

GNU Radio 源码手册

1、GNURadio 的几个例子

1.1 Dial Tone

使用两种模块

`gr::analog::sig_source_f`, 生成 350Hz,440Hz350Hz,440Hz。

`gr::audio::sink`, 将输出连接到语音系统, 以采样率生成输出信号。`audio sink` 可以设置两个输出。

`sig_source_f (freq = 350) -->`

`audio.sink`

`sig_source_f (freq = 440) -->`

`dial_tone.py` 文件:

```
from gnuradio import gr
from gnuradio import audio
from gnuradio.eng_option import eng_option
from optparse import OptionParser
```

try:

```
    from gnuradio import analog
```

except ImportError:

```
    sys.stderr.write("Error: Program requires gr-analog.\n")
```

```
    sys.exit(1)
```

```
class my_top_block(gr.top_block):
```

```
    def __init__(self):
```

```
        gr.top_block.__init__(self)
```

```
        parser = OptionParser(option_class=eng_option)
```

```
        parser.add_option("-O", "--audio-output", type="string", default="",
```

```
                        help="pcm output device name.  E.g., hw:0,0 or
```

```
/dev/dsp")
```

```
        parser.add_option("-r", "--sample-rate", type="eng_float", default=48000,
```

```
                        help="set sample rate to RATE (48000)")
```

```
        (options, args) = parser.parse_args()
```

```
        if len(args) != 0:
```

```
            parser.print_help()
```

```
            raise SystemExit, 1
```

```

sample_rate = int(options.sample_rate)
ampl = 0.1

src0 = analog.sig_source_f(sample_rate, analog.GR_SIN_WAVE, 350, ampl)
src1 = analog.sig_source_f(sample_rate, analog.GR_SIN_WAVE, 440, ampl)
dst = audio.sink(sample_rate, options.audio_output)
self.connect(src0, (dst, 0))
self.connect(src1, (dst, 1))

if __name__ == '__main__':
    try:
        my_top_block().run()
    except KeyboardInterrupt:
        pass

```

2、Flowchart

2.1 操作流程图

GNURadio 基本的结构就是 flowchart。flowchart 是有向无环图，有一个或者多个 source(输入采样点)，一个或者多个 sink(输出采样点或者终止)。

每个程序必须至少创建一个'top_block'作为 flowchart 的顶层结构。这个结构提供了很多全局的方法，'start,' 'stop,' and 'wait'。

GNU Radio 的应用创建 gr_top_block 来实例化 blocks，连接模块，然后开始 gr_top_block。下面给出一个 FIR 滤波器的例子。

```

from gnuradio import gr, blocks, filter, analog
class my_topblock(gr.top_block):
    def __init__(self):
        gr.top_block.__init__(self)
        amp = 1
        taps = filter.firdes.low_pass(1, 1, 0.1, 0.01)
        self.src = analog.noise_source_c(analog.GR_GAUSSIAN, amp)
        self.flt = filter.fir_filter_ccf(1, taps)
        self.snk = blocks.null_sink(gr.sizeof_gr_complex)
        self.connect(self.src, self.flt, self.snk)

if __name__ == "__main__":
    tb = my_topblock()
    tb.start()
    tb.wait()

```

'tb.start()'开始了数据流流过 flowchart，'tb.wait()'等价于知道 gr_top_block 结束后等待线程'join'。 可以利用 'run' 方法来替换这两个方法的先后调用。

2.2 延迟和吞吐量

GNU Radio 运行一个调度器来优化吞吐量。动态调度器使得成块的数据通过 blocks 从 source 流到 sink。数据块的大小和信号处理的速度有关。对于每个 block，能够处理的数据量和输出 buffer 的空间和输入 buffer 中已经收到的数据量有关。

这样操作的结果就是，一个模块可能申请了很多数据来处理(规模可能到几千个采样点)。从速度的角度来看，这样使得大部分的处理时间都用在了处理数据使得系统更有效率。申请小的数据块就意味着会向调度器多次申请。这样做的副作用就是当 block 处理大量数据的时候会出现延迟。

为了解决这个问题，gr_top_block 可以限制 block 可以收到的数据量，也就是上一个 block 的需要输出的数据量。一个 block 只能得到比这个数量少的采样点输入，所以相当于一个 block 的最大延迟。通过限制每次调用申请的数据量，我们相当于增加了调度器的负担，所以降低了全局效率。

可以按照如下方式限制输出数据量：

```
tb.start(1000)
```

```
tb.wait()
```

```
# or
```

```
tb.run(1000)
```

使用这个方法，我们设置了一个全局的 item 数量限制。每个 block 可以通过 'set_max_noutput_items(m)'，重写这个限制。

```
tb.flc.set_max_noutput_items(2000)
```

```
tb.run(1000)
```

在一些情况下，可能想要限制输出缓存的大小。这个能够防止要输出的数据量过大超过了上限，而使得新的输出延迟。你可以为每个 block 的每个输出口设置输出延迟。

```
tb.blk0.set_max_output_buffer(2000)
```

```
tb.blk1.set_max_output_buffer(1, 2000)
```

```
tb.start()
```

```
print tb.blk1.max_output_buffer(0)
```

```
print tb.blk1.max_output_buffer(1)
```

