树外模块

用自己的功能和块扩展GNU Radio

目录

本文大量借鉴了原始文章（但非常过时）“如何编写代码块”？埃里克·布鲁姆（Eric Blossom）写。

什么是树外模块？

树外模块是GNU Radio组件，它不存在于GNU Radio源树中。通常，如果您想使用自己的功能和块来扩展GNU Radio，则由您创建这样的模块（即，除非计划将其提

交给开发人员，否则通常不会在实际的GNU Radio源代码树中添加内容。用于上游集成）。这使您可以自己维护代码，并在主要代码旁具有其他功能。

CGRAN托管了许多这样的模块，CGRAN是GNU Radio相关项目的存储库。如果您自己开发了一些好东西，请将其提交给CGRAN！

这样的模块的一个例子是频谱估计工具箱，它扩展了GNU Radio的频谱估计功能。安装后，您将有更多可用的块（例如，在GNU Radio伴随程序中），其行为与GNU Radio的其余部分相同。但是，开发人员是不同的人。

我可以使用的工具和资源

有一些工具，脚本和文档可以作为第三方程序或GNU Radio的一部分使用。

gr如何写一个块

这是树外模块的示例，该模块作为GNU Radio源树的一部分提供。如果您按照本文档后面的教程进行操作，最终将得到一个看起来很像gr-howto-write-a-block的模块。基本上，这对于作为开发人员的您来说是一个很好的参考，以查看您的模块是否按其应有的方式运行。

由于它是GNU Radio树的一部分，因此已对其进行了测试和维护。因此，GNU Radio的当前版本将始终带有最新的模块结构。

gr-modtool

开发模块时，涉及许多无聊的单调工作：样板代码，makefile编辑等。gr-modtool是一个脚本，旨在通过自动编辑makefile，使用模板并尽可能多地完成工作来帮助所有这些事情对于开发人员，您可以直接进入DSP编码。

请注意，gr-modtool对代码的外观进行了许多假设。自定义的模块越多，并且进行了特定的更改，gr-modtool就越有用。

它托管在CGRAN和github上。

树外创建模块

这是GNU Radio随附的脚本。请注意，它的功能是gr-modtool的子集，因此您实际上并不需要它。但是，如果您不喜欢gr-modtool并且想要手动进行所有makefile编辑等操作，则可以使用此脚本创建初始目录。

Wiki上的开发人员资源

最重要的绝对是块编码指南。虽然这是为GNU Radio主树编写的，但也应将其应用于所有模块。具体来说，看看命名约定！

如果您正在阅读本文，那么您很可能熟悉所有的GNU Radio术语，但是如果您不了解，请在核心概念教程中找到一个顶峰。这包含确定的必备知识。此外，有关编写Python应用程序的教程还介绍了许多关键功能。

CMake，make等

GNU Radio使用CMake作为构建系统。因此，构建模块需要安装cmake，以及您喜欢的任何构建管理器（大多数情况下是“ make”，但也可以使用Eclipse或MS Visual Studio）。

模块的结构

让我们直接进入gr-howto-write-a-block模块，看看它是由什么组成的：

```
gnuradio / gr-howto-write-a-block [master]%ls
应用程序cmake CMakeLists.txt docs grc include lib python swig
```

它由几个子目录组成。任何将用C ++（或C，或任何非Python语言）编写的内容都将放入lib/。对于C ++文件，我们通常将标头放入include/（如果要导出的话）或lib/（如果它们仅在编译时相关但以后不安装）的标头。

当然，Python的东西进来了python/。这包括单元测试（未安装）和已安装的Python模块的一部分。

您可能已经知道，即使GNU Radio块是用C ++编写的，也可以在Python中使用。这是通过SWIG（简化的包装程序和接口生成器）的帮助完成的，SWIG会自动创建粘合代码以使之成为可能。SWIG需要一些有关如何执行此操作的说明，这些说明位于swig/子目录中。

