C++编程建议与UML描述

类与类之间的关系

类是代码中的重要组成元素，它把数据和方法当成一个整体来对待。它的很大一个作用就是限定一个方法和数据的职责。举个例子，我们目标检测的后处理是需要**把模型输出的feature-map解析成一个个障碍物**，完成这个功能需要用到很多参数和数据，但这些数据是给一个目标服务的。那么我们就可以用类来实现这个功能，把这个功能当做整体工程中的一个细胞。

实现的建议

我们现在知道类的功能是信息封装和限定职责，那么怎么去设计一个类比较好呢？

我给一些小建议：

1. **单一职责**，这一条是必须的。假如我设计一个类，这个类的作用包含tracking和detection，那明显是不合理的。单一职责有几个好处：
    - 有效限制各个类的职责，加快问题的定位与程序修改的范围。
    - 降低复杂度，增加可维护性，可读性。
2. **尽量隐藏更多信息**，如把数据信息和一些不会被外部使用的函数隐藏在类的内部。
    - 类的使用者经常不是自己（或者是很久之后的自己），如果我们提供更少和更明确的信息，那么使用者能够最快速得掌握类的使用方法。
    - 如果使用者有随意修改参数的权限，可能会出现不可预计的错误。

优化与关系的描述

part 1

代码整理和优化

我们现在有一个最原始版的detector类，这个类实现的质量不错也很稳定，但是我有点吹毛求疵，提了两个小小的建议：

1. 我使用Initialize做初始化会不会更好呢？
2. 如果我把类中的需要初始化的参数整理成一个数据结构，那么我的Init需要传入的参数就很少，调用起来多方便。

```
class Detector {
public:
  Detector(int input_image_width, int input_image_height, int num_classes,
           std::vector<float> confidence_threshold_arry, float
nms_threshold,
           bool is_tensorflow);
  ~ Detector();
  std::vector<std::vector<float>> Detect(std::vector<float *> reslut_blobs,
                                         NmsMode nms_mode);
protected:

private:
  // nms parameters
  int num_classes_;
```

```
  int num_anchors_;
  int num_stages_;
  int num_total_anchors_;
  int coords_;
  int c_num_;
  float confidence_threshold_;
  std::vector<float> confidence_threshold_arry_; //[MAX_CLASS_NUM] = {0.0};
  float objectness_threshold_;
  float nms_threshold_;
  int input_image_width_;
  int input_image_height_;
  std::vector<float> anchors_;
  std::vector<int> mask_;
  std::vector<int> anchors_scale_;
  bool is_tensorflow_;
  void GetRegionBox(std::vector<float> &pred_bbox, const float *pred,
                    const std::vector<float> &anchors, int n, int index,
                    int i, int j, int w, int h, int num_hw);
};

```

稍稍整理后：

```
struct DetectorParams {
    int num_stages = 3;
    int input_image_width = 384;
    int input_image_height = 640;

    int num_anchors;
    int num_classes;
    int offset_objectness;
    float objectness_threhold;
    int offset_box_objectness;
    int coords;
    float nms_threshold;
    int one_anchor_c_num;
    int c_num;
    std::vector<float> anchors;
    std::vector<int> mask;
    std::vector<int> anchors_scale;
    std::vector<float> confidence_threshold_array;
    NmsMode nms_mode;
};


class Detector {
public:
  Detector();
  ~ Detector();
  void Initialize(const DetectorParams &params);
  std::vector<std::vector<float>> Detect(std::vector<float *> reslut_blobs);
protected:

private:
  // nms parameters
  DetectorParams params_;
```

```
  void GetRegionBox(std::vector<float> &pred_bbox, const float *pred,
                    const std::vector<float> &anchors, int n, int index,
                    int i, int j, int w, int h, int num_hw);
};