上面的接口 blk0 所有端口被设置成缓存为 2000 个 items，而 blk1 只有端口 1 被设置了，其余为默认值。

注意：

- 在运行的开始，缓存的大小就被配置好了。
- 一旦 flowgraph 开始，缓存长度对于一个 block 的值是不能被更改的，即使是 lock()/unlock()。如果要改变缓存大小，必须删除 block 再重新建立。
- 这可能影响到吞吐量。
- 真实的缓存大小实际上是依赖于最小系统粒度。理论上就是一个页的大小，通常是 4096bytes。这就意味着，由指令设置的缓存大小最终会四舍五入到最接近的系统粒度上。

2.3 动态配置流程图

在通信系统运行的时候，经常需要根据输入信号改变系统的状态，这时候需要更新流图。更新意味着改变结构，不独立的参数设置。例如，gr::blocks::add_const_cc 中改变加的常量大小可以由调用 'set_k(k)' 完成。

更新流图有三步：

- 锁定，停止运行，处理数据
- 更新
- 解锁

下面的例子展示了一个流图，首先加入两个 `gr::analog::noise_source_c`，然后由 `gr::blocks::sub_cc` 替代 `gr::blocks::add_cc`。

```
from gnuradio import gr, analog, blocks
import time
class mytb(gr.top_block):
    def __init__(self):
        gr.top_block.__init__(self)
        self.src0 = analog.noise_source_c(analog.GR_GAUSSIAN, 1)
        self.src1 = analog.noise_source_c(analog.GR_GAUSSIAN, 1)
        self.add = blocks.add_cc()
        self.sub = blocks.sub_cc()
        self.head = blocks.head(gr.sizeof_gr_complex, 1000000)
        self.snk = blocks.file_sink(gr.sizeof_gr_complex, "output.32fc")
        self.connect(self.src0, (self.add, 0))
        self.connect(self.src1, (self.add, 1))
        self.connect(self.add, self.head)
        self.connect(self.head, self.snk)

def main():
    tb = mytb()
    tb.start()
    time.sleep(0.01)
    # Stop flowgraph and disconnect the add block
    tb.lock()
    tb.disconnect(tb.add, tb.head)
    tb.disconnect(tb.src0, (tb.add, 0))
    tb.disconnect(tb.src1, (tb.add, 1))
    # Connect the sub block and restart
    tb.connect(tb.sub, tb.head)
    tb.connect(tb.src0, (tb.sub, 0))
    tb.connect(tb.src1, (tb.sub, 1))
    tb.unlock()
    tb.wait()

if __name__ == "__main__":
    main()
```

在更新 `flowchart` 的时候，最大输出 `items` 数量也可以被更改。一个 `block` 也可以调用 `'unset_max_noutput_items()'` 来解锁限制恢复到全局值。下面的例子扩展了上面的例子，增加了设置最大输出 `items` 数量。

```

from gnuradio import gr, analog, blocks
import time
class mytb(gr.top_block):
    def __init__(self):
        gr.top_block.__init__(self)
        self.src0 = analog.noise_source_c(analog.GR_GAUSSIAN, 1)
        self.src1 = analog.noise_source_c(analog.GR_GAUSSIAN, 1)
        self.add = blocks.add_cc()
        self.sub = blocks.sub_cc()
        self.head = blocks.head(gr.sizeof_gr_complex, 1000000)
        self.snk = blocks.file_sink(gr.sizeof_gr_complex, "output.32fc")
        self.connect(self.src0, (self.add,0))
        self.connect(self.src1, (self.add,1))
        self.connect(self.add, self.head)
        self.connect(self.head, self.snk)

def main():
    # Start the gr_top_block after setting some max noutput_items.
    tb = mytb()
    tb.src1.set_max_noutput_items(2000)
    tb.start(100)
    time.sleep(0.01)
    # Stop flowgraph and disconnect the add block
    tb.lock()
    tb.disconnect(tb.add, tb.head)
    tb.disconnect(tb.src0, (tb.add,0))
    tb.disconnect(tb.src1, (tb.add,1))
    # Connect the sub block
    tb.connect(tb.sub, tb.head)
    tb.connect(tb.src0, (tb.sub,0))
    tb.connect(tb.src1, (tb.sub,1))
    # Set new max_noutput_items for the gr_top_block
    # and unset the local value for src1
    tb.set_max_noutput_items(1000)
    tb.src1.unset_max_noutput_items()
    tb.unlock()
    tb.wait()
if __name__ == "__main__":
    main()

```

3、模块类型

3.1 概述

利用 `gnuradio` 的框架，用户可以创建多种类型的模块来实现特定的数据处理。

- Synchronous Blocks (1:1)
- Decimation Blocks (N:1)
- Interpolation Blocks (1:M)
- Basic (a.k.a. General) Blocks (N:M)

3.2 Synchronous Blocks (1:1)

同步模块一个端口的消耗和输出的点数量是一样的。零输入的同步模块叫做 source，零输出的同步模块叫做 sink。

```
#include <gr_sync_block.h>
```

```
class my_sync_block : public gr_sync_block
{
public:
    my_sync_block(...):
        gr_sync_block("my block",
                      gr_make_io_signature(1, 1, sizeof(int32_t)),
                      gr_make_io_signature(1, 1, sizeof(int32_t)))
    {
        //constructor stuff
    }

    int work(int noutput_items,
            gr_vector_const_void_star &input_items,
            gr_vector_void_star &output_items)
    {
        //work stuff...
        return noutput_items;
    }
};
```