如果希望您的块在GNU Radio随附的GNU Radio图形UI中可用，则需要添加块的XML描述并将其放入中grc/。

对于文档，docs/包含一些有关如何从C ++文件和Python文件中提取文档的说明（为此，我们使用Doxygen和Sphinx），并确保它们在Python中可作为文档字符串使用。当然，您也可以在此处添加自定义文档。

最后，apps/子目录包含所有完整的应用程序（包括GRC和独立的可执行文件），这些应用程序与块一起安装到系统中。

有些模块包含另一个目录，examples/可用于保存（猜测内容）示例，这是文档的重要附录，因为其他开发人员可以直接查看代码以了解如何使用块。

构建系统也会带来一些麻烦：CMakeLists.txt文件（每个子目录中都有一个cmake/ 文件）和文件夹。您现在可以忽略后者，因为它主要带给CMake有关如何查找GNU Radio库的说明等。CMakeLists.txt文件需要进行大量编辑以确保模块正确构建。

但是一次只一步！现在，让我们从第一个教程开始。

教程1：创建树外模块

简单方法：gr-modtool

如果您安装了gr-modtool，请使用它。它将创建一个新目录和所有目录。如果您打算在接下来的步骤中使用gr-modtool，则强烈建议您以这种方式创建模块。

运作方式如下：

```
1 ~/tmp % gr_modtool.py create howto
2 Module directory is "./gr-howto".
3 Creating directory...
4 Copying howto example...
5 Unpacking...
6 Replacing occurences of 'howto' to 'howto'...
7 Done.
8 Use 'gr_modtool add' to add a new block to this currently empty mod
```

请注意，gr-modtool实际上使用gr-howto-write-a-block目录作为模板。安装后，它将与“ howto”相关的所有内容重命名为您所称的模块（出于创意，我们在此处也选择了“ howto”）。

困难的方式：手工

如果您现在想了解模块的所有内部工作原理，并且不想安装gr-modtool，请执行以下操作：

1. 将gr-howto-write-a-block复制到新位置（例如~/ src）
2. 重命名目录（例如，gr-howto）
3. 在顶层CMakeLists.txt文件中重命名项目名称
4. 删除* .cc，*。h，*。xml和* .grc文件
5. 从所有CMakeLists.txt文件中删除对这些文件的所有引用

确保不要错过CMakeLists.txt中的任何内容。最好将它们全部打开！

使用CMake

如果您以前从未使用过CMake，现在可以尝试一下。从命令行看，基于CMake的项目的典型工作流是：

```
$ mkdir build#我们目前在模块的顶层目录中
$ cd构建/
$ cmake ../#告诉CMake所有配置文件都在一个目录下
$ make#然后开始构建
```

现在，build/在模块目录中有一个新目录。所有的编译等都在这里完成，因此实际的源代码树上没有临时文件。如果我们更改任何CMakeLists.txt文件，则应重新运行cmake ../。

教程2：用C ++编写一个块（howto_square_ff）

对于第一个示例，我们将创建一个计算其单个float输入的平方的块。该块将接受单个float输入流并产生单个float输出流，即，对于每个传入的float项目，我们输出一个float项目，该浮动项目是该输入项目的平方。

遵循命名约定，我们将使用howto该包作为前缀，并且该块将被称为howto_square_ff，因为它具有浮点输入，浮点输出。

我们将安排该块以及我们在本文中编写的其他块，以howto Python模块结尾。这将使我们能够像这样从Python访问它：

```
1 import howto
2 sqr = howto.square_ff()
```

测试驱动编程

我们可能只是开始敲击C ++代码，但是作为高度发展的现代程序员，我们将首先编写测试代码。毕竟，我们的行为确实有一个很好的规范：将单个浮点数流作为输入，并生成单个浮点数流作为输出。输出应为输入的平方。

