Out-of-tree 模块

用 gnuradio 自带的功能扩展自定义模块

什么是 Out-of-tree 模块

Out-of-tree 模块是不存在于 gnuradio 源代码树中的 gnuradio 组成部分，通常如果你想自己扩展 gnuradio 的功能和模块，Out-of-tree 模块就是你需要创造的这种模块（通常我们不会向实际的 gnuradio 源代码树里面加东西，除非你是想把它上传给开发者们整合使用）。这样可以维护你的代码，并且延续主代码的功能。

工具和资源都可以自己管理

作为一对捆绑式的工具，文件和脚本都可以作为第三方的程序或者 gnuradio 自己的一部分。

gr_modtool 模块编辑的利器

开发一个模块的时候涉及很多单调和枯燥的工作：样板代码，makefile 文件的编辑等，gr_modtool 作为一个脚本，旨在帮助这些所有的事情自动编辑生成文件，尽可能的为开发者做跟多的工作，这样你就可以直接开始 DSP 编码的工作。

需要注意的是，gr_modtool 在你所看到的代码上做了许多的假设，越是你自己定制的，或者有特定变化的，gr_modtool 使用的就越少，但是它可能是启动新的模块的最好的地方。

Cmake，make 等等。

在 gnuradio 中使用 cmake 来作为系统的构建。不管你是否喜欢这个系统构建，你要构建模块就需要安装 cmake（最常见的是 make，但是也可以使用 Eclipse 或者 MS Visual Studio）。

教程 1 创建 Out-of-tree 模块

在下面的的教程中我们将使用名叫 **howto** 的模块。第一步是创建这个模块。利用 gr_modtool 是非常简单的，无论你想要什么样的新模块目录，只需要在命令行输入命令（这是在 gnuradio 源代码树之外的），然后继续。

```
% gr_modtool newmod howto  #####howto 就是你要创建的目录名
Creating out-of-tree module in ./gr-howto... Done.
Use 'gr_modtool add' to add a new block to this currently empty module.
```

如果一切顺利，你将会有意个名叫 gr-howto 的目录。

一个模块的结构

我们可以看看 gr-howto 这个目录是由一下部分组成的。

```
gr-howto % ls
apps  cmake  CMakeLists.txt  docs  examples  grc  include  lib  python  swig
```

他由多个子目录组成，凡是用 C++或者 C（不是 python 写的程序）都将放在 lib 文件中。对于 C++文件我们通常只有头文件放在 include 文件夹中（如果它们要出口），或者放在 lib 文件夹中（如果它们只有在编译时相关，在安装之后无关如_impl.h 文件，你会在接下来的教程中看到里面有些什么）。

当然 python 写的东西将进入 python 文件夹下，这包括未按装的测试单元和已安装的 python 模块。

你可能已经知道虽然 gnuradio 的模块是用 C++写的但是它可以在 python 之中。这是通过 SWIG 的帮助，这是一个简化包装和接口生成器，它会自动创建链接代码来实现这一点。SWIG 需要一些指令来完成这些事情，这些指令在 swig 的子目录中。除非你为你的模块做一些更好的事情，这样你

将不许要去 swig 的子目录。gr_modtool 将会为我们处理所有的这一切。

如果你想让你的模块在 gnuradio companion 中也是可用的，你需要在 GRC 文件夹中添加 XML 描述文件。

docs 文件夹中包含一些说明如何从 C++文件和 python 文件中提取文件的说明（我们使用 Doxygen 和 Sphinx 来完成），并确保它们可以在 python 代码中有效。当然，你可以添加自定义的文件在这里。

在 apps 的子目录中包含一些完整的安装到系统的应用程序（无论是 GRC 还是可单独执行的文件）。

这个构建系统还带来了一些其他的独立的包：cmakelist.txt 文件（其中一个是存在于每一个子目录）和 cmake 的文件夹。你可以无视后者，因为它主要是用于 cmake 如何找到 gnuradio 的库等的一些说明。为了确保你的模块构建正确，cmakelist.txt 这个文件需要更改很多。

教程 2　用 C++写一个模块（square_ff）

对于我们第一个例子，我们创建一个模块用于对浮点输入信号的平方。此模块将接受一个浮点输入流并产生一个浮点输出流。也就是说对于每个输入的浮点信号，输出的浮点信号是输入当然平方。

