

Introduction

Dig into Apollo is mainly to help learning Apollo autopilot system. We first introduce the functions of each module in detail, and then analyzed the code of each module. If you like autonomous driving and want to learn it, let's start this project and discuss anything you want to know!

Note

- If “git clone” on github is too slow, pls try apollo mirror.
- If you have any questions, please feel free to ask in git issue.
- If you need to add new documents or give suggestions, welcome to participate in git discussion.

Getting Started

If you are like me before and don't know how to start the Apollo project. Here are some suggestions.

- First understand the basic module functions. If you are not clear about the general function of the module, it's difficult for you to understand what's the code doing. Here is an beginner level tutorial.
- Then you need to understand the specific methods according to the module, which will be documented in this tutorial. We will analyze the code in depth next. You can learn step by step according to our tutorial. I know it will be a painful process, especially when you are first learning Apollo. If you persist in asking questions and studying, it will become easier in 1-2 months.
- Last but not least, the method in Apollo is almost perfect, but there will be some problems. Try to implement and improve it, find papers, try the latest methods, and hone your skills. I believe you will enjoy this process.

How to learn?

Basic learning:

Watching some introductory tutorials will help you understand the Apollo autopilot system more quickly. I highly recommend tutorial Try to ask some questions, read some blogs, papers, or go to github to add some issues. If you don't have a self-driving car, you can try to deploy a simulation environment, I highly recommend lgsvl. Its community is very friendly! If there are any problems with the simulation environment, such as map creation, or some other problems, welcome to participate in this project Flycars. We will help as much as possible!

Code learning:

First of all, you must understand c++. If you are not very familiar with it, I recommend the book “c++ primer”. This book is very good, but a bit thick. If you just want to start quickly, then try to find some simple tutorial, here I recommend teacher Hou Jie After understanding C++, it is best to have some basic understanding of the modules, which will help you to read the code. This has been explained many times. Use code reading tools to help you read the code. I highly recommend vscode. It supports both Windows and Linux, and has a wealth of plug-ins, which can help you track codes, search and find call relationships. Of course, there are many professional knowledge and professional libraries. I can't repeat the best tutorials one by one here, but I can try to recommend some. Hongyi Li's deep learning, even a math tutorial 3Blue1Brown's math Do some experiments with the simulator. We say “Make your hands dirty”. You can't just watch, you need to try to modify some configurations and see if it takes effect, if possible, you can also try to answer some questions. Remember that autonomous driving is still far from mature. Read some papers, I read a lot of papers, it help me a lot.

Tip

Hope you have fun!

Contributing

This project welcomes contributions and suggestions. Please see our contribution guidelines.

References & resources

Indices and tables

- [Index](#)
- [Module Index](#)
- [Search Page](#)

1. 什么是Apollo

衣带渐宽终不悔，为伊消得人憔悴。

1.1. 简介

apollo是百度的自动驾驶开源框架，根据自动驾驶的功能划分为不同的模块，下面会根据目录结构和功能模块分别介绍和学习”apollo模块”。下面简单介绍下各个模块的作用：

- 定位 - 知道汽车在哪里,这里的定位可能涉及方方面面,比如GPS,但是GPS的精度只有米级别,有下面几个场景会不太合适,比如你在桥洞里,GPS信号不好的情况,另外还有一种情况是,停车的时候,你要知道前后车的距离,另外比如在雪天或者路况复杂的情况,这些情况下,仅有的GPS信号可能不能满足我们的要求,因此无人车又增加了激光雷达来测量周围环境的距离,而且可以精确到厘米。因此产生了高精地图的需求,高精地图可以把周围的3D环境都记录下来,这样我们就可以通过3D图像来匹配周围的场景,来找到自己的位置,而且高精地图还可以记录比地图多的多的东西,比如红绿灯的位置,交通标志,左转还是右转道.通过高精地图我们不仅可以知道道路情况,还可以知道车辆需要获取的一些其他信息,让车辆知道自己的实时位置。
- 感知 - 我们总是希望车辆行驶在马路中间,这样更加安全,这就需要追踪到道路的路牙线,而道路随时会出现拐弯,那么追踪路牙线就用到了图像处理技术,另外还要感知到什么是车辆,什么是行人,主要涉及到图像的语义分割,感知是自动驾驶中最难,而且最具有挑战性的一块,因为只有感知到周围的行人,车辆,以及突发状况,才能为后面规划线路。
- 规划 - 目前已经知道当前道路情况,而且也已经感知到前面的车辆或者行人,如何去规划我们的行驶线路呢,这里需要解决的就是2个点之间的线路,而且行驶中途可能会出现新的情况,又需要重新规划线路,还有一种情况是,通过高精地图,我们已经知道前面需要转弯了,我们可以提前调整线路来适应这种需求。

| -transform 转换，主要是？

| -v2x 顾名思义就vehicle-to-everything，其希望实现车辆与一切可能影响车辆的实体实现信息交互，

目的是减少事故发生，减缓交通拥堵，降低环境污染以及提供其他信息服务。

| -scripts 脚本

| -third_party 第三方库

| -tools 工具目录，基本就是个空目录

1.3. 编译

apollo采用的是bazel来进行编译，因为我对JAVA的maven比较熟，因此对maven的包管理的功能觉得特别好用，第一能解决包自动下载，第二解决依赖传递的问题，第三解决了编译的问题。也就是说你只要引用对应的包，就可以把包的依赖全部解决。而bazel就是c++对应的编译管理，bazel主要是通过WORKSPACE和BUILD文件来进行编译。

2. 如何编译

高行微言，所以修身。

2.1. 编译

编译脚本在`apollo.sh`中实现，通过shell脚本设置一些参数和环境变量，最后通过`bazel`编译。下面我们分析下`apollo.sh`的具体实现。

`apollo.sh`中实现了一些函数，我们先介绍下`build`函数

2.1.1. build

```
function build() {
    if [ "${USE_GPU}" = "1" ] ; then
        echo -e "${YELLOW}Running build under GPU mode. GPU is
required to run the build.${NO_COLOR}"
    else
        echo -e "${YELLOW}Running build under CPU mode. No GPU is
required to run the build.${NO_COLOR}"
    fi
    info "Start building, please wait ..."

    generate_build_targets
    info "Building on $MACHINE_ARCH..."

    MACHINE_ARCH=$(uname -m)
    JOB_ARG="--jobs=$(nproc) --ram_utilization_factor 80"
    if [ "$MACHINE_ARCH" == 'aarch64' ]; then
        JOB_ARG="--jobs=3"
    fi
    info "Building with $JOB_ARG for $MACHINE_ARCH"

    bazel build $JOB_ARG $DEFINES -c $@ $BUILD_TARGETS
    if [ ${PIPESTATUS[0]} -ne 0 ]; then
        fail 'Build failed!'
    fi
```

```

# Build python proto
build_py_proto

# Clear KV DB and update commit_id after compiling.
if [ "$BUILD_FILTER" == 'cyber' ] || [ "$BUILD_FILTER" ==
'drivers' ]; then
    info "Skipping revision recording"
else
    bazel build $JOB_ARG $DEFINES -c $@ $BUILD_TARGETS
    if [ ${PIPESTATUS[0]} -ne 0 ]; then
        fail 'Build failed!'
    fi
    rm -fr data/kv_db*
    REVISION=$(get_revision)
    ./bazel-bin/modules/common/kv_db/kv_db_tool --op=put \
        --key="apollo:data:commit_id" --value="$REVISION"
fi

if [ -d /apollo-simulator ] && [ -e /apollo-
simulator/build.sh ]; then
    cd /apollo-simulator && bash build.sh build
    if [ $? -ne 0 ]; then
        fail 'Build failed!'
    fi
fi
if [ $? -eq 0 ]; then
    success 'Build passed!'
else
    fail 'Build failed'
fi
}

```

2.2. 常见问题

- 如果机器内存小于8G，会出现编译错误的情况，单独编译又没有问题，问题的原因是内存缓冲不足，导致报错。调大内存后解决，错误信息如下。

```
gcc: internal compiler error: Killed
```

我们可以通过在脚本中减少并行编译的线程数来防止上述问题。修改 scripts/apollo_build.sh 中 jobs 的值，减少一点，本例中为2，也就是运行2个线程。

```
local job_args="--jobs=$((nproc) --  
local ram_resources=HOST_RAM*0.7"  
# 替换为  
local job_args="--jobs=2 --  
local ram_resources=HOST_RAM*0.7"
```

2. 因为一些文件需要下载之后才能编译，如果启动离线模式，或者想下载文件之后再进行编译，可以参考。[bazel](#)
[<https://docs.bazel.build/versions/0.18.1/external.html>]

3. 启动容器

橘生淮南则为橘，生于淮北则为枳。

docker的主要的好处是开箱即用，在编译docker的时候安装好需要的环境，在使用的时候就无需担心环境问题带来的影响了。下面我们主要分析下docker文件夹中的脚本，主要涉及docker的编译、启动、以及host相关的内容。

3.1. docker编译

编译docker在目录中apollo\docker\build中，执行的命令为

```
./build_dev.sh ./dev.x86_64.dockerfile
```

在“build_dev.sh”脚本中会执行编译docker的工作，下面我们分析下docker的编译过程。

3.1.1. build_dev.sh

```
# Usage:  
#   ./build_dev.sh ./dev.x86_64.dockerfile  
# 1. 获取dockerfile名称，这里为第一个参数  
DOCKERFILE=$1  
  
# 2. 获取build_dev.sh的目录路径  
CONTEXT=$(dirname "${BASH_SOURCE[0]}")"  
  
REPO=apolloauto/apollo  
ARCH=$(uname -m)  
TIME=$(date +%Y%m%d_%H%M)  
  
TAG="${REPO}:dev-18.04-${ARCH}-${TIME}"  
  
# Fail on first error.  
set -e
```

```
# 3. 编译docker
docker build -t ${TAG} -f ${DOCKERFILE} ${CONTEXT}
echo "Built new image ${TAG}"
```

其中解释下几个参数： -t：设置docker的tag名称 -f：默认不需要设置这个参数， docker会从当前目录中找dockerfile编译，当有多个dockerfile的时候就需要通过”-f”来指定编译的dockerfile CONTEXT： docker编译的资源目录

参考 [<https://docs.docker.com/engine/reference/commandline/build/>]

疑问

1. ARM架构和X86_64架构的docker编译有什么区别？

接着我们来看dockerfile

3.1.2. dev.x86_64.dockerfile

```
# 1. 以nvidia/cuda:10.0做为基础镜像
FROM nvidia/cuda:10.0-cudnn7-devel-ubuntu18.04

ENV DEBIAN_FRONTEND=noninteractive

# 2. 安装软件
RUN apt-get update -y && \
    apt-get install -y \
    apt-transport-https \
    autotools-dev \
    automake \
    bc \
    build-essential \
    cmake \
    cppcheck \
    curl \
    curlftpfs \
    debconf-utils \
    doxygen \
    gdb \
    git \
    google-perf-tools \
```

```
graphviz \
iproute2 \
iutils-ping \
lcov \
libblas-dev \
libssl-dev \
libboost-all-dev \
libcurl4-openssl-dev \
libfreetype6-dev \
liblapack-dev \
libpcap-dev \
libsdl3-dev \
libgtest-dev \
locate \
lsof \
nfs-common \
python-autopep8 \
shellcheck \
software-properties-common \
sshfs \
subversion \
unzip \
uuid-dev \
v4l-utils \
vim \
wget \
libasound2-dev \
zip && \
apt-get clean && rm -rf /var/lib/apt/lists/* && \
echo '\n\n\n' | ssh-keygen -t rsa

# Run installers.
# 3. 拷贝installers中的脚本，并且执行
COPY installers /tmp/installers
# 4. 安装adv，作用？？
RUN bash /tmp/installers/install_adv_plat.sh
# 5. 安装bazel
RUN bash /tmp/installers/install_bazel.sh
# 6. 安装bazel依赖的包
RUN bash /tmp/installers/install_bazel_packages.sh
# 7. 网络文件系统，百度bosfs基于libfuse
RUN bash /tmp/installers/install_bosfs.sh
# 8. 安装conda，包管理软件
```

```
RUN bash /tmp/installers/install_conda.sh
# 9. 安装ffmpeg, 用于视频处理
RUN bash /tmp/installers/install_ffmpeg.sh
# 10. 安装gflag
RUN bash /tmp/installers/install_gflags_glog.sh
# 11. openGL扩展
RUN bash /tmp/installers/install_glew.sh
# 12. google代码规范
RUN bash /tmp/installers/install_google_styleguide.sh
# 13. 安装caffe
RUN bash /tmp/installers/install_gpu_caffe.sh
# 14. 连续系统的大规模非线性优化的软件库
RUN bash /tmp/installers/install_ipopt.sh
# 15. osqp求解器
RUN bash /tmp/installers/install_osqp.sh
# 16. json rpc调用, 是哪个开源库没有备注
RUN bash /tmp/installers/install_libjsonrpc-cpp.sh
# 17. nonlinear optimization非线性优化
RUN bash /tmp/installers/install_nlopt.sh
# 18. node.js版本管理
RUN bash /tmp/installers/install_node.sh
# 19. 视频编解码库
RUN bash /tmp/installers/install_openh264.sh
# 20. ota安全包, 具体是? ?
RUN bash /tmp/installers/install_ota.sh
# 21. 安装pcl点云库
RUN bash /tmp/installers/install_pcl.sh
# 22. poco提供快速的可移植的网络开发
RUN bash /tmp/installers/install_poco.sh
# 23. 安装protobuf
RUN bash /tmp/installers/install_protobuf.sh
# 24. 安装python模块
RUN bash /tmp/installers/install_python_modules.sh
# 25. qp库
RUN bash /tmp/installers/install_qp_oases.sh
# 26. 安装QT
RUN bash /tmp/installers/install_qt.sh
# 27.
RUN bash /tmp/installers/install_supervisor.sh
# 28. 2D图像的变换
RUN bash /tmp/installers/install_undistort.sh
# 29. 增加用户, 并且添加初始化脚本
RUN bash /tmp/installers/install_user.sh
```

```
# 30. 安装yarn, nodejs管理工具? ? ?
RUN bash /tmp/installers/install_yarn.sh
# 31. 性能调试库
RUN bash /tmp/installers/post_install.sh
# 32. 音频编解码
RUN bash /tmp/installers/install_opuslib.sh

# 33. 设置工作路径和用户为apollo
WORKDIR /apollo
USER apollo
```

还有一些没有用到的脚本

1.
install_adolc.sh
2. 安装fast-rtps, 一个网络发现协议
install_fast-rtps.sh
3. 安装pytorch
install_libtorch.sh

3.2. docker脚本

容器相关的脚本在scripts中，下面我逐步分析下这些脚本做了哪些工作。

3.2.1. dev_start.sh

启动容器的脚本

3.2.2. dev_into.sh

进入容器的脚本

3.3. 设置主机

设置host，提供了一些在主机上使用的脚本

3.3.1. cleanup_resources.sh

清楚宿主机上的docker镜像，卷

3.3.2. install_docker.sh & install_nvidia_docker.sh

安装docker

3.3.3. setup_host.sh

```
Apollo_ROOT_DIR="$(`cd "${(dirname "${BASH_SOURCE[0]})"}/../.` && pwd)`"

# Setup core dump format.
if [ -e /proc/sys/kernel ]; then
    echo "${Apollo_ROOT_DIR}/data/core/core_%e.%p" | \
        sudo tee /proc/sys/kernel/core_pattern
fi

# 设置每1分钟进行NTP时间同步
# Setup ntpdate to run once per minute. Log at
# /var/log/syslog.
grep -q ntpdate /etc/crontab
if [ $? -ne 0 ]; then
    echo "*/1 * * * * root ntpdate -v -u us.pool.ntp.org" | \
        sudo tee -a /etc/crontab
fi

# 使用udev管理设备文件
# Add udev rules.
sudo cp -r ${Apollo_ROOT_DIR}/docker/setup_host/etc/* /etc/
#
# Add uvcvideo clock config.
grep -q uvcvideo /etc/modules
if [ $? -ne 0 ]; then
    echo "uvcvideo clock=realtime" | sudo tee -a /etc/modules
fi
```

1. Cyber

凡学之不勤，必其志之未笃也。

Cyber是百度Apollo推出的替代Ros的消息中间件，自动驾驶中的各个模块通过Cyber进行消息订阅和发布，同时Cyber还提供了任务调度、录制Bag包等功能。通过Cyber实现了自动驾驶中间层，这里分为几小结分别对Cyber进行详细的分析。

1.1. Table of Contents

- Cyber源码分析
- Cyber设计思想
- Cyber分布式部署

2. Audio

勤学如春起之苗，不见其增，日有所长。

2.1. Audio模块介绍

audio模块是Apollo 6.0新增加的模块，主要的用途是通过声音来识别紧急车辆（警车，救护车，消防车）。目前的功能还相对比较简单，只能识别单个紧急车辆，同时需要环境风速低于20mph。下面我们会详细分析这个模块的原理以及实现。

2.1.1. 输入输出

audio模块的输入是`/apollo/sensor/microphone`，输入的消息来源于”drivers/microphone”，用到的硬件模块是”RESPEAKER”，目前有双通道和四通道，Apollo用到的硬件是四通道，关于硬件的相关介绍，会在”drivers/microphone”中进行说明。

audio模块的输出是`/apollo/audio_detection`，输出的消息包括：是否检测到紧急车辆，检测到紧急车辆的移动类型（接近还是远离），位置以及角度。

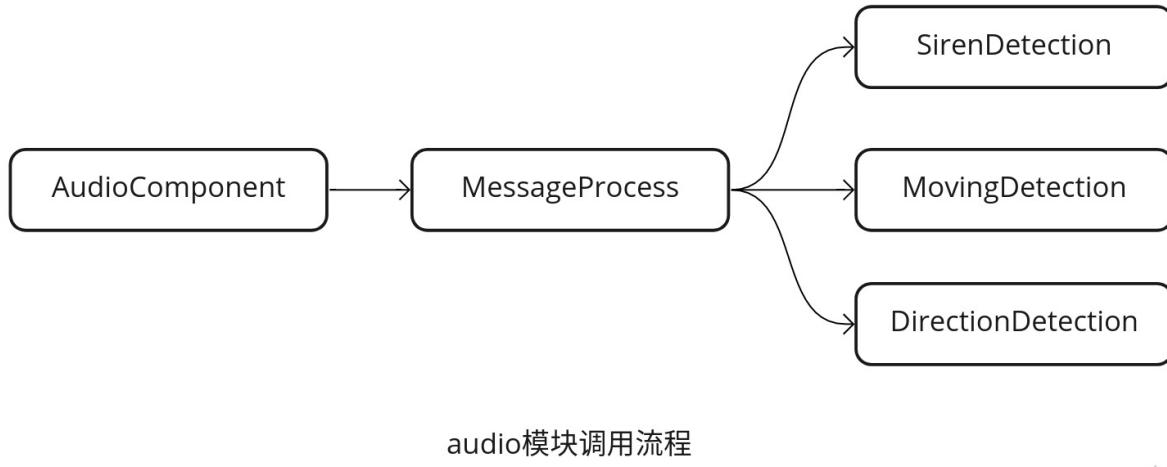
也就是说audio模块单纯的通过声音来识别有没有紧急车辆，以及紧急车辆的位置，为无人驾驶车的感知添加了新的维度。语音交互在车机智能里是非常好的交互方式，目前面临的主要难点是汽车里噪音的影响，这在通过声音进行感知的时候尤其重要。

2.2. 目录结构

audio模块的目录结构如下，整体来说并不复杂，主要的逻辑在”inference”中。

```
|- audio_component.cc
|- audio_component.h // 模块入口
|- BUILD
|- common
|- conf
|- dag
|- data           // 模型文件
|- inference      // 推理
|- launch
|- proto          // 消息格式
|- README.md
|- tools          // 工具
```

Audio模块为事件驱动，当接受到驱动模块的声音输入的时候，就开始解析声音，并且输出结果，下面我们来详细分析具体的处理过程。
audio模块的整体调用流程如图：



2.2.1. 模块入口

audio模块通过”Init()”进行初始化，主要是读取录音机的外参，并且创建”audio_writer_”用于发布消息。初始化好之后，接着通过”Proc”来处理消息。处理的过程通过”OnMicrophone”来完成。

```
bool AudioComponent::Proc(const std::shared_ptr<AudioData>&
audio_data) {
    // TODO(all) remove GetSignals() multiple calls
    AudioDetection audio_detection;
```

```

    MessageProcess::OnMicrophone(*audio_data,
respeaker_extrinsics_file_,
    &audio_info_, &direction_detection_,
&moving_detection_,
    &siren_detection_, &audio_detection);

    FillHeader(node_->Name(), &audio_detection);
    audio_writer_->Write(audio_detection);
    return true;
}

```

2.2.2. 消息处理（MessageProcess）

MessageProcess类实际上是推理模块的一个集合，具体的任务实际上还是在”inference”目录中完成的，主要有3个类，分别为

- **SirenDetection** - 通过深度学习的方法，判断是否是紧急车辆声音。
- **MovingDetection** - 通过声音强度信息和多普勒效应，来判断车辆是靠近还是远离。
- **DirectionDetection** - 通过多个通道之间的差异，计算声音的方向。

下面我们开始分别介绍这3个功能。

2.3. 声音识别（SirenDetection）

声音识别SirenDetection类中只有2个函数，LoadModel进行模型的加载，模型文件在”audio/data”目录中。

```

class SirenDetection {
public:
    bool Evaluate(const std::vector<std::vector<double>>&
signals);

private:
    void LoadModel();

```

Evaluate函数对多个通道声音进行处理，然后输入到模型，并且得出正向和反向的结果，当正向的分数大于反向的时候，则表示检测到了特殊车辆。

```
bool SirenDetection::Evaluate(const
std::vector<std::vector<double>>& signals) {
    // 1. 读取4个通道的数据，装入audio_tensor中
    torch::Tensor audio_tensor = torch::empty(4 * 1 * 72000);
    float* data = audio_tensor.data_ptr<float>();

    for (const auto& channel : signals) {
        for (const auto& i : channel) {
            *data++ = static_cast<float>(i) / 32767.0;
        }
    }

    // 2. 转换声音为张量
    torch::Tensor torch_input =
    torch::from_blob(audio_tensor.data_ptr<float>(),
                     {4, 1,
72000});
    std::vector<torch::jit::IValue> torch_inputs;
    torch_inputs.push_back(torch_input.to(device_));

    // 3. 模型推理
    at::Tensor torch_output_tensor =
    torch_model_.forward(torch_inputs).toTensor()

    .to(torch::kCPU);
    auto torch_output = torch_output_tensor.accessor<float, 2>()
();

    // 4. 投票表决
    float neg_score = torch_output[0][0] + torch_output[1][0] +
                      torch_output[2][0] + torch_output[3][0];
    float pos_score = torch_output[0][1] + torch_output[1][1] +
                      torch_output[2][1] + torch_output[3][1];

    if (neg_score < pos_score) {
        return true;
    } else {
        return false;
    }
}
```

```
    }  
}
```

2.4. 移动检测(MovingDetection)

移动检测在”MovingDetection”类中实现，通过检测最近3帧的声音强度和声音频率，来判断紧急车辆是接近还是远离。MovingDetection中主要有3个函数。

```
// 1. 快速傅里叶变换  
std::vector<std::complex<double>> fft1d(const  
std::vector<double>& signals);  
// 2. 移动检测  
MovingResult Detect(const std::vector<std::vector<double>>&  
signals);  
// 3. 检测单个通道  
MovingResult DetectSingleChannel(  
    const std::size_t channel_index, const  
    std::vector<double>& signal);
```

快速傅里叶变换在fft1d中完成，主要是把时域的东西转换到频域，在这里主要是为了得到声音的频率。

接下来我们看”DetectSingleChannel”的实现。

```
MovingResult MovingDetection::DetectSingleChannel(  
    const std::size_t channel_index, const  
    std::vector<double>& signals) {  
    static constexpr int kStartFrequency = 3;  
    static constexpr int kFrameNumStored = 10;  
    std::vector<std::complex<double>> fft_results =  
        fft1d(signals);  
    // 1. 获取声音信息  
    SignalStat signal_stat = GetSignalStat(fft_results,  
    kStartFrequency);  
    signal_stats_[channel_index].push_back(signal_stat);  
    while (static_cast<int>  
(signal_stats_[channel_index].size()) >  
        kFrameNumStored) {  
        signal_stats_[channel_index].pop_front();
```

```

    }

    // 2. 分析声音强度
    MovingResult power_result =
        AnalyzePower(signal_stats_[channel_index]);
    if (power_result != UNKNOWN) {
        return power_result;
    }

    // 3. 分析声音频率
    MovingResult top_frequency_result =
        AnalyzeTopFrequency(signal_stats_[channel_index]);
    return top_frequency_result;
}

```

可以看到单个通道先通过”GetSignalStat”获取声音信息，并且优先采用声音强度信息，然后采用声音频率。

我们接着上一步骤看如何获取声音强度和声音频率。

```

MovingDetection::SignalStat MovingDetection::GetSignalStat(
    const std::vector<std::complex<double>>& fft_results,
    const int start_frequency) {
    double total_power = 0.0;
    int top_frequency = -1;
    double max_power = -1.0;
    for (int i = start_frequency; i < static_cast<int>(fft_results.size()); ++i) {
        double power = std::abs(fft_results[i]);
        // 1. 对一段时间的声音强度做累加
        total_power += power;
        // 2. 找出一段时间内声音强度最大的作为当时的频率
        if (power > max_power) {
            max_power = power;
            top_frequency = i;
        }
    }
    return {total_power, top_frequency};
}

```

“AnalyzePower”和”AnalyzeTopFrequency”的逻辑相对简单，就是判断过去的三帧，声音强度是否一直减少、增大，频率是否一直减少、增大。以此来判断汽车是远离还是靠近。

最后”Detect”函数会综合4个通道的数据来进行投票表决，然后得到汽车是远离还是靠近，最后输出结果。

2.5. 方向检测(DirectionDetection)

在”DirectionDetection”类中对紧急车辆的方向进行估计，通过2个通道的差异和声音的速度，就可以得到车辆的大概位置。

DirectionDetection类的主要实现在”EstimateSoundSource”函数中，下面我们来分析下具体的实现。

```
std::pair<Point3D, double>
DirectionDetection::EstimateSoundSource(
    std::vector<std::vector<double>>&& channels_vec,
    const std::string& respeaker_extrinsic_file, const int
sample_rate,
    const double mic_distance) {
    // 1. 加载外参
    if (!respeaker2imu_ptr_.get()) {
        respeaker2imu_ptr_.reset(new Eigen::Matrix4d);
        LoadExtrinsics(respeaker_extrinsic_file,
        respeaker2imu_ptr_.get());
    }
    // 2. 计算方向角度
    double degree =
        EstimateDirection(move(channels_vec), sample_rate,
        mic_distance);
    // 3. 计算距离
    Eigen::Vector4d source_position(kDistance * sin(degree),
                                    kDistance * cos(degree), 0,
        1);
    source_position = (*respeaker2imu_ptr_) * source_position;

    Point3D source_position_p3d;
    source_position_p3d.set_x(source_position[0]);
    source_position_p3d.set_y(source_position[1]);
    source_position_p3d.set_z(source_position[2]);
    degree = NormalizeAngle(degree);
    return {source_position_p3d, degree};
}
```

那么如何计算方向角度呢？下面我们看下”EstimateDirection”的实现。

```
double DirectionDetection::EstimateDirection(
    std::vector<std::vector<double>>&& channels_vec, const
int sample_rate,
    const double mic_distance) {
    // 1. 把声音数据放入channels_ts
    std::vector<torch::Tensor> channels_ts;
    auto options =
        torch::TensorOptions().dtype(torch::kFloat64);
    int size = static_cast<int>(channels_vec[0].size());
    for (auto& signal : channels_vec) {
        channels_ts.push_back(torch::from_blob(signal.data(),
{size}, options));
    }

    double tau0, tau1;
    double theta0, theta1;
    const double max_tau = mic_distance / kSoundSpeed;
    // 2. 分别计算通道0和2,1和3的组合来得出角度
    tau0 = GccPhat(channels_ts[0], channels_ts[2], sample_rate,
max_tau, 1);
    theta0 = asin(tau0 / max_tau) * 180 / M_PI;
    tau1 = GccPhat(channels_ts[1], channels_ts[3], sample_rate,
max_tau, 1);
    theta1 = asin(tau1 / max_tau) * 180 / M_PI;

    int best_guess = 0;
    // 3. 得到最优解
    if (fabs(theta0) < fabs(theta1)) {
        best_guess = theta1 > 0 ? std::fmod(theta0 + 360, 360) :
(180 - theta0);
    } else {
        best_guess = theta0 < 0 ? std::fmod(theta1 + 360, 360) :
(180 - theta1);
        best_guess = (best_guess + 90 + 180) % 360;
    }
    best_guess = (-best_guess + 480) % 360;

    return static_cast<double>(best_guess) / 180 * M_PI;
}
```

计算2个通道，从而得到角度信息的实现如下。

```
double DirectionDetection::GccPhat(const torch::Tensor& sig,
                                     const torch::Tensor&
                                     refsig, int fs,
                                     double max_tau, int
                                     interp) {
    const int n_sig = sig.size(0), n_refsig = refsig.size(0),
              n = n_sig + n_refsig;
    torch::Tensor psig = at::constant_pad_nd(sig, {0,
n_refsig}, 0);
    torch::Tensor prefsg = at::constant_pad_nd(refsig, {0,
n_sig}, 0);
    psig = at::rfft(psig, 1, false, true);
    prefsg = at::rfft(prefsg, 1, false, true);

    ConjugateTensor(&prefsg);
    // 1. 复数相乘
    torch::Tensor r = ComplexMultiply(psig, prefsg);
    // 2. 复数取绝对值
    torch::Tensor cc =
        at::irfft(r / ComplexAbsolute(r), 1, false, true,
{interp * n});
    int max_shift = static_cast<int>(interp * n / 2);
    if (max_tau != 0)
        max_shift = std::min(static_cast<int>(interp * fs *
max_tau), max_shift);

    auto begin = cc.index({Slice(cc.size(0) - max_shift,
None)});
    auto end = cc.index({Slice(None, max_shift + 1)});
    cc = at::cat({begin, end});
    // find max cross correlation index
    const int shift = at::argmax(at::abs(cc), 0).item<int>() -
max_shift;
    const double tau = shift / static_cast<double>(interp *
fs);

    return tau;
}
```

Todo: 关于上述过程的详细计算原理，之后需要做进一步的补充？？？

2.6. 工具 (tools)

tools目录提供了一些录制和调试工具。

- **audiosaver.py** - 录制声音并且保存。
- **audio_offline_processing.cc** - 离线测试工具，提供audio模块的离线功能。

至此，我们就得到了是否有紧急车辆，以及车辆的移动方式和方向。实际上audio模块的代码中还遗留有位置信息、感知的结果，估计后面会增加一些新的融合功能。

整体上audio模块还是挺有意思的，当然声音的识别，以及在嘈杂环境如何获取到比较关注的声音，都是业界研究的热点方向，但主要还是集中在室内对人的声音的追踪，室外以及对车或者后续增加到人的场景，还有待发掘。

3. Bridge

上穷碧落下黄泉，两处茫茫皆不见。

3.1. Table of Contents

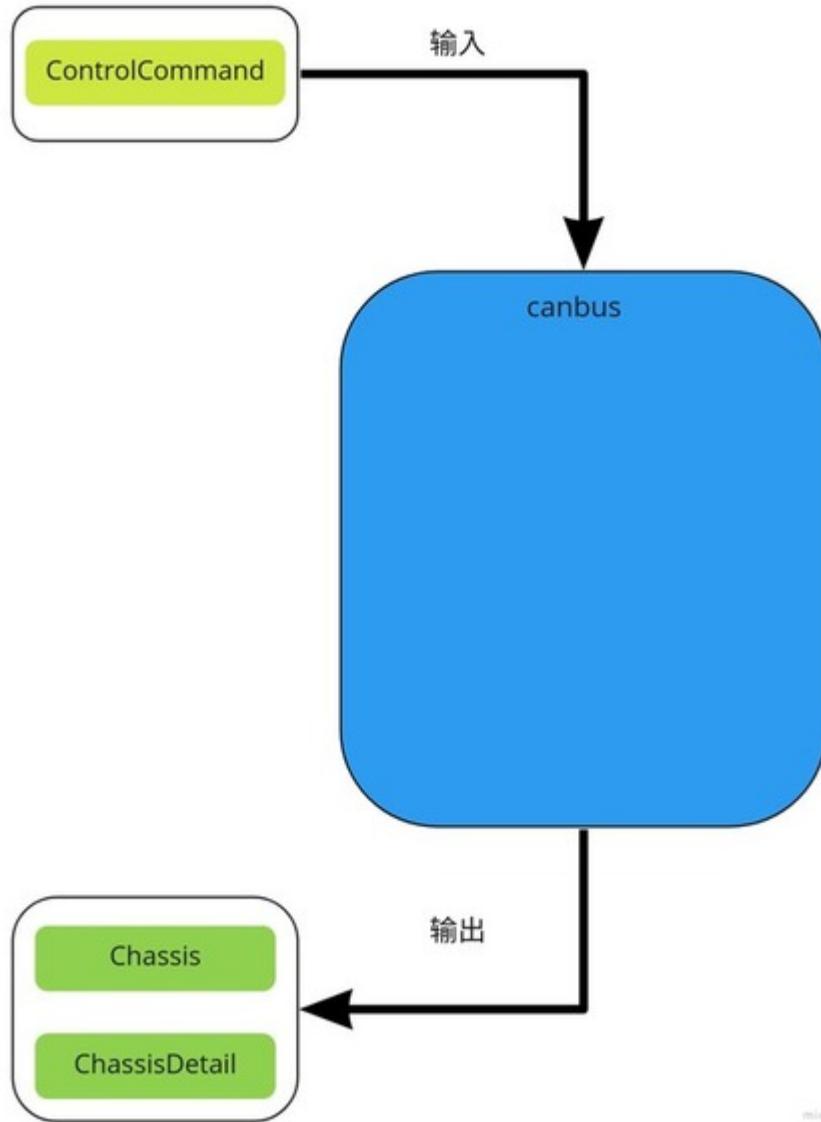
4. Canbus

黑发不知勤学早，白首方悔读书迟。

4.1. Canbus模块介绍

我们先看下什么是Canbus： 控制器局域网 (Controller Area Network, 简称CAN或者CAN bus) 是一种车用总线标准。被设计用于在不需要主机 (Host) 的情况下，允许网络上的节点相互通信。采用广播机制，并利用标识符来定义内容和消息的优先顺序，使得canbus的扩展性良好，同时不基于特殊类型 (Host) 的节点，增加了升级网络的便利性。这里的**Canbus**模块其实可以称为**Chassis**模块，主要的作用是反馈车当前的状态（航向，角度，速度等信息），并且发送控制命令到车线控底盘，可以说**Canbus**模块是车和自动驾驶软件之间的桥梁。由于这个模块和”drivers/canbus”的联系紧密，因此也一起在这里介绍。 Canbus模块是车和自动驾驶软件之间的桥梁，通过canbus驱动 (drivers/canbus)来实现将车身信息发送给apollo上层软件，同时接收控制命令，发送给汽车线控底盘实现对汽车的控制。

