值

标量

允许此参数使用的最小值，指定为双精度标量值。

数据类型： **double**

**designMax** — 允许此参数使用的最大值

标量

允许此参数使用的最大值，指定为双精度标量值。

数据类型： **double**

## 示例

### 添加设计范围设定

```
% Set up the fixed-point configuration object
fixptcfg = coder.config('fixpt');
fixptcfg.TestBenchName = 'dti_test';
fixptcfg.addDesignRangeSpecification('dti', 'u_in', -1.0, 1.0)
fixptcfg.ComputeDerivedRanges = true;
```

```
%Set up C code configuration object
cfg = coder.config('lib');
% Derive ranges and generate fixed-point C code
codegen -config cfg -float2fixed fixptcfg dti -report
```

### 另请参阅

[coder.FixPtConfig](#) | [codegen](#) | [hasDesignRangeSpecification](#) |  
[removeDesignRangeSpecification](#) | [clearDesignRangeSpecifications](#) |  
[getDesignRangeSpecification](#)

# clearDesignRangeSpecifications

清除所有设计范围设定

## 语法

`clearDesignRangeSpecifications()`

## 说明

`clearDesignRangeSpecifications()` 清除所有设计范围设定。

## 示例

清除设计范围设定

```
% Set up the fixed-point configuration object
cfg = coder.config('fixpt');
cfg.TestBenchName = 'dti_test';
cfg.addDesignRangeSpecification('dti', 'u_in', -1.0, 1.0)
cfg.ComputeDerivedRanges = true;
% Verify that the 'dti' function parameter 'u_in' has design range
hasDesignRanges = cfg.hasDesignRangeSpecification('dti','u_in')
% Now remove the design range
cfg.clearDesignRangeSpecifications()
hasDesignRanges = cfg.hasDesignRangeSpecification('dti','u_in')
```

## 另请参阅

`coder.FixPtConfig` | `codegen` | `addDesignRangeSpecification` |  
`removeDesignRangeSpecification` | `hasDesignRangeSpecification` |  
`getDesignRangeSpecification`

# half

构造半精度数值对象

## 说明

使用 **half** 构造函数将半精度数据类型分配给数字或变量。半精度数据类型占用 16 位内存，但其浮点表示使其能够处理比相同大小的整数或定点数据类型更宽的动态范围。有关详细信息，请参阅“Floating-Point Numbers” (Fixed-Point Designer)和“[What is Half Precision?](#)” (Fixed-Point Designer)。

有关支持使用半精度输入的代码生成的函数列表，请参阅“Half Precision Code Generation Support”。

## 创建对象

### 语法

**a = half(v)**

#### 描述

**a = half(v)** 将 **v** 中的值转换为半精度。

#### 输入参数

**v** — 输入数组

标量 | 向量 | 矩阵 | 多维数组

输入数组，指定为标量、向量、矩阵或多维数组。

数据类型: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

## 对象函数

在 MATLAB 中，使用半精度输入的仿真支持这些函数。MATLAB System object 支持半精度数据类型，而 MATLAB System (Simulink) 模块支持具有实数值的半精度数据类型。有关支持使用半精度输入的代码生成的函数列表，请参阅“Half Precision Code Generation Support” (Fixed-Point Designer)。

## 数学和算术

<code>abs</code>	绝对值和复数的模
<code>acos</code>	反余弦（以弧度为单位）
<code>acosh</code>	反双曲余弦
<code>asin</code>	反正弦（以弧度为单位）
<code>asinh</code>	反双曲正弦
<code>atan</code>	反正切（以弧度为单位）
<code>atan2</code>	四象限反正切
<code>atanh</code>	反双曲正切