- 11noutput_items 是输入和输出的缓冲区大小
- 输入的签名 gr_make_io_signature(0, 0, 0)使得模块为 source
- 输出的签名 gr_make_io_signature(0, 0, 0)使得模块为 sink

gr_make_io_signature(int min_streams, int max_streams, int sizeof_stream_item) 控制了接口的数量，以及接口的数据大小。下面给出了 python 的模块样例：

```
class my_sync_block(gr_sync_block):
    def __init__(self):
        gr_sync_block.__init__(self,
                                name = "my sync block",
                                in_sig = [numpy.float32, numpy.float32],
                                out_sig = [numpy.float32],
                                )
    def work(self, input_items, output_items):
        output_items[0][:] = input_items[0] + input_items[1]
        return len(output_items[0])
```

`input_items` 和 `output_items` 是包含列表的列表。`input_items` 的每个端口有一个输入采样点向量。`output_items` 也是一个向量可以将输出点存起来。`output_items[0]` 的长度等于 C++ 中 `noutput_items`。

- `in_sig=None` 的时候，模块为 `source`
- `out_sig=None` 的时候，模块为 `sink`。这时候要用 `len(input_items[0])`
- 不像 C++ 中的 `gr::io_signature` 类，python 可以直接创建指定数据类型的 `list`

3.3 Basic Block

```
#include <gr_block.h>
class my_basic_block : public gr_block
{
public:
    my_basic_adder_block(...):
        gr_block("another adder block",
                in_sig,
                out_sig)
{
    //constructor stuff
}

int general_work(int noutput_items,
                gr_vector_int &ninput_items,
                gr_vector_const_void_star &input_items,
                gr_vector_void_star &output_items)
{
    //cast buffers
    const float* in0 = reinterpret_cast(input_items[0]);
    const float* in1 = reinterpret_cast(input_items[1]);
    float* out = reinterpret_cast(output_items[0]);

    //process data
    for(size_t i = 0; i < noutput_items; i++) {
        out[i] = in0[i] + in1[i];
    }

    //consume the inputs
    this->consume(0, noutput_items); //consume port 0 input
    this->consume(1, noutput_items); //consume port 1 input
    //this->consume_each(noutput_items); //or shortcut to consume on all inputs

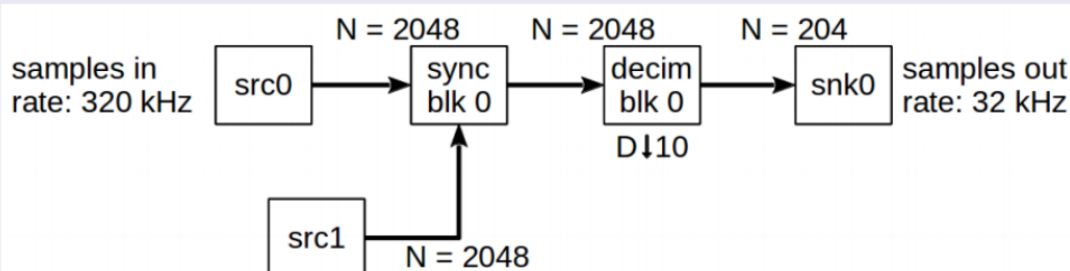
    //return produced
    return noutput_items;
}
};
```

4、GNURadio scheduler

GNURadio 的 scheduler 是数据流调度的核心。gnuradio 的官方文档和教程关于这个部分的介绍很少。为了解释清楚调度器，本文参考了 gnuradio 的其中一个作者 Tom Rondeau 的 slides: gnuradio-note，以及笔者的一些源码阅读。

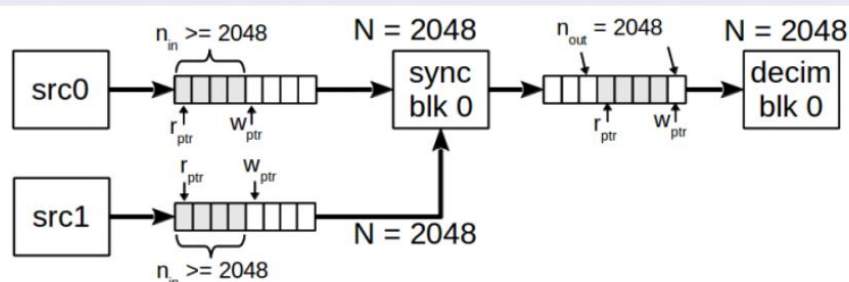
首先看一个例子，后面的讨论都会基于这个简单的例子。创建两个数据流经过一个同步模块，再经过一个十倍欠采样模块，最后输出。

Example of data moving with rate changes.



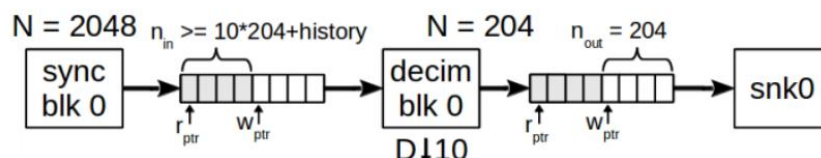
在 gnuradio 里，对于每个模块之间，调度器都会维护一个 buffer。对于一个 block 输入是 input buffer，输出是 output buffer。在 output 区，block 利用 `Wptr` 指针写数据；在 input 区，block 利用 `Rptr` 指针读取数据。

All input streams and output streams must satisfy the constraints.



对于模块 Decimator，我们需要足够的输入来计算输出。

Decimators need enough input to calculate the decimated output.