那么canbus模块的输入是什么？输出是什么呢？



可以看到canbus模块：

- 输入 - 1. ControlCommand (控制命令)
- 输出 - 1. Chassis (汽车底盘信息), 2. ChassisDetail (汽车底盘信息详细信息)

Canbus模块的输入是control模块发送的控制命令，输出汽车底盘信息，这里apollo的上层模块被当做一个can_client来处理，实现接收和发送canbus上的消息。

Canbus模块的目录结构如下：

```
|-- BUILD          // bazel编译文件
|-- canbus_component.cc // canbus主入口
|-- canbus_component.h
|-- canbus_test.cc    // canbus测试
|-- common           // gflag配置
|-- conf             // 配置文件
|-- dag              // dag依赖
|-- launch           // launch加载
|-- proto            // protobuf文件
|-- testdata         // 测试数据
|-- tools            // 遥控汽车和测试canbus总线工具
|-- vehicle          //
```

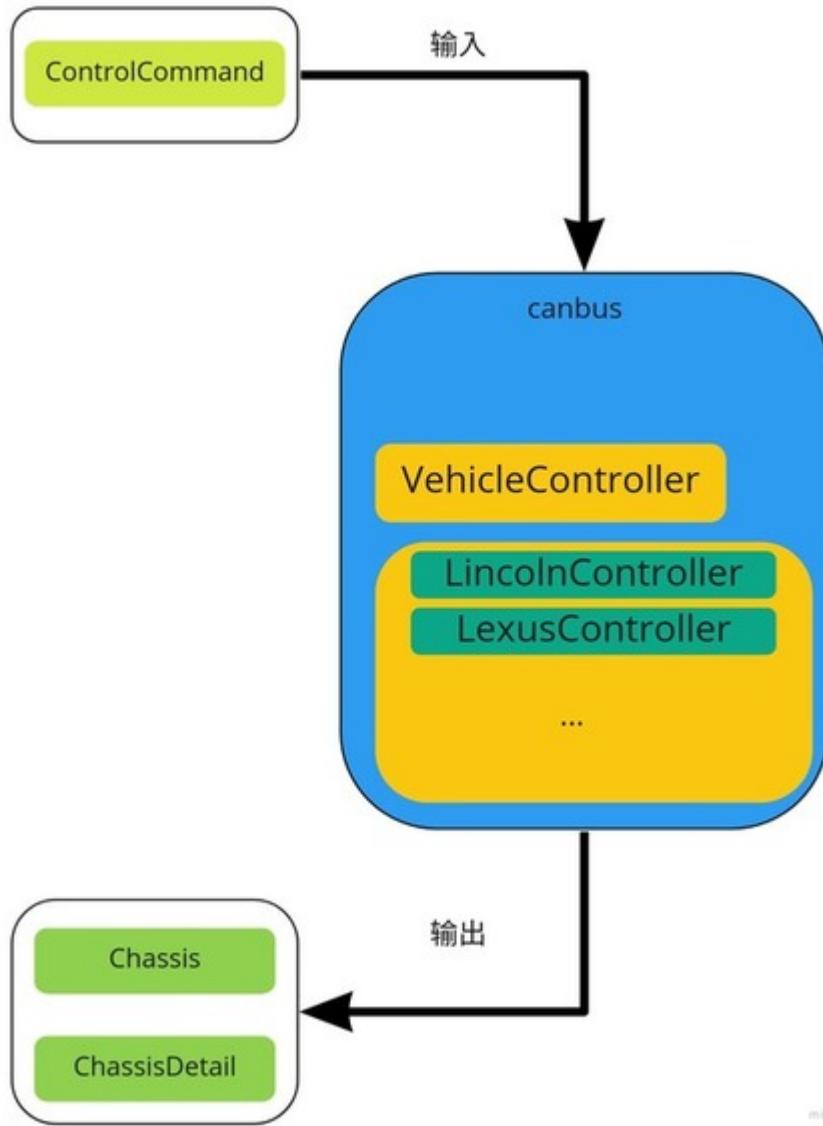
接着我们来分析下Canbus模块的执行流程。

4.2. Canbus模块主流程

Canbus模块的主流程在文件”canbus_component.cc”中， canbus模块为定时触发，每10ms执行一次，发布chassis信息，而ControlCommand则是每次读取到之后触发回调”OnControlCommand”，发送”control_command”到线控底盘。

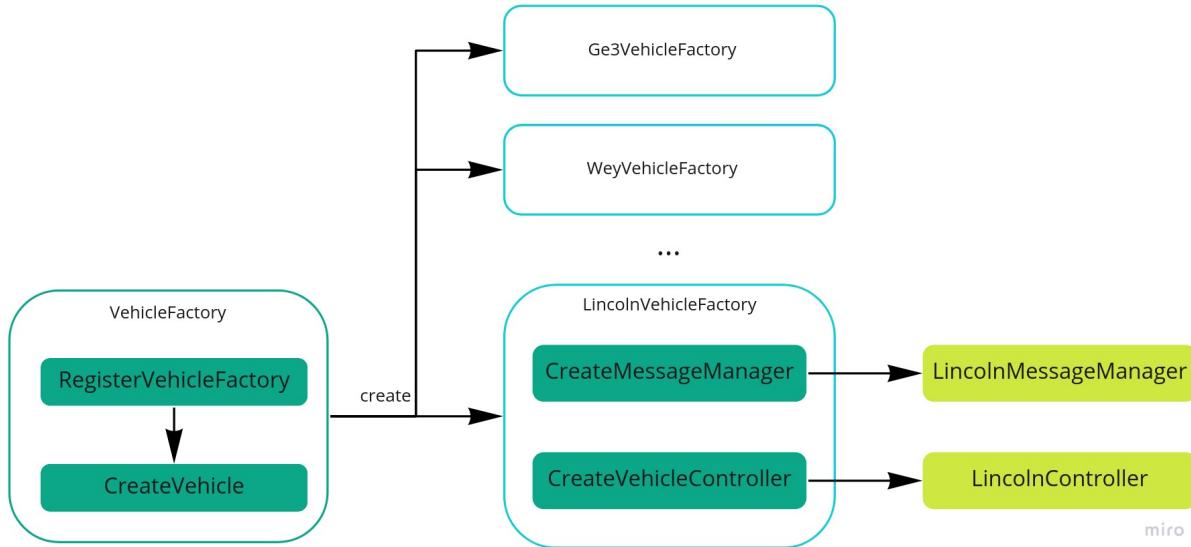
```
bool CanbusComponent::Proc() {
    PublishChassis();
    if (FLAGS_enable_chassis_detail_pub) {
        PublishChassisDetail();
    }
    return true;
}
```

由于不同型号的车辆的canbus命令不一样，在”/vehicle”中适配了不同型号车辆的canbus消息格式，所有的车都继承自Vehicle_controller基类，通过对Vehicle_controller的抽象来发送和读取canbus信息。



4.2.1. 车辆工厂模式(VehicleFactory)

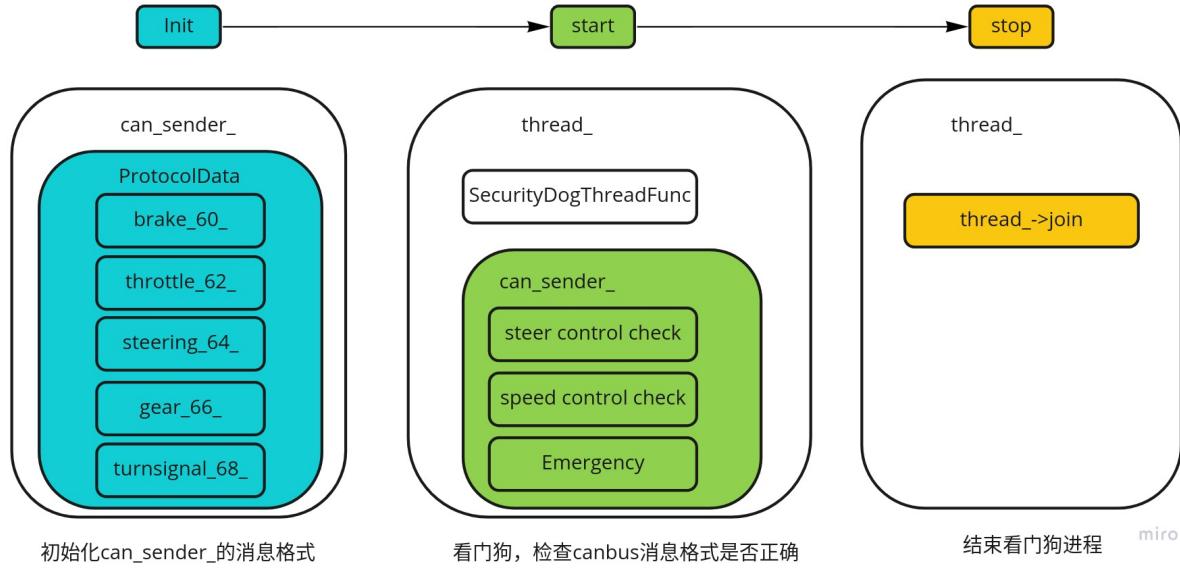
在vehicle中可以适配不同的车型，而每种车型都对应一个 vehicle_controller，创建每种车辆的控制器(VehicleController)和消息管理(MessageManager)流程如下：



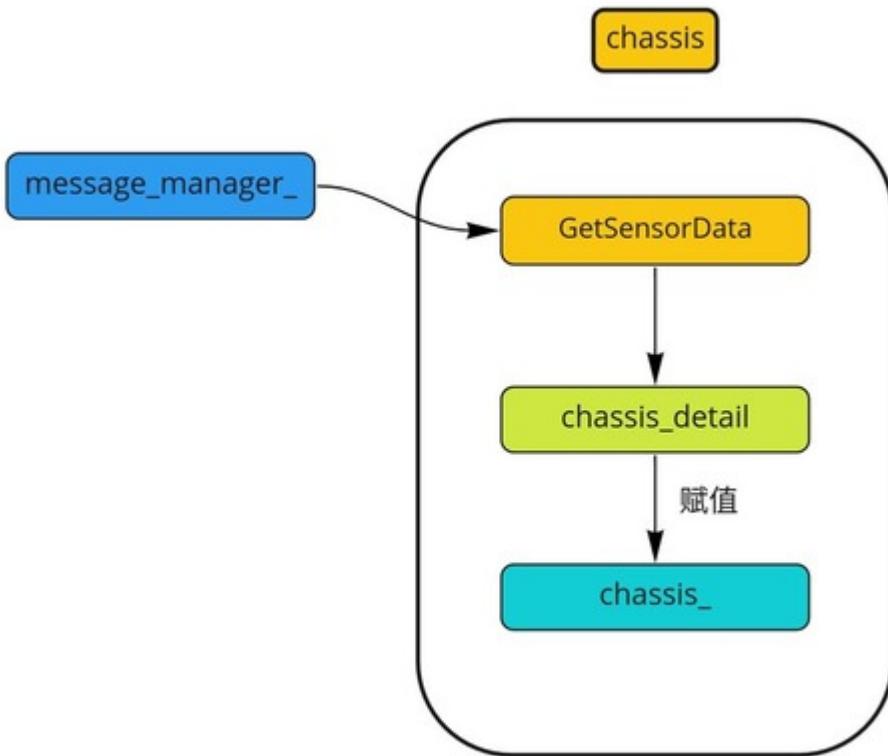
VehicleFactory类通过创建不同的类型**AbstractVehicleFactory**，每个车型自己的Factory在创建出对应的**VehicleController**和**MessageManager**，用林肯来举例子就是： **VehicleFactory**创建**LincolnVehicleFactory**，之后通过**CreateMessageManager**和**CreateVehicleController**创建对应的控制器（**LincolnController**）和消息管理器（**LincolnMessageManager**）。上述代码流程用到了设计模式的工厂模式，通过车辆工厂创造不同的车辆类型。

4.2.2. 车辆控制器(**LincolnController**)

下面以林肯来介绍LincolnController，以及如何接收chassis信息，其它的车型可以以此类推，下面主要分为2部分介绍，第一部分为controller的init->start->stop流程，第二部分为chassis信息获取：



可以看到control模块初始化(init)的过程获取了发送的消息的格式，通过can_sender应该发送那些消息，而启动(start)之后启动一个看门狗，检查canbus消息格式是否正确，最后关闭(stop)模块则是结束看门狗进程。



通过message_manager_获取chassis信息 而chassis的获取则是通过message_manager_获取chassis_detail，之后对chassis进行赋值。

4.3. Canbus(驱动程序)

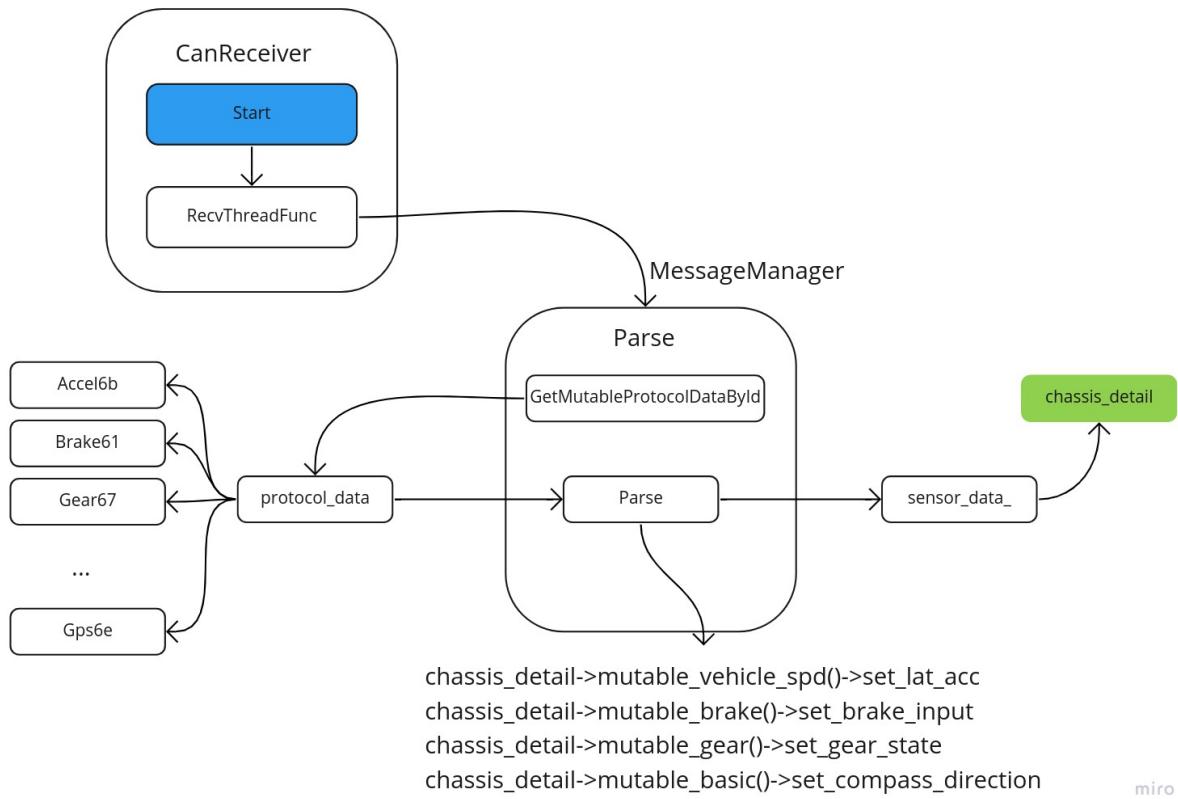
上层的canbus就介绍完成了，而canbus的发送(CanSender)和接收(CanReceiver)，还有消息管理(MessageManager)都是在”drivers/canbus”中实现的。

4.3.1. 消息管理器(MessageManager)

MessageManager是如何获取消息的呢？

MessageManager主要作用是解析和保存canbus数据，而具体的接收和发送则是在”**CanReceiver**”和”**CanSender**”中，拿接收消息举例，也就是说CanReceiver收到消息后，会调用MessageManager中的parse去解析消息，消息的解析协议

在”modules/canbus/vehicle/lincoln/protocol”中，每个消息把自己对应的信息塞到”chassis_detail”中完成了消息的接收。



4.3.2. 消息接收(CanReceiver)

canbus消息的接收在上面有介绍，在`CanReceiver`中的”Start”调用”`RecvThreadFunc`”实现消息的接收，这里会启动一个异步进程去完成接收。

```

template <typename SensorType>
::apollo::common::ErrorCode CanReceiver<SensorType>::Start()
{
    if (is_init_ == false) {
        return ::apollo::common::ErrorCode::CANBUS_ERROR;
    }
    is_running_.exchange(true);

    // 启动异步接收消息
    async_result_ =
    cyber::Async(&CanReceiver<SensorType>::RecvThreadFunc, this);
  
```

```

    return ::apollo::common::ErrorCode::OK;
}

```

RecvThreadFunc通过“can_client_”接收消息，然后通过“MessageManager”去解析消息，在MessageManager中有讲到。

```

template <typename SensorType>
void CanReceiver<SensorType>::RecvThreadFunc() {

    ...
    while (IsRunning()) {
        std::vector<CanFrame> buf;
        int32_t frame_num = MAX_CAN_RECV_FRAME_LEN;

        // 1. can_client_ 接收canbus数据
        if (can_client_->Receive(&buf, &frame_num) != ::apollo::common::ErrorCode::OK) {

            cyber::USleep(default_period);
            continue;
        }
        ...

        for (const auto &frame : buf) {
            uint8_t len = frame.len;
            uint32_t uid = frame.id;
            const uint8_t *data = frame.data;

            // 2. MessageManager解析canbus数据
            pt_manager_->Parse(uid, data, len);
            if (enable_log_) {
                ADEBUG << "recv_can_frame#" << frame.CanFrameString();
            }
        }
        cyber::Yield();
    }
    AINFO << "Can client receiver thread stopped.";
}

```

4.3.3. 消息发送(CanSender)

消息发送对应的是在CanSender中的”Start”调用”PowerSendThreadFunc”，我们可以看具体实现：

```
template <typename SensorType>
common::ErrorCode CanSender<SensorType>::Start() {
    if (is_running_) {
        AERROR << "Cansender has already started.";
        return common::ErrorCode::CANBUS_ERROR;
    }
    is_running_ = true;

    // 启动线程发送消息
    thread_.reset(new std::thread([this] {
        PowerSendThreadFunc(); }));
}

return common::ErrorCode::OK;
}
```

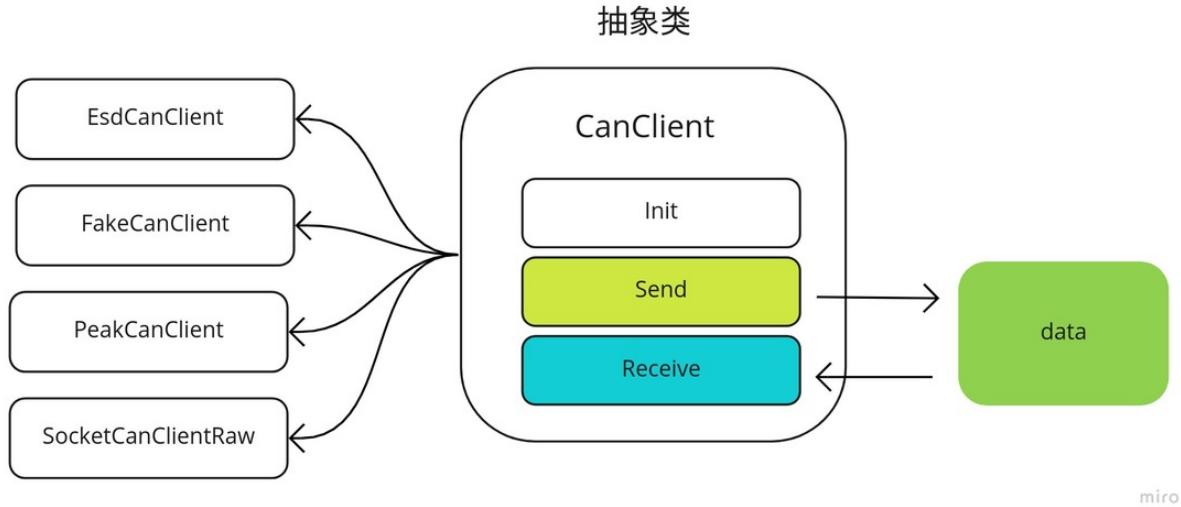
PowerSendThreadFunc再通过”can_client”发送消息：

```
std::vector<CanFrame> can_frames;
CanFrame can_frame = message.CanFrame();
can_frames.push_back(can_frame);

// 通过can_client发送消息
if (can_client_->SendSingleFrame(can_frames) != common::ErrorCode::OK) {
    AERROR << "Send msg failed:" << can_frame.CanFrameString();
}
```

4.3.4. canbus客户端(CanClient)

CanClient是canbus客户端，同时也是canbus的驱动程序，针对不同的canbus卡，对发送和接收进行封装，并且提供给消息发送和接收控制器使用。



拿”EsdCanClient”来举例子，发送在”Send”函数中，调用的是第三方的硬件驱动，目录在”third_party/can_card_library/esd_can”，实现can消息的发送：

```
ErrorCode EsdCanClient::Send(const std::vector<CanFrame>
&frames,
                                int32_t *const frame_num) {
    ...
    // canWrite为第三方库的硬件驱动,
    third_party/can_card_library/esd_can
    // Synchronous transmission of CAN messages
    int32_t ret = canWrite(dev_handler_, send_frames_,
frame_num, nullptr);
    if (ret != NTCAN_SUCCESS) {
        AERROR << "send message failed, error code: " << ret <<
        ", "
        << GetErrorString(ret);
        return ErrorCode::CAN_CLIENT_ERROR_BASE;
    }
    return ErrorCode::OK;
}
```

其他的can卡可以参考上述的流程，至此整个canbus驱动就分析完成了。

4.4. Reference

Controller Area Network (CAN BUS) 通訊協定原理概述

[<https://www.ni.com/zh-tw/innovations/white-papers/06/controller-area-network--can--overview.html>]

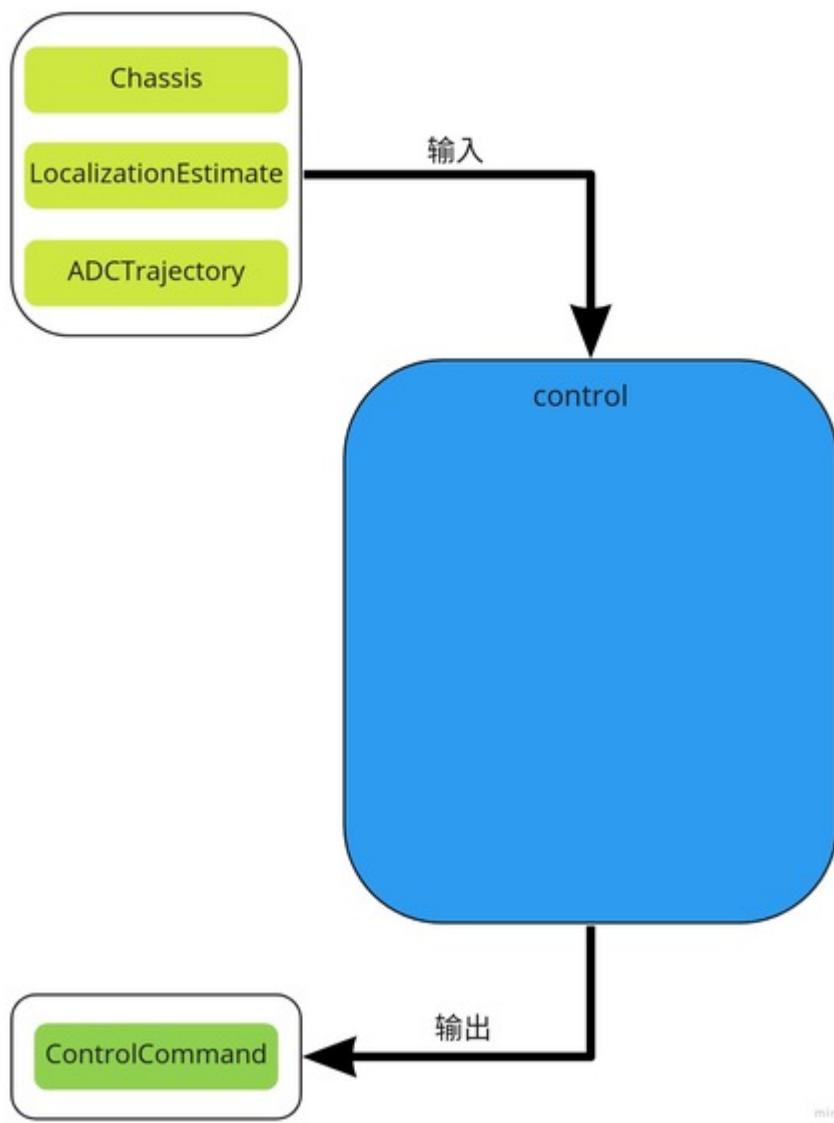
5. Control

岁不寒，无以知松柏；事不难，无以知君子。

5.1. Control模块简介

Apollo控制模块的逻辑相对比较简单，控制模块的作用是根据规划(**planning**模块)生成的轨迹，计算出汽车的油门，刹车和方向盘信号，控制汽车按照规定的轨迹行驶。采用的方法是根据汽车动力学和运动学的知识，对汽车进行建模，实现对汽车的控制。目前apollo主要用到了2种控制方式：PID控制和模型控制。

首先我们需要搞清楚control模块的输入是什么，输出是什么？



可以看到control模块：

- 输入 - Chassis(车辆状态信息), LocalizationEstimate(位置信息), ADCTrajectory(planning模块规划的轨迹)
- 输出 - ControlCommand(油门, 刹车, 方向盘)

Control模块的目录结构如下：

```
|- BUILD      // bazel编译文件
|- common     // PID和控制器的具体实现      --- 算法具体实现
|- conf       // 配置文件                  --- 配置文件
|- control_component.cc // 模块入口
```

```

├── control_component.h
├── control_component_test.cc
├── controller          // 控制器           --- 具体的控制器实现
├── dag                  // dag依赖
├── integration_tests    // 测试
├── launch               // launch加载
├── proto                // protobuf文件，主要是各个控制器的配置数
据结构
└── testdata              // 测试数据
└── tools                 // 工具类

```

下面我们接着分析下control模块的执行流程。

5.2. Control模块主流程

Control模块的入口在”control_component.cc”中，和其它的模块一样，Control模块注册为cyber的一个模块，其中control模块为定时模块，也就是每隔10ms执行一次命令。这里需要注意了planning模块的输出是100ms一次，也就是说100ms才给出一条曲线，而control模块会根据这条曲线，每10ms处理一次，控制汽车按照指定的速度到达指定的位置。Control模块执行的主函数是”ControlComponent::Proc()”，即每隔10ms调用一次该函数，处理的流程在该函数中：

```

bool ControlComponent::Proc() {
    // 1. 读取输入数据，通过拷贝读取输入信息
    chassis_reader_->Observe();
    const auto &chassis_msg = chassis_reader_-
>GetLatestObserved();
    if (chassis_msg == nullptr) {
        AERROR << "Chassis msg is not ready!";
        return false;
    }

    OnChassis(chassis_msg);

    ...
    const auto &trajectory_msg = trajectory_reader_-
>GetLatestObserved();

    ...
}

```

```

const auto &localization_msg = localization_reader_-
>GetLatestObserved();

ControlCommand control_command;
// 2. 生成控制命令
Status status = ProduceControlCommand(&control_command);

...

common::util::FillHeader(node_->Name(), &control_command);
// 3. 发送控制命令
control_cmd_writer_->Write(std::make_shared<ControlCommand>
(control_command));

return true;
}

```

上述的流程大概分为3个阶段：

1. 读取输入数据
2. 生成控制命令
3. 发送控制命令

生成控制命令的核心函数在”ProduceControlCommand”中，其实这个函数还包含了参数检查和_estop(紧急情况)的处理，真正生成命令的函数只有一行：

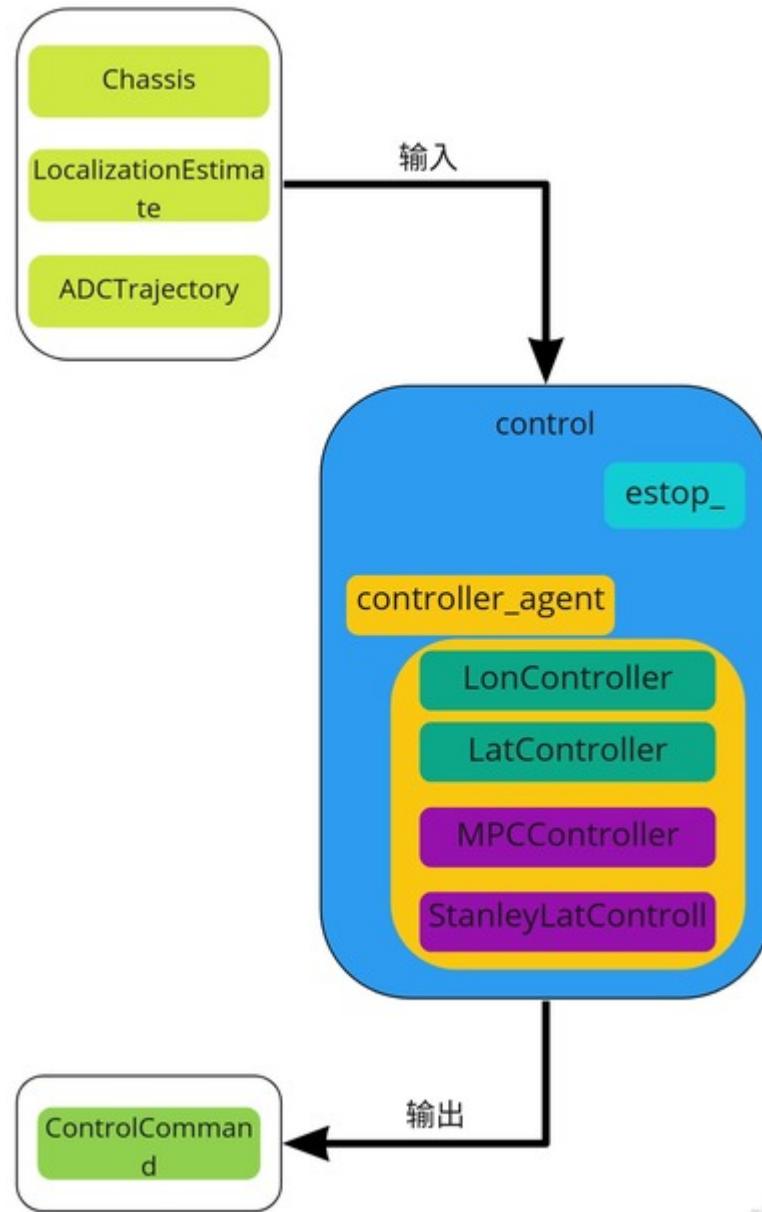
```

Status status_compute =
controller_agent_.ComputeControlCommand(
    &local_view_.localization, &local_view_.chassis,
    &local_view_.trajectory, control_command);

```

5.2.1. 控制器注册

上述过程是通过”controller_agent_”产生控制命令，而”controller_agent_”是一个代理，其它的控制器通过注册到代理控制器，从而实现多个控制器的访问（是否用到了设计模式的代理模



式？）：
包括：

- 纵向控制
- 横向控制
- 模型预测控制
- Stanley横向控制

Stanley横向控制可以参考

[https://www.ri.cmu.edu/pub_files/2009/2/Automatic_Steering_Methods_for_Autonomous_AutoMobile_Path_Tracking.pdf]

那么这些控制器是如何注册到“controller_agent_”的呢？

```
// 1. 在ControlComponent::Init()中调用初始化
bool ControlComponent::Init() {
    // 注册控制器
    if (!controller_agent_.Init(&control_conf_).ok()) {
        monitor_logger_buffer_.ERROR("Control init controller failed! Stopping...");
        return false;
    }
}

// 2. 在ControllerAgent类中注册和初始化
Status ControllerAgent::Init(const ControlConf *control_conf)
{
    // 注册控制器
    RegisterControllers(control_conf);
    // 实例化控制器
    CHECK(InitializeConf(control_conf).ok()) << "Fail to initialize config.";
    // 控制器初始化
    for (auto &controller : controller_list_) {
        if (controller == NULL || !controller->Init(control_conf).ok()) {
            if (controller != NULL) {
                AERROR << "Controller <" << controller->Name() << "> init failed!";
                return Status(ErrorCode::CONTROL_INIT_ERROR,
                             "Failed to init Controller:" +
                             controller->Name());
            } else {
                return Status(ErrorCode::CONTROL_INIT_ERROR,
                             "Failed to init Controller");
            }
        }
        AINFO << "Controller <" << controller->Name() << "> init done!";
    }
}
```

```
    return Status::OK();
}
```

上面就是control模块的主流程，我们接下来先介绍下control模块中的”pad message”和”estop_”，然后再逐个介绍各个控制器。

5.2.2. Pad消息

Pad消息通过发送状态来控制汽车的模式（自动驾驶还是人工驾驶），其中”DrivingAction”一共有3种状态，在”pad_msg.proto”中：

```
enum DrivingAction {
    STOP = 0;
    START = 1;
    RESET = 2;
};
```