```

UML文档描述

单个类的基本元素包含**方法和属性**，其中根据**可见性**，可以分成**公有、私有和受保护**这三类。

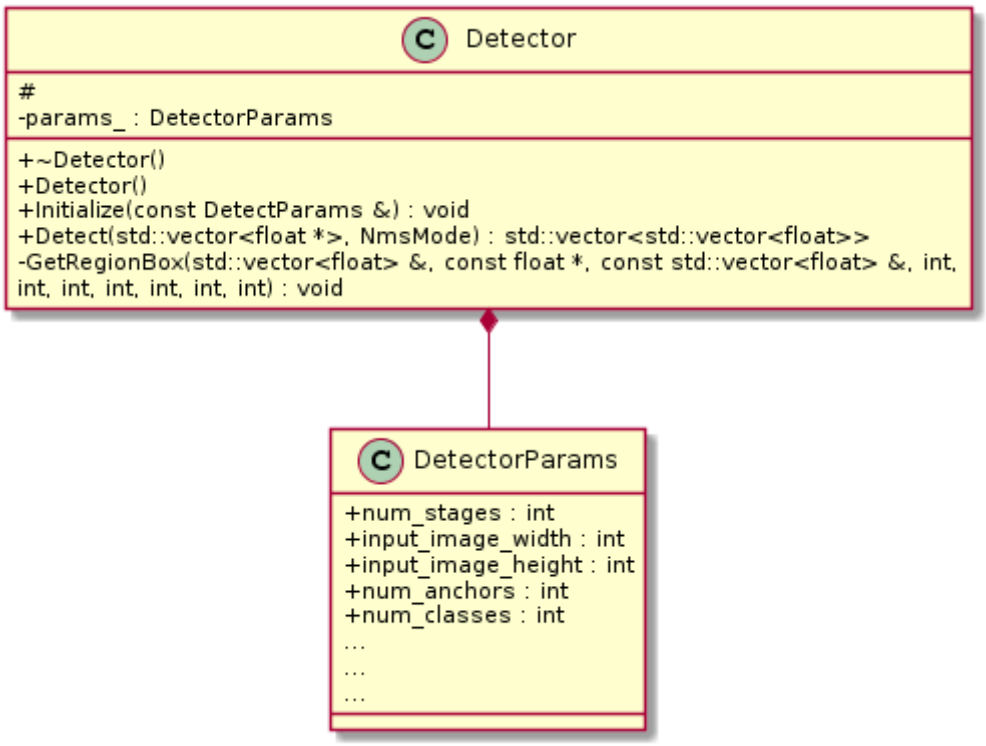
因此UML中类的基本描述包含：

区域	属性
1区	类名 (还可以包含构造型)
2区	类的属性区 (内部数据元素)
3区	行为 - 私有的和公共的

可见性的标注如下：

标记	含义
-	私有的，说明对类外部的调用者是不可见的。
#	保护类型的，只对子类是可见的
+	公共的，对所有都是可见的

我们可以用UML图来表示Detector和DetectorParams这**两个类**， 这里的两个类是**强聚合**的关系，因此用**强聚合**符号来描述：



part2

代码整理与优化

之前我们只有**DetectType1**一种检测方法，而现在我们又增加**DetectType2**的检测方法，那我们如何去**扩展我们的detector**更合适呢？

首先提出有一种简单粗暴的方法，我啥都不考虑**怼接口**就完事了。

```
struct DetectorParams {}
class Detector {
public:
  Detector();
  ~ Detector();
  void Initialize(const DetectorParams &params);
  std::vector<std::vector<float>> DetectType1(std::vector<float *>
reslut_blobs);

  std::vector<std::vector<float>> DetectType2(std::vector<float *>
reslut_blobs);
protected:

private:
  // nms parameters
  DetectorParams params_;
  void GetRegionBoxType1(std::vector<float> &pred_bbox, const float *pred,
                      const std::vector<float> &anchors, int n, int index,
                      int i, int j, int w, int h, int num_hw);
  void GetRegionBoxType2(std::vector<float> &pred_bbox, const float *pred,
                      const std::vector<float> &anchors, int n, int index,
                      int i, int j, int w, int h, int num_hw);

};
```

但是这里两个问题：

1. DetectType1与DetectType2的输出的都是std::vector < std::vector>，但是vector包含的信息是不同的，那么我就需要**两个解析vector的函数**，会增加了**代码的复杂度**。
2. DetectType1与DetectType2 中**95%的内容是重复的**，这会增加了代码的**冗余性**，不好维护。

我们先来解决**第一个问题,** 有没有一个通用的数据结构来表示DetectType1与DetectType2的输出呢？有！

我们可以把函数的输出设置为一个struct，struct可以作为一个**通用的数据传输格式**，非常好解析，也非常好理解

```
struct Obstacle {
  int cutin_type;
  int cls;
  float score;
  float x;
  float y;
  float w;
  float h;
  float dist;
  std::vector<Eigen::Vector2f> box3d;
  std::vector<Eigen::Vector2f> box2d;
};

```

```
typedef std::vector<Obstacle>  ObstacleList;

ObstacleList Detector::DetectType1(std::vector<float *> reslut_blobs);
ObstacleList Detector::DetectType2(std::vector<float *> reslut_blobs);
```