下面的命名规则，命名这个模块为 square_ff，应为它已经是浮点输入和浮点输出型。

我们要修改这个模块，把我们写好的其他东西放到里面，最终放到 howto python 中。这样就可以想这样用 python 来访问它。

```
import howto

sqr = howto.square_ff()
```

创建文件

第一步是为模块创建一个空文件并且编辑 cmakelist.txt 文件。gr_modtool 会帮我们完成工作。在命令行转到 gr-howto 文件夹中，输入：

```
gr-howto % gr_modtool add -t general square_ff
```

#######模块里代码文件名称为 square_ff

```
GNU Radio module name identified: howto
Language: C++
Block/code identifier: square_ff
Enter valid argument list, including default arguments:
Add Python QA code? [Y/n]
Add C++ QA code? [y/N]
Adding file 'square_ff_impl.h'...
Adding file 'square_ff_impl.cc'...
Adding file 'square_ff.h'...
Editing swig/howto_swig.i...
Adding file 'qa_square_ff.py'...
```

```
Editing python/CMakeLists.txt...
Adding file 'howto_square_ff.xml'...
Editing grc/CMakeLists.txt...
```

在命令行中我们指定要添加一个模块，他的类型是 general（但是我们也不知道模块的类型是什么）并且它叫做 square_ff。然后 gr_modtool 会问你是否需要添加 C++和 python 的检测文件。

现在，你可以看看 cmakelist.txt 的不同和 gr_modtool 做了些什么，你还可以看到添加了许多新文件，如果你想要你的模块能够工作，现在必须来编辑这些文件。

测试驱动程序

我们可以直接运行刚才写的 C++代码，但是作为一个高度发展的程序我们必须要先编写测试代码，毕竟我们有一个良好的习惯。取单个的浮点流作为输入，在取单个的浮点流作为输出。输出的必须是输入的平方。

这很简单，我们打开 python 文件夹下的 qa_square_ff.py 我们将编写这个文件如下：

```
from gnuradio import gr, gr_unittest

from gnuradio import blocks
import howto_swig as howto

class qa_square_ff (gr_unittest.TestCase):

    def setUp (self):
        self.tb = gr.top_block ()

    def tearDown (self):
        self.tb = None

    def test_001_square_ff(self):
        src_data = (-3, 4, -5.5, 2, 3)
        expected_result = (9, 16, 30.25, 4, 9)
        src = blocks.vector_source_f(src_data)
        sqr = howto.square_ff()
        dst = blocks.vector_sink_f()
        self.tb.connect(src, sqr)
        self.tb.connect(sqr, dst)
        self.tb.run()
        result_data = dst.data()
        self.assertFloatTuplesAlmostEqual(expected_result, result_data, 6)

if __name__ == '__main__':
    gr_unittest.run(qa_square_ff, "qa_square_ff.xml")
```

gr_unittest 是一个扩展的标准 Python 模块单元测试。gr_unittest 增加了对检查浮点和复数的元组的近似相等的支持。unittest 使用 Python 的反射机制来发现所有运行的方法和运行它们。unittest 包装每次调用 test_*来匹配建立和拆除的调用。当我们运行测试，在这种秩序中 gr_unittest.main 要调用 SETUP，test_001_square_ff，和 tearDown。test_001_square_ff 构建了一个小图，包含三个节点。blocks.vector_source_f（src_data）是源于 src_data 的元素，然后告诉它的完成。howto.square_ff 是我们正在测试的模块。blocks.vector_sink_f 集中 howto.square_ff 的输出。在 run（）方法下运行图，直到所有的模块表示他们完成工作。最后，我们检查的 src_data 执行 square_ff 的结果与我们所期望的一致。

注意，这样的测试通常称为在安装模块之前的测试。这意味着我们需要一些类似的代码，以能够加载模块时测试。cmake 可以适当的改变 PYTHONPATH。此外我们在这个文件中 import 的是 howto_swig 来代替 howto。

为了让 CMake 的实际知道这个测试存在，gr_modtool 修改 python 文件夹的 CMakeLists.txt 如下:

```
########################################################################
######

# Handle the unit tests
########################################################################
include(GrTest)

set(GR_TEST_TARGET_DEPS gnuradio-howto)
set(GR_TEST_PYTHON_DIRS ${CMAKE_BINARY_DIR}/swig)
GR_ADD_TEST(qa_square_ff ${PYTHON_EXECUTABLE} $
{CMAKE_CURRENT_SOURCE_DIR}/qa_square_ff.py)
```