Control模块中只能使用pad消息”RESET”来清空”estop_”的状态。实际上control模块会判断”driving_mode”来决定是否启动自动驾驶，而”driving_mode”是通过发送pad消息状态给”canbus”模块来控制的，这会在canbus模块中详细介绍。总之pad消息的2个作用是：

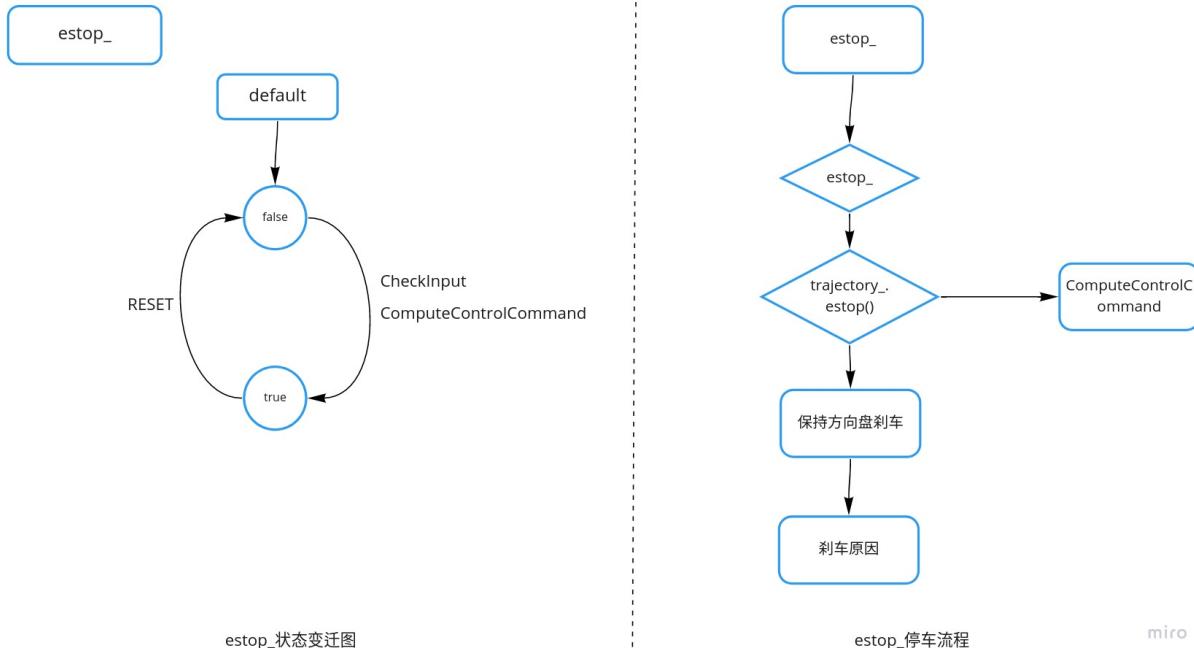
1. 发送消息给canbus模块，来控制**driving_mode**，control模块判断当前**driving_mode**的状态来决定是否启动自动驾驶
2. 通过**reset**来清空**estop_**的状态

5.2.3. estop_ 标志位

那么”estop_”有什么作用呢？estop_标志位的作用是判断control模块是否处于紧急状态，而触发紧急停车。那么在哪些状态下，”estop_”为真，汽车进入紧急停车状态呢？

1. 输入错误(CheckInput返回false)
2. 控制命令计算失败(ComputeControlCommand失败)
3. planning模块直接给出紧急停车

下面是“estop_”的状态变迁图和紧急停车流程：



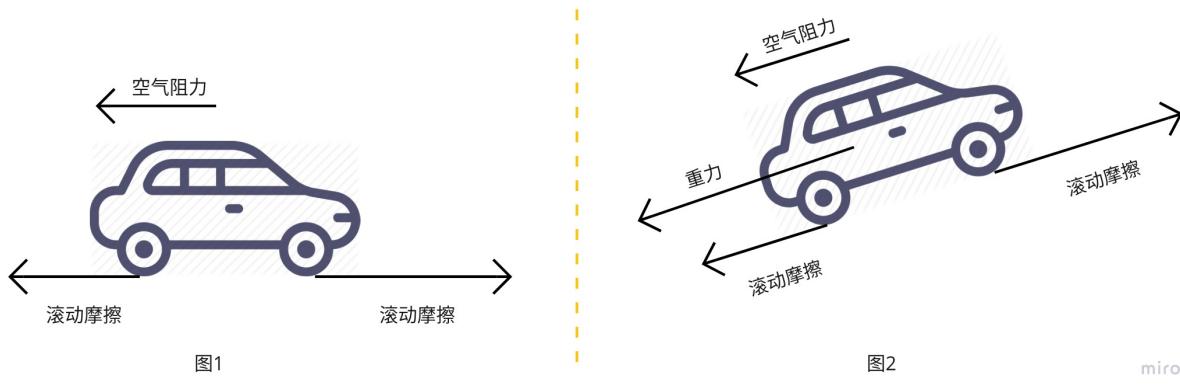
紧急停车的流程很好理解，我们学驾照的时候都知道，紧急停车的时候不能够狂打方向盘，特别是高速的时候，容易侧翻，因此这里的紧急停车流程也是保持方向盘，然后紧急制动。

5.3. 控制器

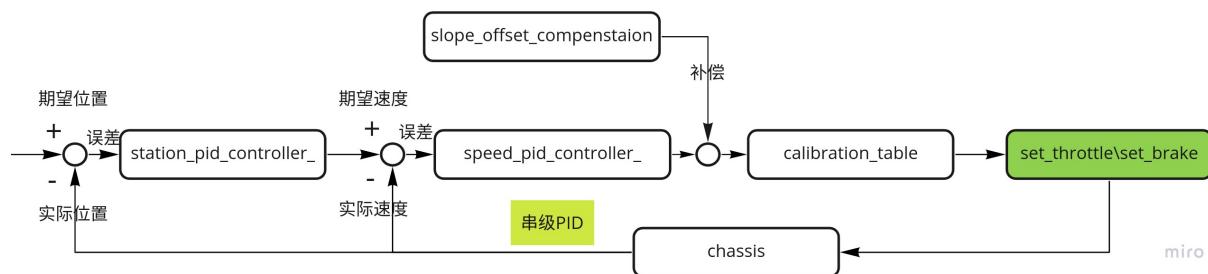
首先我们把汽车的控制分为横向控制和纵向控制2部分，其中纵向控制通过油门和刹车控制车纵向的加减速，而横向控制则通过控制方向盘的转动来控制前轮的方向，从而控制汽车的行驶方向。

5.3.1. LonController控制器

纵向控制器，主要是通过油门和刹车控制车的纵向速度。汽车的纵向模型比较简单，我们可以通过经典力学对汽车进行建模，汽车的纵向受力如果是前轮驱动，那么前轮提供一个向前的滚动摩擦，后轮有一个滚动摩擦阻力再加上风阻，如果有坡度则再加上重力分量，这就构成了汽车的纵向受力模型：



纵向控制要实现的目标是让汽车在指定的时间内到达指定的地点，首先是保证位置准确，如果能够直接通过油门去控制汽车的位置，那么采用单一的PID环就够了。重点是位置是和速度与时间相关的，因此先通过位置PID得到需要的速度，然后再通过速度PID得到需要多大油门，这样的方式叫做串级PID控制器。通过2级PID来控制汽车的速度，从而确定需要踩多少的油门。现在自动驾驶普遍是电动汽车的情况下，汽车的动力实际上由内燃机换成了电动马达，这种控制方式完全可以由伺服控制器来解决，传统的伺服控制器实现了位置控制，速度控制，可以把汽车的控制模块直接由伺服控制器来替代（也许可以解放汽车的控制，如果是线性的电机，那么其实很好办，油门可以直接对应为电机的电流，而电流对应电机输出的扭矩大小），但汽车有个最大的不同点是后退需要换倒车档。下面是串级PID的示意图，其中先根据位置误差得到速度，然后根据速度误差得到油门和刹车，同时在汽车有俯仰角度（汽车行驶在有坡度的路面上）这时候会对PID提供重力分量的补偿：



5.3.2. 校准表

知道速度之后我们可以根据当前速度的误差，来确定是增大油门或者是刹车，而油门和速度的关系是如何对应的呢，举个例子：当你发现车的速度和预期相差为3km/h，你应该踩多大油门呢？当然你可以先假设一个值，比如油门增加2，如果速度低了你再增加到3，总之根据一个经验值，然后再通过PID算法去调节，那怎么去拟合速度和油门的关系曲线呢？我们需要控制汽车到达某个速度，根据牛顿经典力学，只需要知道汽车的初速度和加速度，就可以知道物体一段时候后的速度。因此我们只要找到速度，加速度和油门的关系，就可以通过控制汽车的加速度来让汽车达到某个速度。也就是对速度加速度和油门的关系进行建模，得到它们之间的关系。Apollo采用的是在实际的汽车行驶过程中记录不同速度下，不同的油门值对汽车的加速度的影响，从而得到一张表格，最后通过查表的方式来得到具体的油门和刹车值大小，得到的配置最后保存在conf文件夹中。

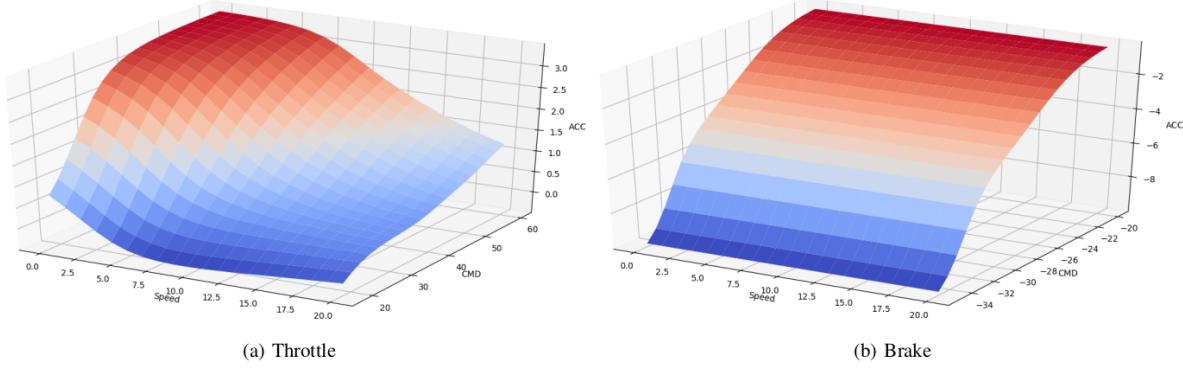
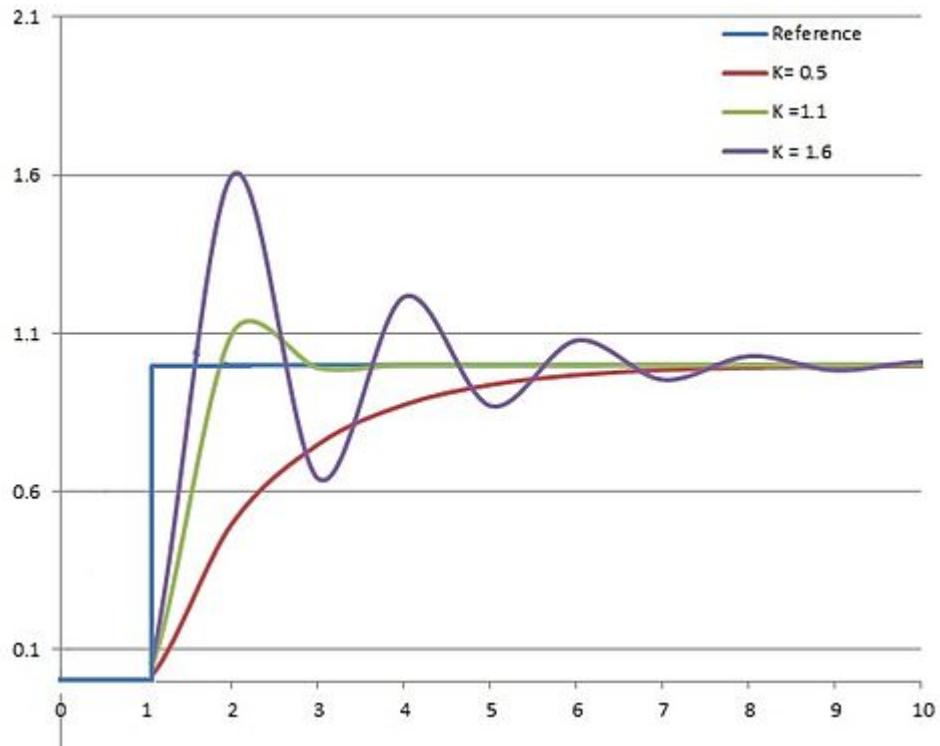


Fig. 2: Offline calibration table 3D view

这个表的生成需要测试每种油门以及每种速度下的表现，但是表不可能列出无限连续的数据，因此最后还是需要通过插值的方式来得到结果。

5.3.3. PID参数调节

什么样的PID参数比较合适？这一部分可以参考WIKI百科的PID参数动画。调试参数的方法是先调节速度环，再调节位置环；先调节比例系数，再调节积分系数。



另外关于PID的控制，正常情况下没有问题，如果遇到路面不平的情况，相当于一张巨大的手在拨弄小车，这样就对PID引入了一个震荡反馈，震荡的引入之后系统是否能够保持稳态呢，这就是PID控制需要研究的范围？？？

5.4. LatController控制器

纵向控制主要是控制速度，而横向控制主要是控制方向。方向盘的角度不一样，车的行驶路径不一样，因此汽车的横向控制主要是通过控制方向盘的转角来控制汽车行驶的角度。纵向控制采用的是**LQR**控制器，为什么不继续采用PID控制器呢？因为每种控制器都有它的适合情况和不适合情况（有一篇文章专门介绍了不同的控制器的优劣，PID主要的缺点是不适合泊车和曲率比较大的情况）。关于LQR控制器主要是一些公式的推导，后面再详细介绍下推导过程。

5.5. MPCController控制器

模型预测控制(MPC)是一类特殊的控制。它的当前控制动作是在每一个采样瞬间通过求解一个有限时域开环最优控制问题而获得。过程的当前状态作为最优控制问题的初始状态，解得的最优控制序列只实施第一个控制作用。这是它与那些使用预先计算控制律的算法的最大不同。本质上模型预测控制求解一个开环最优控制问题。它的思想与具体的模型无关，但是实现则与模型有关。

5.6. StanleyLatController控制器

5.7. 问题

1. 遇到上坡\下坡的情况，原来的系数表就只能提供一个参考，如何提供补偿，PID的每次输入都是从表中查询还是根据调节的结果来的？
2. 遇到下雨的时候，路面的摩擦系数改变和上述问题一样
3. 遇到长下坡，PID调节是否会和人一样“不能长时间踩脚踏板”
4. 转弯的场景是否适合这个参数？
5. 是否所有的场景都适合这个参数？比如自主泊车的情况？
6. 提供的是TrajectoryPoint，里面包含距离，速度，加速度和时间，这3者只要知道2者即可以了。控制的时候如何选择的颗粒度？比如提供的轨迹的时间间隔是否固定，是否每次都需要control模块自己调节（在时间间隔比较长的情况下），PID每次只能参考一个标准，这里是以距离为准，还是以速度为准？如果是，速度还有何意义？planning生成的TrajectoryPoint是否符合物理学规律？

5.8. Reference

- [Apollo代码学习\(二\)——控制模块概述](#)
[<https://blog.csdn.net/u013914471/article/details/82775091>]
- [百度Apollo 2.0 车辆控制算法之LQR控制算法解读](#)
[<https://blog.csdn.net/weijimin1/article/details/85794084>]
- [Apollo代码学习\(五\)——横纵向控制](#)
[<https://blog.csdn.net/u013914471/article/details/83748571>]
- [Apollo自动驾驶入门课程第⑩讲 — 控制 \(下\)](#)
[<https://blog.csdn.net/cg129054036/article/details/83413482>]

- [how_to_tune_control_parameters](#) [https://github.com/ApolloAuto/apollo/blob/master/docs/howto/how_to_tune_control_parameters.md]
- [throttle-affect-the-rpm-of-an-engine](#) [<https://www.physicsforums.com/threads/how-does-the-throttle-affect-the-rpm-of-an-engine.832029/>]
- [PID_controller](#) [https://en.wikipedia.org/wiki/PID_controller]
- [Stanley横向控制](#) [https://www.ri.cmu.edu/pub_files/2009/2/Automatic_Steering_Methods_for_Autonomous_Auto_mobile_Path_Tracking.pdf]
- [Linear-quadratic regulator](#) [https://en.wikipedia.org/wiki/Linear%E2%80%93quadratic_regulator]
- [模型预测控制](#) [<https://baike.baidu.com/item/%E6%A8%A1%E5%9E%8B%E9%A2%84%E6%B5%8B%E6%8E%A7%E5%88%B6>]

6. Data

业精于勤，荒于嬉；行成于思，毁于随。

6.1. data目录结构

data目录的结构如下，

```
.  
├── BUILD  
├── README.md  
├── conf  
├── proto  
├── channel_pool.cc  
├── channel_pool.h          // 设置small_channels_  
├── drive_event_trigger.cc  
├── drive_event_trigger.h  
├── emergency_mode_trigger.cc  
├── emergency_mode_trigger.h  
├── hard_brake_trigger.cc  
├── hard_brake_trigger.h  
├── interval_pool.cc  
├── interval_pool.h  
├── post_record_processor.cc  
├── post_record_processor.h  
├── realtime_record_processor.cc  
├── realtime_record_processor.h  
├── record_processor.cc  
├── record_processor.h  
├── regular_interval_trigger.cc  
├── regular_interval_trigger.h  
├── small_topics_trigger.cc  
├── small_topics_trigger.h  
├── smart_recorder.cc  
├── smart_recorder_gflags.cc  
├── smart_recorder_gflags.h  
└── swerve_trigger.cc  
    └── swerve_trigger.h
```

```
└─ trigger_base.cc  
└─ trigger_base.h
```

主要的实现分为2部分，一部分为trigger，一部分为record。trigger主要的作用是生成对应的时间段，并且放到pool中。record主要用来录制数据，分为在线和离线录制。主要是调用RecordWriter和RecordReader，这2个类实现实时读取。

6.2. RecordViewer

RecordViewer实现了迭代器，并且可以读取文件和实时的流式数据？？RecordViewer中包含RecordReader

```
reader->GetHeader()
```

以上的原理是什么？？如果是文件则从文件读取，如果不是文件应该如何？？

RecordViewer通过Iterator读取msg_buffer_中的消息，通过FillBuffer往msg_buffer_中写入消息，注意这里的”std::multimap”为什么要用map因为每次读取的时候是读取的1s内的reader消息，遍历多个reader的时候，那么时间顺序可能会打乱，因此这里采用红黑树的方式按照key进行排序，所以采用了multimap的结构。

每次在Update中更新消息，

```
bool RecordViewer::Update(RecordMessage* message) {  
    bool find = false;  
    do {  
        // 1. 如果msg_buffer_为空，则填充buffer，填充1s内的消息  
        if (msg_buffer_.empty() && !FillBuffer()) {  
            break;  
        }  
        // 2. 找到buffer中的第一条消息，并且退出  
        auto& msg = msg_buffer_.begin()->second;  
        if (channels_.empty() || channels_.count(msg->channel_name) == 1) {  
            *message = *msg;  
            find = true;
```

```

    }
    msg_buffer_.erase(msg_buffer_.begin());
} while (!find);

return find;
}

```

下面我们看下Process的过程，也就是说实时过程先启动recorder_，然后再从文件中读取消息，然后再重新保存？？？

```

bool RealtimeRecordProcessor::Process() {
    // 保存到文件
    recorder_->Start();

    // Now fast reader follows and reacts for any events
    std::string record_path;
    do {
        if (!GetNextValidRecord(&record_path)) {
            break;
        }
        auto reader = std::make_shared<RecordReader>
(record_path);
        RecordViewer viewer(reader, 0,
std::numeric_limits<uint64_t>::max(),
ChannelPool::Instance()-
> GetAllChannels());

        if (restore_reader_time_ == 0) {
            restore_reader_time_ = viewer.begin_time();
            GetNextValidRecord(&restore_path_);
        }
        // 迭代器++调用RecordViewer::Update读取消息
        for (const auto& msg : viewer) {
            for (const auto& trigger : triggers_) {
                trigger->Pull(msg);
            }
            // 保存消息
            RestoreMessage(msg.time());
        }
    } while (!is_terminating_); // 循环上述步骤
    // Try restore the rest of messages one last time
    RestoreMessage(std::numeric_limits<uint64_t>::max());
}

```

```

    if (monitor_thread && monitor_thread->joinable()) {
        monitor_thread->join();
        monitor_thread = nullptr;
    }
    return true;
}

```

Recorder开始Start

```

bool Recorder::Start() {
    // 1. 创建文件
    writer_.reset(new RecordWriter(header_));
    if (!writer_->Open(output_)) {
        return false;
    }
    std::string node_name = "cyber_recorder_record_" +
    std::to_string(getpid());
    node_ = ::apollo::cyber::CreateNode(node_name);
    // 2. 初始化reader
    if (!InitReadersImpl()) {
        return false;
    }
    message_count_ = 0;
    message_time_ = 0;
    is_started_ = true;

    // 3. 显示进度
    display_thread_ =
        std::make_shared<std::thread>([this]() { this-
>ShowProgress(); });
}

return true;
}

```

InitReadersImpl

```

bool Recorder::InitReadersImpl() {
    std::shared_ptr<ChannelManager> channel_manager =
        TopologyManager::Instance()->channel_manager();

    // get historical writers
    std::vector<proto::RoleAttributes> role_attr_vec;

```

```

channel_manager->GetWriters(&role_attr_vec);
for (auto role_attr : role_attr_vec) {
    FindNewChannel(role_attr);
}

// listen new writers in future
change_conn_ = channel_manager->AddChangeListener(
    std::bind(&Recorder::TopologyCallback, this,
std::placeholders::_1));
if (!change_conn_.IsConnected()) {
    AERROR << "change connection is not connected";
    return false;
}
return true;
}

```

InitReaderImpl 注册callback给reader， 然后写入文件

```

bool Recorder::InitReaderImpl(const std::string&
channel_name,
                               const std::string&
message_type) {
    try {
        std::weak_ptr<Recorder> weak_this = shared_from_this();
        std::shared_ptr<ReaderBase> reader = nullptr;
        auto callback = [weak_this, channel_name](
                           const std::shared_ptr<RawMessage>&
raw_message) {
            auto share_this = weak_this.lock();
            if (!share_this) {
                return;
            }
            share_this->ReaderCallback(raw_message, channel_name);
        };
        ReaderConfig config;
        config.channel_name = channel_name;
        config.pending_queue_size =
            gflags::Int32FromEnv("CYBER_PENDING_QUEUE_SIZE", 50);
        reader = node_->CreateReader<RawMessage>(config,
callback);
        if (reader == nullptr) {
            AERROR << "Create reader failed.";
            return false;
        }
    }
}

```

```
    }
    channel_reader_map_[channel_name] = reader;
    return true;
} catch (const std::bad_weak_ptr& e) {
    AERROR << e.what();
    return false;
}
}
```

7. Dreamview

草木有本心，何求美人折。

dreamview中enable模块是在 backend的hmi接口中实现，调用std::cmd()
执行启动命令。

8. Drivers

一叶遮目，不见泰山

8.1. Table of Contents

Canbus驱动子模块介绍 Radar驱动子模块介绍 Camera驱动子模块介绍
velodyne驱动子模块介绍 hesai驱动子模块介绍 Gnss驱动子模块介绍

8.2. Reference

- [linux](https://www.kernel.org/) [https://www.kernel.org/]

9. Guardian

人不知而不愠，不亦君子乎。

9.1. 简介

Guardian模块的主要作用是监控自动驾驶系统状态，当出现模块为失败状态的时候，会主动切断控制命令输出，并且刹车。

9.2. 触发

guardian模块的触发条件主要有2个。

1. 上报模块状态的消息间隔超过kSecondsTillTimeout（2.5秒）
2. 上报的状态消息中有safety_mode_trigger_time字段 这时候就会触发进入接管。

9.3. TriggerSafetyMode

安全模式的步骤分为2步骤，第一步状态消息中需要紧急刹车或者超声波检测到障碍物，如果检测到障碍物则说明车已经非常接近障碍物了，该检测被称为硬件触发的检测，因为已经发现模块故障，也非常接近障碍物所以刹车会加速。第二步是普通刹车，刹车没有那么急。guardian为定时模块，所以该过程中会一直发送消息，直到车辆停车。当前版本代码屏蔽了上述超声波检测。

9.4. 问题

guardian模块的频率是10ms，因此最大会增加control命令的延时10ms。

10. Localization

虽千万人，吾往矣。

10.1. Localization模块简介

localization模块主要实现了以下2个功能：

1. 输出车辆的位置信息（planning模块使用）
2. 输出车辆的姿态，速度信息（control模块使用）

其中apollo代码中分别实现了3种定位方法：

1. GNSS + IMU定位
2. NDT定位（点云定位）
3. MSF（融合定位）

MSF方法参考论文”Robust and Precise Vehicle Localization Based on Multi-Sensor Fusion in Diverse City Scenes”

10.2. 代码目录

下面是localization的目录结构，在查看具体的代码之前最好看下定位模块的readme文件：

```
├── common          // 声明配置(flags)，从conf目录中读取相应的值
├── conf            // 配置文件存放目录
├── dag             // cyber DAG流
├── launch          // cyber的配置文件，依赖DAG图（这2个和cyber有关
  的后面再分析）
└── msf             // 融合定位(gnss, 点云, IMU融合定位)
    ├── common
    │   ├── io
    │   └── test_data
    └── util
```

```

    |-- local_integ
    |-- local_map
        |-- base_map
        |-- lossless_map
        |-- lossy_map
        |-- ndt_map
        |-- test_data
    |-- local_tool
        |-- data_extraction
        |-- local_visualization
        |-- map_creation
    └── params
        ├── gnss_params
        ├── vehicle_params
        └── velodyne_params
    └── ndt                         // ndt定位
        ├── map_creation
        ├── ndt_locator
        └── test_data
            └── ndt_map
            └── pcds
    └── proto                        // 消息格式
    └── rtk                          // rtk定位
    └── testdata                     // imu和gps的测试数据

```

通过上述目录可以知道，定位模块主要实现了rtk, ndt, msf这3个定位方法，分别对应不同的目录。proto文件夹定义了消息的格式，common和conf主要是存放一些配置和消息TOPIC。下面我们逐个分析RTK定位、NDT定位和MSF定位。

10.3. RTK定位流程

RTK定位是通过GPS和IMU的信息做融合然后输出车辆所在的位置。RTK通过基准站的获取当前GPS信号的误差，用来校正无人车当前的位置，可以得到厘米级别的精度。IMU的输出频率高，可以在GPS没有刷新的情况下（通常是1s刷新一次）用IMU获取车辆的位置。下面是RTK模块的目录结构。

```

    |-- BUILD                         // bazel编译文件
    |-- rtk_localization.cc          // rtk定位功能实现模块

```

```
|-- rtk_localization_component.cc          // rtk消息发布模块
|-- rtk_localization_component.h
|-- rtk_localization.h
└-- rtk_localization_test.cc             // 测试
```

其中”rtk_localization_component.cc”注册为标准的cyber模块，RTK定位模块在”Init”中初始化，每当接收到”localization::Gps”消息就触发执行”Proc”函数。

```
class RTKLocalizationComponent final
    : public cyber::Component<localization::Gps> {
public:
    RTKLocalizationComponent();
    ~RTKLocalizationComponent() = default;

    bool Init() override;

    bool Proc(const std::shared_ptr<localization::Gps>
&gps_msg) override;
```

下面我们分别查看这2个函数。

1. Init函数 Init函数实现比较简单，一是初始化配置信息，二是初始化IO。初始化配置信息主要是读取一些配置，例如一些topic信息等。下面主要看下初始化IO。

```
bool RTKLocalizationComponent::InitIO() {
    // 1. 读取IMU信息，每次接收到localization::CorrectedImu消息，则回调执行“RTKLocalization::ImuCallback”
    corrected_imu_listener_ = node_
>CreateReader<localization::CorrectedImu>(
        imu_topic_, std::bind(&RTKLocalization::ImuCallback,
localization_.get(),
                           std::placeholders::_1));
    CHECK(corrected_imu_listener_);
    // 2. 读取GPS状态信息，每次接收到GPS状态消息，则回调执行“RTKLocalization::GpsStatusCallback”
    gps_status_listener_ = node_
>CreateReader<drivers::gnss::InsStat>(
        gps_status_topic_,
std::bind(&RTKLocalization::GpsStatusCallback,
```

```

        localization_.get(),
std::placeholders::_1));
CHECK(gps_status_listener_);

// 3.发布位置信息和位置状态信息
localization_talker_ =
    node_->CreateWriter<LocalizationEstimate>
(localization_topic_);
CHECK(localization_talker_);

localization_status_talker_ =
    node_->CreateWriter<LocalizationStatus>
(localization_status_topic_);
CHECK(localization_status_talker_);
return true;
}

```

也就是说，RTK模块同时还接收IMU和GPS的状态信息，然后触发对应的回调函数。具体的实现在”RTKLocalization”类中，我们先看下回调的具体实现。以”GpsStatusCallback”为例，每次读取到gps状态信息之后，会把信息保存到”gps_status_list_”列表中。”ImuCallback”类似，也是接收到IMU消息后，保存到”imu_list_”列表中。

```

void RTKLocalization::GpsStatusCallback(
    const std::shared_ptr<drivers::gnss::InsStat>
&status_msg) {
    std::unique_lock<std::mutex> lock(gps_status_list_mutex_);
    if (gps_status_list_.size() < gps_status_list_max_size_) {
        gps_status_list_.push_back(*status_msg);
    } else {
        gps_status_list_.pop_front();
        gps_status_list_.push_back(*status_msg);
    }
}

```

2. Proc 在每次接收到”localization::Gps”消息后，触发执行”Proc”函数。这里注意如果需要接收多个消息，这里是3个消息，则选择最慢的消息作为触发，否则，如果选择比较快的消息作为触发，这样会导致作为触发的消息刷新了，而其它的消息还没有刷新。所以这里采用的是GPS消息作为触发消息，IMU的消息刷新快。下面我们看具体的实现。

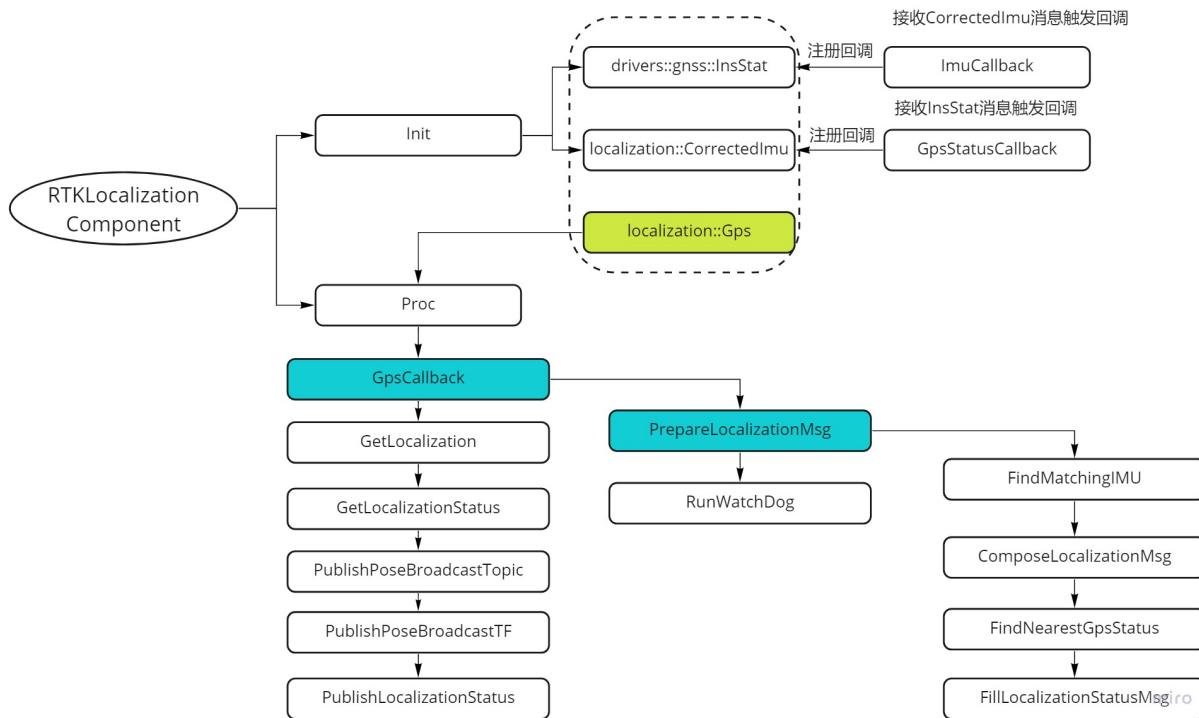
```
bool RTKLocalizationComponent::Proc(
    const std::shared_ptr<localization::Gps>& gps_msg) {
    // 1. 通过RTKLocalization处理GPS消息回调
    localization_->GpsCallback(gps_msg);

    if (localization_->IsServiceStarted()) {
        LocalizationEstimate localization;
        // 2. 获取定位消息
        localization_->GetLocalization(&localization);
        LocalizationStatus localization_status;
        // 3. 获取定位状态
        localization_-
    >GetLocalizationStatus(&localization_status);

        // publish localization messages
        // 4. 发布位置信息
        PublishPoseBroadcastTopic(localization);
        // 5. 发布位置转换信息
        PublishPoseBroadcastTF(localization);
        // 6. 发布位置状态信息
        PublishLocalizationStatus(localization_status);
        ADEBUG << "[OnTimer]: Localization message publish
success!";
    }

    return true;
}
```

具体的执行过程如下图所示。



主要的执行过程在”GpsCallback”中，然后通过”GetLocalization”和”GetLocalizationStatus”获取结果，最后发布对应的位置信息、位置转换信息和位置状态信息。由于”GpsCallback”主要执行过程在”PrepareLocalizationMsg”中，因此我们主要分析”PrepareLocalizationMsg”的实现。