接着我们复习一下 block 的工作函数。

4.1 general_work()和 work()

`general_work()`和 `work()`是 block 工作的核心函数，数据流的操作都在这里完

成。

```
int block::general_work(int noutput_items,  
    gr_vector_int &ninput_items,  
    gr_vector_const_void_star &input_items,  
    gr_vector_void_star &output_items)
```

input_items 是一个 vector 包含一组指针指向 input buffer。output_items 是一个 vector 包含一组指针指向 output buffer。general_work()方法不指定输入输出的关系，只是指定输入和输出的数量。noutput_items 是最小的 output 数量。ninput_items 是 input buffer。

```
int block::work(int noutput_items,  
    gr_vector_const_void_star &input_items,  
    gr_vector_void_star &output_items)
```

work 函数指定了 input 和 output 的关系。通过 noutput_items 确定 ninput_items。有了这些知识，我们开始看 scheduler 的工作方式

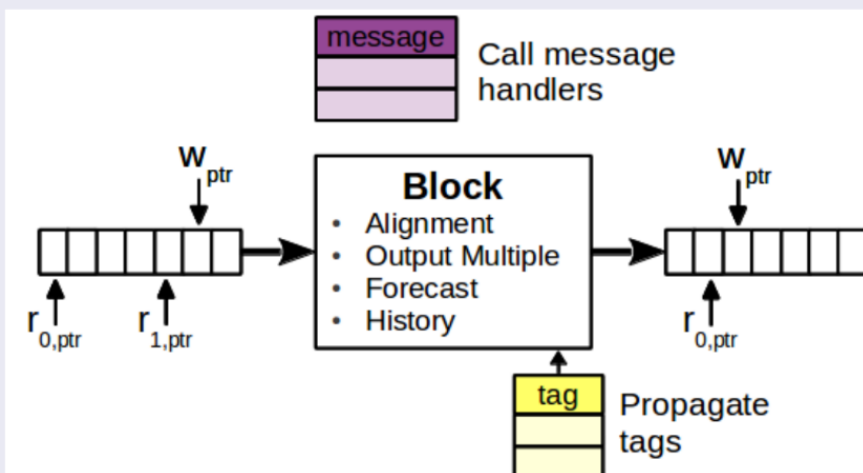
4.2 Scheduler 的基本功能

GNURadio 的调度器会处理 block 的需求也就是对于数据流和数据指针的调度，以及控制 buffer 缓冲区的大小。除此之外，buffer，messages 流和 stream tags 也会由调度器控制。Block 之间会传递三种类型的数据：采样点数据 data，消息 messages，数据标签 tags。下面我们分别看一下，对于三种类型，调度器需要作什么。

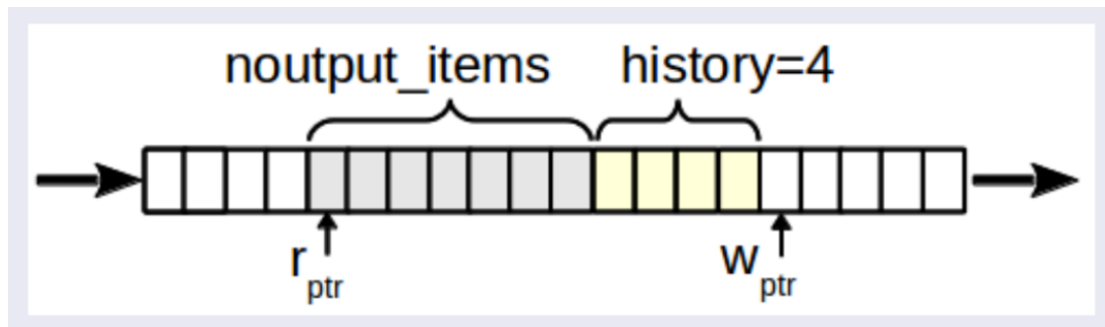
Data 调度

对于 Data, blocks 有几个需求：alignment, output multiple, forecast, history。alignment 和 output multiple 都是为了控制输出的数据量要满足一定的倍数。forecast 和 history 都是控制 buffer 的数据满足读取的需求。调度器调度数据主要就是满足 alignment, output multiple, forecast, history 的需求。

The scheduler handles the buffer states, block requirements, messages, and stream tags



- **alignment:** 将输出对齐到一定倍数，不一定保证。
- **output multiple:** 将输出对齐到一定倍数，保证实现。如不满足会等待。
- **forecast:** 利用 `ninput_items_required[i]` 告诉调度器，对于每个输出需要多少输入。
- **history:** 利用 `set_history()` 方法，高速 scheduler 进一步调整 buffer 的长度。如果我们将 history 设置为 N，那么 buffer 里的前 N 个数据中的 N-1 个数据为历史数据（即使你已经用过了）。history 保证了 buffer 里至少有 N-1 个数据。



当我们给定输出的数据数量 `noutput_items`，那么我们可以计算输入数据量 `ninput_items_required[i]`:

```
//forecast()
ninput_items_required[i]=noutput_items+history()-1; // default
ninput_items_required[i]=noutput_items*decimation()+history()-1; // Decim
ninput_items_required[i]=noutput_items/interpolation()+history()-1; // Interp
```