ceil	向正无穷舍入
conj	复共轭
conv	卷积和多项式乘法
conv2	二维卷积
cos	以弧度为单位的参数的余弦
cosh	双曲余弦
cospi	准确计算 $\cos(X*\pi)$
cumsum	累积和
dot	点积
exp	指数
expm1	针对较小的 x 值正确计算 $\exp(x)-1$
fft	快速傅里叶变换
fft2	二维快速傅里叶变换
fftn	N 维快速傅里叶变换
fftshift	将零频分量移到频谱中心
fix	向零舍入
floor	向负无穷舍入
fma	Multiply and add using fused multiply add approach
hypot	平方和的平方根 (斜边)
ifft	快速傅里叶逆变换
ifft2	二维快速傅里叶逆变换
ifftn	多维快速傅里叶逆变换
ifftshift	逆零频平移
imag	复数的虚部
ldivide	数组左除
log	自然对数
log10	常用对数 (以 10 为底)
log1p	针对较小的 x 值正确计算 $\log(1+x)$
log2	以 2 为底的对数和浮点数分解
mean	数组的均值
minus	减法
mldivide	求解关于 x 的线性方程组 $Ax = B$
mod	除后的余数 (取模运算)
mrdivide	求解关于 x 的线性方程组 $xA = B$
mtimes	矩阵乘法
plus	添加数字, 追加字符串
pow10	以 10 为底的幂和缩放半精度数
pow2	浮点数的以 2 为底的幂运算和缩放
power	按元素求幂
prod	数组元素的乘积
rdivide	数组右除
real	复数的实部
rem	除后的余数
round	舍入至最近的小数或整数
rsqrt	Reciprocal square root
sign	Sign 函数 (符号函数)
sin	参数的正弦, 以弧度为单位
sinh	双曲正弦
sinpi	准确地计算 $\sin(X*\pi)$
sqrt	平方根
sum	数组元素总和
tan	以弧度表示的参数的正切

tanh	双曲正切
times	乘法
uminus	一元减法
uplus	一元加法

## 数据类型

allfinite	Determine if all array elements are finite
anynan	Determine if any array element is NaN
cast	将变量转换为不同的数据类型
cell	元胞数组
double	双精度数组
eps	浮点相对精度
flintmax	浮点格式的最大连续整数
Inf	创建所有值均为 Inf 的数组
int16	16 位有符号整数数组
int32	32 位有符号整数数组
int64	64 位有符号整数数组
int8	8 位有符号整数数组
isa	确定输入是否具有指定数据类型
isfloat	确定输入是否为浮点数据类型
isinteger	确定输入是否为整数数组
islogical	确定输入是否为逻辑数组
isnan	确定哪些数组元素为 NaN
isnumeric	确定输入是否为数值数组
isobject	确定输入是否为 MATLAB 对象
isreal	确定数组是否使用复数存储
logical	将数值转换为逻辑值
NaN	创建所有值均为 NaN 的数组
realmax	最大的正浮点数
realmin	最小标准浮点数
single	单精度数组
storedInteger	Stored integer value of fi object
typecast	在不更改基础数据的情况下转换数据类型
uint16	16 位无符号整数数组
uint32	32 位无符号整数数组
uint64	64 位无符号整数数组
uint8	8 位无符号整数数组

## 关系和逻辑运算符

all	确定所有的数组元素是为非零还是 true
and	计算逻辑 AND
Short-Circuit AND	Logical AND with short-circuiting
any	确定是否有任何数组元素非零
eq	确定相等性
ge	决定大于或等于
gt	确定大于
isequal	确定数组相等性
isequaln	测试数组相等性, 将 NaN 值视为相等
le	确定小于等于
lt	确定小于
ne	确定不相等性

not	计算逻辑 NOT
or	计算逻辑 OR
Short-Circuit OR	具有短路功能逻辑 OR

## 向量和矩阵运算

cat	串联数组。
chol	Cholesky 分解
circshift	循环平移数组
colon	向量创建、数组下标和 for 循环迭代
complex	创建复数数组
ctranspose	复共轭转置
empty	创建指定类的空数组
eye	单位矩阵
flip	翻转元素顺序
fliplr	将数组从左向右翻转
flipud	将数组从上向下翻转
horzcat	Horizontal concatenation for heterogeneous arrays
iscolumn	确定输入是否为列向量
isempty	确定数组是否为空
isfinite	确定哪些数组元素为有限
isinf	确定哪些数组元素为无限值
ismatrix	确定输入是否为矩阵
isrow	确定输入是否为行向量
isscalar	确定输入是否为标量
issorted	确定数组是否已排序
isvector	确定输入是否为向量
length	最大数组维度的长度
lu	LU 矩阵分解
max	数组的最大元素
min	数组的最小元素
ndims	数组维度数目
numel	数组元素的数目
ones	创建全部为 1 的数组
permute	置换数组维度
repelem	重复数组元素副本
repmat	重复数组副本
reshape	重构数组
size	数组大小
sort	对数组元素排序
squeeze	删除长度为 1 的维度
transpose	转置向量或矩阵
vertcat	Vertically concatenate for heterogeneous arrays
zeros	创建全零数组