10.3.1. 获得定位信息

PrepareLocalizationMsg函数的具体实现如下。

```

void RTKLocalization::PrepareLocalizationMsg(
    const localization::Gps &gps_msg, LocalizationEstimate
*localization,
    LocalizationStatus *localization_status) {
    // find the matching gps and imu message
    double gps_time_stamp = gps_msg.header().timestamp_sec();
    CorrectedImu imu_msg;
    // 1. 寻找最匹配的IMU信息
    FindMatchingIMU(gps_time_stamp, &imu_msg);
    // 2. 根据GPS和IMU信息，给位置信息赋值
    ComposeLocalizationMsg(gps_msg, imu_msg, localization);
}
    
```

```

    drivers::gnss::InsStat gps_status;
    // 3.查找最近的GPS状态信息
    FindNearestGpsStatus(gps_time_stamp, &gps_status);
    // 4.根据GPS状态信息，给位置状态信息赋值
    FillLocalizationStatusMsg(gps_status, localization_status);
}

```

下面我们逐个分析上述4个过程。

10.3.2. FindMatchingIMU

在队列中找到最匹配的IMU消息，其中区分了队列的第一个，最后一个，以及如果在中间位置则进行插值。插值的时候根据距离最近的原则进行反比例插值。

```

bool RTKLocalization::FindMatchingIMU(const double
                                      gps_timestamp_sec,
                                      CorrectedImu *imu_msg)
{
    // 加锁，这里有疑问，为什么换个变量就没有锁了呢？
    std::unique_lock<std::mutex> lock(imu_list_mutex_);
    auto imu_list = imu_list_;
    lock.unlock();

    // 在IMU队列中找到最新的IMU消息
    // scan imu buffer, find first imu message that is newer
    // than the given
    // timestamp
    auto imu_it = imu_list.begin();
    for (; imu_it != imu_list.end(); ++imu_it) {
        if ((*imu_it).header().timestamp_sec() -
gps_timestamp_sec >
            std::numeric_limits<double>::min()) {
            break;
        }
    }

    if (imu_it != imu_list.end()) { // found one
        if (imu_it == imu_list.begin()) {

```

```

        AERROR << "IMU queue too short or request too old. "
        << "Oldest timestamp[" <<
imu_list.front().header().timestamp_sec()
        << "], Newest timestamp["
        << imu_list.back().header().timestamp_sec() <<
"], GPS timestamp["
        << gps_timestamp_sec << "]";
*imu_msg = imu_list.front(); // the oldest imu
} else {
    // here is the normal case
    auto imu_it_1 = imu_it;
    imu_it_1--;
    if (!(*imu_it).has_header() || !
(*imu_it_1).has_header()) {
        AERROR << "imul and imu_it_1 must both have header.";
        return false;
    }
    // 根据最新的IMU消息和它之前的消息做插值。
    if (!InterpolateIMU(*imu_it_1, *imu_it,
gps_timestamp_sec, imu_msg)) {
        AERROR << "failed to interpolate IMU";
        return false;
    }
}
else {
    // 如果没有找到，则取最新的20ms以内的消息，如果超过20ms则报错。
    // give the newest imu, without extrapolation
*imu_msg = imu_list.back();
if (imu_msg == nullptr) {
    AERROR << "Fail to get latest observed imu_msg.";
    return false;
}

if (!imu_msg->has_header()) {
    AERROR << "imu_msg must have header.";
    return false;
}

if (std::fabs(imu_msg->header().timestamp_sec() -
gps_timestamp_sec) >
    gps_imu_time_diff_threshold_) {
    // 20ms threshold to report error
    AERROR << "Cannot find Matching IMU. IMU messages too

```

```

old. "
        << "Newest timestamp[" <<
imu_list.back().header().timestamp_sec()
        << "], GPS timestamp[" << gps_timestamp_sec <<
"]";
    }
}

return true;
}

```

接下来我们看线性插值

1. InterpolateIMU 根据上述函数得到2个IMU消息分别对角速度、线性加速度、欧拉角进行插值。原则是根据比例，反比例进行插值。

```

bool RTKLocalization::InterpolateIMU(const CorrectedImu
&imu1,
                                      const CorrectedImu
&imu2,
                                      const double
timestamp_sec,
                                      CorrectedImu *imu_msg) {

    if (timestamp_sec - imu1.header().timestamp_sec() <
        std::numeric_limits<double>::min()) {
        AERROR << "[InterpolateIMU1]: the given time stamp[" <<
timestamp_sec
            << "] is older than the 1st message["
            << imu1.header().timestamp_sec() << "]";
        *imu_msg = imu1;
    } else if (timestamp_sec - imu2.header().timestamp_sec() >
        std::numeric_limits<double>::min()) {
        AERROR << "[InterpolateIMU2]: the given time stamp[" <<
timestamp_sec
            << "] is newer than the 2nd message["
            << imu2.header().timestamp_sec() << "]";
        *imu_msg = imu2;
    } else {
        // 线性插值
        *imu_msg = imu1;
    }
}

```

```

    imu_msg->mutable_header()-
>set_timestamp_sec(timestamp_sec);

    double time_diff =
        imu2.header().timestamp_sec() -
imu1.header().timestamp_sec();
    if (fabs(time_diff) >= 0.001) {
        double frac1 =
            (timestamp_sec - imu1.header().timestamp_sec()) /
time_diff;
        // 1. 分别对角速度、线性加速度、欧拉角进行插值
        if (imu1 imu().has_angular_velocity() &&
            imu2 imu().has_angular_velocity()) {
            auto val =
InterpolateXYZ(imu1 imu().angular_velocity(),
imu2 imu().angular_velocity(), frac1);
            imu_msg->mutable_imu()->mutable_angular_velocity()-
>CopyFrom(val);
        }
        ...
    }
    return true;
}

```

2. InterpolateXYZ 根据距离插值，反比例，即frac1越小，则越靠近p1，frac1越大，则越靠近p2

```

template <class T>
T RTKLocalization::InterpolateXYZ(const T &p1, const T &p2,
                                    const double frac1) {
    T p;
    double frac2 = 1.0 - frac1;
    if (p1.has_x() && !std::isnan(p1.x()) && p2.has_x() &&
!std::isnan(p2.x())) {
        p.set_x(p1.x() * frac2 + p2.x() * frac1);
    }
    if (p1.has_y() && !std::isnan(p1.y()) && p2.has_y() &&
!std::isnan(p2.y())) {
        p.set_y(p1.y() * frac2 + p2.y() * frac1);
    }
}

```

```

    }

    if (p1.has_z() && !std::isnan(p1.z()) && p2.has_z() &&
!std::isnan(p2.z())))
    {
        p.set_z(p1.z() * frac2 + p2.z() * frac1);
    }
    return p;
}

```

10.3.3. ComposeLocalizationMsg

填充位置信息，这里实际上涉及到姿态解算，具体是根据GPS和IMU消息对位置信息进行赋值。需要注意需要根据航向对IMU的信息进行转换。

```

void RTKLocalization::ComposeLocalizationMsg(
    const localization::Gps &gps_msg, const
localization::CorrectedImu &imu_msg,
    LocalizationEstimate *localization) {
localization->Clear();

FillLocalizationMsgHeader(localization);

localization-
>set_measurement_time(gps_msg.header().timestamp_sec());

// combine gps and imu
auto mutable_pose = localization->mutable_pose();
// GPS消息包含位置信息
if (gps_msg.has_localization()) {
    const auto &pose = gps_msg.localization();
    // 1. 获取位置
    if (pose.has_position()) {
        // position
        // world frame -> map frame
        mutable_pose->mutable_position()-
>set_x(pose.position().x() -
map_offset_[0]);
        mutable_pose->mutable_position()-
>set_y(pose.position().y() -
map_offset_[1]);
    }
}
}

```

```

        mutable_pose->mutable_position()-
>set_z(pose.position().z() -
map_offset_[2]);
    }
    // 2. 获取方向
    // orientation
    if (pose.has_orientation()) {
        mutable_pose->mutable_orientation()-
>CopyFrom(pose.orientation());
        double heading = common::math::QuaternionToHeading(
            pose.orientation().qw(), pose.orientation().qx(),
            pose.orientation().qy(), pose.orientation().qz());
        mutable_pose->set_heading(heading);
    }
    // linear velocity
    // 3. 获取速度
    if (pose.has_linear_velocity()) {
        mutable_pose->mutable_linear_velocity()-
>CopyFrom(pose.linear_velocity());
    }
}

if (imu_msg.has_imu()) {
    const auto &imu = imu_msg.imu();
    // linear acceleration
    // 4. 获取imu的线性加速度
    if (imu.has_linear_acceleration()) {
        if (localization->pose().has_orientation()) {
            // linear_acceleration:
            // convert from vehicle reference to map reference
            // 为什么需要做旋转? ? ? 转换为车当前方向的速度? ? ?
            Vector3d orig(imu.linear_acceleration().x(),
                          imu.linear_acceleration().y(),
                          imu.linear_acceleration().z());
            Vector3d vec = common::math::QuaternionRotate(
                localization->pose().orientation(), orig);
            mutable_pose->mutable_linear_acceleration()-
>set_x(vec[0]);
            mutable_pose->mutable_linear_acceleration()-
>set_y(vec[1]);
            mutable_pose->mutable_linear_acceleration()-
>set_z(vec[2]);
        }
    }
}

```

```

    // linear_acceleration_vfr
    // 设置线性加速度
    mutable_pose->mutable_linear_acceleration_vrf()-
>CopyFrom(
    imu.linear_acceleration());
} else {
    AERROR << "[PrepareLocalizationMsg]: "
        << "fail to convert linear_acceleration";
}
}

// 5. 设置角速度，也需要根据航向转换
// angular velocity
...
// 6. 设置欧拉角
// euler angle
if (imu.has_euler_angles()) {
    mutable_pose->mutable_euler_angles()-
>CopyFrom(imu.euler_angles());
}
}
}
}

```

10.3.4. FindNearestGpsStatus

获取最近的Gps状态信息，这里实现的算法是遍历查找
“gps_time_stamp”最近的状态，GPS状态信息不是按照时间顺序排列的？？？

```

bool RTKLocalization::FindNearestGpsStatus(const double
gps_timestamp_sec,

drivers::gnss::InsStat *status) {

    ...
// 1. 遍历查找最近的GPS状态信息
double timestamp_diff_sec = 1e8;
auto nearest_itr = gps_status_list.end();
for (auto itr = gps_status_list.begin(); itr !=
gps_status_list.end();
++itr) {

```

```

        double diff = std::abs(itr->header().timestamp_sec() -
gps_timestamp_sec);
        if (diff < timestamp_diff_sec) {
            timestamp_diff_sec = diff;
            nearest_itr = itr;
        }
    }
...
}

```

10.3.5. FillLocalizationStatusMsg

获取位置状态，一共有3种状态：稳定状态(INS_RTKFIXED)、浮动状态(INS_RTKFLOAT)、错误状态(ERROR)。由于代码比较简单，这里就不分析了。

10.3.6. 发布消息

最后通过以下几个函数发布消息。

1. PublishPoseBroadcastTopic // 发布位置信息
2. PublishPoseBroadcastTF // 发布位置转换transform信息
3. PublishLocalizationStatus // 发布位置状态信息

以上就是整个RTK的定位流程，主要的思路是通过接收GPS和IMU信息结合输出无人车的位置信息，这里还有一个疑问是为什么最后输出的定位信息的位置是直接采用的GPS的位置信息，没有通过IMU信息对位置信息做解算，还是说在其它模块中实现的？？？

10.4. Reference

- [Robust and Precise Vehicle Localization Based on Multi-Sensor Fusion in Diverse City Scenes](#) [<https://ieeexplore.ieee.org/document/8461224>]

11. Map

事者，生于虑，成于务，失于傲。

11.1. Map模块简介

其实我们只需要知道map模块的主要功能是“加载openstreet格式的地图，并且提供一系列的API给其他模块使用”。然后再根据具体的场景来了解地图各个部分的作用，就算是对map模块比较了解了。

11.2. Map目录结构

本章主要介绍下apollo代码的map模块，map的代码目录结构如下：

```
├── data          // 生成好的地图
│   └── demo
├── hdmap         // 高精度地图
│   ├── adapter    // 从xml文件读取地图(openstreet保存格式为xml)
│   │   └── xml_parser
│   └── test-data
├── pnc_map       // 给规划控制模块用的地图
│   └── testdata
├── proto         // 地图各元素的消息格式(人行横道，车道线等)
└── relative_map // 相对地图
    ├── common
    ├── conf
    ├── dag
    ├── launch
    ├── proto
    ├── testdata
    │   └── multi_lane_map
    └── tools
└── testdata      // 测试数据?
    └── navigation_dummy
└── tools         // 工具
```

apollo的高精度地图采用了opendrive格式，opendrive是一个统一的地图标准，这样保证了地图的通用性。其中map模块主要提供的功能是读取高精度地图，并且转换成apollo程序中的Map对象。直白一点就是说把xml格式的opendrive高精度地图，读取为程序能够识别的格式。map模块没有实现的功能是高精度地图的制作，简略的制图过程将在下面章节介绍。

11.3. 地图数据结构

由于openstreet格式是一个标准，可以参考官方网站。下面主要介绍下apollo是如何读取xml地图，并且使用的。地图的读取在adapter中，其中xml_parser目录提供解析xml的能力。

而”opendrive_adapter.cc”则实现了地图的加载，转换为程序中的Map对象。然后地图在”hdmap_impl.cc”中提供一系列api接口给其他模块使用。下面先介绍下地图消息格式，主要在proto目录。“map.proto”分为地图头部信息和结构体，头部信息主要介绍了地图的基本信息“版本，时间，投影方法，地图大小，厂家等”。结构体主要是道路的不同组成部分，包括“人行横道，路口区域，车道，停车观察，信号灯，让路标志，重叠区域，禁止停车，减速带，道路，停车区域，路边的小路，或者行人走的路”。

11.3.1. 地图信息头

首先是地图的基本信息

```
message Header {  
    optional bytes version = 1;      //地图版本  
    optional bytes date = 2;        //地图时间  
    optional Projection projection = 3; //投影方法  
    optional bytes district = 4;     //区  
    optional bytes generation = 5;   //  
    optional bytes rev_major = 6;    //  
    optional bytes rev_minor = 7;    //  
    optional double left = 8;       //左  
    optional double top = 9;        //上  
    optional double right = 10;     //右  
    optional double bottom = 11;    //底
```

```
    optional bytes vendor = 12;           //供应商
}
```

下面是地图的道路信息，其中有2个标志(StopSign, YieldSign)是美国才有的，后来查看了下知乎发现对应到国内是(停，让)，具体的含义都是一样，停车的意思是到路口先停止，看下有没有车，然后再开始启动，让车就是先让行，比如交汇路口，理应让直行的车辆先通过，然后再汇入道路。[参考](https://www.zhihu.com/question/20512694) [https://www.zhihu.com/question/20512694] 下面在介绍下overlap，overlap在注释里的解释是“任何一对在地图上重合的东西，包括（车道，路口，人行横道）”，比如路口的人行横道和道路是重叠的，还有一些交通标志和道路也是重叠的，这是创造的一个逻辑概念。（不知道这样理解是否正确）

```
message Map {
    optional Header header = 1;           //上面所说地图基本信息

    repeated Crosswalk crosswalk = 2;    //人行横道
    repeated Junction junction = 3;      //交叉路口
    repeated Lane lane = 4;              //车道
    repeated StopSign stop_sign = 5;      //停车标志
    repeated Signal signal = 6;          //信号灯
    repeated YieldSign yield = 7;         //让车标志
    repeated Overlap overlap = 8;         //重叠区域
    repeated ClearArea clear_area = 9;    //禁止停车区域
    repeated SpeedBump speed_bump = 10;   //减速带
    repeated Road road = 11;              //道路
    repeated ParkingSpace parking_space = 12; //停车区域
    repeated Sidewalk sidewalk = 13;       //路边的小路，或者行人走的路,
现在的版本已经去掉？但是其他模块有些还有sidewalk
}
```

11.3.2. 人行横道

map_crosswalk.proto 人行横道(google图片搜索出了彩虹人行横道和三维人行横道，就问深度学习该怎么办？)

```
message Crosswalk {
    optional Id id = 1;                  //编号

    optional Polygon polygon = 2;         //多边形
```

```
    repeated Id overlap_id = 3;      //重叠ID
}
```



11.3.3. 路口

map_junction.proto 路口，道路汇聚点

```
message Junction {
    optional Id id = 1;      //编号

    optional Polygon polygon = 2;      //多边形

    repeated Id overlap_id = 3;      //重叠id
}
```



11.3.4. 车道

map.lane.proto 车道线，介绍的比较复杂

```
// A lane is part of a roadway, that is designated for use by  
// a single line of vehicles.  
// Most public roads (include highways) have more than two  
lanes.  
message Lane {  
    optional Id id = 1; //编号  
  
    // Central lane as reference trajectory, not necessary to  
    be the geometry central.  
    optional Curve central_curve = 2; //中心曲线  
  
    // Lane boundary curve.  
    optional LaneBoundary left_boundary = 3; //左边界  
    optional LaneBoundary right_boundary = 4; //右边界
```

```

// in meters.
optional double length = 5;                                //长度

// Speed limit of the lane, in meters per second.
optional double speed_limit = 6;                            //速度限制

repeated Id overlap_id = 7;                                 //重叠区域id

// All lanes can be driving into (or from).
repeated Id predecessor_id = 8;                            //前任id
repeated Id successor_id = 9;                             //继任者id

// Neighbor lanes on the same direction.
repeated Id left_neighbor_forward_lane_id = 10;        //前面左边邻居id
repeated Id right_neighbor_forward_lane_id = 11;       //前面右边邻居id

enum LaneType {                                         //车道类型
    NONE = 1;                                         //无
    CITY_DRIVING = 2;                                  //城市道路
    BIKING = 3;                                       //自行车
    SIDEWALK = 4;                                     //人行道
    PARKING = 5;                                      //停车
};
optional LaneType type = 12;                            //车道类型

enum LaneTurn {                                         //转弯类型
    NO_TURN = 1;                                     //直行
    LEFT_TURN = 2;                                    //左转弯
    RIGHT_TURN = 3;                                   //右转弯
    U_TURN = 4;                                       //掉头
};
optional LaneTurn turn = 13;                           //转弯类型

repeated Id left_neighbor_reverse_lane_id = 14;        //保留(后面?)左边邻居
repeated Id right_neighbor_reverse_lane_id = 15;       //右边邻居

optional Id junction_id = 16;

```

```

// Association between central point to closest boundary.
repeated LaneSampleAssociation left_sample = 17;           //中
心点与最近左边界之间的关联
repeated LaneSampleAssociation right_sample = 18;          //中
心点与最近右边界之间的关联

enum LaneDirection {
    FORWARD = 1;      //前
    BACKWARD = 2;     //后, 潮汐车道借用的情况?
    BIDIRECTION = 3;  //双向
}
optional LaneDirection direction = 19;           //车道方向

// Association between central point to closest road
boundary.
repeated LaneSampleAssociation left_road_sample = 20;      //中
心点与最近左路边界之间的关联
repeated LaneSampleAssociation right_road_sample = 21;
//中心点与最近右路边界之间的关联
}

```



11.3.5. 停止信号

map_stop_sign.proto 停止信号

```
message StopSign {  
    optional Id id = 1;           //编号  
    repeated Curve stop_line = 2;   //停止线, Curve曲线应该是基础  
                                    //类型  
    repeated Id overlap_id = 3;     //重叠id  
  
    enum StopType {  
        UNKNOWN = 0;             //未知  
        ONE WAY = 1;            //只有一车道可以停  
        TWO WAY = 2;  
        THREE WAY = 3;  
        FOUR WAY = 4;  
        ALL WAY = 5;  
    };  
    optional StopType type = 4;  
}
```



11.3.6. 交通信号标志

map_signal.proto 交通信号标志

```
message Subsignal {
    enum Type {
        UNKNOWN = 1;      //未知
        CIRCLE = 2;       //圈???
        ARROW_LEFT = 3;   //左边
        ARROW_FORWARD = 4; //前面
        ARROW_RIGHT = 5;  //右边
        ARROW_LEFT_AND_FORWARD = 6; //左前
        ARROW_RIGHT_AND_FORWARD = 7; //右前
        ARROW_U_TURN = 8;  //掉头
    };
    optional Id id = 1;
    optional Type type = 2;

    // Location of the center of the bulb. now no data support.
    optional apollo.common.PointENU location = 3;      //也是基础类
型?
}

message Signal {
    enum Type {
        UNKNOWN = 1;
        MIX_2_HORIZONTAL = 2;
        MIX_2_VERTICAL = 3;
        MIX_3_HORIZONTAL = 4;
        MIX_3_VERTICAL = 5;
        SINGLE = 6;
    };
    optional Id id = 1;
    optional Polygon boundary = 2;      //多边形
    repeated Subsignal subsignal = 3;   //子信号
    // TODO: add orientation. now no data support.
    repeated Id overlap_id = 4;        //重叠id
    optional Type type = 5;           //这里的类型是主要指交通标识的个数及
位置? ?
    // stop line
```

```
repeated Curve stop_line = 6;           //在哪里结束?  
}
```



11.3.7. 让行

map_yield_sign.proto 让行标志（美国才有）

```
message YieldSign {  
    optional Id id = 1;           //编号  
  
    repeated Curve stop_line = 2;   //在哪里结束  
  
    repeated Id overlap_id = 3;     //重叠id  
}
```



11.3.8. 重叠区域

map_overlap.proto 这里只介绍了 LaneOverlapInfo，其他的还没有对应的格式

```
message LaneOverlapInfo {  
    optional double start_s = 1; //position (s-coordinate)  
    optional double end_s = 2; //position (s-coordinate)  
    optional bool is_merge = 3;  
}  
// Information about one object in the overlap.  
message ObjectOverlapInfo {  
    optional Id id = 1;  
  
    oneof overlap_info {  
        LaneOverlapInfo lane_overlap_info = 3;  
        SignalOverlapInfo signal_overlap_info = 4;  
        StopSignOverlapInfo stop_sign_overlap_info = 5;  
        CrosswalkOverlapInfo crosswalk_overlap_info = 6;  
        JunctionOverlapInfo junction_overlap_info = 7;  
        YieldOverlapInfo yield_sign_overlap_info = 8;  
        ClearAreaOverlapInfo clear_area_overlap_info = 9;  
        SpeedBumpOverlapInfo speed_bump_overlap_info = 10;  
        ParkingSpaceOverlapInfo parking_space_overlap_info = 11;  
    }  
}
```

```

        SidewalkOverlapInfo sidewalk_overlap_info = 12;
    }
}

// Here, the "overlap" includes any pair of objects on the
map
// (e.g. lanes, junctions, and crosswalks).
message Overlap {
    optional Id id = 1;

    // Information about one overlap, include all overlapped
    objects.
    repeated ObjectOverlapInfo object = 2;
}

```

逻辑概念，没有具体的规则显示这个区域

11.3.9. 禁止停车

map_clear_area.proto 禁止停车

```

// A clear area means in which stopping car is prohibited

message ClearArea {
    optional Id id = 1;           //编号
    repeated Id overlap_id = 2;   //重叠id
    optional Polygon polygon = 3; //多边形
}

```



map_speed_bump.proto 渏速带

```
message SpeedBump {  
    optional Id id = 1;           // 编号  
    repeated Id overlap_id = 2;   // 重叠区域  
    repeated Curve position = 3; // 曲线位置  
}
```



11.3.10. 道路信息

map_road.proto 道路的信息，是由一些RoadSection组成

```
// road section defines a road cross-section, At least one  
section must be defined in order to  
// use a road, If multiple road sections are defined, they  
must be listed in order along the road  
message RoadSection {  
    optional Id id = 1;  
    // lanes contained in this section
```

```

repeated Id lane_id = 2;
// boundary of section
optional RoadBoundary boundary = 3;
}

// The road is a collection of traffic elements, such as
lanes, road boundary etc.
// It provides general information about the road.
message Road {
    optional Id id = 1;
    repeated RoadSection section = 2;

    // if lane road not in the junction, junction id is null.
    optional Id junction_id = 3;
}

```



11.3.11. 停车区域

`map_parking.proto` 停车区域

```

// ParkingSpace is a place designated to park a car.
message ParkingSpace {
    optional Id id = 1;

```

```
    optional Polygon polygon = 2;  
  
    repeated Id overlap_id = 3;  
  
    optional double heading = 4;  
}
```



11.3.12. 行人道路

map_sidewalk.proto 路边的小路，或者行人走的路

// A sidewalk (American English) or pavement (British English), also known as a footpath or footway, is a path along the side of a road.

```
message Sidewalk {  
    optional Id id = 1;  
    repeated Id overlap_id = 2;  
    optional Polygon polygon = 3;  
}
```



其中还包括剩下的4个没有介绍 map_id.proto 这里的map_id是基础id?

```
message Id {  
    optional string id = 1;          //id, 字符类型  
}
```

map_speed_control.proto 限制速度

```
message SpeedControl {  
    optional string name = 1;  
    optional apollo.hdmap.Polygon polygon = 2;  
    optional double speed_limit = 3;  
}
```

map_geometry.proto 地图的几何形状?

```
// Polygon, not necessary convex.  
message Polygon {  
    repeated apollo.common.PointENU point = 1;  
}
```

```

// Straight line segment.
message LineSegment {
    repeated apollo.common.PointENU point = 1;
}

// Generalization of a line.
message CurveSegment {
    oneof curve_type {
        LineSegment line_segment = 1;
    }
    optional double s = 6; // start position (s-coordinate)
    optional apollo.common.PointENU start_position = 7;
    optional double heading = 8; // start orientation
    optional double length = 9;
}

// An object similar to a line but that need not be straight.
message Curve {
    repeated CurveSegment segment = 1;
}

```

map_pnc_junction.proto PNC路口（具体的场景是什么？？）

```

message PNCJunction {
    optional Id id = 1;

    optional Polygon polygon = 2;

    repeated Id overlap_id = 3;
}

```

11.4. opendriver地图解析

上面只是简单的介绍了下地图的数据格式，具体的应用场景，还需要结合planning模块进一步学习。我们再回过头来看adapter模块，其中xml_parser就是针对道路的不同元素部分做的解析。

```

|-- adapter
|   |-- BUILD
|   |-- coordinate_convert_tool.cc      // 坐标转换工具
|   |-- coordinate_convert_tool.h

```

```

    |-- opendrive_adapter.cc          // 加载opendrive格式地图
    |-- opendrive_adapter.h
    |-- proto_organizer.cc           //
    |-- proto_organizer.h
    |-- xml_parser                  // xml_parser针对道路的不同元素做相应
解析
    |   |-- common_define.h
    |   |-- header_xml_parser.cc
    |   |-- header_xml_parser.h
    |   |-- junctions_xml_parser.cc
    |   |-- junctions_xml_parser.h
    |   |-- lanes_xml_parser.cc
    |   |-- lanes_xml_parser.h
    |   |-- objects_xml_parser.cc
    |   |-- objects_xml_parser.h
    |   |-- roads_xml_parser.cc
    |   |-- roads_xml_parser.h
    |   |-- signals_xml_parser.cc
    |   |-- signals_xml_parser.h
    |   |-- status.h
    |   |-- util_xml_parser.cc
    |   |-- util_xml_parser.h

```

11.5. 高精度地图API

最后在看下hdmap_impl.cc，主要实现了一系列的api来查找道路中的元素。由于实现的接口太多，后面有时间了看是否能够整理下api文档。关于pnc_map和relative_map还没有介绍，关于一些道路元素的使用场景没有介绍。

11.6. tools

tools的目录结构如下，主要是一些制作和转换地图的工具。

```

.
├── BUILD
├── bin_map_generator.cc      // txt地图转换为bin地图
├── map_datachecker          // 远程地图采集，这里只是采集了pose？
├── map_tool.cc               // 地图加上偏移
└── map_xysl.cc              // 功能很多，用来查找lane，以及lane上

```

的点转SL
└── proto_map_generator.cc // 转换opendrive格式的地图为apollo
proto的地图
└── quaternion_euler.cc // 4维旋转转3维旋转
└── refresh_default_end_way_point.cc // 更新routing的默认地标
└── sim_map_generator.cc // 生成sim map

下面简单介绍下各个工具的实现以及作用。

11.6.1. sim_map_generator

通过base_map生成sim_map，其中sim_map去掉了base_map中的

```
left_sample  
right_sample  
left_road_sample  
right_road_sample
```

主要的作用为获取当前道路的宽度。

另外对central_curve, left_boundary和right_boundary进行了降采样，减少了点数。

11.6.2. refresh_default_end_way_point

更新routing POI中的默认点的位置信息，这里的默认点就是比较典型的地标，方便选择routing位置。

11.6.3. quaternion_euler

4维旋转转3维

11.6.4. proto_map_generator

转换opendrive格式的地图为apollo proto的地图

11.6.5. map_xysl

功能很多，用来查找lane，以及lane上的点转SL

11.6.6. map_tool

地图整体加上位置偏移。

11.6.7. bin_map_generator

txt地图转换为bin地图。

11.7. 如何制作高精度地图

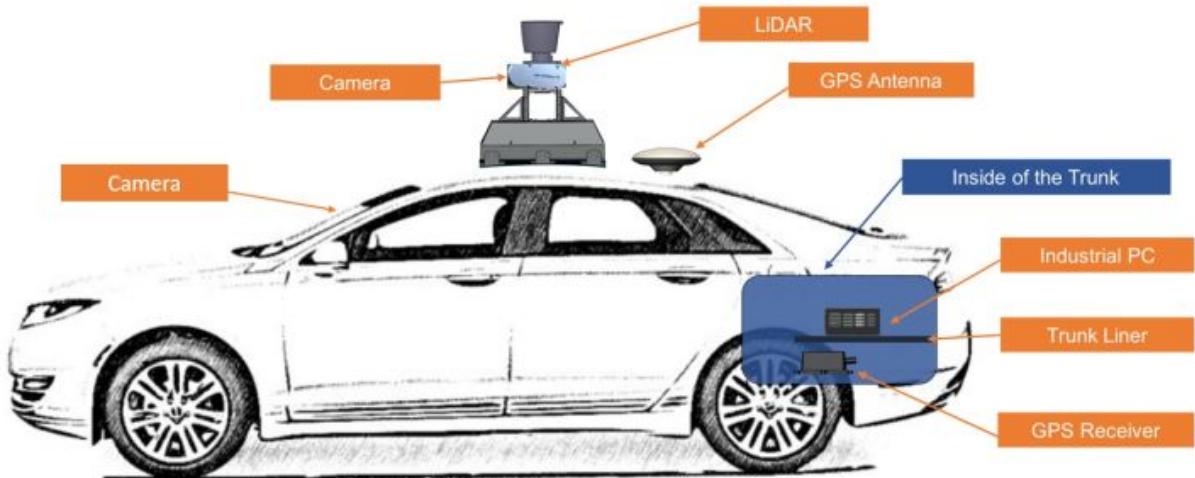
前面介绍了为什么需要高精度地图，那么我们如何制作一张高精度地图呢？制作一张高精度地图可以大概分为3个过程：采集、加工、转换。

11.7.1. 采集

如何采集地图？

我们需要一些传感器来获取数据，下面是需要的传感器列表：

1. lidar
2. 摄像头
3. gnss
4. imu



知乎 @王方浩

- **lidar** 主要是来采集点云数据，因为激光雷达可以精确的反应出位置信息，所以激光雷达可以知道路面的宽度，红绿灯的高度，以及一些其他的信息，当然现在也有厂家基于视觉SLAM（纯摄像头测距）来制作地图的，有兴趣的也可以看下相关介绍。
- **摄像头** 主要是来采集一些路面的标志，车道线等，因为图像的像素信息更多，而位置信息不太精确，所以采用摄像头来识别车道线，路面的一些标志等。
- **gnss** 记录了车辆的位置信息，记录了当前采集点的坐标。
- **imu** 用来捕获车辆的角度和加速度信息，用来校正车辆的位置和角度。

需要的操作系统和软件：

- ubuntu 16.04
- apollo 用apollo的录制bag功能，可以把传感器的数据都录制下来，提供生成高精地图的原始数据。其实在录制数据之前，需要对上面所说的传感器进行校准工作，这部分的工作比较专业，涉及到坐标系转换，也涉及到一些传感器的知识，所以对非专业人士来说不是那么好理解。或者开发一系列工具来实现校准。接下来就是采集了，采集过程中需要多次采集来保证采集的数据比较完整，比如你在路口的时候，从不同的角度开车过去看到的建筑物的轮廓是不一样的，这些轮廓就是激光雷达扫描到的数据。所

以遇到路口，或者多车道的情况，尽可能的多采集几次，才能收集到比较完整的地图信息。并且速度不要太快，apollo上的介绍是不超过60km/h（这里没有特别说明会出现什么问题）。以下是我的一点个人想法：

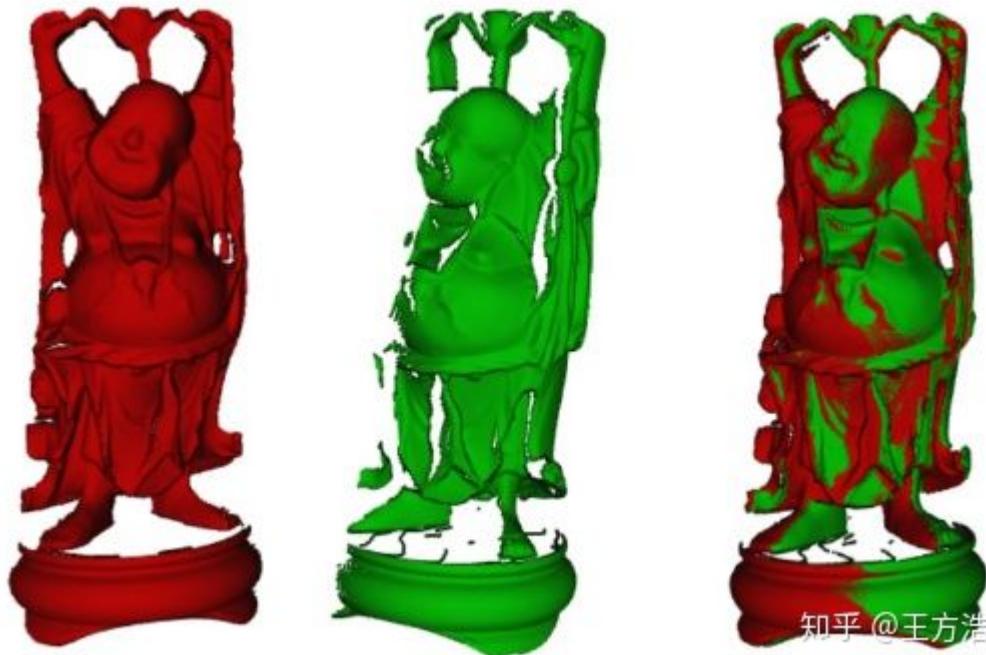
上面的采集方案依赖很多，首先需要一系列的硬件，其次是需要apollo，并且熟悉apollo的启动流程，最后还需要传感器校准的知识。实际上采集的过程中我们不需要自动驾驶。可以开发一个轻量级的采集方案，硬件全部集中到一个盒子中，软件只需要提供录制bag包的能力就可以了，这点ros都可以做到，最后校准由于硬件都是一体化的盒子，只需要校准一个传感器就可以把其中所有传感器的坐标系确定。相对于上面的方案来说更加轻量，可能只需要邮寄一套设备就可以开始录制地图了。here的地图是分层的，比如路面是很少更新的，而路灯，车道标识，或者红绿灯可能会更换，所以路面信息可能需要激光雷达去采集一次，而路灯，车道标识，红绿灯等可以通过摄像头的方案来更新，因为高精度地图需要实时更新，上面的方案可能更加适合一些地图更新的场景。