经过这样的 `forecast` 设置，可以保证输入满足输出的需求。

Buffer 和 latency 调度

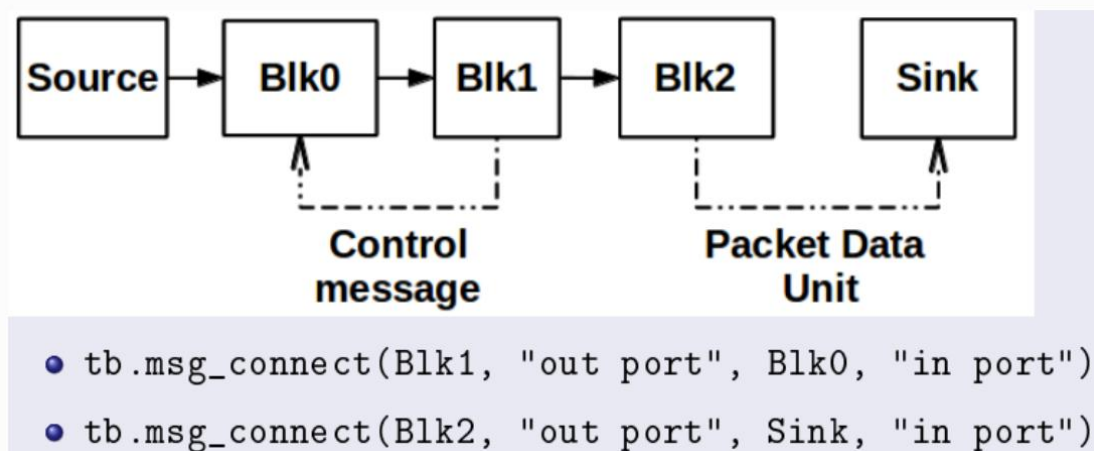
调度器也会控制缓冲区大小和延迟。又一下几个方法完成。

```
// Caps the maximum noutput_items.
// Will round down to nearest output multiple, if set.
// Does not change the size of any buffers.
set_max_noutput_items(int)
// Sets the maximum buer size for all output buers.
// Buffer calculations are based on a number of factors, this limits overall size.
// On most systems, will round to nearest page size.
set_max_output_buffer(long)
// Sets the minimum buer size for all output buers.
// On most systems, will round to nearest page size.
set_min_output_buffer(long)
```

Messages 调度

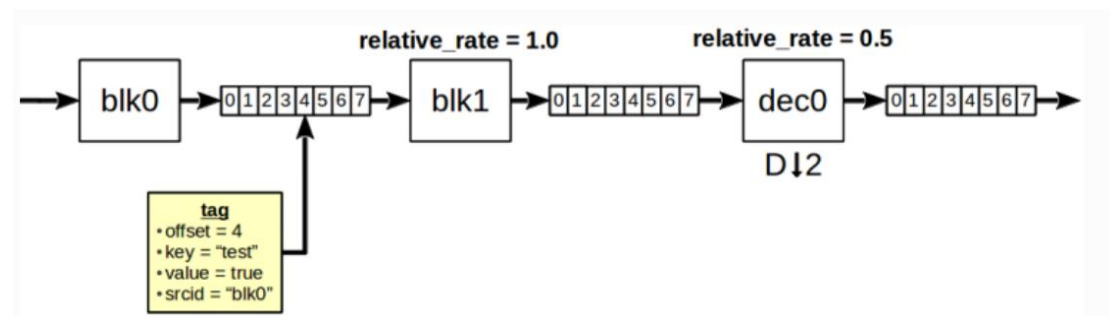
Message 可以用来传递一些控制信息，或者数据包 Packet Unit Data。每个 block 可以创建自己的 Messages queue。当 messages 传递的时候，messages 会放到 subscriber 的 queue 里。Messages 的优先级是高于 data 的，在后面的整体操作流

程中，优先处理 messages。调度器 dispatch 处理 messages 是通过调用 block 的 handler 实现的。Messages 的 queue 大小是由 max_nmsgs 控制的。



Stream Tags 调度

Stream tags 是帮助 block 标记和识别处理过的数据。对于一个指定的 samples，我们打上一些 tag。tag 会逐级传递。随着 data rate 的变化，tag 的位置会更新。tag_propagation_policy 标签的传递规则是有 block 的构造器控制的。tag 的处理是在 general_work 后面。tag_propagation_policy 有两种 TPP_ALL_TO_ALL 和 TPP_ONE_TO_ONE。第一种会把所有 Tag 都标上每一个 samples，后一种是一对一的。