## 图形

area	二维 alpha 形状的面积
bar	条形图
barh	水平条形图
fplot	绘制表达式或函数
line	创建基本线条
plot	二维线图

plot3	三维点或线图
plotmatrix	散点图矩阵
rgbplot	绘制颜色图
scatter	散点图
scatter3	三维散点图
xlim	设置或查询 x 坐标轴范围
ylim	设置或查询 y 坐标轴范围
zlim	设置或查询 z 坐标轴范围

## 深度学习

activations	计算深度学习网络层激活
classify	Classify data using trained deep learning neural network
predict	Reconstruct the inputs using trained autoencoder
predictAndUpdateState	Predict responses using a trained recurrent neural network and update the network state

要显示支持函数的列表，请在 MATLAB 命令行窗口中输入：

```
methods(half(1))
```

## 示例

### 将值转换为半精度

要将双精度数字转换为半精度，请使用 **half** 函数。

```
a = half(pi)
```

```
a =
```

```
half
```

```
3.1406
```

您也可以使用 **half** 函数将现有变量转换为半精度。

```
v = single(magic(3))
```

```
v = 3x3 single matrix
```

```
8 1 6
3 5 7
4 9 2
```

```
a = half(v)
```

```
a =
```

```
3x3 half matrix
```

```
8 1 6
```

3	5	7
4	9	2

## 限制

- 不支持结合使用半精度和逻辑类型的算术运算。
- 有关其他用法说明和限制，请参阅“Half Precision Code Generation Support”。

## 版本历史记录

在 R2018b 中推出

## 扩展功能

### C/C++ 代码生成

使用 MATLAB® Coder™ 生成 C 代码和 C++ 代码。

- 有关支持使用半精度输入的代码生成的函数列表和任何相关联的限制，请参阅“Half Precision Code Generation Support”。
- 如果您的目标硬件不支持半精度，则 **half** 将用作存储类型，算术运算以单精度执行。
- 有些函数仅将 **half** 用作存储类型，以单精度执行算术运算，不管目标硬件是什么。
- 对于深度学习代码生成，半精度输入会转换为单精度，计算以单精度执行。
- 在 MATLAB 中，**isobject** 函数以半精度输入返回 true。在生成的代码中，此函数返回 false。

### GPU 代码生成

使用 GPU Coder™ 为 NVIDIA® GPU 生成 CUDA® 代码。

- 有关支持使用半精度输入的代码生成的函数列表和任何相关联的限制，请参阅“Half Precision Code Generation Support”。
- 要生成和执行具有半精度数据类型的代码，CUDA 计算能力必须为 5.3 或更高。
- 要生成和执行具有半精度数据类型的代码，CUDA 工具包的版本必须为 10.0 或更高。
- 您必须将内存分配 (**malloc**) 模式设置为 'Discrete'，才能生成 CUDA 代码。
- GPU 代码生成不支持半精度复数数据类型。
- 如果您的目标硬件不支持半精度，则 **half** 将用作存储类型，算术运算以单精度执行。
- 有些函数仅将 **half** 用作存储类型，以单精度执行算术运算，不管目标硬件是什么。
- 对于深度学习代码生成，半精度输入会转换为单精度，计算以单精度执行。要将计算量减半，请在 **coder.DeepLearningConfig** 中将库目标设置为 'tensorrt'，并将数据类型设置为 'FP16'。
- 在 MATLAB 中，**isobject** 函数以半精度输入返回 true。在生成的代码中，此函数返回 false。

## 另请参阅

**single** | **double**

### 主题

“Half Precision Code Generation Support”  
 “Floating-Point Numbers” (Fixed-Point Designer)  
 “What is Half Precision?” (Fixed-Point Designer)