11.7.2. 加工

如何加工上述地图？

首先需要生成一张原始的地图，这里我们采用点云生成原始的地图，因为点云的距离位置信息比较准确，因为点云数据是0.1s采集一帧，下面我们可以做一个计算。如果车速是100km/h，对应27.8m/s。即0.1s车行驶的距离是2.78m，而激光雷达的扫描距离大概是150m，所以前后2帧大部分地方是重合的。因为数据是一帧一帧的，我们需要把上面的说的每一帧进行合并，生成一张完整的地图，有点类似全景照片拼接，这样我们就可以得到一张原始的采集路段的地图。这里用到了点云的配准技术，有2种算法ICP和NDT，基于上面的算法，可以把点云的姿态进行变换并且融合。具体的介绍可以[参考](#)

[<https://blog.csdn.net/xs1997/article/details/76795041>]。



知乎 @王方浩

上图红色

和绿色的部分是从不同方位扫描得到的结果，最后是配准融合之后的结果，可以把地球想象成上图这个模型放大了1000万倍的效果，我们的车相当于一个扫描设备，把每次扫描的结果拼接起来，就制作好了一张点云地图。点云拼接好了之后，我们就需要在道路上标出路沿，车道线，红绿灯，路口，一些交通标识等。大部分的工作都可以用深度学习结合图像的方法去解决，查找出上面的一些信息并且标识出来，目前有些场景还是需要人工标识出来，比如路口停止线和红绿灯的关系，如果一些特殊场景的车道线等，需要人工去做一些校正。上面的过程可以说是一个简易的制图过程。实际上这里还需要讲下高精地图的格式，因为如果没有一个统一的格式，高精度地图是没有太多意义的。我们可以把高精度地图分为三层：

- 地图图层 地图图层主要是道路的信息，比如道路的路沿，车道线，路口信息，主要是道路的一些基本信息。
- 定位图层 定位图层主要是具备独特的目标或特征，比如红绿灯，交通标志，道路的点云数据等。
- 动态图层 动态图层主要是一些实时路况，修路或者封路等需要实时推送或者更新的数据。通过下面的加工流程：

点云地图校准 -> 地图标注加工 -> 高精度地图

这样就生成了一张高精度地图，当然加工过程中首要的目标是提高效率和质量，尽量的采用算法自动化处理会很大的提高效率，这可能是后面地图厂家的核心竞争力。因为地图需要实时更新，谁的效率更高，谁的图就越新，用的人越多，之后的数据也越完善。

11.7.3. 转换

转换主要是得到一个通用的自动驾驶系统可以使用的高精度地图。

上面的高精地图格式可能还是原始的数据格式，需要转换为apollo中高精度地图的格式，apollo中高精度地图采用了opendrive的格式，并且做了改进，总之这是一个通用的标准，这个很重要，否则每个厂家的数据如果不兼容，会导致很大的问题，你需要开发一系列的转换工具，去处理不同地图的差异，并且不同的自动驾驶系统和不同的地图厂家采用的方式不一样，会带来很多兼容性问题。

11.8. Reference

- [convert opendrive to base_map.xml](#) [https://github.com/ApolloAuto/apollo/issues/603]
- [点云拼接注册](#) [https://blog.csdn.net/xs1997/article/details/76795041]
- [百度技术讲堂](#) [http://bit.baidu.com/Course/detail/id/282.html]
- [四维图新](#) [https://www.navinfo.com/product/autodri_map]
- [Apollo 2.5地图采集功能使用指南](#) [https://link.zhihu.com/?target=https%3A//github.com/ApolloAuto/apollo/blob/master/docs/quickstart/apollo_2_5_map_collection_guide_cn.md]

12. Monitor

人不知而不愠，不亦君子乎。

12.1. 简介

monitor模块主要是监控硬件和软件状态，当出现故障的时候，显示故障原因，并且输出状态给guardian模块进行紧急处理。

12.2. MonitorManager

12.2.1. 初始化Init

12.2.2. StartFrame

12.2.3. EndFrame

12.3. hardware

12.4. software

软件主要打开功能安全functional_safety_monitor,

13. Perception

温故而知新，可以为师矣

13.1. Perception模块简介

首先简单看下perception的目录结构：

```
.  
├── BUILD  
├── Perception_README_3_5.md  
└── README.md  
├── base          // 基础类  
├── camera        // 相机相关           --- 子模块流程  
├── common        // 公共目录  
├── data          // 相机的内参和外参  
├── fusion         // 传感器融合  
├── inference     // 深度学习推理模块  
├── lib            // 一些基础的库，包括线程、时间等  
├── lidar          // 激光雷达相关           --- 子模块流程  
├── map            // 地图  
├── model          // 深度学习模型  
├── onboard        // 各个子模块的入口       --- 子模块入口  
├── production    // 感知模块入口（深度学习模型也存放在里面） --- 通过cyber启动子模块  
├── proto          // 数据格式，protobuf  
├── radar          // 毫米波               --- 子模块流程  
└── testdata       // 上述几个模块的测试数据  
└── tool           // 离线测试工具
```

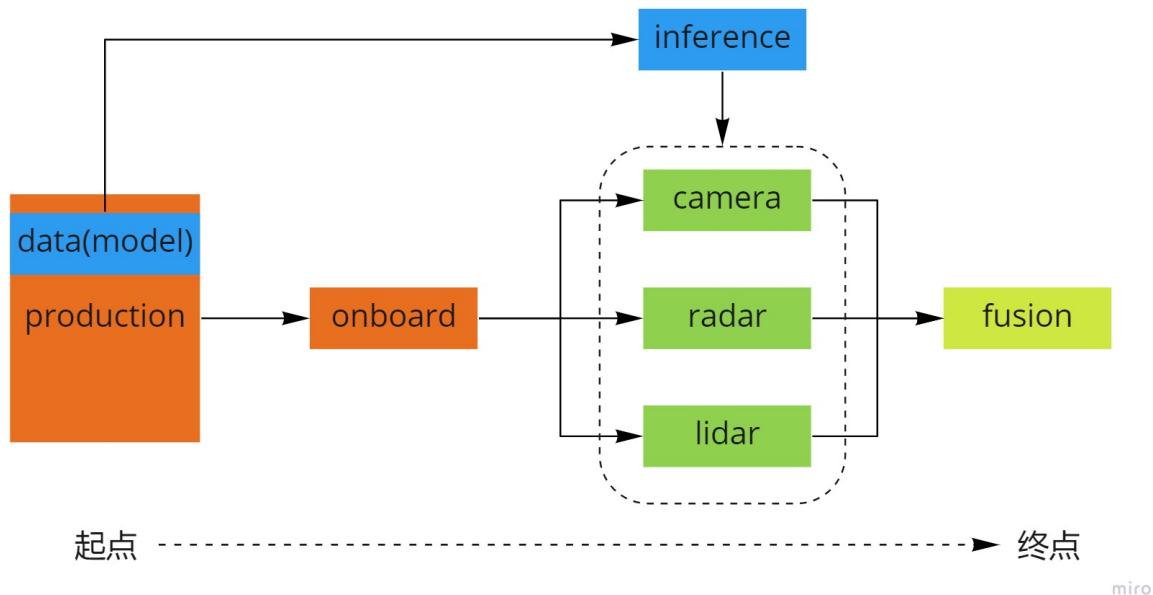
下面介绍几个重要的目录结构：

- **production**目录 - 感知模块的入口在**production**目录，通过**lanuch**加载对应的**dag**，启动感知模块，感知模块包括多个子模块，在**onboard**目录中定义。
- **onboard**目录 - 定义了多个子模块，分别用来处理不同的传感器信息（Lidar,Radar,Camera）。各个子模块的入口在**onboard**目录中，

每个传感器的流程大概相似，可以分为预处理，物体识别，感兴趣区域过滤以及追踪。

- inference目录 - 深度学习推理模块，我们知道深度学习模型训练好了之后需要部署，而推理则是深度学习部署的过程，实际上部署的过程会对模型做加速，主要实现了**caffe**, **TensorRT**和**paddlepaddle**3种模型部署。训练好的深度模型放在”modules\perception\production\data”目录中，然后通过推理模块进行加载部署和在线计算。
- camera目录 - 主要实现车道线识别，红绿灯检测，以及障碍物识别和追踪。
- radar目录 - 主要实现障碍物识别和追踪（由于毫米波雷达上报的就是障碍物信息，这里主要是对障碍物做追踪）。
- lidar目录 - 主要实现障碍物识别和追踪（对点云做分割，分类，识别等）。
- fusion目录 - 对上述传感器的感知结果做融合。

整个模块的流程如图：



可以看到感知模块由production模块开始，由fusion模块结束。

13.2. production 目录

production中主要是存放：

1. 配置和lanuch和dag启动文件
2. 存放训练好的模型

```
.  
└── conf    // 配置文件  
└── dag     // dag启动文件  
└── data    // 训练好的模型  
└── launch  // cyber launch加载dag
```

该文件中有多个lanuch文件，同时一个lanuch文件中包含多个dag文件，也就是说一个lanuch文件会启动多个子模块。

13.3. onboard 目录

onboard目录定义了多个子模块，每个子模块对应一个功能，包括：车道线识别，障碍物识别，红绿灯识别，传感器融合，场景分割等。

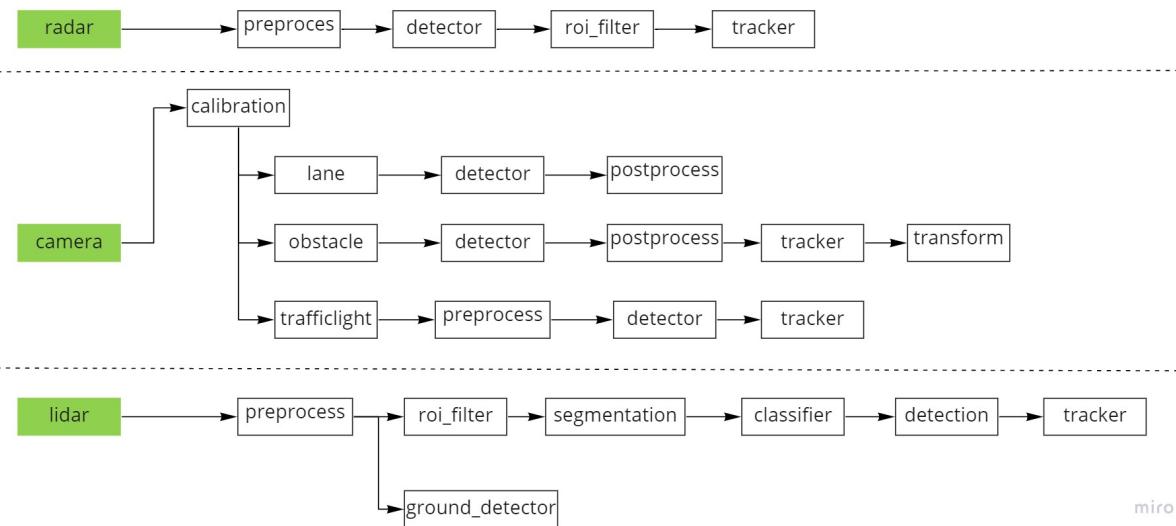
```
.  
└── common_flags  
└── component           // 子模块入口  
└── inner_component_messages  
└── msg_buffer  
└── msg_serializer  
└── proto  
└── transform_wrapper
```

实际上几个子模块可能合并为一个模块，如何确定模块是否合并呢？我们可以查看”onboard/component”目录中的BUILD文件。

```
name = "perception_component_inner_lidar",  
srcs = [  
    "fusion_component.cc",  
    "lidar_output_component.cc",  
    "radar_detection_component.cc",  
    "recognition_component.cc",  
    "segmentation_component.cc",  
    "detection_component.cc",  
,
```

在BUILD文件中上述几个模块被编译为一个模块”libperception_component_lidar”。也就是说在dag中实际上只需要启动**libperception_component_lidar**这一个模块就相当于启动了上述几个模块。

看完了perception模块的入口，以及各个子模块的定义，那么各个子模块的功能如何实现的呢？实际上感知各个子模块的功能是通过lidar,radar和camera3种传感器实现的，每种传感器分别都执行了目标识别和追踪的任务，最后通过fusion对传感器的数据做融合，执行代码分别在”perception/radar”, ”perception/lidar”, ”perception/camera”目录中。这里有2种查看代码的方式，一种是正序的方式，根据具体的功能，例如从物体识别子模块入手，分别查看lidar,radar和camera模块中的物体识别功能，另一种是倒序的方式，根据传感器划分，先查看传感器分别实现了哪些功能，然后回过头来看各个子模块是如何把上述功能整合起来的。这里我们采用第2种方式，先看各个传感器的执行流程如下图。



从图中可以看到，每个传感器都实现了物体识别的功能，而摄像头还实现了车道线识别和红绿灯检测的功能，每个传感器执行的任务流水线也大概相似，先进行预处理，然后做识别，最后过滤并且追踪目标。其中物体识别用到了推理引擎inference。

接下来来我们分别查看各个传感器的具体实现。我们先从radar开始看起，主要是radar模块相对比较简单。

13.4. 子模块介绍

radar子模块介绍 camera子模块介绍 lidar子模块介绍 fusion子模块
inference推理子模块

13.5. Reference

- [A Beginner's Guide to Convolutional Neural Networks](#)
[<https://skymind.ai/wiki/convolutional-network>]
- [cnn](#) [<https://cs231n.github.io/convolutional-networks/>]
- [traffic light dataset](#) [<https://hci.iwr.uni-heidelberg.de/node/6132/download/3d66608cfb112934ef40175e9a20c81f>]
- [pytorch-tutorial](#) [<https://github.com/yunjey/pytorch-tutorial>]
- [全连接层的作用是什么？](#) [<https://www.zhihu.com/question/41037974>]
- [索伯算子](#)
[<https://zh.wikipedia.org/wiki/%E7%B4%A2%E8%B2%9D%E7%88%BE%E7%AE%97%E5%AD%90>]
- [卷积](#) [<https://zh.wikipedia.org/wiki/%E5%8D%B7%E7%A7%AF>]
- [TensorRT\(1\)-介绍-使用-安装](#) [<https://arleyzhang.github.io/articles/7f4b25ce/>]
- [高性能深度学习支持引擎实战——TensorRT](#)
[<https://zhuanlan.zhihu.com/p/35657027>]

14. Planning

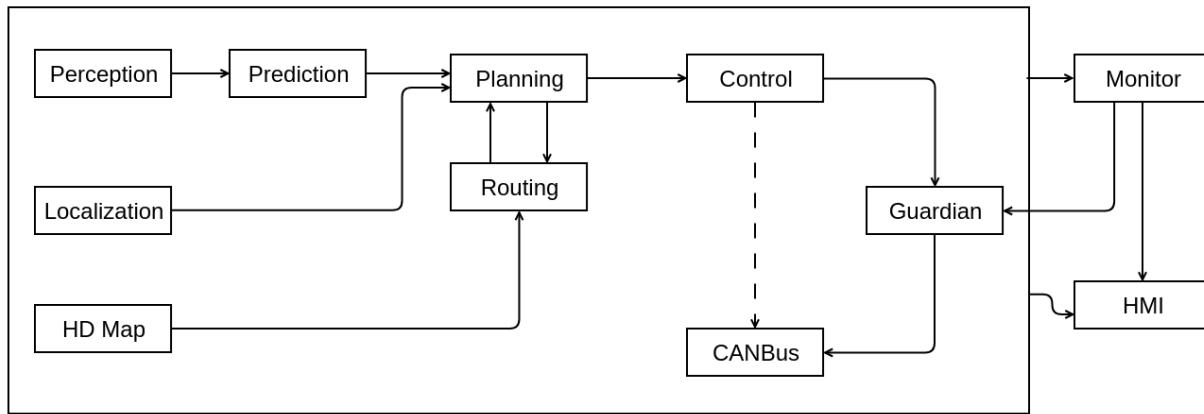
吾尝终日而思矣 不如须臾之所学也

14.1. Planning模块简介

规划(planning)模块的作用是根据感知预测的结果，当前的车辆信息和路况规划出一条车辆能够行驶的轨迹，这个轨迹会交给控制(control)模块，控制模块通过油门，刹车和方向盘使得车辆按照规划的轨迹运行。规划模块的轨迹是短期轨迹，即车辆短期内行驶的轨迹，长期的轨迹是routing模块规划出的导航轨迹，即起点到目的地的轨迹，规划模块会先生成导航轨迹，然后根据导航轨迹和路况的情况，沿着短期轨迹行驶，直到目的地。这点也很好理解，我们开车之前先打开导航，然后根据导航行驶，如果前面有车就会减速或者变道，超车，避让行人等，这就是短期轨迹，结合上述的方式直到行驶到目的地。

14.1.1. Planning输入输出

我们先看下Apollo的数据流向：



可以看到规划(planning)模块的上游是Localization, Prediction, Routing模块，而下游是Control模块。Routing模块先规划出一条导航线路，然后Planning模块根据这条线路做局部优化，如果Planning模块发现短期规划的线路行不通（比如前面修路，或者错过了路口），会触发Routing

模块重新规划线路，因此这两个模块的数据流是双向的。Planning模块的输入在”planning_component.h”中，接口如下：

```
bool Proc(const  
std::shared_ptr<prediction::PredictionObstacles>&  
          prediction_obstacles,  
          const std::shared_ptr<canbus::Chassis>& chassis,  
          const  
std::shared_ptr<localization::LocalizationEstimate>&  
          localization_estimate) override;
```

输入参数为：

1. 预测的障碍物信息(prediction_obstacles)
2. 车辆底盘(chassis)信息(车辆的速度，加速度，航向角等信息)
3. 车辆当前位置(localization_estimate)

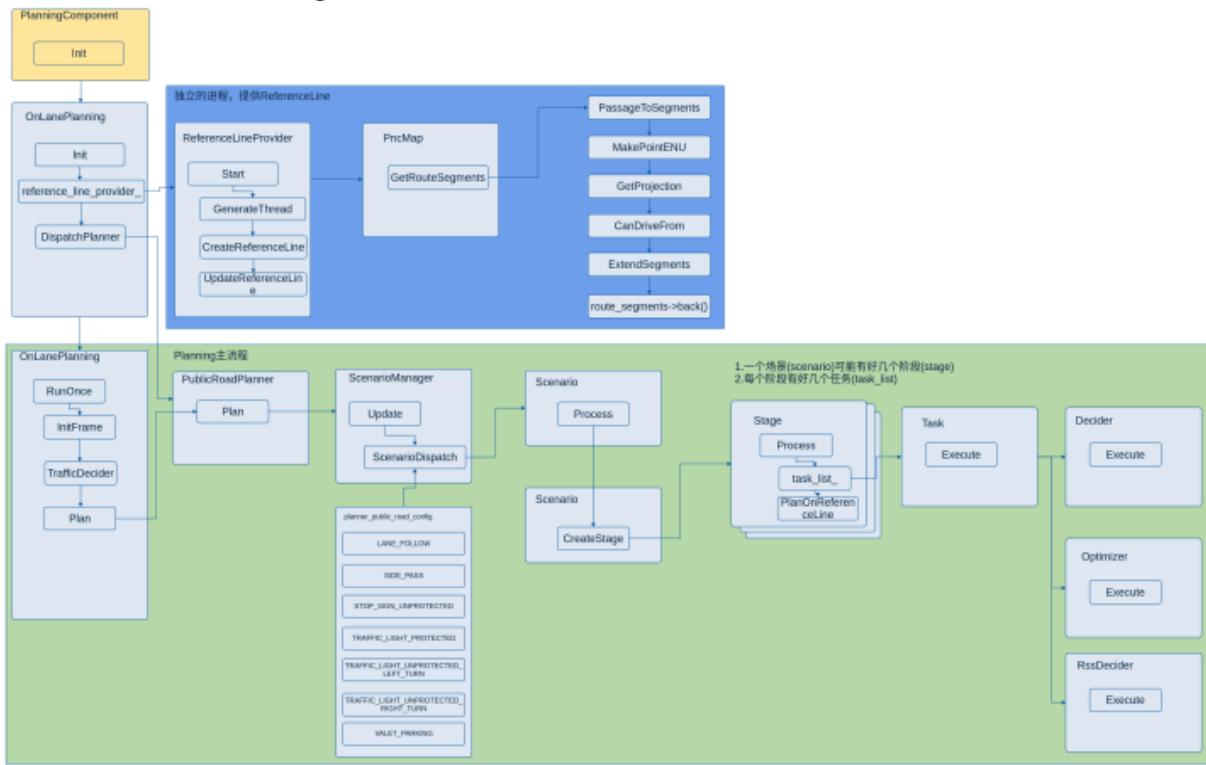
实际上还有高精度地图信息，不在参数中传入，而是在函数中直接读取的。

Planning模块的输出结果在”PlanningComponent::Proc()”中，为规划好的线路，发布到Control模块订阅的Topic中。输出结果为：规划好的路径。

```
planning_writer_->Write(std::make_shared<ADCTrajectory>  
(adc_trajectory_pb));
```

14.1.2. Planning整个流程

下图是整个Planning模块的执行过程:



1. 模块的入口是PlanningComponent，在Cyber中注册模块，订阅和发布消息，并且注册对应的Planning类。
2. Planning的过程之前是定时器触发，即每隔一段固定的时间执行一次，现已经改为事件触发，即只要收集完成对应TOPIC的消息，就会触发执行，这样好处是提高的实时性。
3. Planning类主要实现了2个功能，一个是启动ReferenceLineProvider来提供参考线，后面生成的轨迹都是在参考线的基础上做优化，ReferenceLineProvider启动了一个单独的线程，每隔50ms执行一次，和Planning主流程并行执行。Planning类另外的一个功能是执行Planning主流程。
4. Planning主流程先是选择对应的Planner，我们这里主要分析PublicRoadPlanner，在配置文件中定义了Planner支持的场景(Scenario)，把规划分为具体的几个场景来执行，每个场景又分为几个阶段(Stage)，每个阶段会执行多个任务(Task)，任务执行完成后，对应的场景就完成了。不同场景间的切换是由一个状态机(ScenarioDispatch)来控制的。规划控制器根据ReferenceLineProvider提供的参考线，在不同的场景下做切换，生

成一条车辆可以行驶的轨迹，并且不断重复上述过程直到到达目的地。

接下来我们逐步分析整个planning模块的代码结构。

14.2. Planning模块入口

14.2.1. 模块注册

Planning模块的入口

为”planning_component.h”和”planning_component.cc”两个文件，实现的功能如下：

```
// 订阅和发布消息
std::shared_ptr<cyber::Reader<perception::TrafficLightDetection>>
    traffic_light_reader_;
std::shared_ptr<cyber::Reader<routing::RoutingResponse>>
    routing_reader_;
std::shared_ptr<cyber::Reader<planning::PadMessage>>
    pad_message_reader_;
std::shared_ptr<cyber::Reader<relative_map::MapMsg>>
    relative_map_reader_;

std::shared_ptr<cyber::Writer<ADCTrajectory>>
    planning_writer_;
std::shared_ptr<cyber::Writer<routing::RoutingRequest>>
    rerouting_writer_;

// 在Cyber中注册模块
CYBER_REGISTER_COMPONENT(PlanningComponent)
```

14.2.2. 模块初始化

除了注册模块，订阅和发布消息之外，planning模块实现了2个主要函数”init”和”proc”。Init中实现了模块的初始化：

```
if (FLAGS_open_space_planner_switchable) {
    planning_base_ = std::make_unique<OpenSpacePlanning>();
```

```

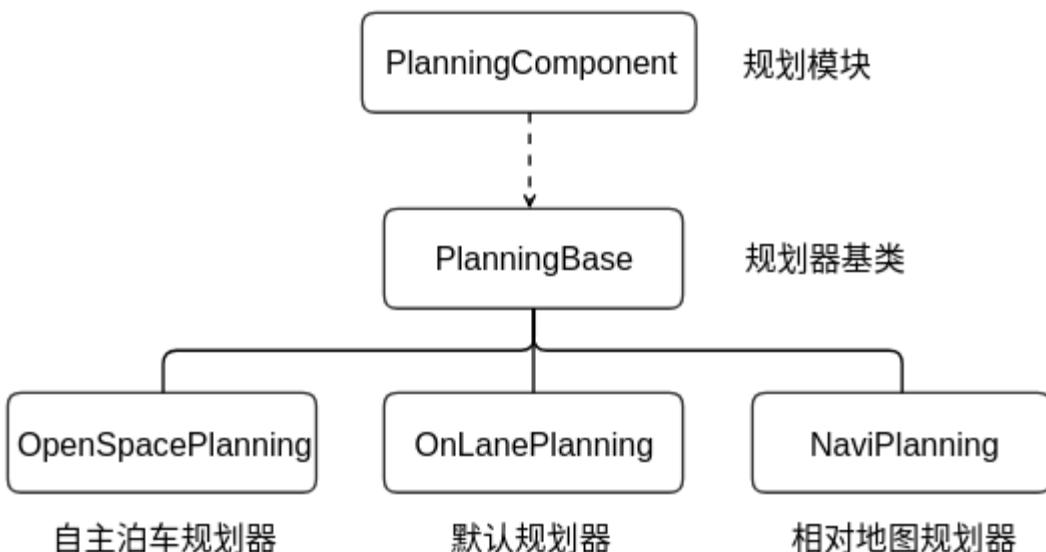
} else {
    if (FLAGS_use_navigation_mode) {
        planning_base_ = std::make_unique<NaviPlanning>();
    } else {
        planning_base_ = std::make_unique<OnLanePlanning>();
    }
}

```

上面实现了3种Planning的注册，planning模块根据配置选择不同的Planning实现方式，”FLAGS_open_space_planner_switchable”和”FLAGS_use_navigation_mode”在Planning模块的conf目录中。因为上述2个配置默认都为false，Planning默认情况下的实现是”OnLanePlanning”。下面介绍下这3种Planning的区别。

- **OpenSpacePlanning** - 主要的应用场景是自主泊车和狭窄路段的掉头。[参考](#)
[https://github.com/ApolloAuto/apollo/blob/master/docs/specs/Open_Space_Planner.md]
- **NaviPlanning** -
- **OnLanePlanning** - 主要的应用场景是开放道路的自动驾驶。

模块之间的关系如下：



可以看到”OpenSpacePlanning”，”NaviPlanning”和”OnLanePlanning”都继承自同一个基类，并且在PlanningComponent中通过配置选择一个具体

的实现进行注册。

Init接下来实现了具体的消息发布和消息订阅，我们只看具体的一个例子：

```
// 读取routing模块的消息
routing_reader_ = node_->CreateReader<RoutingResponse>(
    FLAGS_routing_response_topic,
    [this](const std::shared_ptr<RoutingResponse>& routing)
{
    AINFO << "Received routing data: run routing
callback."
        << routing->header().DebugString();
    std::lock_guard<std::mutex> lock(mutex_);
    routing_.CopyFrom(*routing);
});
// 读取红绿灯
traffic_light_reader_ = ...
// 是否使用导航模式
if (FLAGS_use_navigation_mode) {
    pad_message_reader_ = ...
// 读取相对地图
    relative_map_reader_ = ...
}
// 发布规划好的线路
planning_writer_ =
    node_->CreateWriter<ADCTrajectory>
(FLAGS_planning_trajectory_topic);
// 发布重新规划请求
rerouting_writer_ =
    node_->CreateWriter<RoutingRequest>
(FLAGS_routing_request_topic);
```

至此，Planning模块的初始化就完成了。

14.2.3. 模块运行

Proc主要是检查数据，并且执行注册好的Planning，生成路线并且发布。

```

bool PlanningComponent::Proc(...) {
    // 1. 检查是否需要重新规划线路。
    CheckRerouting();

    // 2. 数据放入local_view_中，并且检查输入数据。
    ...

    // 3. 执行注册好的Planning，生成线路。
    planning_base_->RunOnce(local_view_, &adc_trajectory_pb);

    // 4. 发布消息
    planning_writer_->Write(std::make_shared<ADCTrajectory>
        (adc_trajectory_pb));
}

```

整个”PlanningComponent”的分析就完成了，可以看到”PlanningComponent”是Planning模块的入口，在Apollo3.5引入了Cyber之后，实现了Planning模块在Cyber中的注册，订阅和发布topic消息。同时实现了3种不同的Planning，根据配置选择其中的一种并且运行。由于默认的Planning是开放道路的OnLanePlanning，我们接下来主要分析这个Planning。

14.3. OnLanePlanning

每次Planning会根据以下2个信息作为输入来执行：

1. Planning上下文信息
2. Frame结构体(车辆信息，位置信息等所有规划需要用到的信息，在/planning/common/frame.h中)

```

uint32_t sequence_num_ = 0;
LocalView local_view_;
const hdmap::HDMap *hdmap_ = nullptr;
common::TrajectoryPoint planning_start_point_;
common::VehicleState vehicle_state_;
std::list<ReferenceLineInfo> reference_line_info_;

bool is_near_destination_ = false;

/**

```

```

* the reference line info that the vehicle finally choose to
drive on
*/
const ReferenceLineInfo *drive_reference_line_info_ =
nullptr;

ThreadSafeIndexedObstacles obstacles_;
std::unordered_map<std::string, const
perception::TrafficLight *>
traffic_lights_;

ChangeLaneDecider change_lane_decider_;
ADCTrajectory current_frame_planned_trajectory_; // last
published trajectory

std::vector<routing::LaneWaypoint> future_route_waypoints_;

```

14.3.1. 初始化

OnLanePlanning的初始化逻辑在Init中，主要实现分配具体的Planner，启动参考线提供器(reference_line_provider_)，代码分析如下：

```

Status OnLanePlanning::Init(const PlanningConfig& config) {
    ...
    // 启动参考线提供器，会另启动一个线程，执行一个定时任务，每隔50ms提供一
    次参考线。
    reference_line_provider_ =
    std::make_unique<ReferenceLineProvider>(hdmap_);
    reference_line_provider_->Start();

    // 为Planning分配具体的Planner。
    planner_ = planner_dispatcher_->DispatchPlanner();
    ...
}

```

可以看到”DispatchPlanner”在”OnLanePlanning”实例化的时候就指定了。

```

class OnLanePlanning : public PlanningBase {
public:
    OnLanePlanning() {
        planner_dispatcher_ =
std::make_unique<OnLanePlannerDispatcher>();
    }
}

```

在看”OnLanePlannerDispatcher”具体的实现，也是根据配置选择具体的”Planner”，默认为”PUBLIC_ROAD”规划器：

```

// 配置文件
standard_planning_config {
    planner_type: PUBLIC_ROAD
    planner_type: OPEN_SPACE
    ...
}

// OnLanePlannerDispatcher具体实现
std::unique_ptr<Planner>
OnLanePlannerDispatcher::DispatchPlanner() {
    PlanningConfig planning_config;

    apollo::cyber::common::GetProtoFromFile(FLAGS_planning_config
_file,
                                         &planning_config);
    if (FLAGS_open_space_planner_switchable) {
        return planner_factory_.CreateObject(
            // OPEN_SPACE规划器

            planning_config.standard_planning_config().planner_type(1));
    }
    return planner_factory_.CreateObject(
        // PUBLIC_ROAD规划器

        planning_config.standard_planning_config().planner_type(0));
}

```

14.3.2. 事件触发

OnLanePlanning的主要逻辑在”RunOnce()”中，在Apollo 3.5之前是定时器触发，3.5改为事件触发，即收到对应的消息之后，就触发执行，这

样做的好处是增加了实时性 [参考](https://github.com/ApolloAuto/apollo/issues/6572) [https://github.com/ApolloAuto/apollo/issues/6572]。

```
void OnLanePlanning::RunOnce(const LocalView& local_view,
                             ADCTrajectory* const
ptr_trajectory_pb) {

    // 初始化Frame
    status = InitFrame(frame_num, stitching_trajectory.back(),
vehicle_state);
    ...

    // 判断是否符合交通规则
    for (auto& ref_line_info : *frame_-
>mutable_reference_line_info()) {
        TrafficDecider traffic_decider;
        traffic_decider.Init(traffic_rule_configs_);
        auto traffic_status =
traffic_decider.Execute(frame_.get(), &ref_line_info);
        if (!traffic_status.ok() || !ref_line_info.IsDrivable())
{
            ref_line_info.SetDrivable(false);
            AWARN << "Reference line " <<
ref_line_info.Lanes().Id()
                << " traffic decider failed";
            continue;
        }
    }

    // 执行计划
    status = Plan(start_timestamp, stitching_trajectory,
ptr_trajectory_pb);

    ...
}

Status OnLanePlanning::Plan(
    const double current_time_stamp,
    const std::vector<TrajectoryPoint>& stitching_trajectory,
    ADCTrajectory* const ptr_trajectory_pb) {

    ...
}
```

```

    // 调用具体的(PUBLIC_ROAD)Planner执行
    auto status = planner_->Plan(stitching_trajectory.back(),
frame_.get(),
                           ptr_trajectory_pb);
...
}

```

上述就是”OnLanePlanning”的执行过程，先是Planner分发器根据配置，选择具体的planner，然后初始化Frame，(PUBLIC_ROAD)planner根据输入帧执行”Plan”方法。