建立树与树安装

当您运行 cmake 的时候，你通常运行在一个单独的目录（如 build/），这就是构建树。安装的路径是$prefix/lib/$pythonversion/dist-packages，$prefix 是无论你在哪里（通常是 /usr/local/）进行 cmake 的配置和-DCMAKE_INSTALL_PREFIX 的开关。

在编译过程中，该库将被复制到 build 目录，只有在安装过程中，文件安装到安装目录，从而使 gnuradio 能够应用我们提供的模块。我们写的应用程序，使它们在访问安装目录中的代码和库。在另一方面，我们希望我们的测试代码构建树，在那里我们可以在安装之前发现问题运行。

C++代码（第一部分）

现在，我们已经有了一个测试案例，让我们写的 C++代码。所有的信号处理块从 gr::block 或它的一个子类派生。gr_modtool 已经为我们提供了三个文件定义块:
lib/square_ff_impl.h,

lib/square_ff_impl.cc

include/howto/square_ff.h.

所有我们需要做的是修改它们，供我驱策。

首先，我们来看看我们的头文件。因为我们正在编写的块就是这么简单，所以我们没有必要真正改变他们（在 include 文件夹加入头文件往往是相当完整的运行 gr_modtool 后，除非我们需要添加一些公共方法，如赋值方法，即 getter 和 setter）。这使得我们可以用 lib/square_ff_impl.cc。

gr_modtool 的暗示是你必须通过增加<+>符号改变的代码。

让我们通过这一次:

square_ff_impl::square_ff_impl()

```
      : gr::block("square_ff",
                  gr::io_signature::make(1, 1, sizeof (float)), // input
signature
                  gr::io_signature::make(1, 1, sizeof (float))) // output
signature
    {
        // empty constructor
    }
```

该构造函数本身是空的，因为平方块没有必要设立什么。

唯一有兴趣的部分是输入和输出签名的定义：在输入端，我们有 1 个端口，允许浮点的投入。输出端口是相同的。

void

```
    square_ff_impl::forecast (int noutput_items, gr_vector_int
&ninput_items_required)
    {
      ninput_items_required[0] = noutput_items;
    }
```

forecast()是一个函数，它告诉调度有多少投入项目需要从 noutput_items 输出。在这种情况下，它们是相同的。

index 0 表示这是用于第一端口，但是我们只有一个路径。这通常是在很多模块指向 forecast 的情况。举例来说，你可以看看 gr::block, gr::sync_block, gr::sync_decimator,和 gr::sync_interpolator 是怎么来定义默认的预测功能记载的事情，比如率变动和 history。

最后，还有 general_work（），这是纯虚的 gr::block，所以我们一定要重写。 general_work（）是不实际的信号处理方法：

```
    int

    square_ff_impl::general_work (int noutput_items,
                                  gr_vector_int &ninput_items,
                                  gr_vector_const_void_star &input_items,
                                  gr_vector_void_star &output_items)
    {
      const float *in = (const float *) input_items[0];
      float *out = (float *) output_items[0];

      for(int i = 0; i < noutput_items; i++) {
        out[i] = in[i] * in[i];
      }

      // Tell runtime system how many input items we consumed on
      // each input stream.
      consume_each (noutput_items);

      // Tell runtime system how many output items we produced.
      return noutput_items;
    }
```

for 循环可复制输入缓冲区到输出缓冲区的平方，分别有一个指针到输入和一个指针到输出缓冲区。

使用 Cmake

如果您以前从未使用 CMake 的，这是很好的时间来进行尝试。一个在命令行中看到的 CMake 为基础的项目的典型的工作流程是这样的:

```
$ mkdir build         # 我们目前在模块的顶层目录

$ cd build/

$ cmake ../           # 告诉 CMake，它的所有配置文件为一目录了

$ make                # 并开始建设（应该在前一节之后工作）
```

现在我们有一个新的目录 build/在我们的模块的目录。所有的编译等在这里完成，所以实际的源代码树不是充斥着临时文件。如果我们改变任何的 CMakeLists.txt 文件，我们应该重新运行 cmake 的.. /（不过说实话，cmake 的自动检测这些变化和重播，当你下一次运行 make）。