这有多难？原来，这很容易！查看此代码，我们将其另存为python/qa_howto.py：

```
 1 from gnuradio import gr, gr_unittest
 2 import howto_swig # Can't import howto because that module does no
 3
 4 class qa_howto (gr_unittest.TestCase):
 5
 6     def setUp (self):
 7         self.tb = gr.top_block ()
 8
 9     def tearDown (self):
10         self.tb = None
11
12     def test_001_square_ff (self):
13         src_data = (-3, 4, -5.5, 2, 3)
14         expected_result = (9, 16, 30.25, 4, 9)
15         src = gr.vector_source_f (src_data)
16         sqr = howto_swig.square_ff ()
17         dst = gr.vector_sink_f ()
18         self.tb.connect (src, sqr)
19         self.tb.connect (sqr, dst)
20         self.tb.run ()
21         result_data = dst.data ()
22         self.assertFloatTuplesAlmostEqual (expected_result, result
23
24 if __name__ == '__main__':
25     gr_unittest.main ()
```

gr_unittest是标准Python模块unittest的扩展。gr_unittest增加了对检查浮点数和复数元组的近似相等性的支持。Unittest使用Python的反射机制来查找所有以test_开头的方法并运行它们。Unittest用对setUp和tearDown的匹配调用包装对test_ *的每个调用。有关详细信息，请参见Python unittest文档。

当我们运行测试时，gr_unittest.main将调用setUp，test_001_square_ff和tearDown。

test_001_square_ff构建一个包含三个节点的小图。
gr.vector_source_f(src_data)将获取的元素，src_data然后说完成了。
howto.square_ff是我们正在测试的块。gr.vector_sink_f收集的输出
howto.square_ff。

该run()方法运行图形，直到所有块都指示它们完成为止。最后，我们检查
square_ffon 的执行结果是否src_data符合我们的期望。

请注意，通常在安装模块之前调用此测试。这意味着我们需要一些技巧才能在测试时加载块。CMake通过适当地更改PYTHONPATH来处理大多数事情。同样，我们导入howto_swig而不是howto在此文件中。

为了使CMake真正知道此测试的存在，我们在处添加了一行
python/CMakeLists.txt，因此看起来像这样：

```
########################################################### ######################
＃处理单元测试
########################################################### ######################
包括（GrTest）

设置（GR_TEST_TARGET_DEPS gnuradio-howto）
设置（GR_TEST_PYTHON_DIRS $ {CMAKE_BINARY_DIR} / swig）
GR_ADD_TEST（qa_howto $ {PYTHON_EXECUTABLE} $ {CMAKE_CURRENT_SOURCE_D
```

构建树与安装树

运行cmake时，通常会在单独的目录（例如build/）中运行它。这是构建树。安装树的路径为$ prefix / lib / pythonversion / site-packages，其中$ prefix是您在/usr/local/使用-DCMAKE_INSTALL_PREFIX开关进行配置（通常为）期间为CMake指定的值。

在编译期间，库被复制到构建树中。仅在安装期间，文件才会安装到安装树中，从而使我们的块可用于GNU Radio应用程序。

我们编写应用程序，以便它们访问安装树中的代码和库。另一方面，我们希望我们的测试代码在构建树上运行，在安装树中我们可以检测到问题。

make test

我们用于make test运行测试（build/ 在调用cmake之后从子目录运行此测试）。这将调用一个Shell脚本，该脚本设置PYTHONPATH环境变量，以便我们的测试使用我们的代码和库的生成树版本。然后，它将运行所有名称均为qa _ *。py格式的文件，并报告总体成功或失败。

要使用未安装的代码版本，需要执行许多幕后操作（请查看cmake/目录以获取便宜的刺激）。

当然，我们现在不能调用它-没有要测试的东西。

C ++代码（第1部分）

现在我们有了一个测试用例，让我们编写C ++代码。所有信号处理块均源自gr_block其子类或其中之一。现在就去查看Doxygen生成的手册上的块文档！

快速浏览文档可以发现，由于它general_work()是纯虚拟的，因此我们绝对需要重写它。general_work() 是执行实际信号处理的方法。对于我们的平方示例，我们将需要重写它，并提供构造函数和析构函数以及一些其他东西，以利用boost shared_ptrs的优势。