14.4. Planner

我们先看下Planner目录结构，一共实现了5种Planner：

```

.
├── BUILD
├── navi_planner_dispatcher.cc
├── navi_planner_dispatcher.h
├── navi_planner_dispatcher_test.cc
├── on_lane_planner_dispatcher.cc
├── on_lane_planner_dispatcher.h
├── on_lane_planner_dispatcher_test.cc
├── planner_dispatcher.cc
├── planner_dispatcher.h
└── planner.h
    ├── lattice           // lattice planner
    ├── navi              // navi planner
    ├── open_space         // open space planner
    ├── public_road        // public road planner
    └── rtk               // rtk planner

```

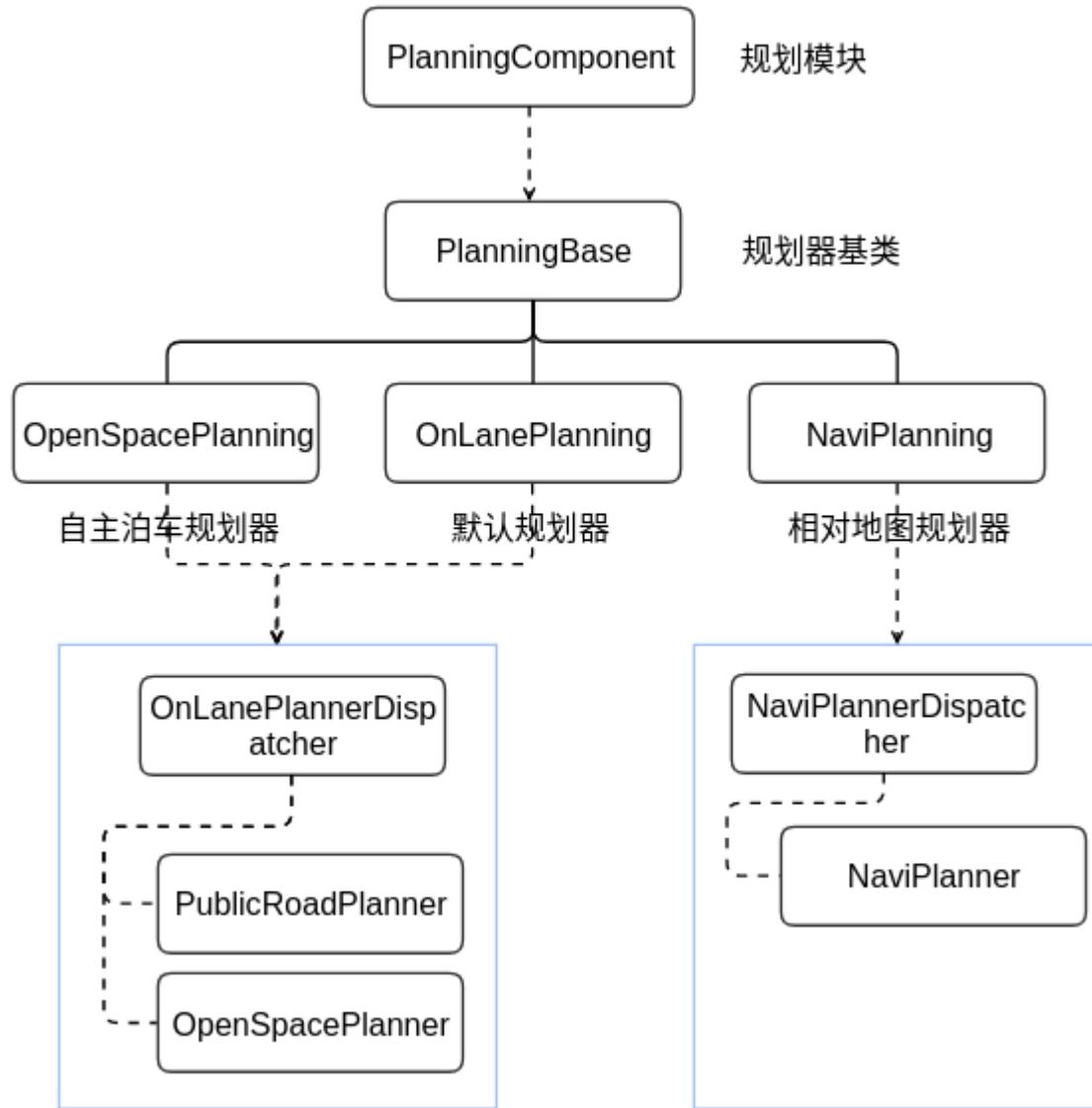
可以看到Planner目录分别实现了Planner发布器和具体的Planner，关于发布器我们后面会根据流程图来介绍，这里先介绍一下5种不同的Planner。

- **rtk** - 根据录制的轨迹来规划行车路线
- **public_road** - 开放道路的轨迹规划器
- **lattice** - 基于网格算法的轨迹规划器

- **navi** - 基于实时相对地图的规划器
- **open_space** - 自主泊车规划器

14.4.1. Planner注册场景

下面我们整理一下planner模块的流程:



1. PlanningComponent在cyber中注册
2. 选择Planning
3. 根据不同的Dispatcher，分发Planner

下面我们主要介绍”PublicRoadPlanner”，主要的实现还是在Init和Plan中。init中主要是注册规划器支持的场景(scenario)。

```
Status PublicRoadPlanner::Init(const PlanningConfig& config)
{
    // 读取public Road配置
    const auto& public_road_config =
        config_.standard_planning_config().planner_public_road_config
();

    // 根据配置注册不同的场景
    for (int i = 0; i <
public_road_config.scenario_type_size(); ++i) {
        const ScenarioConfig::ScenarioType scenario =
            public_road_config.scenario_type(i);
        supported_scenarios.insert(scenario);
    }
    scenario_manager_.Init(supported_scenarios);
}
```

我们看下”PublicRoadPlanner”支持的场景有哪些？

```
// 还是在"/conf/planning_config.pb.txt"中
standard_planning_config {
    planner_type: PUBLIC_ROAD
    planner_type: OPEN_SPACE
    planner_public_road_config {
        // 支持的场景
        scenario_type: LANE_FOLLOW // 车道线保持
        scenario_type: SIDE_PASS // 超车
        scenario_type: STOP_SIGN_UNPROTECTED // 停止
        scenario_type: TRAFFIC_LIGHT_PROTECTED // 红绿灯
        scenario_type: TRAFFIC_LIGHT_UNPROTECTED_LEFT_TURN // 红绿灯左转
        scenario_type: TRAFFIC_LIGHT_UNPROTECTED_RIGHT_TURN // 红绿灯右转
        scenario_type: VALET_PARKING // 代客泊车
    }
}
```

14.4.2. 运行场景

接着看”Plan”中的实现:

```
 Status PublicRoadPlanner::Plan(const TrajectoryPoint&
planning_start_point,
                                Frame* frame,
                                ADCTrajectory*
ptr_computed_trajectory) {
    DCHECK_NOTNULL(frame);
    // 更新场景，决策当前应该执行什么场景
    scenario_manager_.Update(planning_start_point, *frame);
    // 获取当前场景
    scenario_ = scenario_manager_.mutable_scenario();
    // 执行当前场景的任务
    auto result = scenario_->Process(planning_start_point,
frame);

    // 当前场景完成
    if (result == scenario::Scenario::STATUS_DONE) {
        // only updates scenario manager when previous scenario's
status is
        // STATUS_DONE
        scenario_manager_.Update(planning_start_point, *frame);
    } else if (result == scenario::Scenario::STATUS_UNKNOWN) {
        // 当前场景失败
        return Status(common::PLANNING_ERROR, "scenario returned
unknown");
    }
    return Status::OK();
}
```

可以看到”Planner”模块把具体的规划转化成一系列的场景，每次执行规划之前先判断更新当前的场景，然后针对具体的场景去执行。下面 我们先看下”Scenario”模块，然后把这2个模块串起来讲解。

14.5. Scenario

我们同样先看下”Scenario”的目录结构:

```
.
├── bare_intersection
```

```
|-- BUILD
|-- lane_follow           // 车道线保持
|-- narrow_street_u_turn   // 狹窄掉头
|-- scenario.cc
|-- scenario.h
|-- scenario_manager.cc
|-- scenario_manager.h
|-- side_pass              // 超车
|-- stage.cc
|-- stage.h
|-- stop_sign               // 停止
|-- traffic_light           // 红绿灯
|-- util
|-- valet_parking           // 代客泊车
```

其中需要知道场景如何转换，以及每种场景如何执行。几种场景的介绍可以先看下Apollo的官方文档[planning](#) [<https://github.com/ApolloAuto/apollo/blob/master/modules/planning/README.md>]，主要的场景是lane_follow，side_pass和stop_sign。

14.5.1. 场景转换

场景转换的实现在”scenario_manager.cc”中，其中实现了场景注册，创建场景和更新场景的功能。

```
bool ScenarioManager::Init(
    const std::set<ScenarioConfig::ScenarioType>&
supported_scenarios) {
    // 注册场景
    RegisterScenarios();
    default_scenario_type_ = ScenarioConfig::LANE_FOLLOW;
    supported_scenarios_ = supported_scenarios;
    // 创建场景，默认为lane_follow
    current_scenario_ = CreateScenario(default_scenario_type_);
    return true;
}

// 更新场景
void ScenarioManager::Update(const common::TrajectoryPoint&
ego_point,
                             const Frame& frame) {
```

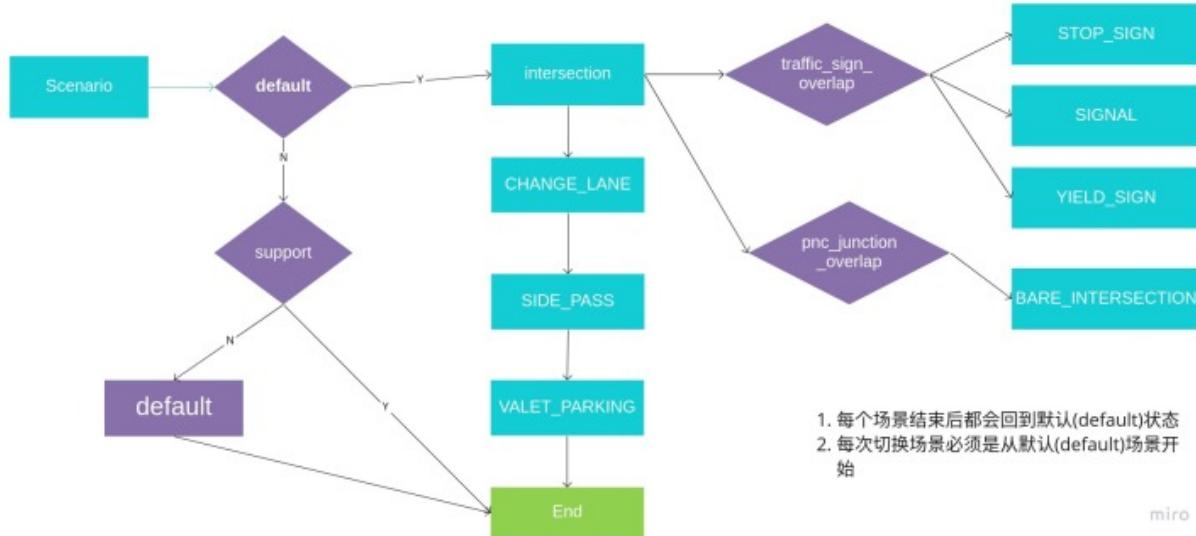
```

CHECK(!frame.reference_line_info().empty());
// 保留当前帧
Observe(frame);
// 场景分发
ScenarioDispatch(ego_point, frame);
}

// 通过一个有限状态机，决定当前的场景
void ScenarioManager::ScenarioDispatch(const
common::TrajectoryPoint& ego_point,
                                         const Frame& frame) {
    ...
}

```

其中”ScenarioDispatch”的状态切换可以参考下图:

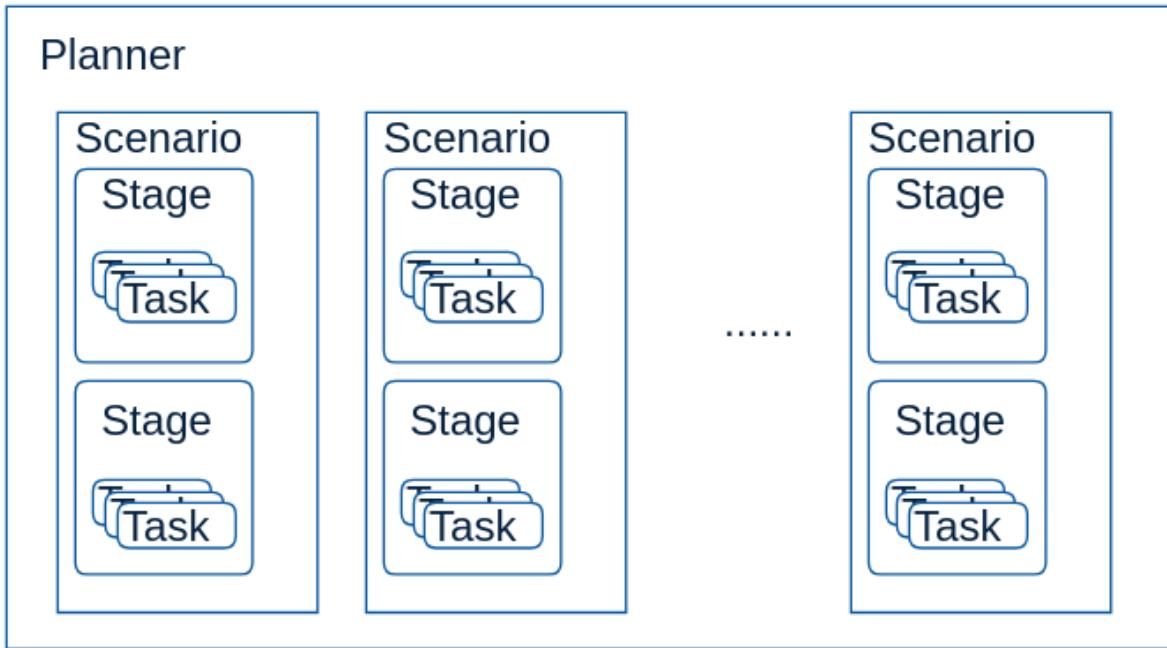


可以看到，每次切换场景必须是从默认场景(LANE_FOLLOW)开始，即每次场景切换之后都会回到默认场景。

ScenarioDispatch目前的代码还没完全完成(有些分支TODO)，而且个人感觉这个实现不够简介和优秀，逻辑看起来有些混乱，不知道是否可以用状态机改进？

14.5.2. 场景运行

场景的执行在”scenario.cc”和对应的场景目录中，实际上每个场景又分为一个或者多个阶段(stage)，每个阶段又由不同的任务(task)组成。执行一个场景，就是顺序执行不同阶段的不同任务。



下面我们来看一个具体的例子， Scenario对应的stage和task在”planning/conf/scenario”中。

```
// Scenario对应的Stage
scenario_type: SIDE_PASS
stage_type: SIDE_PASS_APPROACH_OBSTACLE
stage_type: SIDE_PASS_GENERATE_PATH
stage_type: SIDE_PASS_STOP_ON_WAITPOINT
stage_type: SIDE_PASS_DETECT_SAFETY
stage_type: SIDE_PASS_PASS_OBSTACLE
stage_type: SIDE_PASS_BACKUP

// Stage对应的Task
stage_type: SIDE_PASS_APPROACH_OBSTACLE
enabled: true
task_type: DP_POLY_PATH_OPTIMIZER
task_type: PATH_DECIDER
task_type: SPEED_BOUNDS_PRIORI_DECIDER
task_type: DP_ST_SPEED_OPTIMIZER
task_type: SPEED_DECIDER
task_type: SPEED_BOUNDS_FINAL_DECIDER
```

```
task_type: QP_SPLINE_ST_SPEED_OPTIMIZER
```

```
// 以此类推
```

由于Scenario都是顺序执行，只需要判断这一阶段是否结束，然后转到下一个阶段就可以了。具体的实现在：

```
Scenario::ScenarioStatus Scenario::Process(
    const common::TrajectoryPoint& planning_init_point,
    Frame* frame) {
    ...

    // 如果当前阶段完成，则退出
    if (current_stage_->stage_type() ==
        ScenarioConfig::NO_STAGE) {
        scenario_status_ = STATUS_DONE;
        return scenario_status_;
    }

    // 进入下一阶段执行或者错误处理
    auto ret = current_stage_->Process(planning_init_point,
frame);
    switch (ret) {
        case Stage::ERROR: {
            ...
        }
        case Stage::RUNNING: {
            ...
        }
        case Stage::FINISHED: {
            ...
        }
        default: {
            ...
        }
    }
    return scenario_status_;
}
```

我们接着看一下Stage中”Process”的执行：

```
Stage::StageStatus LaneFollowStage::Process(
    const TrajectoryPoint& planning_start_point, Frame*
frame) {
    ...
    // 根据参考线规划
}
```

```

        auto cur_status =
            PlanOnReferenceLine(planning_start_point, frame,
&reference_line_info);
        ...
}

// LANE_FOLLOW中的PlanOnReferenceLine
Status LaneFollowStage::PlanOnReferenceLine(
    const TrajectoryPoint& planning_start_point, Frame*
frame,
    ReferenceLineInfo* reference_line_info) {

    // 顺序执行stage中的任务
    for (auto* optimizer : task_list_) {
        const double start_timestamp = Clock::NowInSeconds();
        // 任务
        ret = optimizer->Execute(frame, reference_line_info);
    }

    // 增加障碍物的代价
    for (const auto* obstacle :
        reference_line_info->path_decision()-
>obstacles().Items()) {
        if (obstacle->IsVirtual()) {
            continue;
        }
        if (!obstacle->IsStatic()) {
            continue;
        }
        if (obstacle->LongitudinalDecision().has_stop()) {
            ...
        }
    }

    // 返回参考线
    reference_line_info->SetTrajectory(trajectory);
    reference_line_info->SetDrivable(true);
    return Status::OK();
}

```

上面是用”LaneFollowStage”中的”PlanOnReferenceLine”来举例子，不同场景中的”PlanOnReferenceLine”实现可能也不一样，这样设计的好处是，当发现一个场景有问题，需要修改不会影响到其他的场景。同时也可以针对不同场景做优化，比通用的规划更加适合单独的场景。每种场景都有一个专门的目录来进行优化。

接下来我们看下Task是如何执行的。

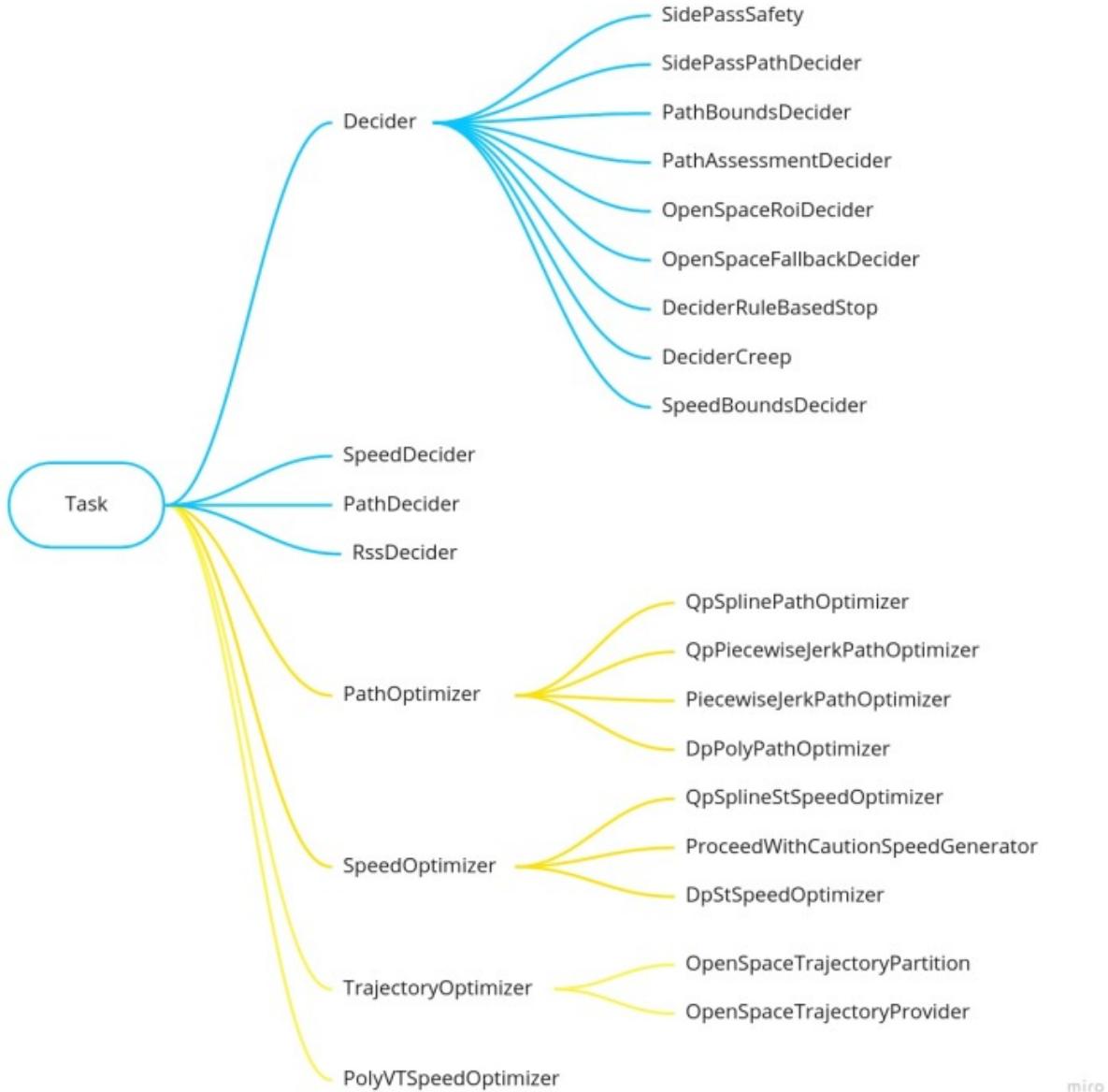
14.6. Task

我们先看Task的目录结构：

```
.  
|   -- BUILD  
|   -- deciders      // 决策器  
|   -- optimizers    // 优化器  
|   -- rss  
|   -- smoothers     // 平滑器  
|   -- task.cc  
|   -- task_factory.cc  
|   -- task_factory.h  
|   -- task.h
```

可以看到每个Task都可以对应到一个决策器或者优化器（平滑器不作为Task，单独作为一个类）。

每个Task都实现了”Execute”方法，而每个决策器和优化器都继承至Task类。可以参考下图：



Task类的生成用到了设计模式的工厂模式，通过”TaskFactory”类生产不同的Task类。

14.6.1. DP & QP

Task中的决策器和优化器采用的方法有DP和QP:

- **DP** 动态规划
- **QP** 二次规划

QP方法的路径优化和速度优化可以参考apollo文档:

- [QP-Spline-Path Optimizer](#)
[https://github.com/ApolloAuto/apollo/blob/master/docs/specs/qp_spline_path_optimizer.md]
- [QP-Spline-ST-Speed Optimizer](#)
[https://github.com/ApolloAuto/apollo/blob/master/docs/specs/qp_spline_st_speed_optimizer.md]

14.7. Reference

- [解析百度Apollo之决策规划模块](#) [<https://paul.pub/apollo-planning/#id-planning%E4%B8%8Eplanner>]
- [Open Space Planner Algorithm](#)
[https://github.com/ApolloAuto/apollo/blob/master/docs/specs/Open_Space_Planner.md]

15. Prediction

悟已往之不谏,知来者之可追

15.1. 介绍

首先建议先阅读官方文档(readme.md), 里面说明了数据流向, 也就是说预测模块是直接接收的感知模块给出的障碍物信息, 这和CV领域的传统预测任务有区别, CV领域的预测任务不需要先识别物体, 只需要根据物体的特征, 对比前后2帧, 然后得出物体的位置, 也就说甚至不需要物体识别, 业界之所以不这么做的原因是检测物体太耗时了。当然也有先检测物体再做跟踪的, 也就是说目前apollo中的物体检测实际上是采用的第二种方法, 这也可以理解, 反正感知模块一定会工作, 而且一定要检测物体, 所以何不把这个信息直接拿过来用呢? 这和人类似, 逐帧跟踪指定特征的对象, 就是物体的轨迹, 然后再根据现有的轨迹预测物体未来的轨迹。预测的轨迹和障碍物信息发送给规划(planning)模块使用。

15.2. 目录结构

预测模块的目录结构如下:

```
.  
├── BUILD  
├── common          // common目录, 公用类  
├── conf            // 启动配置  
├── container       // 1. 消息容器  
├── dag             // 启动文件dag  
├── data            // 模型文件路径  
├── evaluator       // 3. 评估者  
├── images          // 文档(图片)  
├── launch          // 启动, 加载模块  
├── network          // 工具  
└── pipeline         // 预测模块主入口  
└── prediction_component.cc
```

```
|--- prediction_component.h  
|--- prediction_component_test.cc  
|--- predictor // 4. 预测器  
|--- proto // protobuf消息格式  
|--- README_cn.md // 文档(中文介绍, 建议直接看英文)  
|--- README.md // 文档(英文介绍)  
|--- scenario // 2. 场景  
|--- submodules // 子模块???  
|--- testdata // 测试数据  
|--- util // 工具类
```

可以看到预测模块主要是分为2大块功能，一是实时的预测执行过程，一是工具类(离线验证？)：

- 在线预测流程 - container -> scenario -> evaluator -> predictor
- 离线流程 - pipeline (util) 提取bag包中的数据给离线测试用？

15.3. 预测模块(PredictionComponent类)

预测模块和其它模块一样，都是在cyber中注册，具体的实现在”prediction_component.h”和”prediction_component.cc”中，我们知道cyber模块有2种消息触发模式，一种是定时器触发，一种是消息触发，而预测为消息触发模式。预测模块的输入消息为：

1. **perception::PerceptionObstacles** - 感知模块输出的障碍物信息
2. **planning::ADCTrajectory** - 规划模块输出的行驶路径
3. **localization::LocalizationEstimate** - 车辆当前的位置

输出消息：

1. **prediction::PredictionObstacles** - 预测模块输出的障碍物信息

预测模块和所有其它模块一样，都实现了”cyber::Component”基类中的”Init()”和”Proc()”方法，分别进行初始化和消息触发调用，调用由框架自动执行，关于cyber如何调用和执行每个模块，可以参考cyber模块的介绍，下面我们将主要介绍这2个方法。

15.3.1. 初始化(Init())

预测模块的初始化在”PredictionComponent::Init()”中进行，主要是注册消息读取和发送控制器，用来读取和发送消息，需要注意的是初始化过程中也对”**MessageProcess**”类进行了初始化，而”**MessageProcess**”实现了预测模块的整个消息处理流程。

```
bool PredictionComponent::Init() {
    component_start_time_ = Clock::NowInSeconds();

    // 预测模块消息处理流程初始化
    if (!MessageProcess::Init()) {
        return false;
    }

    // 规划模块消息读取者
    planning_reader_ = node_->CreateReader<ADCTrajectory>(
        FLAGS_planning_trajectory_topic, nullptr);

    // 定位模块消息读取者
    localization_reader_ =
        node_->CreateReader<localization::LocalizationEstimate>
    (
        FLAGS_localization_topic, nullptr);
    // 故事读取???
    storytelling_reader_ = node_->CreateReader<storytelling::Stories>(
        FLAGS_storytelling_topic, nullptr);
    // 预测消息发送者
    prediction_writer_ =
        node_->CreateWriter<PredictionObstacles>
    (FLAGS_prediction_topic);
    // 中间消息的发送者，这一块的目的是什么？？
    container_writer_ =
        node_->CreateWriter<SubmoduleOutput>
    (FLAGS_container_topic_name);

    adc_container_writer_ = node_->CreateWriter<ADCTrajectoryContainer>(
        FLAGS_adccontainer_topic_name);

    perception_obstacles_writer_ = node_->CreateWriter<PerceptionObstacles>(
        FLAGS_perception_obstacles_topic_name);
```

```
    return true;
}
```

下面我们接着看消息回调执行函数

15.3.2. 回调执行(Proc)

回调执行函数会执行以下过程:

```
bool PredictionComponent::Proc(
    const std::shared_ptr<PerceptionObstacles>&
perception_obstacles) {
    // 1. 如果使用lego, 则执行子过程
    if (FLAGS_use_lego) {
        return ContainerSubmoduleProcess(perception_obstacles);
    }
    // 2. 否则就执行端到端的过程
    return PredictionEndToEndProc(perception_obstacles);
}
```

下面我们分别看下这2个过程有什么差异? 我们先看

15.3.3. ContainerSubmoduleProcess

子过程的函数如下:

```
bool PredictionComponent::ContainerSubmoduleProcess(
    const std::shared_ptr<PerceptionObstacles>&
perception_obstacles) {
    constexpr static size_t kHistorySize = 10;
    const auto frame_start_time = absl::Now();
    // Read localization info. and call OnLocalization to
    update
    // the PoseContainer.
    // 读取定位信息, 并且更新位置容器
    localization_reader_->Observe();
    auto ptr_localization_msg = localization_reader_-
>GetLatestObserved();
    if (ptr_localization_msg == nullptr) {
        AERROR << "Prediction: cannot receive any localization
```

```

message.";
    return false;
}
MessageProcess::OnLocalization(*ptr_localization_msg);

// Read planning info. of last frame and call OnPlanning to
update
// the ADCTrajectoryContainer
// 读取规划路径，并且更新路径容器
planning_reader_->Observe();
auto ptr_trajectory_msg = planning_reader_-
>GetLatestObserved();
if (ptr_trajectory_msg != nullptr) {
    MessageProcess::OnPlanning(*ptr_trajectory_msg);
}

// Read storytelling message and call OnStorytelling to
update the
// StoryTellingContainer
// 读取故事消息，并且更新故事容器？？
storytelling_reader_->Observe();
auto ptr_storytelling_msg = storytelling_reader_-
>GetLatestObserved();
if (ptr_storytelling_msg != nullptr) {
    MessageProcess::OnStoryTelling(*ptr_storytelling_msg);
}

MessageProcess::ContainerProcess(*perception_obstacles);
// 障碍物容器指针
auto obstacles_container_ptr =
    ContainerManager::Instance()-
>GetContainer<ObstaclesContainer>(
        AdapterConfig::PERCEPTION_OBSTACLES);
CHECK_NONNULL(obstacles_container_ptr);
// 路径规划容器指针
auto adc_trajectory_container_ptr =
    ContainerManager::Instance()-
>GetContainer<ADCTrajectoryContainer>(
        AdapterConfig::PLANNING_TRAJECTORY);
CHECK_NONNULL(adc_trajectory_container_ptr);
// 输出障碍物信息
SubmoduleOutput submodule_output =
    obstacles_container_ptr-

```

```

>GetSubmoduleOutput(kHistorySize,
frame_start_time);
// 发布消息
container_writer_->Write(submodule_output);
adc_container_writer_-
>Write(*adc_trajectory_container_ptr);
perception_obstacles_writer_->Write(*perception_obstacles);
return true;
}

```

看起来上述函数只是计算中间过程，并且发布消息到订阅节点。具体的用途需要结合业务来分析（具体的业务场景是什么？？？）。

15.3.4. PredictionEndToEndProc

端到端的过程函数如下：

```

bool PredictionComponent::PredictionEndToEndProc(
    const std::shared_ptr<PerceptionObstacles>&
perception_obstacles) {

    // Update relative map if needed
    // 如果是导航模式，需要判断地图是否准备好
    if (FLAGS_use_navigation_mode && !PredictionMap::Ready()) {
        AERROR << "Relative map is empty.";
        return false;
    }

    // Read localization info. and call OnLocalization to
    update
    // the PoseContainer.
    // 读取定位消息，并且处理消息
    localization_reader_->Observe();
    auto ptr_localization_msg = localization_reader_-
>GetLatestObserved();
    MessageProcess::OnLocalization(*ptr_localization_msg);

    // Read storytelling message and call OnStorytelling to
    update the
    // StoryTellingContainer
    // 读取并且处理故事消息？？
}

```

```

storytelling_reader_->Observe();
auto ptr_storytelling_msg = storytelling_reader_-
>GetLatestObserved();
if (ptr_storytelling_msg != nullptr) {
    MessageProcess::OnStoryTelling(*ptr_storytelling_msg);
}

// Read planning info. of last frame and call OnPlanning to
update
// the ADCTrajectoryContainer
// 读取并且处理规划消息
planning_reader_->Observe();
auto ptr_trajectory_msg = planning_reader_-
>GetLatestObserved();
if (ptr_trajectory_msg != nullptr) {
    MessageProcess::OnPlanning(*ptr_trajectory_msg);
}

// Get all perception_obstacles of this frame and call
OnPerception to
// process them all.
// 处理障碍物消息
auto perception_msg = *perception_obstacles;
PredictionObstacles prediction_obstacles;
MessageProcess::OnPerception(perception_msg,
&prediction_obstacles);

// 填充发布预测的障碍物轨迹消息
// Postprocess prediction obstacles message

prediction_obstacles.set_start_timestamp(frame_start_time_);
...
common::util::FillHeader(node_->Name(),
&prediction_obstacles);
prediction_writer_->Write(prediction_obstacles);
return true;
}

```

15.4. 消息处理(MessageProcess)

可以看到上述过程都是在MessageProcess中处理完成的，那么我们先看下MessageProcess的执行过程。

15.4.1. 初始化(Init)

消息处理的初始化首先在”PredictionComponent::Init()”中调用，下面我们看下实现了哪些功能：

```
bool MessageProcess::Init() {
    // 1. 初始化容器
    InitContainers();
    // 2. 初始化评估器
    InitEvaluators();
    // 3. 初始化预测器
    InitPredictors();

    // 如果为导航模式，则判断地图是否加载
    if (!FLAGS_use_navigation_mode && !PredictionMap::Ready())
    {
        AERROR << "Map cannot be loaded.";
        return false;
    }

    return true;
}
```