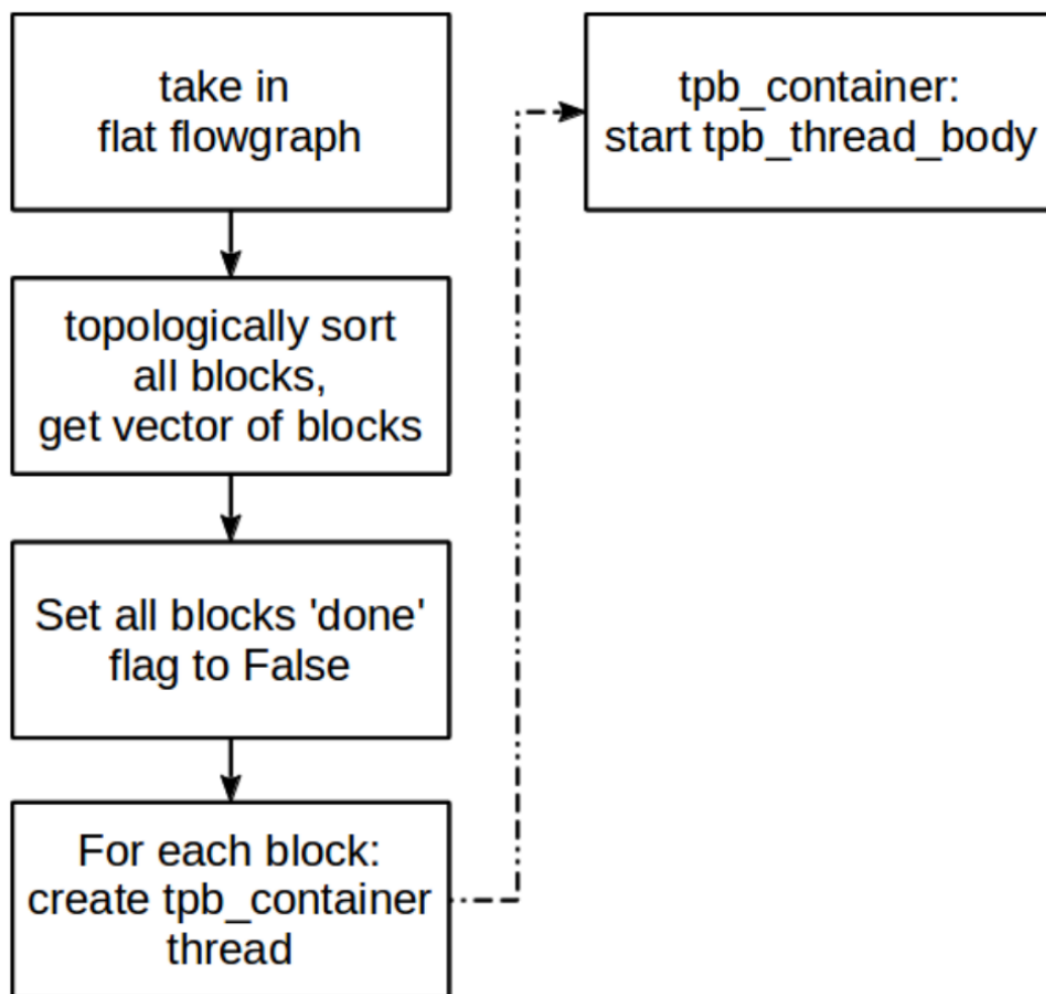
综上，调度器需要完成以下的任务：

- 计算 input 有多少可用的点
- 计算 output 有多空间
- 确定限制条件: history, alignment, forecast
- 必要的调整或者重试
- call general_work，给 block 恰当的指针和数据
- 从 general_work 的返回值更新指针

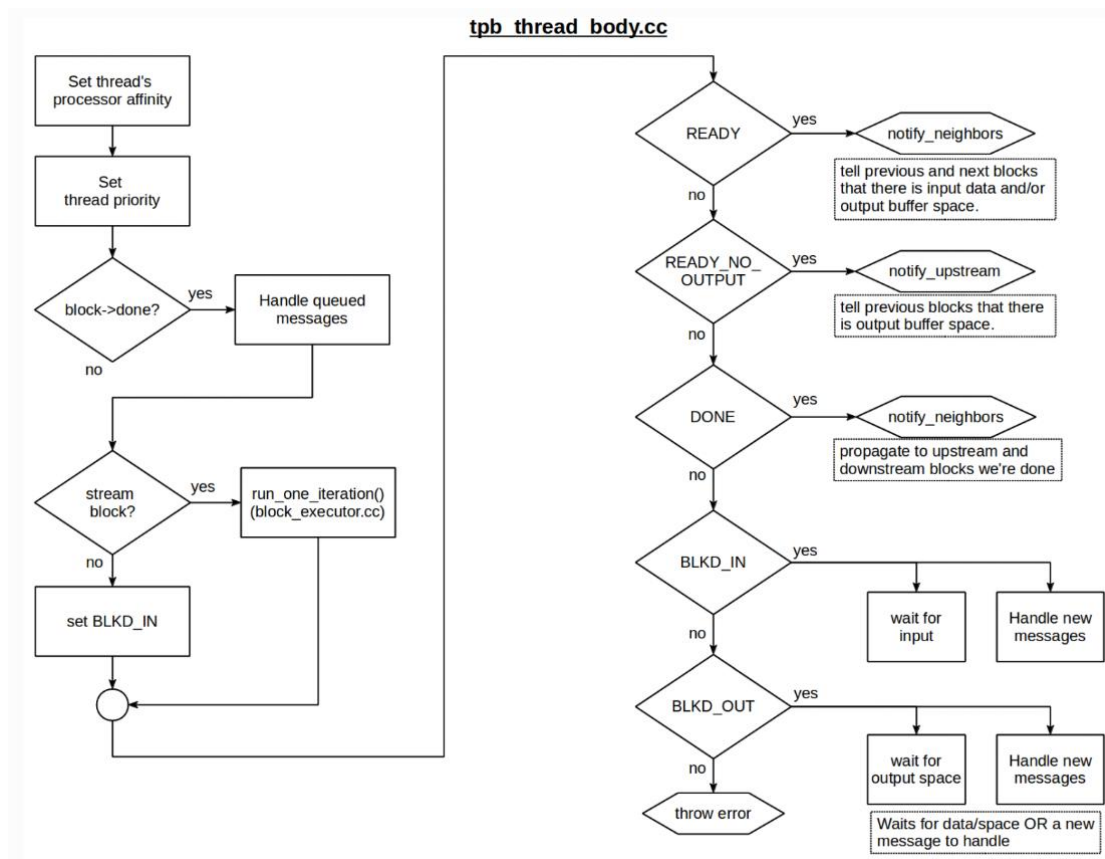
4.3 Scheduler Flow Chart

有了上面的基础，我们就做好了了解 scheduler 如何调度一个完整的 gnuradio flow chart 的准备。起初，调度器会为每个模块初始化创建一个线程。tpb_container 为 block 的线程池。

scheduler tpb.cc



`tpb_thread_body` 会控制所有线程。首先设置线程优先级。如果 block 就绪了，就可以处理传递的 `messages`。如果 `input` 的数据量不够，会将 block 设置为 `BLKD_IN`。直到数据流满足了需求，进入核心函数 `run_one_iteration()`。这个函数在 `block_executor.cc` 文件中实现。如果函数结束，`ready` 状态的时候，会通知与这个 block 相邻的其他 block。告诉他们，`input` 和 `output` 缓冲区的状态。如果是 `READY_NO_OUTPUT`，则说明没有数据输出，通知上一 block。如果 `DONE`，传递 `DONE` 的消息到其他所有 block。