我们来试一下 - 运行 make test

因为我们的 C++代码之前写过代码质量保证 QA，马上就可以看到，如果我们所做的事情是正确的。

我们用 make test 运行我们的测试（从 build/子目录中运行此，调用 cmake 和 make）。这将调用一个 shell 脚本设置 PYTHONPATH 环境变量，以便我们的测试使用构建树版本的代码和库。然后运行其中的形式是 qa_*.py 名称的所有文件。报告的整体成功或失败。

在后面有很多操作来使用我们没有安装的代码版本（看 cmake 的/目录中一个简单的动作。）

如果你完成了 square_ff 模块，这应该能正常运行:

```
gr-howto/build % make test

Running tests...
Test project /home/braun/tmp/gr-howto/build
    Start 1: test_howto
1/2 Test #1: test_howto .......................   Passed    0.01 sec
    Start 2: qa_square_ff
2/2 Test #2: qa_square_ff .....................   Passed    0.38 sec

100% tests passed, 0 tests failed out of 2

Total Test time (real) =   0.39 sec
```

如果事情在测试期间出现故障，我们可以挖得更深一些。当我们运行 make test，我们实际上调用 CMake 的程序 ctest，其中有许多选项，我们可以传递给它的更详细信息。说我们忘了乘 in[i] * in[i]和这样实际上不是信号整形。如果我们只是运行 make test，甚至只是 CTEST，我们会得到这样的:

```
gr-howto/build $ ctest
Test project /home/braun/tmp/gr-howto/build
```

```
Start 1: test_howto
1/2 Test #1: test_howto ..................... Passed 0.02 sec
Start 2: qa_square_ff
2/2 Test #2: qa_square_ff .....................***Failed 0.21 sec
50% tests passed, 1 tests failed out of 2
Total Test time (real) = 0.23 sec
The following tests FAILED:
2 - qa_square_ff (Failed)
Errors while running Ctest
```

要了解我们qa_square_ff测试发生了什么事，我们运行cmake -V -R square。'-V'标志为我们提供了详细的输出和“-R”标志是一个正则表达式只运行那些匹配的测试。

```
gr-howto/build  $ ctest -V -R square
UpdateCTestConfiguration  from :/home/braun/tmp/gr-
howto/build/DartConfiguration.tcl
UpdateCTestConfiguration  from :/home/bruan/tmp/gr-
howto/build/DartConfiguration.tcl
Test project /home/braun/tmp/gr-howto/build
Constructing a list of tests
Done constructing a list of tests
Checking test dependency graph...
Checking test dependency graph end
test 2
    Start 2: qa_square_ff

2: Test command: /bin/sh "/home/bruan/tmp/gr-
howto/build/python/qa_square_ff_test.sh"
2: Test timeout computed to be: 9.99988e+06
2: F
2: ======================================================================
2: FAIL: test_001_t (__main__.qa_square_ff)
2: ----------------------------------------------------------------------
2: Traceback (most recent call last):
2:   File "/home/braun/tmp/gr-howto/python/qa_square_ff.py", line 44, in
test_001_t
2:     self.assertFloatTuplesAlmostEqual(expected_result, result_data, 6)
2:   File "/opt/gr/lib/python2.7/dist-packages/gnuradio/gr_unittest.py", line
90, in assertFloatTuplesAlmostEqual
2:     self.assertAlmostEqual (a[i], b[i], places, msg)
2: AssertionError: 9 != -3.0 within 6 places
2:
2: ----------------------------------------------------------------------
2: Ran 1 test in 0.002s
2:
2: FAILED (failures=1)
1/1 Test #2: qa_square_ff .....................***Failed    0.21 sec

0% tests passed, 1 tests failed out of 1

Total Test time (real) =   0.21 sec

The following tests FAILED:
      2 - qa_square_ff (Failed)
Errors while running Ctest
```

这就告诉我们，“9！=-3.0”因为我们预期的输出是（-3）^2=9，但真正得到的-3输入。我们可以利用这些信息来回去和修复我们的块，直到测试通过。

更多的C + +代码（只有更好） - 子类的通用模式

gr::block 允许极大的灵活性，对于输入流的消费和生产输出流。熟练使用 forecast()和 consume()（见下文）来建立变量模块。这是一个可构造的模块，用来消耗每个不同输入数据率和产生输出数据率的数据，这是输入数据的一个功能.另一方面，这是很常见的信号处理块具有输入速率和输出速率之间的固定关系.很多都是 1:1，而另一些则 1: N 或 N: 1 的关系。你一定以为同样的事情在以前的模块的 general_work()功能中: 如果消耗的项目数是相同生产的项目数，为什么我要告诉 GNU Radio 的完全相同数量的两倍？