现在，我们需要编写一个头文件，一个.cc文件，然后编辑
include/CMakeLists.txt和lib/CMakeLists.txt。和以前一样，我们可以使用gr-modtool完成所有操作：

```
tmp / gr-howto％gr_modtool.py添加-t general square_ff
在目录中操作。
确定的GNU Radio模块名称：howto
代码类型：常规
```

```
块/代码标识符：square_ff
完整的块/代码标识符为：howto_square_ff
输入有效的参数列表，包括默认参数：
添加Python质量检查代码？[是/否] n
添加C ++质量检查代码？[是/否] n
遍历lib ...
正在添加文件“ howto_square_ff.h” ...
正在添加文件“ howto_square_ff.cc” ...
横行...
编辑swig / howto_swig.i ...
遍历python ...
正在编辑python / CMakeLists.txt ...
横移式起重机
正在添加文件“ howto_square_ff.xml” ...
正在编辑grc / CMakeLists.txt ...
```

如您所见，gr-modtool的作用甚至更多：它创建GRC绑定（尽管XML无效，但稍后会更多），并对SWIG定义有所作用。但最重要的是，它创建了我们想要的头文件和cc文件。现在，我们可以使用我们最喜欢的编辑器对其进行编辑。（还请注意，由于我们已经python/在上一节中放置了qa * .py文件，因此我们跳过了它的自动生成。）

当然，我们也可以手动执行此操作：从另一个项目复制.h和.cc文件，使用search / replace相应地重命名所有内容，然后编辑CMakeLists.txt。

最后，我们实际上编辑代码以进行平方。让我们直接进入结果。首先，我们从howto_square_ff.h放入的头文件（）开始include/。立即打开：

来源：gr-howto-write-a-block / include / howto_square_ff.h

C ++文件被调用howto_square_ff.cc并驻留在中lib/。也打开它：

来源：gr-howto-write-a-block / lib / howto_square_ff.cc

这里有一些注意事项：

- 头文件仅包含标准定义等，并且对于所有类型的块看起来都非常相似。实际上，对于这个简单的示例，如果您使用gr-modtool，则头文件根本不需要任何手动编辑（文档除外）
- io签名（在.cc文件的构造函数定义中）指定我们有一个输入端口，该端口接受'float'类型的项目。输出是相同的。
- 所有工作均在general_work()功能中完成。有一个分别指向输入和输出缓冲区的指针，以及一个将输入缓冲区的平方复制到输出缓冲区的for循环。
- 该general_work()方法的最后两个语句告诉GNU Radio从输入缓冲区读取了多少项目（即已消耗），以及将多少项目写入了输出缓冲区（return语句）。
- 如果您使用gr-modtool，则将对howto_square_ff.h的引用添加到其中，include/CMakeLists.txt并将howto_square_ff.cc的引用添加到了lib/CMakeLists.txt。也请查看这些文件，以了解构建系统的工作方式。
- 另外，对howto_square_ff.h的引用已添加到文件中swig/howto_swig.i。因为此块非常简单，所以将SWIG指向头文件并告诉它“创建看起来像C ++类的Python对象”就足够了。由于GNU Radio附带了一些其自身的SWIG魔术，因此在大多数情况下，它可以正常工作。

很简单，不是吗？

更多的C ++代码（但更好）-通用模式的子类

gr_block在输入流的消耗和输出流的生产方面提供了极大的灵活性。巧妙地使用forecast()和consume()（请参阅下文）允许构建可变速率块。可以构造在每个输入上以不同速率消耗数据的块，并以与输入数据内容有关的速率生成输出。

另一方面，信号处理块在输入速率和输出速率之间具有固定的关系是非常普遍的。许多人是1：1，而其他人则是1：N或N：1关系。您必须general_work()在上一个块的功能中想到了同样的事情：如果消耗的项目数量与生产的项目数量相同，为什么我必须告诉GNU Radio两次完全相同的数量？