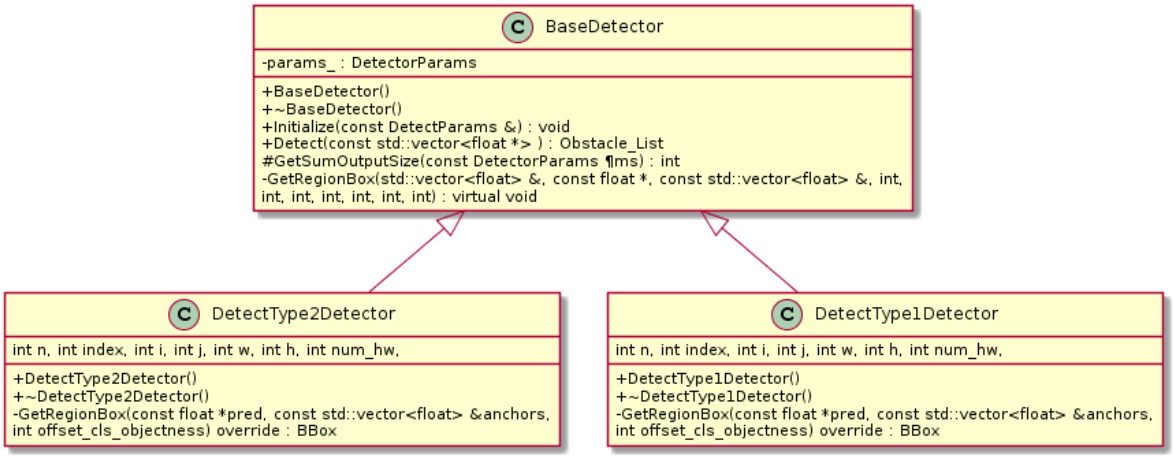
我们来解决**第二个问题**，有没有办法来**最大化来提高代码的复用性**呢？有！

首先我们可以子类继承父类的方法复用相同的代码与函数，然后通过动态绑定的方法调用不能被复用的代码。 经过分析之后发现，DetectType1和DetectType2中只有GetRegion函数不一样，那么我们Detect函数就可以被完全复用了。我们只需要在子类中实现各自的GetRegion函数就可以了。

```
class BaseDetector {
  public:
    BaseDetector();
    virtual ~BaseDetector();

    void Initialize(const DetectorParams &params);
    ObstacleList Detect(const std::vector<float *> &result_blobs);
    std::vector<int> GetOutputSizes(const DetectorParams &params);
  protected:
    int GetSumOutputSize(const DetectorParams &params);

  private:
    DetectorParams params_;
    virtual BBox GetRegionBox(const float *pred, const std::vector<float>
&anchors,
                              int n, int index, int i, int j, int w, int h,
                              int num_hw,int offset_cls_objectness) = 0;
};


class DetectType2Detector : public BaseDetector {
    public:
        DetectType2Detector();
        ~DetectType2Detector();
    private:
        BBox GetRegionBox(const float *pred, const std::vector<float>
&anchors,
                          int n, int index, int i, int j, int w, int h,
                          int num_hw, int offset_cls_objectness) override;
};


class DetectType1Detector : public BaseDetector {
    public:
        DetectType1Detector();
        ~DetectType1Detector();
  private:
        BBox GetRegionBox(const float *pred, const std::vector<float>
&anchors,
                          int n, int index, int i, int j, int w, int h,
                          int num_hw, int offset_cls_objectness) override;
};
```