另一个常见的需求是需要检查多个输入样本，以产生一个单一的输出样本。这是正交的输入和输出速率之间的关系。例如，一个非抽取，非内插 FIR 滤波器需要检查 N 个输入样本为它产生的，其中 N 是抽头的滤波器的数目的每个输出样本。然而，它仅消耗单个输入采样以产生单一的输出。我们称这个概念“history”，但你也可以把它看成“look-ahead”。

1 gr::sync_block

gr::sync_block 是从 gr::block 派生并实现了一个 1:1 的可选的“history”。既然我们知道了输入到输出的速度，某些简化是可能的。从实现者的观点看，主要的变化是，我们定义了一个 work（）方法，而不是 general_work（）。work（）具有一个稍微不同的调用顺序；它省略了不必要的 ninput_items 参数，并安排 consume_each（）代表我们被调用。

让我们来添加源于 gr::sync_block 的模块 square2_ff 并调用它。首先，我们编辑 qa_square_ff.py 增加另一个测试:

```
    def test_002_square2_ff(self):

        src_data = (-3, 4, -5.5, 2, 3)
        expected_result = (9, 16, 30.25, 4, 9)
        src = blocks.vector_source_f(src_data)
        sqr = howto.square2_ff()
        dst = blocks.vector_sink_f()
        self.tb.connect(src, sqr, dst)
        self.tb.run()
        result_data = dst.data()
        self.assertFloatTuplesAlmostEqual(expected_result, result_data, 6)
```

你可以看到它是完全一样的测试和以前一样，只是用 square2_ff 的。

然后，我们使用 gr_modtool 添加模块文件，跳过 QA 代码（因为我们已经有一个）:

```
gr-howto % gr_modtool add -t sync square2_ff

GNU Radio module name identified: howto
Language: C++
Block/code identifier: square2_ff
Enter valid argument list, including default arguments:
Add Python QA code? [Y/n] n
Add C++ QA code? [Y/n] n
Adding file 'square2_ff_impl.h'...
Adding file 'square2_ff_impl.cc'...
Adding file 'square2_ff.h'...
Editing swig/howto_swig.i...
Adding file 'howto_square2_ff.xml'...
Editing grc/CMakeLists.txt...
```

在 square2_ff_impl.cc 构造函数中完成与前面相同的方法，除了父类被变为 gr::sync_block.

```
    square2_ff_impl::square2_ff_impl()

      : gr::sync_block("square2_ff",
                       gr::io_signature::make(1, 1, sizeof (float)),
                       gr::io_signature::make(1, 1, sizeof (float)))
    {}

// [...] skip some lines ...

    int
    square2_ff_impl::work(int noutput_items,
                          gr_vector_const_void_star &input_items,
                          gr_vector_void_star &output_items)
    {
      const float *in = (const float *) input_items[0];
      float *out = (float *) output_items[0];

      for(int i = 0; i < noutput_items; i++) {
        out[i] = in[i] * in[i];
      }

      // Tell runtime system how many output items we produced.
      return noutput_items;
    }
```

work 功能是真正的区别（同时，我们没有一个 forecast()功能了）。我们来看看它的更详细的下一节。

这给了我们一些东西让我们少出错少写代码，如果模块 history 需要大于 1，请调用 set_history（）构造函数或任何时间的要求的变化。

GR:: sync_block 提供了一个 forecast 用来处理的 history 要求。

2 gr::sync_decimator

GRGR:: sync_decimator 从 GR:: sync_block 派生并实现了一个 N: 1 块可选的 history。

3 gr::sync_interpolator

GR:: sync_interpolator 从 GR:: sync_block 派生并实现了 1: N 挡可选的 history。

有了这些知识，应该明确的是，howto_square_ff 应该是一个没有 history 的 gr::sync_block

现在，返回到我们的 build 目录，并运行 make。因为 gr_modtool 添加 square2_ff 块所需的 CMakeLists.txt 文件，cmake 的自动重新运行，为我们接着是 make。