另一个常见要求是需要检查多个输入样本以生成单个输出样本。这与输入和输出速率之间的关系正交。例如，非抽取，非内插FIR滤波器需要为其产生的每个输出采样检查N个输入采样，其中N是滤波器中的抽头数。但是，它仅消耗单个输入样本即可产生单个输出。我们称这个概念为“历史”，但您也可以将其视为“先行”。

- gr_sync_block

gr_sync_block源自gr_block，并实现了具有可选历史记录的1：1块。假设我们知道输入到输出的速率，则可以进行某些简化。从实现者的角度来看，主要的变化是我们定义了一个工作方法而不是general_work()。work()呼叫顺序略有不同；它忽略了不必要的ninput_items参数，并安排consume_each()以我们的名义被调用。

```
1   /*!
2    * \brief Just like gr_block::general_work, only this arranges t
3    *  call consume_each for you.
4    *
5    * The user must override work to define the signal processing c
6    */
7   virtual int work (int noutput_items,
8                     gr_vector_const_void_star &input_items,
9                     gr_vector_void_star &output_items) = 0;
```

这使我们无需担心的事情也更少，无需编写代码。如果块要求的历史记录大于1，请set_history()在构造函数中调用，或在条件更改时随时调用。

gr_sync_block提供了处理历史记录要求的预测版本。

- gr_sync_decimator

gr_sync_decimator从gr_sync_blockN：1块派生并实现该块，带有可选历史记录。

- gr_sync_interpolator

gr_sync_interpolator派生自gr_sync_block并实现具有可选历史记录的1：N块。

有了这些知识也应该清楚，howto_square_ff应该是一个gr_sync_block没有历史。

因此，让我们编写另一个块，该块的作用与之前相同，但它是一个同步块。gr-modtool的另一个调用是我们的朋友：

```
tmp / gr-howto%gr_modtool.py添加-t sync square2_ff
在目录中操作。
确定的GNU Radio模块名称：howto
代码类型：同步
块/代码标识符：square2_ff
完整的块/代码标识符为：howto_square2_ff
输入有效的参数列表，包括默认参数：
添加Python质量检查代码？[是/否] n
添加C ++质量检查代码？[是/否] n
遍历lib ...
正在添加文件“ howto_square2_ff.h” ...
正在添加文件“ howto_square2_ff.cc” ...
横行...
编辑swig / howto_swig.i ...
横移式起重机
正在添加文件“ howto_square2_ff.xml” ...
正在编辑grc / CMakeLists.txt ...
```

同样，我们将跳过质量检查文件的生成，因为我们将仅使用另一个文件。
实际上，测试是完全相同的。这是qa_howto.py两个块的文件：

来源：gr-howto-write-a-block / python / qa_howto.py

make test现在运行将产生一个测试运行，qa_howto.py该测试不应失败。

内部work()功能

如果您使用的是同步块（包括抽取器和插值器），这就是gr_modtool生成的框架代码的样子：

```
 1 int
 2 my_block_name::work (int noutput_items,
 3       gr_vector_const_void_star &input_items,
 4       gr_vector_void_star &output_items)
 5 {
 6     const float *in = (const float *) input_items[0];
 7     float *out = (float *) output_items[0];
 8
 9     // Do <+signal processing+>
10
11     // Tell runtime system how many output items we produced.
12     return noutput_items;
13 }
```

那么，给定历史记录，向量，多个输入端口等，这真的是您所需要的吗？是的！由于同步块具有固定的输出输入速率，因此您只需要知道输出项的数量即可计算出可用的输入项数。

示例-加法器块：source：gnuradio-core / src / lib / gengen / gr_add_XX.cc.t

该块具有未知数量的输入和可变的矢量长度。可以通过input_items.size()和检查连接的端口数output_items.size()。for遍历所有可用项目的外循环上升到noutput_items*d_vlen。输出项目的数量与输入项目的数量相同，因为它是一个同步块，并且您可以信任GNU Radio提供此数量的项目。在这种情况下，一项是样本的向量，但是我们要添加各个样本，因此for循环会考虑这一点。