5、AGC 自动增益控制

这里整理一下 GNURadio 的自动增益控制是如何实现的。自动增益控制模块是在 `analog` 大类里实现，并一共定义了三种自动增益控制: `agc`, `agc2`, `agc3`。`agc` 是最普通的自动增益控制, `agc`, `agc2` 增加了 `attack` 和 `decay` 控制。`attack` 指的是 `agc` 可以多快的响应功率迅速增加的信号, `decay` 指的是 `agc` 可以多快的响应功率迅速减小的信号。这两个时间决定了 AGC 的带宽。通常来说, 我们要求 AGC 的带宽要小于信号的最小频率, 这样才不会影响信号的解调。

```
class ANALOG_API agc_cc
```

```
{
```

```
public:
```

```
    /*!
```

```
    * Construct a complex value AGC loop implementation object.
```

```
    *
```

```
    * \param rate the update rate of the loop.
```

```
    * \param reference reference value to adjust signal power to.
```

```
    * \param gain initial gain value.
```

```
    * \param max_gain maximum gain value (0 for unlimited).
```

```
    */
```

```
    agc_cc(float rate = 1e-4,
           float reference = 1.0,
           float gain = 1.0,
           float max_gain = 0.0)
```

```

        : _rate(rate), _reference(reference), _gain(gain), _max_gain(max_gain){};

virtual ~agc_cc(){};

float rate() const { return _rate; }
float reference() const { return _reference; }
float gain() const { return _gain; }
float max_gain() const { return _max_gain; }

void set_rate(float rate) { _rate = rate; }
void set_reference(float reference) { _reference = reference; }
void set_gain(float gain) { _gain = gain; }
void set_max_gain(float max_gain) { _max_gain = max_gain; }

gr_complex scale(gr_complex input)
{
    gr_complex output = input * _gain;

    _gain += _rate * (_reference - std::sqrt(output.real() * output.real() +
                                                output.imag() *
output.imag()));
    if (_max_gain > 0.0 && _gain > _max_gain) {
        _gain = _max_gain;
    }
    return output;
}

void scaleN(gr_complex output[], const gr_complex input[], unsigned n)
{
    for (unsigned i = 0; i < n; i++) {
        output[i] = scale(input[i]);
    }
}

protected:
    float _rate;          // adjustment rate
    float _reference;     // reference value
    float _gain;          // current gain
    float _max_gain;     // max allowable gain
};

```

6、Polymorphic Types

6.1 介绍

Polymorphic Types(多态)是一种高级的数据类型，被设计成通用类型用来在 block 和 thread 之间传递数据。在 stream tags 和 message passed 接口中使用的很多。下面由一段 python 代码看 pmt 的使用方法：

```
>>> import pmt
>>> P = pmt.from_long(23)
>>> type(P)
<class 'pmt.pmt_swig.swig_int_ptr'>
>>> print P
23
>>> P2 = pmt.from_complex(1j)
>>> type(P2)
<class 'pmt.pmt_swig.swig_int_ptr'>
>>> print P2
0+1i
>>> pmt.is_complex(P2)
True
```

我们利用 from_long 和 from_complex 导入了一个长整数和一个复数。但是他们的类型是一样的，都是 pmt。这样我们就可以把这些变量利用 swig 传入 C++。同样 C++代码如下：

```
#include <pmt/pmt.h>
// [...]
pmt::pmt_t P = pmt::from_long(23);
std::cout << P << std::endl;
pmt::pmt_t P2 = pmt::from_complex(gr_complex(0, 1)); // Alternatively:
pmt::from_complex(0, 1)
std::cout << P2 << std::endl;
std::cout << pmt::is_complex(P2) << std::endl;
```

有两个特点在 C++和 python 都很重要。首先，我们可以很容易的打印 pmt 的内容。PMT 内置了把值转化成 string 的方法（某些类型的数据不行）。而且，PMT 必须显式的知道他们的类型，所以我们可以查询他们的类型，比如调用 is_complex 方法。

non-PMT 和 PMT 的转化使用 from_x 和 to_x 方法：

```
pmt::pmt_t P_int = pmt::from_long(42);
int i = pmt::to_long(P_int);
pmt::pmt_t P_double = pmt::from_double(0.2);
double d = pmt::to_double(P_double);
```

string 是一个比较特殊的类型，他的转化是特殊的方法。

```
pmt::pmt_t P_str = pmt::string_to_symbol("spam");
pmt::pmt_t P_str2 = pmt::intern("spam");
std::string
str = pmt::symbol_to_string(P_str);
```

pmt::intern 是 symbol_to_string 的另外一种方法。

在 python 中，我们可以使用弱类型。

7、Metadata Information

7.1 Introduction

元数据文件在文件头有额外的元数据存储着有关采样点类型的信息。原始文件，二进制文件不带有任何额外信息。所以这类文件必须被特殊处理。系统中的任何改变，比如采样率或者接受机的频率都没有在文件中体现，元数据的文件头解决了这类问题。