“Generate Code for Sobel Edge Detection That Uses Half-Precision Data Type”

类

---

## coder.BuildConfig 类

包: `coder`

代码生成期间的编译上下文

### 描述

代码生成器会创建一个此类的对象，以便于访问编译上下文。编译上下文封装代码生成器使用的设置，包括：

- 目标语言
- 代码生成目标
- 目标硬件
- 编译工具链

在为 `coder.ExternalDependency` 类编写的方法中使用 `coder.BuildConfig` 方法。

### 类属性

抽象 `true`

有关类属性的信息，请参阅“类属性”。

## 创建对象

代码生成器创建此类的对象。

## 方法

### 公共方法

<code>getHardwareImplementation</code>	Get handle of copy of hardware implementation object
<code>getStdLibInfo</code>	获取标准库信息
<code>getTargetLang</code>	Get target code generation language
<code>getToolchainInfo</code>	Returns handle of copy of toolchain information object
<code>isCodeGenTarget</code>	Determine if build context represents specified target
<code>isMatlabHostTarget</code>	Determine if hardware implementation object target is MATLAB host computer

## 示例

### 使用 `coder.BuildConfig` 方法访问 `coder.ExternalDependency` 方法中的编译上下文

此示例说明如何使用 `coder.BuildConfig` 方法来访问 `coder.ExternalDependency` 方法中的编译上下文。在此示例中，您使用：

- `coder.BuildConfig.isMatlabHostTarget` 来验证代码生成目标是 MATLAB 主机。如果主机不是 MATLAB，则报告错误。



- `coder.BuildConfig.getStdLibInfo` 来获取链接时和运行时库文件扩展名。使用此信息来更新编译信息。

为包含函数 `adder` 的外部库编写一个类定义文件。

```
%=====
% This class abstracts the API to an external Adder library.
% It implements static methods for updating the build information
% at compile time and build time.
%=====

classdef AdderAPI < coder.ExternalDependency
 %#codegen

 methods (Static)

 function bName = getDescriptiveName(~)
 bName = 'AdderAPI';
 end

 function tf = isSupportedContext(buildContext)
 if buildContext.isMatlabHostTarget()
 tf = true;
 else
 error('adder library not available for this target');
 end
 end

 function updateBuildInfo(buildInfo, buildContext)
 % Get file extensions for the current platform
 [~, linkLibExt, execLibExt, ~] = buildContext.getStdLibInfo();

 % Add file paths
 hdrFilePath = fullfile(pwd, 'codegen', 'dll', 'adder');
 buildInfo.addIncludePaths(hdrFilePath);

 % Link files
 linkFiles = strcat('adder', linkLibExt);
 linkPath = hdrFilePath;
 linkPriority = '';
 linkPrecompiled = true;
 linkLinkOnly = true;
 group = '';
 buildInfo.addLinkObjects(linkFiles, linkPath, ...
 linkPriority, linkPrecompiled, linkLinkOnly, group);

 % Non-build files for packaging
 nbFiles = 'adder';
 nbFiles = strcat(nbFiles, execLibExt);
 buildInfo.addNonBuildFiles(nbFiles, '');
 end

 %API for library function 'adder'
 function c = adder(a, b)
 if coder.target('MATLAB')
 % running in MATLAB, use built-in addition
 c = a + b;
 else

```

```
% Add the required include statements to the generated function code
coder.cinclude('adder.h');
coder.cinclude('adder_initialize.h');
coder.cinclude('adder_terminate.h');
c = 0;

% Because MATLAB Coder generated adder, use the
% housekeeping functions before and after calling
% adder with coder.ceval.

coder.ceval('adder_initialize');
c = coder.ceval('adder', a, b);
coder.ceval('adder_terminate');
end
end
end
end
```

## 版本历史记录

在 R2013b 中推出

### 另请参阅

[coder.target](#) | [coder.ExternalDependency](#) | [coder.HardwareImplementation](#) |  
[coder.make.ToolchainInfo](#)

### 主题

"Develop Interface for External C/C++ Code"  
"Build Process Customization"

## 对象

---



# 工具

---