再次，运行 make test 将产生与 qa_square_ff.py 一样不会失败的测试运行。

work()功能的内部信息

如果您使用的是同步块（包括抽取和内插器），这是怎样由 gr_modtool 产生的代码的框架:

```
  int
```

```
  my_block_name::work(int noutput_items,
                      gr_vector_const_void_star &input_items,
                      gr_vector_void_star &output_items)
  {
    const float *in = (const float *) input_items[0];
    float *out = (float *) output_items[0];

    // Do <+signal processing+>

    // Tell runtime system how many output items we produced.
    return noutput_items;
  }
```

因此，鉴于history，vectors（向量），multiple input ports（多个输入端口）等，然而这是真的你所需要的？是的！因为同步模块有一个固定的输出输入速率，所有你需要知道的是输出项的数量，你可以计算多个输入项如何使用。

为 GRC 提供你制作的模块

您现在可以安装你的模块，但它不会提供 GRC。这是因为 gr_modtool 无法在你写一个模块之前创建有效的 XML 文件。当你调用 gr_modtool 加载生成的 XML 代码只是一些框架代码。一旦你写完模块，gr_modtool 有一个功能，以帮助您建立您的 XML 代码。对于 HOWTO 例子，你可以通过调用 square2_ff 块调用它。

```
gr-howto % gr_modtool makexml square2_ff

GNU Radio module name identified: howto
Warning: This is an experimental feature. Don't expect any magic.
Searching for matching files in lib/:
Making GRC bindings for lib/square2_ff_impl.cc...
Overwrite existing GRC file? [y/N] y
```

需要注意的是 gr_modtool 附加创建一个无效的 GRC 文件，这样我们就可以覆盖。

在大多数情况下，gr_modtool 不能找出所有的参数本身，你将不得不手工修改相应的 XML 文件。

在这种情况下，由于该块是如此简单，在 XML 实际上是有效的。看看 grc/howto_square2_ff.xml:

```
<block>

  <name>Square2 ff</name>
  <key>howto_square2_ff</key>
  <category>HOWTO</category>
  <import>import howto</import>
  <make>howto.square2_ff()</make>
  <sink>
    <name>in</name>
    <type>float</type>
  </sink>
  <source>
    <name>out</name>
    <type>float</type>
  </source>
</block>
```

也许你想把自动生成的名称更改为更好的东西。

如果你这样做从 build 目录用 make install，您可以在 GRC 使用块。如果 GRC 已经在运行，你可以点击在 GRC 工具栏上的“刷新块”按钮;它是在右手边一个蓝色圆形箭头。您现在应该看到在模块树“HOWTO”。

还有更多：更多的 gr::block 方法

你会注意到有大量方法可用来配置你的区块。

下面是一些比较重要的:

set_history():

如果你的块需要的 history（即像 FIR 滤波器），调用此构造函数。GNU Radio 确保你有一个给定数量的'old'的项目可用。

你可以有最小的 history 是 1，也就是说，对于每一个输出的项目，则需要 1 个输入项。如果你选择一个较大的值，N，这意味着你的输出项目是从当前输入项，并从 N-1 上一个输入项计算。

调度器采用这种方法照顾你。如果设置了 history 长度为 N，在输入缓冲区中的前 N 个项目包括 N-1 以前的（即使你已经消耗掉他们）。

forecast()

该系统需要知道有多少数据是必需的，以确保有效性在每个输入数组。如前所述，forecast()方法提供了这个信息，因此你你写一个 gr::block 衍生物（同步块，这是隐式的）必须重写它。

默认实现的功能块 forecast()表明有一个 1: 1 的关系在 noutput_items 中并且要求了每个输入流。这些项目的大小是由 gr::block 的构造函数中 gr::io_signature::make 定义。输入和输出项目的大小，当然可以有所不同，这仍然有资格作为一个 1:1 的关系。当然，如果你有这种关系，你不会想用 gr::block！

```
  // default implementation:  1:1

  void
  gr::block::forecast(int noutput_items,
                      gr_vector_int &ninput_items_required)
  {
    unsigned ninputs = ninput_items_required.size ();
    for(unsigned i = 0; i < ninputs; i++)
      ninput_items_required[i] = noutput_items;
  }
```