UML文档描述

这里类与类之间就又多了一种新的关系----**继承**， 我们用继承符号描述父类与子类的关系。



part 3

代码整理和优化

我们已经把detector这个模块的功能实现得差不多了，需要知道以下三件事才能使用这个detector模块：

1. 如何去实例化
2. 如何去初始化
3. 如何去调用功能接口

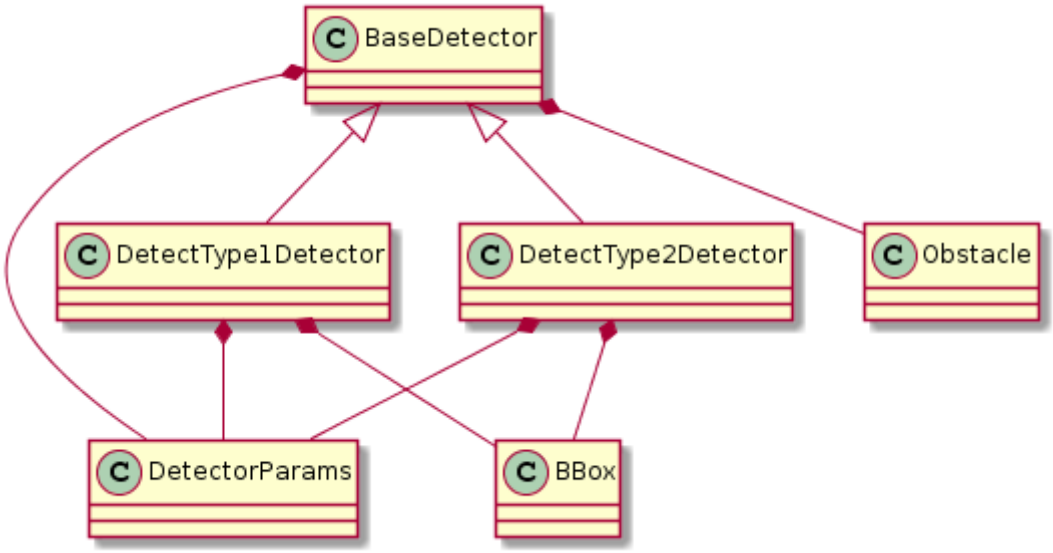
但是现在我使用它需要掌握的信息还是太多了，使用方法能不能**更加简单**一点呢？OK！我们使用**factory设计模式**去控制类的实例化和实例化。

```
enum struct DetectorType {DetectType1, DetectType2};

class DetectorFactory {
  public:
    DetectorFactory();
    ~DetectorFactory();
    std::shared_ptr<BaseDetector> CreateDetector(DetectorType option);
  private:

};

DetectorFactory::DetectorFactory() {}
DetectorFactory::~DetectorFactory() {}

std::shared_ptr<BaseDetector> DetectorFactory::CreateDetector(DetectorType option) {
  std::shared_ptr<BaseDetector> detector = nullptr;
  switch (option) {
    case DetectorType::DetectType1: {
      detector = std::make_shared<DetectType1Detector>();
      DetectorParams params= GetDetectType1DetectorParams();
      detector->Initialize(params);
      break;
    }
    case DetectorType::DetectType2: {
      detector = std::make_shared<DetectType2Detector>();
      DetectorParams params= GetDetectType2DetectorParams();
```

```
      detector->Initialize(params);
      break;
    }
    default: {
      assert(0);
    }
  }
  return detector;
}
```

UML文档描述

我在这里用UML完整得描述一下detector模块，其实在工程中类与类之间95%关系，强聚合（组合）和继承（实现）。



模块与模块之间的关系

到这里我们的detector模块已经比较完整了，那我们如何去描述模块和模块之间的关系？如何去设计好一个模块呢？

实现的建议：

一般情况下，每个子文件夹都中内容都包含一个模块，我针对模块的实现也给出小建议。

1. 模块之间的代码**尽量不要出现耦合**的情况（common模块和data模块除外），模块应该各司其职，一个模块完成独立的一个功能。模块之前的交互只应该是输入与输出的数据。

   好处：

   - 有效限制各个模块的职责，加快问题的定位与程序修改的范围。
   - 可读性， 可维护性。
2. **单一职责**，一个模块只执行模块名表述的职责。
   - 可读性， 可维护性。

优化与关系描述

part 1

整理与优化

整理代码之后，工程的文件结构如下，我在这里做了一些细微的优化。

1. 建立一个common模块，将一些在不同模块间传输的数据结构放入data_type.h中，将对象获取初始化参数的模块放在params.cpp中，之后统一使用params.cpp来获取工程初始化类的参数。
2. 将detector模块中通用的函数放入detector/common.cpp中。

```
|-- src // 代码
     |-- common
        |-- data_type.h
        |-- params.cpp
        |-- params.h
   |-- detector
    |-- base_detector.h
    |-- base_detector.cpp
    |-- common.h
    |-- common.cpp
    |-- detector_factory.cpp
    |-- detector_factory.h
    |-- DetectType2_detector.h
    |-- DetectType2_detector.cpp
    |-- DetectType1_detector.h
    |-- DetectType1_detector.cpp
   |-- perception
    |-- front_obstacle_perception.h
    |-- front_obstacle_perception.cpp
     |-- tracking
...
...
...
```

...