上述子过程的初始化就是从配置文件读取配置，并且初始化对应的类，结构相对比较简单，这里就不一一介绍了。

15.4.2. 消息处理

定位，规划和故事的消息处理相对比较简单，主要是向对应的容器中插入数据（每个容器都实现了Insert()方法），下面着重介绍感知模块消息的处理过程，该过程也输出了最后的结果。

```
void MessageProcess::OnPerception(
    const perception::PerceptionObstacles&
perception_obstacles,
    PredictionObstacles* const prediction_obstacles) {
    // 1. 分析场景和处理容器中的数据
```

```

ContainerProcess(perception_obstacles);

// 获取障碍物容器
auto ptr_obstacles_container =
    ContainerManager::Instance()-
>GetContainer<ObstaclesContainer>(
    AdapterConfig::PERCEPTION_OBSTACLES);
// 获取规划曲线容器
auto ptr_ego_trajectory_container =
    ContainerManager::Instance()-
>GetContainer<ADCTrajectoryContainer>(
    AdapterConfig::PLANNING_TRAJECTORY);

// Insert features to FeatureOutput for offline_mode
// 离线模式，保存障碍物曲线？？
if (FLAGS_prediction_offline_mode ==
PredictionConstants::kDumpFeatureProto) {
    for (const int id :
        ptr_obstacles_container-
>curr_frame_movable_obstacle_ids()) {
        Obstacle* obstacle_ptr = ptr_obstacles_container-
>GetObstacle(id);
        if (obstacle_ptr == nullptr) {
            AERROR << "Null obstacle found.";
            continue;
        }
        if (!obstacle_ptr->latest_feature().IsInitialized()) {
            AERROR << "Obstacle [" << id << "] has no latest
feature.";
            continue;
        }
        // TODO(all): the adc trajectory should be part of
features for learning
        //           algorithms rather than part of the
feature.proto
        /*
         *obstacle_ptr->mutable_latest_feature()->
mutable_adc_trajectory_point() =
            ptr_ego_trajectory_container-
>adc_trajectory().trajectory_point();
        */
        FeatureOutput::InsertFeatureProto(obstacle_ptr-
>latest_feature());
    }
}

```

```

        ADEBUG << "Insert feature into feature output";
    }
    // Not doing evaluation on offline mode
    return;
}

// Make evaluations
// 2. 进行评估
EvaluatorManager::Instance() ->Run(ptr_obstacles_container);
if (FLAGS_prediction_offline_mode ==
    PredictionConstants::kDumpDataForLearning ||

    FLAGS_prediction_offline_mode ==
PredictionConstants::kDumpFrameEnv) {
    return;
}

// Make predictions
// 3. 进行预测
PredictorManager::Instance() ->Run(perception_obstacles,
ptr_ego_trajectory_container,
ptr_obstacles_container);

// Get predicted obstacles
// 4. 输出预测结果
*prediction_obstacles = PredictorManager::Instance() -
>prediction_obstacles();
}

```

上面的消息处理过程实际上是整个预测的过程，分为以下几个步骤：



下面主要分析各个模块的输入是什么，输出是什么？以及它们的作用？

15.5. 容器(container)

容器的作用主要是存储对应类型的消息，用来给评估器(evaluator)使用。

15.5.1. 容器基类(Container)

首先介绍容器基类”Container”类，主要申明了”Insert”方法：

```
class Container {
public:
    Container() = default;

    virtual ~Container() = default;

    // 子类重写Insert方法
    virtual void Insert(const ::google::protobuf::Message&
message) = 0;
};
```

15.5.2. 容器管理(ContainerManager)

上述基类有4个扩展类：PoseContainer，ObstaclesContainer，ADCTrajectoryContainer和StoryTellingContainer，分别存储不同的消息类型，而这些容器的管理和注册在ContainerManager中，下面我们看下ContainerManager类的具体实现。

```
class ContainerManager {
public:
    // 初始化
    void Init(const common::adapter::AdapterManagerConfig
&config);

    // 获取容器模板
    template <typename T>
    T *GetContainer(const
common::adapter::AdapterConfig::MessageType &type) {
        auto key_type = static_cast<int>(type);
        if (containers_.find(key_type) != containers_.end()) {
            return static_cast<T *>(containers_[key_type].get());
        }
        return nullptr;
```

```
}

// gtest 可以访问被测试类的私有成员
FRIEND_TEST(FeatureExtractorTest, junction);
FRIEND_TEST(ScenarioManagerTest, run);

private:
    // 注册容器
    void RegisterContainer(
        const common::adapter::AdapterConfig::MessageType
&type);

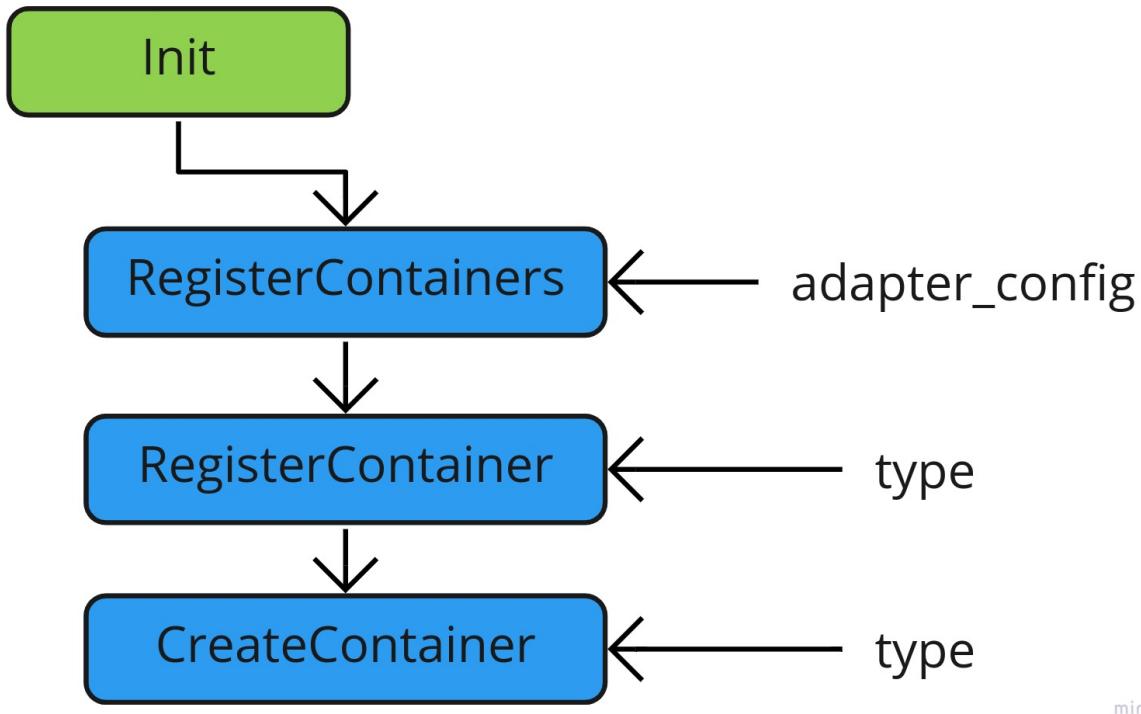
    // 创建容器
    std::unique_ptr<Container> CreateContainer(
        const common::adapter::AdapterConfig::MessageType
&type);

    // 注册所有的容器
    void RegisterContainers();

private:
    // 容器map，保存所有已经创建的容器
    std::unordered_map<int, std::unique_ptr<Container>>
containers_;

    // 配置文件
    common::adapter::AdapterManagerConfig config_;
    // 单例
    DECLARE_SINGLETON(ContainerManager)
};
```

ContainerManager类的执行过程如下：



可以看到容器管理器在Init中根据配置文件创建并且注册对应类型的容器，并且放到”containers_”中，而”containers_”为map结构，其中key为容器的类型，而值为对应的容器。注册之后通过”GetContainer”获取对应类型的容器。总结起来就是”ContainerManager”负责从配置文件注册并且管理对应的容器。

15.5.3. 姿态容器(PoseContainer)

姿态容器主要是根据”localization::LocalizationEstimate”消息转换为障碍物(perception::PerceptionObstacle)信息，这其实是把本车转换为障碍物。

```
class PoseContainer : public Container {
public:
    PoseContainer() = default;

    virtual ~PoseContainer() = default;

    // 插入消息
    void Insert(const ::google::protobuf::Message& message)
```

```

override;

    // 获取障碍物信息(这里就是本车)
    const perception::PerceptionObstacle*
ToPerceptionObstacle();

    double GetTimestamp();

private:
    // 根据本车位置转换为障碍物信息
    void Update(const localization::LocalizationEstimate&
localization);

public:
    // 类型赋值为车
    static const perception::PerceptionObstacle::Type type_ =
        perception::PerceptionObstacle::VEHICLE;

private:
    // 障碍物指针
    std::unique_ptr<perception::PerceptionObstacle>
obstacle_ptr_;
};


```

姿态容器(PoseContainer)中主要注意”PoseContainer::Update”方法的实现，即把当前车辆转换为障碍物信息，由于该函数实现比较简单，这里就不展开了。

15.5.4. 规划轨迹容器(ADCTrajectoryContainer)

ADCTrajectoryContainer类的具体实现如下：

```

class ADCTrajectoryContainer : public Container {
public:

    ADCTrajectoryContainer();
    virtual ~ADCTrajectoryContainer() = default;

    // 插入消息
    void Insert(const ::google::protobuf::Message& message)
override;

```

```
// 是否受到保护???
bool IsProtected() const;

// 确认点是否在路口之中
bool IsPointInJunction(const common::PathPoint& point)
const;

// 当前的道路序列是否和规划的轨迹有重叠???
bool HasOverlap(const LaneSequence& lane_sequence) const;

// 设置位置
void SetPosition(const common::math::Vec2d& position);

// 获取规划轨迹中的路口
std::shared_ptr<const hdmap::JunctionInfo> ADCJunction()
const;

// 获取到路口的距离
double ADCDistanceToJunction() const;

// 获取规划轨迹
const planning::ADCTrajectory& adc_trajectory() const;

// 道路是否在参考线中
bool IsLaneIdInReferenceLine(const std::string& lane_id)
const;

bool IsLaneIdInTargetReferenceLine(const std::string&
lane_id) const;
// 获取道路序列
const std::vector<std::string>& GetADCLaneIDSequence()
const;

const std::vector<std::string>&
GetADCTargetLaneIDSequence() const;
// 设置路口???
void SetJunction(const std::string& junction_id, const
double distance);

private:
// 设置路口的形状
void SetJunctionPolygon();
```

```

// 设置道路序列
void SetLaneSequence();

void SetTargetLaneSequence();
// 道路id转换为字符串
std::string ToString(const std::unordered_set<std::string>&
lane_ids);

std::string ToString(const std::vector<std::string>&
lane_ids);

private:
planning::ADCTrajectory adc_trajectory_; // 规划轨迹
common::math::Polygon2d adc_junction_polygon_; // 轨迹的路口形状
std::shared_ptr<const hdmap::JunctionInfo>
adc_junction_info_ptr_;
double s_dist_to_junction_;
std::unordered_set<std::string> adc_lane_ids_;
std::vector<std::string> adc_lane_seq_; // 轨迹道路序列
std::unordered_set<std::string> adc_target_lane_ids_;
std::vector<std::string> adc_target_lane_seq_;
};

```

还是一样查看对应的”Insert”函数中实现了哪些功能？

```

void ADCTrajectoryContainer::Insert(
    const ::google::protobuf::Message& message) {
adc_lane_ids_.clear();
adc_lane_seq_.clear();
adc_target_lane_ids_.clear();
adc_target_lane_seq_.clear();
adc_junction_polygon_ = std::move(Polygon2d());
// 获取规划轨迹
adc_trajectory_.CopyFrom(dynamic_cast<const ADCTrajectory&>
(message));

// 设置道路序列
SetLaneSequence();

// 设置目标道路序列

```

```
    SetTargetLaneSequence();  
}
```

可以看到”Insert”函数主要实现了规划轨迹的赋值操作，也就是保存规划轨迹的信息，而其它函数的功能，主要用来场景(**scenario**)识别中，我们在看到具体的场景识别的时候再展开分析。

故事容器(**StoryTellingContainer**)

StoryTellingContainer类和其它上述容器的实现类似：

```
class StoryTellingContainer : public Container {  
public:  
    StoryTellingContainer() = default;  
    virtual ~StoryTellingContainer() = default;  
  
    // 插入storytelling::Stories信息  
    void Insert(const ::google::protobuf::Message& message)  
override;  
  
    // 获取路口信息  
    std::shared_ptr<const hdmap::JunctionInfo> ADCJunction()  
const;  
  
    // 获取路口Id  
    const std::string& ADCJunctionId() const;  
  
    // 计算到路口的距离  
    double ADCDistanceToJunction() const;  
  
private:  
    apollo::storytelling::CloseToJunction close_to_junction_;  
};
```

StoryTellingContainer类实际上接收”**storytelling**”模块的消息，来获取当前车和路口的距离。

15.5.5. 障碍物容器(**ObstaclesContainer**)

我们最后介绍障碍物容器，实际上障碍物分为2块，一种是障碍物(Obstacle)，一种是障碍物簇(ObstacleClusters)，下面我们分别开始介绍：

```
class ObstaclesContainer : public Container {
public:
    ObstaclesContainer();
    explicit ObstaclesContainer(const SubmoduleOutput&
submodule_output);
    virtual ~ObstaclesContainer() = default;

    // 插入障碍物(PerceptionObstacles)信息
    void Insert(const ::google::protobuf::Message& message)
override;

    // 插入障碍物信息和时间戳
    void InsertPerceptionObstacle(
        const perception::PerceptionObstacle&
perception_obstacle,
        const double timestamp);

    // 根据特征创建并且插入对应的障碍物
    void InsertFeatureProto(const Feature& feature);

    // 创建道路图结构
    void BuildLaneGraph();

    // 创建路口的特征
    void BuildJunctionFeature();

    // 根据id获取障碍物
    Obstacle* GetObstacle(const int id);

    // 清空障碍物容器???
    void Clear();

    void CleanUp();

    size_t NumOfObstacles() { return ptr_obstacles_.size(); }

    // 根据id获取障碍物
    const apollo::perception::PerceptionObstacle&
GetPerceptionObstacle(
```

```
    const int id);

    // 当前帧的移动障碍物id
    const std::vector<int>& curr_frame_movable_obstacle_ids();

    // 当前帧不能移动的障碍物id
    const std::vector<int>&
    curr_frame_unmovable_obstacle_ids();

    // 当前帧中不能忽略的障碍物信息
    const std::vector<int>&
    curr_frame_considered_obstacle_ids();

    // 设置不能忽略的障碍物id
    void SetConsideredObstacleIds();

    // 当前帧中的障碍物id
    std::vector<int> curr_frame_obstacle_ids();

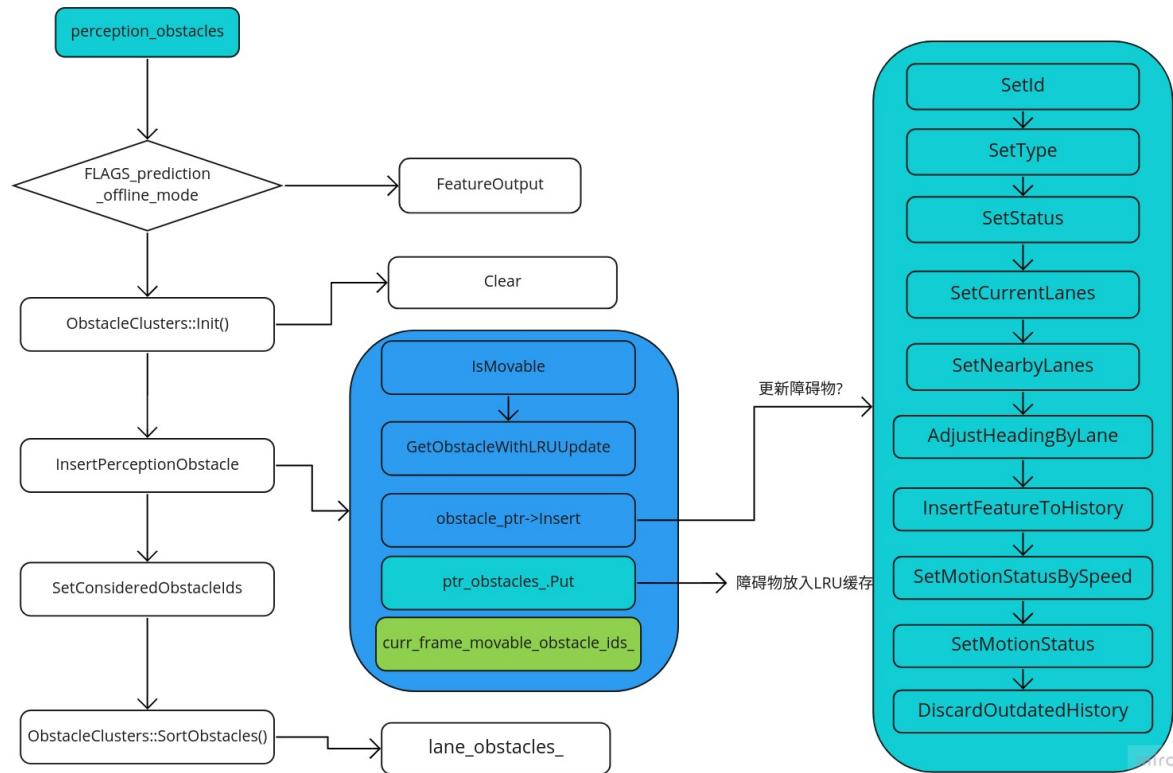
    double timestamp() const;

    // 获取子模块输出
    SubmoduleOutput GetSubmoduleOutput(
        const size_t history_size, const absl::Time&
frame_start_time);

private:
    // 根据最近最少使用的原则获取障碍物
    Obstacle* GetObstacleWithLRUUpdate(const int obstacle_id);
    // 是否移动
    bool IsMovable(const perception::PerceptionObstacle&
perception_obstacle);

private:
    double timestamp_ = -1.0;
    common::util::LRUCache<int, std::unique_ptr<Obstacle>>
ptr_obstacles_;
    std::vector<int> curr_frame_movable_obstacle_ids_;
    std::vector<int> curr_frame_unmovable_obstacle_ids_;
    std::vector<int> curr_frame_considered_obstacle_ids_;
};
```

可以看到障碍物容器主要是根据不同的障碍物测量保存障碍物信息，下面我们看下相应的流程图：



可以看到上述过程中的主要功能实现都在”Obstacle”类中，下面我们主要分析下”Obstacle”类中的几个函数。

我们接着看”Obstacle”类的消息插入函数：

```

bool Obstacle::Insert(const PerceptionObstacle&
perception_obstacle,
                      const double timestamp,
                      const int prediction_obstacle_id) {
    // 是否是过时的消息
    if (ReceivedOlderMessage(timestamp)) {
        return false;
    }

    // 把障碍物的信息赋值给feature，包括Id, Type, Status
    Feature feature;
    if (!SetId(perception_obstacle, &feature,
prediction_obstacle_id)) {
        return false;
    }
  
```

```

    }

    SetType(perception_obstacle, &feature);

    SetStatus(perception_obstacle, timestamp, &feature);

    // Set obstacle lane features
    // 设置障碍物lane特征
    if (type_ != PerceptionObstacle::PEDESTRIAN) {
        SetCurrentLanes(&feature);
        SetNearbyLanes(&feature);
    }

    // 根据lane调整车辆的方向
    if (FLAGS_adjust_vehicle_heading_by_lane &&
        type_ == PerceptionObstacle::VEHICLE) {
        AdjustHeadingByLane(&feature);
    }

    // 将障碍物特征保存到历史(feature_history_)
    InsertFeatureToHistory(feature);

    // 设置障碍物运动状态(set_is_still)
    if (FLAGS_use_navigation_mode) {
        SetMotionStatusBySpeed();
    } else {
        SetMotionStatus();
    }

    // 删除过去的历史特征(feature_history_)
    DiscardOutdatedHistory();
    return true;
}

```

其中还用到了卡尔曼(KalmanFilter)滤波？？？

我们接着看BuildLaneGraph函数：

```

void Obstacle::BuildLaneGraph() {

    Feature* feature = mutable_latest_feature();
    // No need to BuildLaneGraph for those non-moving
}

```

```

obstacles.

if (feature->is_still() && id_ != FLAGS_ego_vehicle_id) {
    ADEBUG << "Not build lane graph for still obstacle";
    return;
}
if (feature->lane().lane_graph().lane_sequence_size() > 0)
{
    ADEBUG << "Not build lane graph for an old obstacle";
    return;
}
double speed = feature->speed();
double t_max = FLAGS_prediction_trajectory_time_length;
auto estimated_move_distance = speed * t_max;

double road_graph_search_distance = std::fmax(
    estimated_move_distance,
FLAGS_min_prediction_trajectory_spatial_length);

bool is_in_junction = HasJunctionFeatureWithExits();
std::unordered_set<std::string> exit_lane_id_set;
if (is_in_junction) {
    for (const auto& exit : feature-
>junction_feature().junction_exit()) {
        exit_lane_id_set.insert(exit.exit_lane_id());
    }
}

// BuildLaneGraph for current lanes:
// Go through all the LaneSegments in current_lane,
// construct up to max_num_current_lane of them.
int seq_id = 0;
int curr_lane_count = 0;
for (auto& lane : feature->lane().current_lane_feature()) {
    std::shared_ptr<const LaneInfo> lane_info =
        PredictionMap::LaneById(lane.lane_id());
    LaneGraph lane_graph = ObstacleClusters::GetLaneGraph(
        lane.lane_s(), road_graph_search_distance, true,
        lane_info);
    if (lane_graph.lane_sequence_size() > 0) {
        ++curr_lane_count;
    }
    for (const auto& lane_seq : lane_graph.lane_sequence()) {
        if (is_in_junction && !HasJunctionExitLane(lane_seq,

```

```

        exit_lane_id_set)) {
            continue;
        }
        LaneSequence* lane_seq_ptr =
            feature->mutable_lane()->mutable_lane_graph()-
>add_lane_sequence();
        lane_seq_ptr->CopyFrom(lane_seq);
        lane_seq_ptr->set_lane_sequence_id(seq_id++);
        lane_seq_ptr->set_lane_s(lane.lane_s());
        lane_seq_ptr->set_lane_l(lane.lane_l());
        lane_seq_ptr->set_vehicle_on_lane(true);
        lane_seq_ptr->set_lane_type(lane.lane_type());
        SetLaneSequenceStopSign(lane_seq_ptr);
        ADEBUG << "Obstacle [" << id_ << "] set a lane sequence
["
            << lane_seq.ShortDebugString() << ".";
    }
    if (curr_lane_count >= FLAGS_max_num_current_lane) {
        break;
    }
}

// BuildLaneGraph for neighbor lanes.
int nearby_lane_count = 0;
for (auto& lane : feature->lane().nearby_lane_feature()) {
    std::shared_ptr<const LaneInfo> lane_info =
        PredictionMap::LaneById(lane.lane_id());
    LaneGraph lane_graph = ObstacleClusters::GetLaneGraph(
        lane.lane_s(), road_graph_search_distance, false,
    lane_info);
    if (lane_graph.lane_sequence_size() > 0) {
        ++nearby_lane_count;
    }
    for (const auto& lane_seq : lane_graph.lane_sequence()) {
        if (is_in_junction && !HasJunctionExitLane(lane_seq,
exit_lane_id_set)) {
            continue;
        }
        LaneSequence* lane_seq_ptr =
            feature->mutable_lane()->mutable_lane_graph()-
>add_lane_sequence();
        lane_seq_ptr->CopyFrom(lane_seq);
        lane_seq_ptr->set_lane_sequence_id(seq_id++);
    }
}

```

```

        lane_seq_ptr->set_lane_s(lane.lane_s());
        lane_seq_ptr->set_lane_l(lane.lane_l());
        lane_seq_ptr->set_vehicle_on_lane(false);
        lane_seq_ptr->set_lane_type(lane.lane_type());
        SetLaneSequenceStopSign(lane_seq_ptr);
        ADEBUG << "Obstacle [" << id_ << "] set a lane sequence
    [
        << lane_seq.ShortDebugString() << ].";
    }
    if (nearby_lane_count >= FLAGS_max_num_nearby_lane) {
        break;
    }
}

if (feature->has_lane() && feature-
>lane().has_lane_graph()) {
    SetLanePoints(feature);
    SetLaneSequencePath(feature->mutable_lane()-
>mutable_lane_graph());
}
ADEBUG << "Obstacle [" << id_ << "] set lane graph
features.";
}

```

15.5.6. 道路图(RoadGraph)

道路图的实现在”road_graph.h”和”road_graph.cc”中，构建道路图在”BuildLaneGraph”方法中，而主要的实现则在”ConstructLaneSequence”中：

```

void RoadGraph::ConstructLaneSequence(
    const bool search_forward_direction, const double
    accumulated_s,
    const double curr_lane_seg_s, std::shared_ptr<const
    LaneInfo> lane_info_ptr,
    const int graph_search_horizon, const bool
    consider_lane_split,
    std::list<LaneSegment>* const lane_segments,
    LaneGraph* const lane_graph_ptr) const {
    // Sanity checks.
    if (lane_info_ptr == nullptr) {
        AERROR << "Invalid lane.";
    }
}

```

```

        return;
    }
    if (graph_search_horizon < 0) {
        AERROR << "The lane search has already reached the
limits";
        AERROR << "Possible map error found!";
        return;
    }

    // Create a new lane_segment based on the current
lane_info_ptr.
    double curr_s =
        curr_lane_seg_s >= 0.0 ? curr_lane_seg_s :
lane_info_ptr->total_length();
    LaneSegment lane_segment;
    lane_segment.set_adc_s(curr_s);
    lane_segment.set_lane_id(lane_info_ptr->id().id());
    lane_segment.set_lane_turn_type(
        PredictionMap::LaneTurnType(lane_info_ptr->id().id()));
    lane_segment.set_total_length(lane_info_ptr-
>total_length());
    if (search_forward_direction) {
        lane_segment.set_start_s(curr_s);
        lane_segment.set_end_s(std::fmin(curr_s + length_ -
accumulated_s,
                                         lane_info_ptr-
>total_length()));
    } else {
        lane_segment.set_start_s(
            std::fmax(0.0, curr_s - (length_ - accumulated_s)));
        lane_segment.set_end_s(curr_s);
    }
    if (search_forward_direction) {
        lane_segments->push_back(std::move(lane_segment));
    } else {
        lane_segments->push_front(std::move(lane_segment));
    }

    // End condition: if search reached the maximum search
distance,
    // or if there is no more successor lane_segment.
    if (search_forward_direction) {
        if (lane_segment.end_s() < lane_info_ptr->total_length())

```

```

    ||

        lane_info_ptr->lane().successor_id().empty()) {
            LaneSequence* sequence = lane_graph_ptr-
>add_lane_sequence();
            for (const auto& it : *lane_segments) {
                *(sequence->add_lane_segment()) = it;
            }
            lane_segments->pop_back();
            return;
        }
    } else {
        if (lane_segment.start_s() > 0.0 ||
            lane_info_ptr->lane().predecessor_id().empty()) {
            LaneSequence* sequence = lane_graph_ptr-
>add_lane_sequence();
            for (const auto& it : *lane_segments) {
                *(sequence->add_lane_segment()) = it;
            }
            lane_segments->pop_front();
            return;
        }
    }

    // Otherwise, continue searching for subsequent
    lane_segments.
    double new_accumulated_s = 0.0;
    double new_lane_seg_s = 0.0;
    std::vector<std::shared_ptr<const hdmap::LaneInfo>>
candidate_lanes;
    std::set<std::string> set_lane_ids;
    if (search_forward_direction) {
        new_accumulated_s = accumulated_s + lane_info_ptr-
>total_length() - curr_s;
        // Redundancy removal.
        for (const auto& successor_lane_id : lane_info_ptr-
>lane().successor_id()) {
            set_lane_ids.insert(successor_lane_id.id());
        }
        for (const auto& unique_id : set_lane_ids) {

candidate_lanes.push_back(PredictionMap::LaneById(unique_id))
;
}

```

```

// Sort the successor lane_segments from left to right.
std::sort(candidate_lanes.begin(), candidate_lanes.end(),
IsAtLeft);
// Based on other conditions, select what successor lanes
should be used.
if (!consider_lane_split) {
    candidate_lanes = {

PredictionMap::LaneWithSmallestAverageCurvature(candidate_lanes)};
}
} else {
    new_accumulated_s = accumulated_s + curr_s;
    new_lane_seg_s = -0.1;
    // Redundancy removal.
    for (const auto& predecessor_lane_id :
        lane_info_ptr->lane().predecessor_id()) {
        set_lane_ids.insert(predecessor_lane_id.id());
    }
    for (const auto& unique_id : set_lane_ids) {

candidate_lanes.push_back(PredictionMap::LaneById(unique_id))
;
}
}
bool consider_further_lane_split =
    !search_forward_direction ||
    (FLAGS_prediction_offline_mode ==
     PredictionConstants::kDumpFeatureProto) ||
    (FLAGS_prediction_offline_mode ==
     PredictionConstants::kDumpDataForLearning) ||
    (consider_lane_split && candidate_lanes.size() == 1);
// Recursively expand lane-sequence.
for (const auto& candidate_lane : candidate_lanes) {
    ConstructLaneSequence(search_forward_direction,
new_accumulated_s,
                                new_lane_seg_s, candidate_lane,
                                graph_search_horizon - 1,
consider_further_lane_split,
                                lane_segments, lane_graph_ptr);
}
if (search_forward_direction) {
    lane_segments->pop_back();
}

```

```

    } else {
        lane_segments->pop_front();
    }
}

```

上述道路图的作用是什么？？？如何构造道路图？？？

15.6. 场景(scenario)

根据本车的位置，和高精度地图，解析当前车辆所在的场景。

15.6.1. 场景管理器(ScenarioManager)

场景管理器只有2个函数，一个是Run，一个是输出当前场景scenario。下面我们分析如何获取当前场景：

```

void ScenarioManager::Run() {
    // 获取环境特征
    auto environment_features =
FeatureExtractor::ExtractEnvironmentFeatures();
    // 通过环境特征分析场景
    auto ptr_scenario_features =
ScenarioAnalyzer::Analyze(environment_features);
    // 给当前场景赋值
    current_scenario_ = ptr_scenario_features->scenario();

    // TODO(all) other functionalities including lane, junction
filters
}

```

15.7. 评估者(evaluator)

“Evaluator”类为基类，其它类继承至该类，而”EvaluatorManager”类做为管理类，负责管理三种评估者，分别为：自行车，行人，汽车。

15.7.1. 评估者基类(Evaluator)

在”Evaluator”基类中申明方法”Evaluate”，其它子类重新构造实现了上述方法。另外基类中还实现了2个辅助函数：

1. 提供世界坐标到物体坐标转换

```
std::pair<double, double> WorldCoordToObjCoord(
    std::pair<double, double> input_world_coord,
    std::pair<double, double> obj_world_coord, double
obj_world_angle)
```

2. 提供向量转矩阵？？？

```
Eigen::MatrixXf VectorToMatrixXf(const std::vector<double>&
nums,
                                    const int start_index,
                                    const int end_index)

Eigen::MatrixXf VectorToMatrixXf(const std::vector<double>&
nums,
                                    const int start_index,
                                    const int end_index,
                                    const int output_num_row,
                                    const int output_num_col)
```

15.7.2. 评估者管理器(EvaluatorManager)

评估者管理器主要是对评估器进行管理和注册。

```
class EvaluatorManager {
public:

    virtual ~EvaluatorManager() = default;

    void Init(const PredictionConf& config);

    Evaluator* GetEvaluator(const ObstacleConf::EvaluatorType&
type);

    void Run(ObstaclesContainer* obstacles_container);

    void EvaluateObstacle(Obstacle* obstacle,
```

```

        ObstaclesContainer*
obstacles_container,
                           std::vector<Obstacle*> dynamic_env);

    void EvaluateObstacle(Obstacle* obstacle,
                           ObstaclesContainer*
obstacles_container);

private:
    void BuildObstacleIdHistoryMap(ObstaclesContainer*
obstacles_container);

    void DumpCurrentFrameEnv(ObstaclesContainer*
obstacles_container);

    void RegisterEvaluator(const ObstacleConf::EvaluatorType&
type);

    std::unique_ptr<Evaluator> CreateEvaluator(
        const ObstacleConf::EvaluatorType& type);

    void RegisterEvaluators();

private:
    std::map<ObstacleConf::EvaluatorType,
std::unique_ptr<Evaluator>> evaluators_;
    ...

    std::unordered_map<int, ObstacleHistory>
obstacle_id_history_map_;

    DECLARE_SINGLETON(EvaluatorManager)
};

```

下面我们分别查看3种不同的评估者： 自行车评估者， 行人评估者和车辆评估者。

15.7.3. 自行车评估者(CyclistKeepLaneEvaluator)

自行车评估者主要是评估自行车保持当前车道的概率。评估主要是函数”Evaluate”中进行的， 函数的参数”**obstacles_container**”没有使用，

使用的只是”**obstacle_ptr**”障碍物指针。评估的过程主要是根据生成的”LaneGraph”计算自行车在lane中出现的概率。

```
bool CyclistKeepLaneEvaluator::Evaluate(
    Obstacle* obstacle_ptr, ObstaclesContainer*
obstacles_container) {
    // 设置评估者类型
    obstacle_ptr->SetEvaluatorType(evaluator_type_);
    // 特征是否初始化，该方法为protobuf自带方法，确认required字段是否都已
    经赋值，看起来这里没必要判断？？
    // https://github.com/protocolbuffers/protobuf/issues/2900
    int id = obstacle_ptr->id();
    if (!obstacle_ptr->latest_feature().IsInitialized()) {
        return false;
    }

    Feature* latest_feature_ptr = obstacle_ptr-
>mutable_latest_feature();
    // 检查特征是否包含lane graph
    if (!latest_feature_ptr->has_lane() ||
        !latest_feature_ptr->lane().has_lane_graph() ||
        !latest_feature_ptr->lane().has_lane_feature()) {
        return false;
    }

    LaneGraph* lane_graph_ptr =
        latest_feature_ptr->mutable_lane()-
>mutable_lane_graph();
    // 检查lane graph的道路序列是否为空
    if (lane_graph_ptr->lane_sequence().empty()) {
        return false;
    }

    std::string curr_lane_id =
        latest_feature_ptr->lane().lane_feature().lane_id();
    // 计算每个lane序列的概率
    for (auto& lane_sequence : *lane_graph_ptr-
>mutable_lane_sequence()) {
        const double probability =
ComputeProbability(curr_lane_id, lane_sequence);
        lane_sequence.set_probability(probability);
    }
}
```

```

    return true;
}