虽然在 square_ff 实施 1:1 的工作，但它不会是适当的内插计数器，或者是 noutput_items 和输入的要求之间的更为复杂的模块。这说明使用从 gr::sync_block，gr::sync_interpolator 或者 gr::sync_decimator 中派生的类来代替 gr::block，你可以经常避免实施预测。

set_output_multiple()

当实现你的 general_work()例程，它会偶尔方便于运行时系统确保你只要求生产一批是某些特定的值的多个输出项。这算法可能自然地适用于一个固定大小数据的模块的发生。调用 set_output_multiple 构造函数来指定这个要求。默认的输出多为 1。

完成你的工作和安装

首先，查看这个清单：

你写了一个或多个数据块，包括 QA 码吗？

确实让测试通过？
有可用的 GRC 绑定（如果你想）吗？

在这种情况下，您可以继续安装模块。在 Linux 计算机上，这将意味着回到 build 目录，并调用 make install：

```
$ cd build/

$ make install # or sudo make install
```

使用 ubuntu 你可能还需要调用 ldconfig：

```
$ sudo ldconfig   （注：ubuntu 系统加上这句命令，更新库）
```

否则，你会得到无法找到你刚刚安装的库的错误消息。

其他类型的块

Sources and sinks

Sources 和 sinks 都是来自于 gr::sync_block。他们唯一不同的是，Sources 没有输入端而 sinks 没有输出端。这可以从 gr::sync_block 构造函数传递的 gr::io_signature::make 中看出。来看一看"file_source.{h,cc}":source:gr-blocks/lib/file_source_impl.cc 和 file_sink_impl.{h,cc} 是一些很直接的例子。

Hierarchical blocks（分层模块）

对于层次模块的概念，看这一点。当然它们也可一用 C++来写。gr_modtool 为 C++和 python 的分层模块支持框架代码。

```
~/gr-howto % gr_modtool.py add -t hier hierblockcpp_ff

GNU Radio module name identified: howto
Language: C++
Block/code identifier: hierblockcpp_ff
Enter valid argument list, including default arguments:
Add Python QA code? [Y/n]
Add C++ QA code? [y/N]
Adding file 'hierblockcpp_ff_impl.h'...
Adding file 'hierblockcpp_ff_impl.cc'...
Adding file 'hierblockcpp_ff.h'...
Editing swig/howto_swig.i...
Adding file 'howto_hierblockcpp_ff.xml'...
Editing grc/CMakeLists.txt...
```

使用-l python 来用 python 创建这样的模块。

教程 3：在 Python 编写的信号处理模块

注意：用 Python 编写信号处理模块伴随着性能损失。用 Python 编写模块的最常见的原因是因为你想快速作出模块，而无需使用 C++。

从前面的教程中，您已经了解了模块以及它们如何工作。让我们加快进程，另一平方的代码是用纯 python 写的，我们叫它 square3_ff()。

添加测试案例

因此，首先，我们通过编辑 qa_square_ff.py 添加另一个测试用例。离开了另外两个测试案例，QA 现在看起来像这样：

```
from gnuradio import gr, gr_unittest

from gnuradio import blocks
import howto_swig
from square3_ff import square3_ff

class qa_square_ff (gr_unittest.TestCase):

    def setUp (self):
        self.tb = gr.top_block ()

    def tearDown (self):
        self.tb = None

    # [...] Skipped the other test cases

    def test_003_square3_ff (self):
        src_data = (-3, 4, -5.5, 2, 3)
        expected_result = (9, 16, 30.25, 4, 9)
        src = blocks.vector_source_f (src_data)
        sqr = square3_ff ()
        dst = blocks.vector_sink_f ()
        self.tb.connect (src, sqr)
        self.tb.connect (sqr, dst)
        self.tb.run ()
        result_data = dst.data ()
        self.assertFloatTuplesAlmostEqual (expected_result, result_data, 6)

if __name__ == '__main__':
    gr_unittest.main ()
```

实际测试案例长相酷似以前的一样，只是更换 square3_ff ( ) 模块的定义。其他唯一的区别是在 import 语句：我们现正导入一个名为 square3_ff 模块，从中我们拿出新模块。

增加模块代码

把测试单元放到该放的的地方，我们使用 gr_modtool 来增加称为 square3_ff.py 的文件到 Python/目录下：

```
gr-howto % gr_modtool add -t sync -l python square3_ff

GNU Radio module name identified: howto
```

```
Language: Python
Block/code identifier: square3_ff
Enter valid argument list, including default arguments:
Add Python QA code? [Y/n] n
Adding file 'square3_ff.py'...
Adding file 'howto_square3_ff.xml'...
Editing grc/CMakeLists.txt...
```