示例-gr_unpack_k_bits_bb：来源：gnuradio-core / src / lib / general / gr_unpack_k_bits_bb.cc

这是一个将字节分开并产生各个位的块。同样，在编译时也不知道每个字节有多少位。但是，每个输入项有固定数量的输出项，因此我们可以简单地除以noutput_items/d_k得到正确数量的输入项。这将永远是正确的，因为GNU Radio知道输入输出比率，并且将确保该比率noutput_items始终是该整数比率的倍数。

示例- 源代码中的历史记录：gr-digital / lib / digital_diff_phasor_cc.cc

如果使用长度为k的历史记录，则GNU Radio将保留输入缓冲区的k-1个条目，而不是丢弃它们。这意味着，如果GNU Radio告诉您输入缓冲区有N个项目，则实际上它有N + k-1个您可以使用的项目。

考虑上面的例子。我们需要上一个项目，因此历史记录设置为k = 2。如果仔细检查for循环，您会发现在noutput_items项目之外，noutput_items+1实际上是在读取项目。这是可能的，因为历史记录的输入缓冲区中还有一个额外的项目。

消费noutput_items完商品后，最后一项不会被丢弃，可用于下一次调用work()。

救命！我的测试失败了！

恭喜你！如果您的测试失败，则您的质量检查代码已经付清了钱。显然，您想先修复所有问题，然后再继续。

You can use the command ctest -V (instead of make test, again, all in your build/ subdirectory) to get all the output from the tests. You can also use ctest -V -R REGEX to only run tests that match REGEX, if you have many tests and want to narrow it down. If you can't figure out the problem from the output of your QA code, put in print statements and show intermediary results. If you need more info on debugging blocks, check out the debugging tutorial.

Making your blocks available in GRC

You can now install your module, but it will not be available in GRC. That's because gr-modtool can't create valid XML files before you've even written a block. The XML code generated when you call gr_modtool add is just some skeleton code.

Once you've finished writing the block, gr_modtool has a function to help you create the XML code for you. For the howto example, you can invoke it on the square2_ff block by calling

```
$ gr_modtool.py makexml square2_ff
```

In most cases, gr_modtool can't figure out all the parameters by itself and you will have to edit the appropriate XML file by hand. The GRC wiki site has a description available.

For the blocks written in tutorial 2, the valid XML files look like this:

source:gr-howto-write-a-block/grc/howto_square_ff.xml
source:gr-howto-write-a-block/grc/howto_square2_ff.xml

There's more: additional gr_block-methods

If you've read the gr_block documentation (which you should have), you'll have noticed there are a great number of methods available to configure your block.

Here's some of the more important ones:

set_history()

If you're block needs a history (e.g. something like an FIR filter), call this in the constructor. GNU Radio then makes sure you have the given number of 'old' items available.

forecast()

Looking at general_work() you may have wondered how the system knows how much data it needs to ensure is valid in each of the input arrays. The forecast() method provides this information.

The default implementation of forecast() says there is a 1:1 relationship between noutput_items and the requirements for each input stream. The size of the items is defined by gr_io_signatures in the constructor of gr_block. The sizes of the input and output items can of course differ; this still qualifies as a 1:1 relationship.

```
 1
 2   // default implementation:  1:1
 3   void
 4   gr_block::forecast (int noutput_items,
 5                       gr_vector_int &ninput_items_required)
 6   {
 7     unsigned ninputs = ninput_items_required.size ();
 8     for (unsigned i = 0; i < ninputs; i++)
 9       ninput_items_required[i] = noutput_items;
10   }
```

Although the 1:1 implementation worked for howto_square_ff, it wouldn't be appropriate for interpolators, decimators, or blocks with a more complicated relationship between noutput_items and the input requirements. That said, by deriving your classes from gr_sync_block, gr_sync_interpolator or gr_sync_decimator instead of gr_block, you can often avoid implementing forecast.

