AI学习笔记--MediaPipe--Calculators

Calculators 作为计算的模块单元，是 MediaPipe 中最基础的元素之一。再上述的 Example 中我们熟知了一个 App 可以使用多种计算单元作为算法的容器，计算 input 和 output 的数据流媒体。

这里我们探讨一下 PacketClonerCalculator 的实现。PacketClonerCalculator 只有非常简单的功能，也使用了很多 Calculator 模块来计算 Graphs，他的主要作用是将输入的数据流复制到输出的数据流中。

PacketClonerCalculators的使用场景是当收到的数据流对应的时间戳并不是和原始需要的相一致。假设我们有一个房间，里面有一个麦克风，一个光传感器和一个正在收集感官数据的摄像机。每个传感器独立工作，并间歇地收集数据。假设每个传感器的输出为:

- Microphone 可应用于判断房间的噪声分贝指数
- Light sensor 可应用于房间内的光照强度
- video Camera，可应用于采集房间内的图形数据。

在这个例子中，我们设计一个简单的目标是处理从这三个传感器中获取来的二进制数据。一旦我们拥有了从 Mic、light sensor、Camera 当中获取的数据后，那样也意味着，在 MediaPipe 处理中，我们拥有了 3 个输入源数据。他们分别是：

- room_mic_signal——这个输入流中的每个数据包都是整数数据，表示带有时间戳的房间中音频的音量大小。
- room_lightening_sensor——这个输入流中的每个数据包都是整数数据，表示房间的照明亮度与时间戳之间的关系。
- room_video_tick_signal——这个输入流中的每个数据包都是视频数据的 imageframe，表示从房间中的摄像机收集的视频，带有时间戳。

如下的代码是一段**PacketClonerCalculator的实现部分。可以使用重写 setContract()，open（）和 process（）等方法处理，也可以使用current_成员变量进行处理。返回缓存的输入数据 package。**

```
#include <vector>

#include "absl/strings/str_cat.h"
#include "mediapipe/calculators/core/packet_cloner_calculator.pb.h"
#include "mediapipe/framework/calculator_framework.h"

namespace mediapipe {

// For every packet received on the last stream, output the latest packet
// obtained on all other streams. Therefore, if the last stream outputs at
a
// higher rate than the others, this effectively clones the packets from
```

```
the
// other streams to match the last.
//
// Example config:
// node {
//   calculator: "PacketClonerCalculator"
//   input_stream: "first_base_signal"
//   input_stream: "second_base_signal"
//   input_stream: "tick_signal"
//   output_stream: "cloned_first_base_signal"
//   output_stream: "cloned_second_base_signal"
// }
//
// Related:
//   packet_cloner_calculator.proto: Options for this calculator.
//   merge_input_streams_calculator.cc: One output stream.
//   packet_inner_join_calculator.cc: Don't output unless all inputs are
new.
class PacketClonerCalculator : public CalculatorBase {
public:
  static ::mediapipe::Status GetContract(CalculatorContract* cc) {
    const int tick_signal_index = cc->Inputs().NumEntries() - 1;
    for (int i = 0; i < tick_signal_index; ++i) {
      cc->Inputs().Index(i).SetAny();
      cc->Outputs().Index(i).SetSameAs(&cc->Inputs().Index(i));
    }
    cc->Inputs().Index(tick_signal_index).SetAny();
    return ::mediapipe::OkStatus();
  }

    // 打开方法，取得头描述等等。
  ::mediapipe::Status Open(CalculatorContext* cc) final {
    // Load options.
    const auto calculator_options =
        cc->Options<mediapipe::PacketClonerCalculatorOptions>();
    output_only_when_all_inputs_received_ =
        calculator_options.output_only_when_all_inputs_received();

    // Parse input streams.
    tick_signal_index_ = cc->Inputs().NumEntries() - 1;
    current_.resize(tick_signal_index_);
    // Pass along the header for each stream if present.
    for (int i = 0; i < tick_signal_index_; ++i) {
      if (!cc->Inputs().Index(i).Header().IsEmpty()) {
        cc->Outputs().Index(i).SetHeader(cc->Inputs().Index(i).Header());
      }
    }
    return ::mediapipe::OkStatus();
  }

    //用于处理的函数方法，如果不为空，copy 到 current_中
```

```
  ::mediapipe::Status Process(CalculatorContext* cc) final {
    // Store input signals.
    for (int i = 0; i < tick_signal_index_; ++i) {
      if (!cc->Inputs().Index(i).Value().IsEmpty()) {
        current_[i] = cc->Inputs().Index(i).Value();
      }
    }

    // Output according to the TICK signal.
    if (!cc->Inputs().Index(tick_signal_index_).Value().IsEmpty()) {
      if (output_only_when_all_inputs_received_) {
        // Return if one of the input is null.
        for (int i = 0; i < tick_signal_index_; ++i) {
          if (current_[i].IsEmpty()) {
            return ::mediapipe::OkStatus();
          }
        }
      }
      // Output each stream.
      for (int i = 0; i < tick_signal_index_; ++i) {
        if (!current_[i].IsEmpty()) {
          cc->Outputs().Index(i).AddPacket(
              current_[i].At(cc->InputTimestamp()));
        } else {
          cc->Outputs().Index(i).SetNextTimestampBound(
              cc->InputTimestamp().NextAllowedInStream());
        }
      }
    }
    return ::mediapipe::OkStatus();
  }

private:
  std::vector<Packet> current_;
  int tick_signal_index_;
  bool output_only_when_all_inputs_received_;
};

REGISTER_CALCULATOR(PacketClonerCalculator);

}  // namespace mediapipe
```