记得不要为我们现有的模块添加任何 QA 文件。接下来，编辑新文件 python/square3_ff.py。它应该看起来有点像这样：

```
import numpy

from gnuradio import gr

class square3_ff(gr.sync_block):
    " Squaring block "
    def __init__(self):
        gr.sync_block.__init__(
            self,
            name = "square3_ff",
            in_sig = [numpy.float32], # Input signature: 1 float at a time
            out_sig = [numpy.float32], # Output signature: 1 float at a time
        )

    def work(self, input_items, output_items):
        output_items[0][:] = input_items[0] * input_items[0] # Only works
because numpy.array
        return len(output_items[0])
```

一些马上要注意的事项：

1　模块的类是从 gr.sync_block 中派生，就像 C++的版本是从 gr:: sync_block 衍生。

2　它构造了一个名称和输入/输出签名被设置和 work()函数

其他类型的 Python 块

就像 C ++的变种，有四种类型中的 Python 模块：

gr.sync_block

gr.decim_block

gr.interp_block

gr.basic_block - gr::block 的 python 版本

像他们的 C++版本中一样，你可以重写 forecast(), work(), 和 general_work()方法。所不同的是，参数列表的工作职能始终是如前面的例子：

```
    def work(self, input_items, output_items):
```

```
    # Do stuff

    def general_work(self, input_items, output_items):
    # Do stuff
```

通过 len(input_items[PORT_NUM])中得到的输入/输出中的项目数。

小结：

3.7 跟 3.6 比只是在源码包里没有 gr-howto-write-a-block，得先自己生成，大致方法如下：

```
$ gr_modtool newmod howto     #生成模块
$ cd gr-howto
$ gr_modtool add -t general square_ff      #添加代码
$ mkdir build/
$ cd build && cmake .. && make
$ make test
$ sudo make install
$ sudo ldconfig    (注: ubuntu 系统加上这句命令，更新库)
```

如果想在 gnuradio-companion 中使用自己的模块，还必须在 grc 目录下生成 xml 文件。在终端输入

gr-modtool.py makexml square_ff

打开 xml 文件并参照例程进行修改。

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

最后回到 gr-howto 目录，输入 sudo make install 即可安装。输入 gnuradio-companion 即可看到你自己生成的模块。至此全部过程结束。

gr-modtool 这个脚本为我们编写自己的数字信息处理模块提供了很大的方便，如果想清楚的知道每一部分的作用，可能还要一点点的手动安装

关于 GNU Radio 里面涉及的一些东西

include: c++头文件
lib: c++源码
python: python 语言文件，这个就是模块导入要用到的，我们在 import **时,实际上就是先访问这个 py 文件，然后通过它去访问所涉及的.so 文件（共享库文件）

swig: 关于接口黏合的，实际上称之为粘合工具的，我们的 python 脚本里面用到最多的就是 self.connect(a,b)，a 和 b 实际上是共享库文 件，C++直接编译生成的共享库文件，使用 python 的 self.connect(a,b)是连不上的，swig 的作用就是让 python“认 识”C++的共享库文件。模块里面的.i 后缀的是 swig 能识别的文件。

GNU Radio 里面所有以 qa 开头的文件，称之为质量保证文件，实际上就是测试文件，是让我们自己编写测试用例的，比如说，我写了实现输入数据的平方运算，测 试用例里面的 input 为（1,2,3,4,5），那我们的 expected_result（预期结果）就应该是（1,4,9,16,25），模块写完进 行测试的时候，只有当 input 之后实际运行的结果和 expected_result 一致时，测试方才通过。

doc 文件夹里面的东西，用到的工具实际上是 doxygen 工具，它的作用是根据你写程序的注释情况生成相关的程序说明文档，这个文件夹可以不要，删掉也没有关系。

grc 文件夹，里面的东西实际上就是同步生成图形界面有关的模块，也就是 gnuradio-companion 可以调用的图形模块，主要使用 xml 语言进行标 注，我们要做的不是去修改 xml 语言，而是文件里面的的模块名 howto_square 修改为我们自己写的模块的模块名就 OK 了。