我们利用 `gr::blocks::file_meta_sink` 写入元数据文件，利用 `gr::blocks::file_meta_source` 读取元数据文件。

元数据文件的文件头描述了一个数据分片的信息。比如，item size，数据类型 (complex)，采样率，首个采样点的时间戳，文件头的大小和分片大小。

第一个静态区保存着：

- version: (char) version number (usually set to METADATA_VERSION)
- rx_rate: (double) Stream's sample rate
- rx_time: (pmt::pmt_t pair - (uint64_t, double)) Time stamp (format from UHD)
- size: (int) item size in bytes - reflects vector length if any.
- type: (int) data type (enum below)
- cplx: (bool) true if data is complex
- strt: (uint64_t) start of data relative to current header
- bytes: (uint64_t) size of following data segment in bytes

额外的分局存储在每一个收到的 tags 里。

- rx_rate: the sample rate of the stream.
- rx_time: the time stamp of the first item in the segment.

在一个文件中的数据类型是不会变得。因为 GNU Radio 的 block 只能在构造函数中的 IO signature 设置数据类型，所以之后的数据类型改变不会被接受。

元数据文件的类型

GNU Radio 支持两种：

- inline: headers 和数据在同一行
- detached: headers 在一个单独的 header file 里

inline 是标准的方法。如果使用 detached 方法，headers 简单地插入到 detached header file；数据文件是标准的无中断的原始二进制格式。

8、message passing

8.1 Introduction

元数据文件在文件头有额外的元数据存储着有关采样点类型的信息。原始文件，二进制文件不带有任何额外信息。所以这类文件必须被特殊处理。系统中的任何改变，比如采样率或者接受机的频率都没有在文件中体现，元 GNURadio

的最初设计是为了处理数据流，比特和采样点为基本的处理单位。为了传输控制信息，元数据，包结构，Gnuradio 引入了标签流。标签流和数据流是并行处理的。标签流是为了存储 metadata，控制信息的。标签和采样点是关联的，和数据流一起传输。这个模型使得模块可以识别一些特殊的事件，采取一定的措施。缺点是：标签流只能单向流动，而且只能在模块的 work 函数访问。优点是，标签流和数据流是等同步的。

设计这个机制的两个目的是：

- 下行的模块可以回传数据给上行模块
- 外部程序可以用这个接口和 GNURADIO 通信

这个模块严重依赖多态类型实现（PMT）。

8.2 Message Passing API

message passing 的接口在 gr::basic_block 中得到了实现，gr::basic_block 是所有 block 的父类。每个 block 都有一个消息队列可以存储消息，并向上传输消息。而且可以区分输入和输出端口。

端口是在构造器中声明的：

```
void message_port_register_in(pmt::pmt_t port_id)
```

```
void message_port_register_out(pmt::pmt_t port_id)
```

每个接口有一个端口 id。其他 block 要和这个端口通信，接收或者发送 message，必须订阅这个端口。subscribe 的 API 如下：

```
void message_port_pub(pmt::pmt_t port_id, pmt::pmt_t msg);
```

```
void message_port_sub(pmt::pmt_t port_id, pmt::pmt_t target);
```

```
void message_port_unsub(pmt::pmt_t port_id, pmt::pmt_t target);
```

任何 block 订阅了另一个 block 的输出端口后，会在发布消息的时候收到消息。在一个 block 内部，当他要发布消息的时候，他会向每个订阅了他输出的端口的 block 的消息队列发送消息。

8.3 Message Handler Functions

订阅了 block 的消息的端口后，必须声明一个处理方法。利用 gr::basic_block::message_port_register_in 订阅了一个端口后，我们必须把这个端口绑定到一个消息处理器上。这个部分利用的是 boost 的 bind 函数：

```
set_msg_handler(pmt::pmt_t port_id,
```

```
boost::bind(&block_class::message_handler_function, this, _1));
```

- 'port_id' 是输入的端口 id
- 'block_class::message_handler_function' 是处理这个端口消息的函数
- this 和 _1 是 boost 绑定函数的

```
void block_class::message_handler_function(pmt::pmt_t msg);
```

8.4 Connecting Messages through the Flowgraph

这个机制的接口是独立于数据流的，所以创建模块的时候不需要

8.5 Code Examples

下面利用 `gr::blocks::message_debug` 和 `gr::blocks::tagged_stream_to_pdu`。
`gr::blocks::message_debug` 模块是用来调试消息传递的模块。有三个输入口：

- `print`，打印所有信息到标准输出流。
- `store`，把消息存储到 `list` 里，和 `gr::blocks::message_debug::get_message(int i)` 连接，取出第 `i` 个消息。
- `pdu_print`，把 PDU 消息转化成标准流。

```
{  
    message_port_register_in(pmt::mp("print"));  
    set_msg_handler(pmt::mp("print"),  
        boost::bind(&message_debug_impl::print, this, _1));  
    message_port_register_in(pmt::mp("store"));  
    set_msg_handler(pmt::mp("store"),  
        boost::bind(&message_debug_impl::store, this, _1));  
    message_port_register_in(pmt::mp("print_pdu"));  
    set_msg_handler(pmt::mp("print_pdu"),  
        boost::bind(&message_debug_impl::print_pdu, this, _1));  
}
```