Note that if you've already got a history set, you usually don't need to set this.

set_output_multiple()

在执行general_work()例程时，偶尔需要运行时系统来确保仅要求您生成一些特定值的倍数的输出项。如果您的算法自然适用于固定大小的数据块，则可能会发生这种情况。在构造函数中调用set_output_multiple以指定此要求。默认输出倍数为1。

完成工作并安装

首先，通过以下清单：

- 您是否编写了一个或多个模块，包括质量检查代码？
- 是否make test通过了吗？
- 是否有可用的GRC绑定（如果您要这样做）？

在这种情况下，您可以继续安装模块。在Linux机器上，这意味着要返回到build目录并调用make install：

```
$ cd构建/
$ sudo make install
```

使用Ubuntu，通常还需要调用ldconfig：

```
$ sudo ldconfig
```

否则，您将收到一条错误消息，提示找不到刚刚安装的库。

其他类型的块

源和汇

源和汇源于gr_sync_block.唯一不同的是，源无输入，汇无输出。这反映在传递给gr_sync_block构造函数的gr_io_signatures中。看一看“ gr_file_source。{h，cc}”：source：gnuradio-core / src / lib / io / gr_file_source.cc和gr_file_sink。{h，cc}，了解一些非常简单的示例。另请参见有关编写Python应用程序的教程。

层次块

For the concept of hierarchical blocks, see this. Of course, they can also be written in C++. gr-modtool supports skeleton code for hierarchical blocks both in Python and C++.

```
tmp/gr-howto % gr_modtool.py add -t hiercpp hierblockcpp_ff
Operating in directory .
GNU Radio module name identified: howto
Code is of type: hiercpp
Block/code identifier: hierblockcpp_ff
Full block/code identifier is: howto_hierblockcpp_ff
Enter valid argument list, including default arguments:
Add Python QA code? [Y/n] n
Add C++ QA code? [Y/n] n
Traversing lib...
Adding file 'howto_hierblockcpp_ff.h'...
Adding file 'howto_hierblockcpp_ff.cc'...
Traversing swig...
Editing swig/howto_swig.i...
Traversing grc...
Adding file 'howto_hierblockcpp_ff.xml'...
Editing grc/CMakeLists.txt...
tmp/gr-howto % gr_modtool.py add -t hierpython hierblockpy_ff
Operating in directory .
GNU Radio module name identified: howto
Code is of type: hierpython
```

```
Block/code identifier: hierblockpy_ff
Full block/code identifier is: howto_hierblockpy_ff
Enter valid argument list, including default arguments:
Add Python QA code? [Y/n] n
Traversing python...
Adding file 'hierblockpy_ff.py'...
Traversing grc...
Adding file 'howto_hierblockpy_ff.xml'...
Editing grc/CMakeLists.txt...
```

Everything at one glance: Cheat sheet for editing modules/components:

Here's a quick list for all the steps necessary to build blocks and out-of-tree modules:

1. Create (do this once per module): `gr_modtool create MODULENAME`
2. Add a block to the module: `gr_modtool add BLOCKNAME`
3. Create a build directory: `mkdir build/`
4. 调用制作过程：（`cd build && cmake .. && make`请注意，只有在更改CMake文件时，才需要调用cmake）
5. 调用测试：`make test`或`ctest`或`ctest -V`更多详细信息
6. 安装（仅在一切正常且没有测试失败时）： `sudo make install`
7. Ubuntu用户：重新加载库： `sudo ldconfig`
8. 从源代码树中删除块： `gr_modtool rm REGEX`
9. 通过从CMake文件中删除它们来禁用块： `gr_modtool disable REGEX`

用Python编写信号处理块

这位于单独的页面上。

调试块

调试GNU Radio可作为单独的教程获得。

注：树木外模块（译者：出处，翻译整理适当参考！）