通常来说，一个 Calculator只用一个 CC 文件就可以了，不需要.h 头文件定义，主要是因为，mediaPipe需要注册到 REGISTER_CALCULATOR 宏中，并且，MediaPipe 会去搜相关注册过的Calculator。

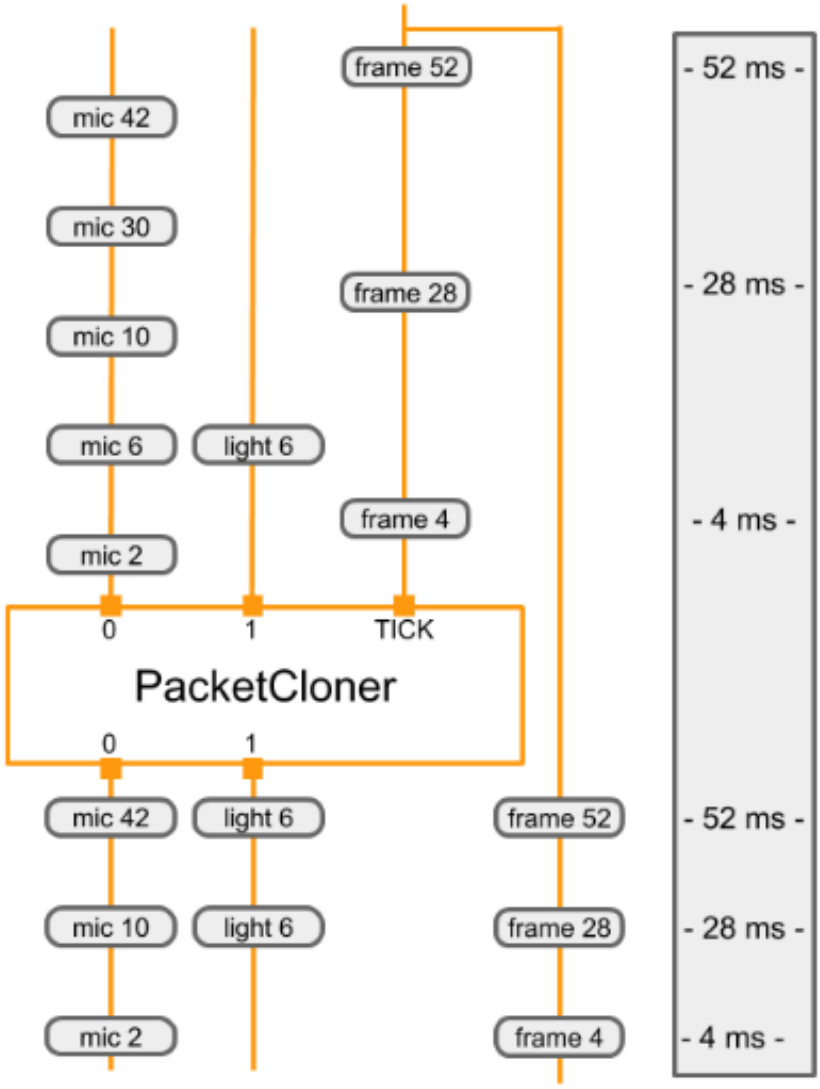
下面是一个简单的 MediaPipe Graph ,它拥有 3 个输入流，1 个 Node 和 3 个输出。代码如下：

```
input_stream: "room_mic_signal"
input_stream: "room_lighting_sensor"
```

```
input_stream: "room_video_tick_signal"

node {
   calculator: "PacketClonerCalculator"
   input_stream: "room_mic_signal"
   input_stream: "room_lighting_sensor"
   input_stream: "room_video_tick_signal"
   output_stream: "cloned_room_mic_signal"
   output_stream: "cloned_lighting_sensor"
   output_stream: "cloned_video_tick_signal"
}
```

下图显示了PacketClonerCalculator如何根据它的一系列输入包定义它的输出。



每当PacketClonerCalculator在它的TICK输入流上收到一个包时，它就从它的每个输入流中输出最近的包。输出包的顺序由输入包的顺序和它们的时间戳决定。时间戳显示在图表的右侧。|