```

下面我们看下概率计算是如何进行的？计算的过程也很简单，就是判断lane序列的第一条lane是否和当前lane相同，相同则返回1，不相同则返回0。

```

double CyclistKeepLaneEvaluator::ComputeProbability(
    const std::string& curr_lane_id, const LaneSequence&
lane_sequence) {
    if (lane_sequence.lane_segment().empty()) {
        AWARNING << "Empty lane sequence.";
        return 0.0;
    }
    // 如果序列的第一条车道和当前车道id一致，那么则返回1，否则返回0
    std::string lane_seq_first_id =
lane_sequence.lane_segment(0).lane_id();
    if (curr_lane_id == lane_seq_first_id) {
        return 1.0;
    }
    return 0.0;
}

```

15.7.4. 行人互动评估者(PedestrianInteractionEvaluator)

行人互动评估主要是估计行人的概率，采用了深度学习的LSTM模型，关于行人预测可以参考论文”Social LSTM:Human Trajectory Prediction in Crowded Spaces”，这里我们对代码进行分析。

- 首先初始化加载深度学习模型。

```

void PedestrianInteractionEvaluator::LoadModel() {
    // 设置线程数为1
    torch::set_num_threads(1);
    if (FLAGS_use_cuda && torch::cuda::is_available()) {
        ADEBUG << "CUDA is available";
        // 设置CUDA设备
        device_ = torch::Device(torch::kCUDA);
    }
    // 加载训练好的LTSM模型文件
    torch_position_embedding_ = torch::jit::load(

```

```

FLAGS_torch_pedestrian_interaction_position_embedding_file,
device_);
torch_social_embedding_ = torch::jit::load(
FLAGS_torch_pedestrian_interaction_social_embedding_file,
device_);
torch_single_lstm_ = torch::jit::load(
    FLAGS_torch_pedestrian_interaction_single_lstm_file,
device_);
torch_prediction_layer_ = torch::jit::load(
FLAGS_torch_pedestrian_interaction_prediction_layer_file,
device_);
}

```

2. 评估行人行为，函数比较长，输入为障碍物，输出为行人轨迹”latest_feature_ptr->add_predicted_trajectory”，关于Feature的描述在”feature.proto”中。其中函数参数”obstacles_container”没有使用（原因为继承至基类的方法，重写的时候保存了参数），下面我们开始分析具体的代码：

```

bool PedestrianInteractionEvaluator::Evaluate(
    Obstacle* obstacle_ptr, ObstaclesContainer*
obstacles_container) {
    // Sanity checks.
    CHECK_NOTNULL(obstacle_ptr);

    obstacle_ptr->SetEvaluatorType(evaluator_type_);

    int id = obstacle_ptr->id();
    if (!obstacle_ptr->latest_feature().IsInitialized()) {
        AERROR << "Obstacle [" << id << "] has no latest
feature.";
        return false;
    }
    Feature* latest_feature_ptr = obstacle_ptr-
>mutable_latest_feature();
    CHECK_NOTNULL(latest_feature_ptr);

    // Extract features, and:
    // - if in offline mode, save it locally for training.
}

```

```

// - if in online mode, pass it through trained model to
evaluate.
    std::vector<double> feature_values;
    // 这里只是把最新的行人时间戳和位置提取出来了，并且线性存储在数组中(把数
据展开)，
    // 也就是说数组的第一个元素是时间戳，第二为ID，第三和第四为位置。
    ExtractFeatures(obstacle_ptr, &feature_values);
    if (FLAGS_prediction_offline_mode ==
        PredictionConstants::kDumpDataForLearning) {
        FeatureOutput::InsertDataForLearning(*latest_feature_ptr,
        feature_values,
                                "pedestrian",
        nullptr);
        ADEBUG << "Saving extracted features for learning
locally.";
        return true;
    }

    static constexpr double kShortTermPredictionTimeResolution
= 0.4;
    static constexpr int kShortTermPredictionPointNum = 5;
    static constexpr int kHiddenStateUpdateCycle = 4;

    // Step 1 Get social embedding
    torch::Tensor social_pooling = GetSocialPooling();
    std::vector<torch::jit::IValue> social_embedding_inputs;

    social_embedding_inputs.push_back(std::move(social_pooling.to
(device_)));
    torch::Tensor social_embedding =
    torch_social_embedding_.forward(social_embedding_inputs)
        .toTensor()
        .to(torch::kCPU);

    // Step 2 Get position embedding
    double pos_x = feature_values[2];
    double pos_y = feature_values[3];
    double rel_x = 0.0;
    double rel_y = 0.0;
    if (obstacle_ptr->history_size() > kHiddenStateUpdateCycle
- 1) {
        rel_x = obstacle_ptr->latest_feature().position().x() -

```

```

        obstacle_ptr->feature(3).position().x();
    rel_y = obstacle_ptr->latest_feature().position().y() -
        obstacle_ptr->feature(3).position().y();
}

torch::Tensor torch_position = torch::zeros({1, 2});
torch_position[0][0] = rel_x;
torch_position[0][1] = rel_y;
std::vector<torch::jit::IValue> position_embedding_inputs;

position_embedding_inputs.push_back(std::move(torch_position.
to(device_)));
torch::Tensor position_embedding =
    torch_position_embedding_.forward(position_embedding_inputs)
        .toTensor()
        .to(torch::kCPU);

// Step 3 Conduct single LSTM and update hidden states
torch::Tensor lstm_input =
    torch::zeros({1, 2 * (kEmbeddingSize + kHiddenSize)});
for (int i = 0; i < kEmbeddingSize; ++i) {
    lstm_input[0][i] = position_embedding[0][i];
}

if (obstacle_id_lstm_state_map_.find(id) ==
    obstacle_id_lstm_state_map_.end()) {
    obstacle_id_lstm_state_map_[id].ht = torch::zeros({1, 1,
    kHiddenSize});
    obstacle_id_lstm_state_map_[id].ct = torch::zeros({1, 1,
    kHiddenSize});
    obstacle_id_lstm_state_map_[id].timestamp = obstacle_ptr-
>timestamp();
    obstacle_id_lstm_state_map_[id].frame_count = 0;
}
torch::Tensor curr_ht = obstacle_id_lstm_state_map_[id].ht;
torch::Tensor curr_ct = obstacle_id_lstm_state_map_[id].ct;
int curr_frame_count =
    obstacle_id_lstm_state_map_[id].frame_count;

if (curr_frame_count == kHiddenStateUpdateCycle - 1) {
    for (int i = 0; i < kHiddenSize; ++i) {
        lstm_input[0][kEmbeddingSize + i] = curr_ht[0][0][i];
    }
}

```

```

        lstm_input[0][kEmbeddingSize + kHiddenSize + i] =
curr_ct[0][0][i];
    }

    std::vector<torch::jit::IValue> lstm_inputs;
    lstm_inputs.push_back(std::move(lstm_input.to(device_)));
    auto lstm_out_tuple =
torch_single_lstm_.forward(lstm_inputs).toTuple();
    auto ht = lstm_out_tuple->elements()[0].toTensor();
    auto ct = lstm_out_tuple->elements()[1].toTensor();
    obstacle_id_lstm_state_map_[id].ht = ht.clone();
    obstacle_id_lstm_state_map_[id].ct = ct.clone();
}
obstacle_id_lstm_state_map_[id].frame_count =
    (curr_frame_count + 1) % kHiddenStateUpdateCycle;

// Step 4 for-loop get a trajectory
// Set the starting trajectory point
Trajectory* trajectory = latest_feature_ptr-
>add_predicted_trajectory();
trajectory->set_probability(1.0);
TrajectoryPoint* start_point = trajectory-
>add_trajectory_point();
start_point->mutable_path_point()->set_x(pos_x);
start_point->mutable_path_point()->set_y(pos_y);
start_point->mutable_path_point()-
>set_theta(latest_feature_ptr->theta());
start_point->set_v(latest_feature_ptr->speed());
start_point->set_relative_time(0.0);

for (int i = 1; i <= kShortTermPredictionPointNum; ++i) {
    double prev_x = trajectory->trajectory_point(i -
1).path_point().x();
    double prev_y = trajectory->trajectory_point(i -
1).path_point().y();
    CHECK(obstacle_id_lstm_state_map_.find(id) !=
        obstacle_id_lstm_state_map_.end());
    torch::Tensor torch_position = torch::zeros({1, 2});
    double curr_rel_x = rel_x;
    double curr_rel_y = rel_y;
    if (i > 1) {
        curr_rel_x =
            prev_x - trajectory->trajectory_point(i -

```

```

2).path_point().x();
    curr_rel_y =
        prev_y - trajectory->trajectory_point(i -
2).path_point().y();
    }
    torch_position[0][0] = curr_rel_x;
    torch_position[0][1] = curr_rel_y;
    std::vector<torch::jit::IValue>
position_embedding_inputs;

position_embedding_inputs.push_back(std::move(torch_position.
to(device_)));
    torch::Tensor position_embedding =

torch_position_embedding_.forward(position_embedding_inputs)
    .toTensor()
    .to(torch::kCPU);
    torch::Tensor lstm_input =
        torch::zeros({1, kEmbeddingSize + 2 * kHiddenSize});
for (int i = 0; i < kEmbeddingSize; ++i) {
    lstm_input[0][i] = position_embedding[0][i];
}

auto ht = obstacle_id_lstm_state_map_[id].ht.clone();
auto ct = obstacle_id_lstm_state_map_[id].ct.clone();

for (int i = 0; i < kHiddenSize; ++i) {
    lstm_input[0][kEmbeddingSize + i] = ht[0][0][i];
    lstm_input[0][kEmbeddingSize + kHiddenSize + i] = ct[0]
[0][i];
}
    std::vector<torch::jit::IValue> lstm_inputs;
    lstm_inputs.push_back(std::move(lstm_input.to(device_)));
    auto lstm_out_tuple =
torch_single_lstm_.forward(lstm_inputs).tuple();
    ht = lstm_out_tuple->elements()[0].toTensor();
    ct = lstm_out_tuple->elements()[1].toTensor();
    std::vector<torch::jit::IValue> prediction_inputs;
    prediction_inputs.push_back(ht[0]);
    auto pred_out_tensor =
torch_prediction_layer_.forward(prediction_inputs)
        .toTensor()
        .to(torch::kCPU);

```

```

        auto pred_out = pred_out_tensor.accessor<float, 2>();
        TrajectoryPoint* point = trajectory-
>add_trajectory_point();
        double curr_x = prev_x + static_cast<double>(pred_out[0]
[0]);
        double curr_y = prev_y + static_cast<double>(pred_out[0]
[1]);
        point->mutable_path_point()->set_x(curr_x);
        point->mutable_path_point()->set_y(curr_y);
        point->set_v(latest_feature_ptr->speed());
        point->mutable_path_point()->set_theta(
            latest_feature_ptr->velocity_heading());
        point-
>set_relative_time(kShortTermPredictionTimeResolution *
static_cast<double>(i));
    }

    return true;
}

```

车辆评估者涉及的评估者模型比较多，下面我们逐个介绍。

15.7.5. CostEvaluator

评估车辆的横向偏移概率，通过计算当前车的宽度和车道的横向差值，然后通过”Sigmoid”函数映射到0-1的概率空间。

15.7.6. CruiseMLPEvaluator

MLP为”多层神经网络”，相比上述过程，新增加了lane相关的特征。

15.7.7. JunctionMapEvaluator

加入了语义地图，模型未知？？？

15.7.8. JunctionMLPEvaluator

路口多层神经网络，没有利用地图

15.7.9. LaneAggregatingEvaluator

道路合并评估器

15.7.10. LaneScanningEvaluator

道路扫描？？？

15.7.11. MLPEvaluator

多层神经网络评估器？？？

15.7.12. SemanticLSTMxEvaluator

语义LSTM评估器

评估器主要是用深度学习的方法进行预测对应的概率，而且依赖事先建好的图，所以弄清楚上述2个过程很关键。

15.8. 预测器(predictor)

“Predictor”类为基类，其它类继承至该类，而”PredictorManager”类作为管理类。最后通过预测器预测障碍物的轨迹。

15.8.1. 预测器基类(Predictor)

预测者基类主要申明了”Predict”方法，在子预测器中重构。输入是评估器给出的概率(trajectory)，输出则是预测的轨迹。

15.8.2. 预测管理器(PredictorManager)

预测管理器主要是对预测器进行创建，管理和注册。主要的实现 在”PredictObstacle”中：

下面我们分别介绍几种预测器。

15.8.3. EmptyPredictor

对静止的物体进行预测，没有任何轨迹输出。

15.8.4. ExtrapolationPredictor

外推法预测器，

15.8.5. FreeMovePredictor

自由移动？？？

15.8.6. InteractionPredictor

15.8.7. JunctionPredictor

15.8.8. LaneSequencePredictor

生成曲线在”DrawLaneSequenceTrajectoryPoints”中实现。

15.8.9. MoveSequencePredictor

15.8.10. SequencePredictor

15.8.11. SingleLanePredictor

上述所有评估器的过程都类似，都是找到lane之后对lane做一个平滑的曲线？？？最后都调用了”PredictionMap::SmoothPointFromLane”。

15.9. Reference

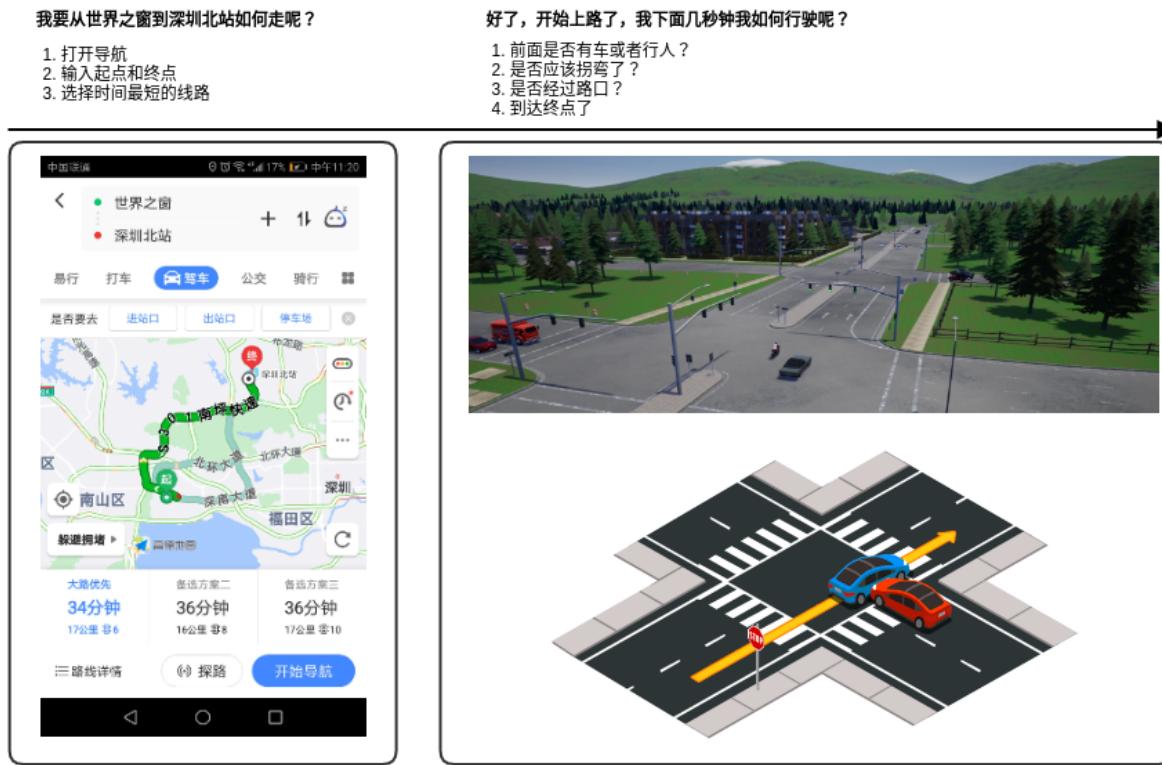
- [Apollo 5.0 障碍物行为预测技术](#)
[<https://www.cnblogs.com/liuzubing/p/11388485.html>]
- [Apollo自动驾驶入门课程第⑥讲—预测](#)
[<https://cloud.tencent.com/developer/news/310036>]

16. Routing

青，取之于蓝而青于蓝；冰，水为之而寒于水。

16.1. Routing模块简介

Routing类似于现在开车时用到的导航模块，通常考虑的是起点到终点的最优路径（通常是最短路径），Routing考虑的是起点到终点的最短路径，而Planning则是行驶过程中，当前一小段时间如何行驶，需要考虑当前路况，是否有障碍物。Routing模块则不需要考虑这些信息，只需要做一个长期的规划路径即可，过程如下：



这也和我们开车类似，上车之后，首先搜索目的地，打开导航（Routing做的事情），而开始驾车之后，则会根据当前路况，行人车辆信息来适当调整直到到达目的地（Planning做的事情）。

- **Routing** - 主要关注起点到终点的长期路径，根据起点到终点之间的道路，选择一条最优路径。
- **Planning** - 主要关注几秒钟之内汽车的行驶路径，根据当前行驶过程中的交通规则，车辆行人等信息，规划一条短期路径。

16.2. 基础知识

16.2.1. Demo

[演示地址](https://daohu527.github.io/) [https://daohu527.github.io/] 我们通过”OSM Pathfinding”作为例子，来详细讲解整个过程，感谢@mplewis。首先我们通过如下的视频演示看下Routing寻找路径的过程，查找的是深圳南山区的地图：

1. 首先选择查找算法，有: A*, Breadth First Search, Greedy Best First Search, Uniform Cost Search, Depth First Search。
2. 选择起点
3. 选择终点
4. 选择开始，开始寻找路径

上面的项目是基于OSM(openstreetmap)获取的地图数据，如果需要自己制作地图，首先在OSM的[官网](https://www.openstreetmap.org/) [https://www.openstreetmap.org/]导出地图，导出的文件格式为“map.osm”，可以通过浏览器打开查看，然后在项目的tools目录，把OSM地图转成项目用到的(Graph)图。制作demo的过程如下：

1. 获取地图信息 - 由于OSM的地图都是开源的，所以我们只需要找到对应的区域，并且选择导出，就可以导出地图的原始数据。地图的数据格式为OSM格式。
2. 构建图 - 根据上述的信息，构建有向图，下载的格式对渲染比较友好，但是对查找最短路径不友好，因此要转换成有向图的格式(apollo的routing模块也是经过了如下的转换)。
3. 查找最短路径 - 根据上述的信息，查找一条最短路径。

如果图的规模太大，以1000个举例，只算两个点之间互相有连接的情况， 1000×1000 就是100万个点，如果点的规模更大，那么就需要采用redis数据库来提高查找效率了。

下面我们先介绍上面的例子是如何工作的。

16.2.2. 地图

首先我们以openstreetmap为例来介绍下地图是如何组成的。[开放街道地图](https://www.openstreetmap.org/) [https://www.openstreetmap.org/]（英语：OpenStreetMap，缩写为OSM）是一个建构自由内容之网上地图协作计划，目标是创造一个内容自由且能让所有人编辑的世界地图，并且让一般的移动设备有方便的导航方案。因为这个地图是一个开源地图，所以可以灵活和自由的获取地图资源。接着看下openstreetmap的基本元素：**Node**  节点表示由其纬度和经度定义的地球表面上的特定点。每个节点至少包括id号和一对坐标。节点也可用于定义独立点功能。例如，节点可以代表公园长椅或水井。节点也可以定义道路(Way)的形状，节点是一切形状的基础。

```
<node id="25496583" lat="51.5173639" lon="-0.140043"  
version="1" changeset="203496" user="80n" uid="1238"  
visible="true" timestamp="2007-01-28T11:40:26Z">  
    <tag k="highway" v="traffic_signals"/>  
</node>
```

Way  道路是包含2到2,000个有序节点的折线组成，用于表示线性特征，例如河流和道路。道路也可以表示区域（实心多边形）的边界，例如建筑物或森林。在这种情况下，道路的第一个和最后一个节点将是相同的。这被称为“封闭的方式”。

```
<way id="5090250" visible="true" timestamp="2009-01-  
19T19:07:25Z" version="8" changeset="816806" user="Blumpsy"  
uid="64226">  
    <nd ref="822403"/>  
    <nd ref="21533912"/>  
    <nd ref="821601"/>  
    <nd ref="21533910"/>  
    <nd ref="135791608"/>  
    <nd ref="333725784"/>  
    <nd ref="333725781"/>  
    <nd ref="333725774"/>  
    <nd ref="333725776"/>  
    <nd ref="823771"/>
```

```
<tag k="highway" v="residential"/>
<tag k="name" v="Clipstone Street"/>
<tag k="oneway" v="yes"/>
</way>
```

Relation  关系是记录两个或更多个数据元素（节点，方式和/或其他关系）之间的关系的多用途数据结构。例子包括：

- 路线关系，列出形成主要（编号）高速公路，自行车路线或公交路线的方式。
- 转弯限制，表示你无法从一种方式转向另一种方式。
- 描述具有孔的区域（其边界是“外部方式”）的多面体（“内部方式”）。

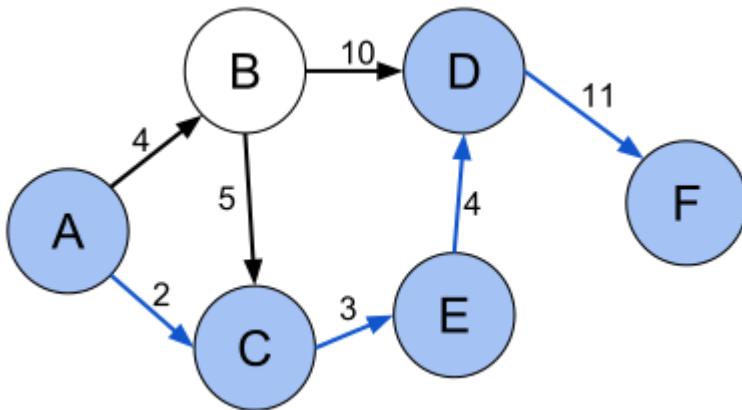
Tag  所有类型的数据元素（节点，方式和关系）以及变更集都可以包含标签。标签描述了它们所附着的特定元素的含义。标签由两个自由格式文本字段组成; ‘Key’ 和 ‘Value’。例如，“高速公路”=“住宅”定义了一条道路。元素不能有2个带有相同“key”的标签，“key”必须是唯一的。例如，您不能将元素标记为amenity = restaurant和amenity = bar。

我们看到的地图，实际上是由一些Node和Way组成，需要展示地图时候，通过读取地图中的Node和Way的数据实时画(渲染)出来，例如2个Node组成了一条道路，那么就在这两点之间画一条直线，并且标记为道路，如果是封闭区域，并且根据数据，画出一个多边形，并把它标记为湖泊或者公园。有很多地图渲染引擎，以下是openstreet推荐的地图引擎：

1. Osmarender: 一个基于可扩展样式表语言转换 (XSLT) 的渲染器,能够创建可缩放矢量图形(SVG), SVG可以用浏览器观看或转换成位图。
2. Mapnik: 一个用C++写的非常快的渲染器,可以生成位图(png, jpeg)和矢量图形(pdf, svg, postscript)。

16.2.3. 最短距离

我们先看一下经典的例子：最短路径。在图论中，最短路径问题是在图中的两个顶点之间找到路径，使得其边的权重之和最小化的问题。而在地图上找到两个点之间最短路径的问题可以被建模为图中最短路径问题的特殊情况，其中顶点对应于交叉点并且边缘对应于路段，每个路段对应于路段的长度。



最短路径算法：

- Dijkstra算法
- A*算法
- Bellman-Ford算法
- SPFA算法（Bellman-Ford算法的改进版本）
- Floyd-Warshall算法
- Johnson算法
- Bi-Direction BFS算法

在地图上查找两个点之间的最短距离，我们就可以把道路的长度当做边，路口当做节点，通过把道路抽象为一个有向图，然后通过上述算法，查找到当前2点之间的最短路径，并且输出，这就是每次我们查找起点和终点的过程。所以要查找起点到终点之间的路径，需要经过以下几个步骤：

1. 获取地图的原始数据，节点和道路信息。
2. 通过上述信息，构建有向图。
3. 采用最短路径算法，找到2点之间的最短距离。

实际上，真实场景的地图导航，查找2点之间的路径，可能不是实时计算出来的，假设有100个人在查询“北京机场”到“天安门”的路

线，第一个人的路线可能是实时计算得到的，而其他99个人都是用的缓存的数据，在第一个人查到之后，后面的99个人就不需要重复计算了。对于频繁查找的路线，也可以在晚上统一计算，然后保存起来，可以利用存储换速度，除非道路有变化（新修路或者道路维修，桥断了），然后再重新计算。多样化的需求，比如可以选择高速优先还是不走高速，地铁优先，少换乘等，这些都需要构建层次和结构化的信息。根据用户的反馈实时的更新路况，比如路上有10个人在用地图导航，发现在某一段大家都开的很慢，或者有用户反馈堵车，就更新当前路况。所以地图是一个强者越强的市场，用户越多，数据就更新的就越快，地图就越准确，用的人就越多；用的人越少，数据更新的就越慢，假设一条路上只有一个用户，一个人开的慢，并不能反馈当前道路拥堵，用该地图的其它用户过去之后，发现堵车，就导致用户体验很差，下次就不会再用这个地图了。所以地图必须需要一定的用户量才能活下去，而且强者越强。

17. Routing模块分析

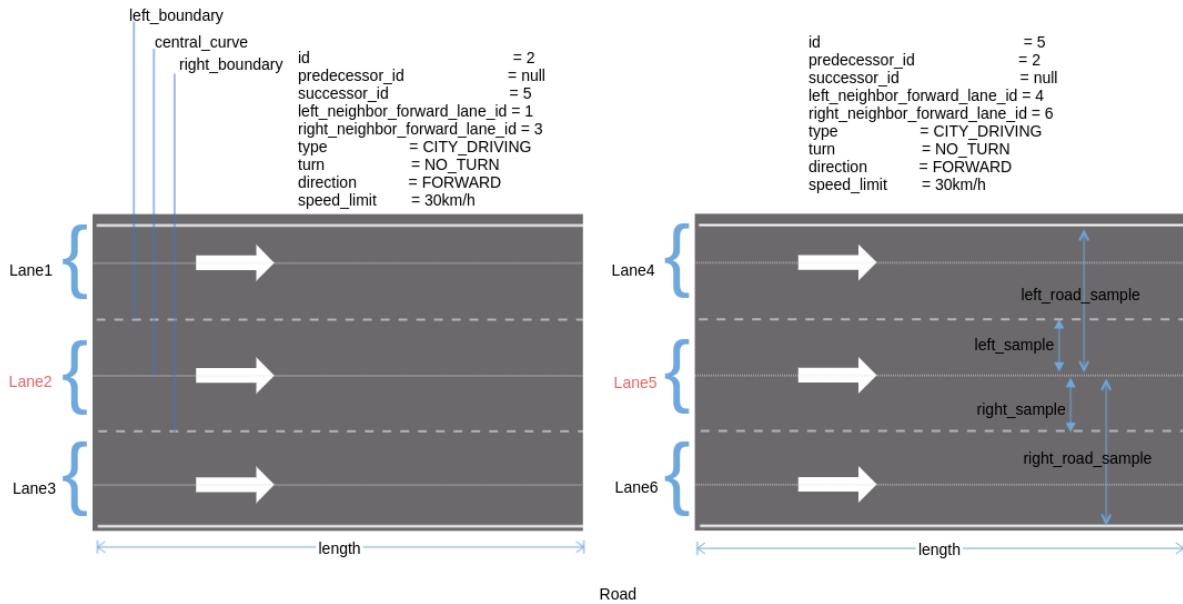
分析Routing模块之前，我们只需要能够解决以下几个问题，就算是把routing模块掌握清楚了。

1. 如何从A点到B点
2. 如何规避某些点 - 查找的时候发现是黑名单里的节点，则选择跳过。
3. 如何途径某些点 - 采用分段的形式，逐段导航（改进版的算法是不给定点的顺序，自动规划最优的线路）。
4. 如何设置固定线路，而且不会变？最后routing输出的结果是什么？固定成文件的形式。

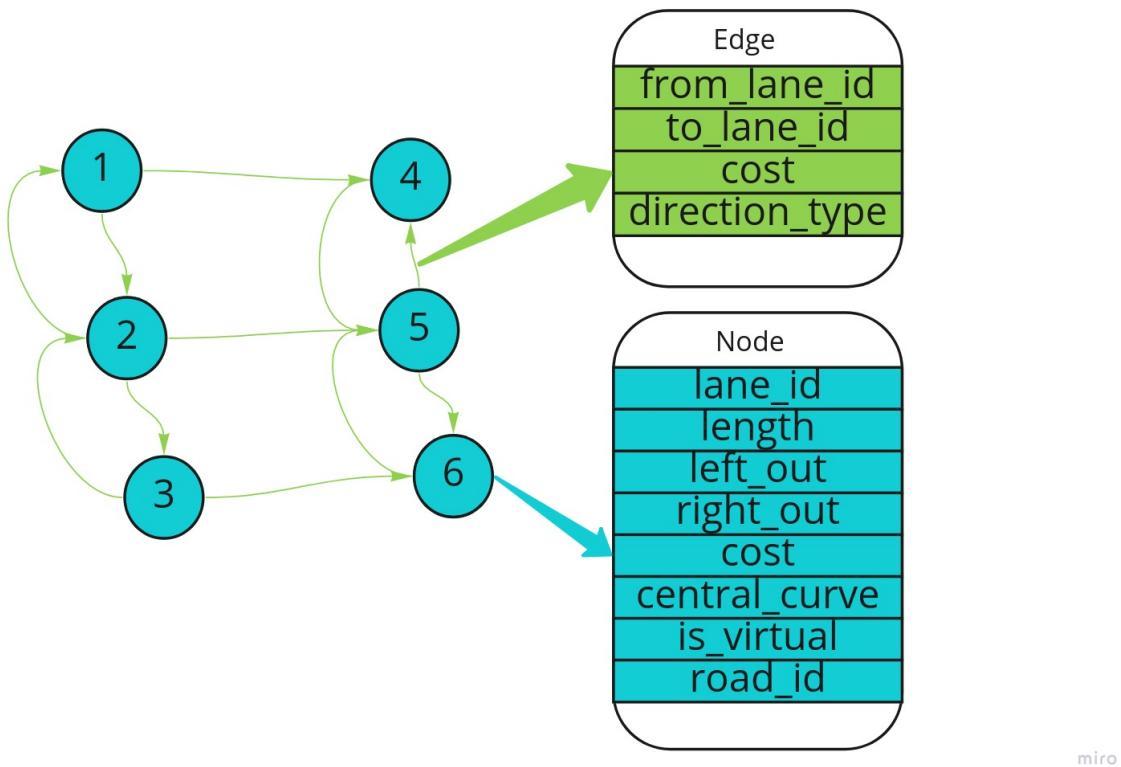
17.1. 创建Routing地图

通过上面的介绍可以知道，routing需要的是一个拓扑结构的图，要想做routing，第一步就是要把原始的地图转换成包含拓扑结构的图，apollo中也实现了类似的操作，把base_map转换为routing_map，这里的

`base_map`就是高精度地图，而`routing_map`则是导航地图，`routing_map`的结构为一个有向图。对应的例子在”modules/map/data/demo”中，这个例子比较简陋，因为`routing_map.txt`中只包含一个节点(Node)，没有边(Edge)信息。apollo建图的实现在”routing/topo_creator”中，首先apollo的拓扑图中的节点和上面介绍的传统的节点不一样，我们前面的例子中，节点就是路的起点和终点，边就是路，而自动驾驶中的道路是车道线级别的，原来的这种定义点和边的方式就不适用了（定义不了车道），所以apollo中引用的新的概念，apollo中的点就是一条车道，而边则是车道和车道之间的连接，点对应具体的车道，而边则是一个虚拟的概念，表示车道之间的关系。下面我们可以先看下apollo中道路(road)和车道(lane)的概念。



可以看到一条道路(road)，包含多个车道(lane)，图中一条道路分成了2段，每一段包含3条车道(lane)，车道的信息见图中，主要标识了车道唯一id，左边界，右边界，参考线，长度，前车道，后车道，左边相邻的车道，右边相邻的车道等，通过这些结构化的信息，我们就知道车道之间的相互关系，也就知道了我们能否到达下一个车道，从而规划出一条到达目的地的车道线级别的路线，Planning模块在根据规划好的线路进行行驶，因为已经到车道线级别了，所以相对规划起来就简单很多。最后我们会建立一张如下的图，其中节点是一个个的lane，而边则代表lane之间的连接。



miro

其中节点和边的结构在protobuf中定义，在文件”modules/routing/proto/topo_graph.proto”中，其中：

- **NODE** - 包括车道唯一id，长度，左边出口，右边出口（这里的出口对应车道虚线的部分，或者自己定义的一段允许变道的路段），路段代价（限速或者拐弯的路段会增加成本，代价系数在 `routing_config.pb.txt` 中定义），中心线（虚拟的，用于生成参考线），是否可见，车道所属的道路id。
- **EDGE** - 则包括起始车道id，到达车道id，切换代价，方向（向前，向左，向右）。我们以上图中的例子来说明：

```
// 以lane2举例子
id = 2
predecessor_id = null // 上一车道id, 不考虑变道
的情况
successor_id = 5 // 下一车道id, 不考虑变道
的情况
left_neighbor_forward_lane_id = 1 // 左边邻居车道
right_neighbor_forward_lane_id = 3 // 右边邻居车道
type = CITY_DRIVING
```

```

turn                                = NO_TURN    // 没有拐弯，有些车道
本身是曲线，如路口的左拐弯，右拐弯车道
direction                           = FORWARD   // 前向，反向，或者双
向
speed_limit                         = 30        // 限速30km/h

// 以lane5举例子
id                                 = 5
predecessor_id                     = 2 // 上一车道id，不考虑变道的情
况
successor_id                       = null      // 下一车道id，不考虑
变道的情况
left_neighbor_forward_lane_id     = 4         // 左边邻居车道
right_neighbor_forward_lane_id   = 6         // 右边邻居车道
type                               = CITY_DRIVING
turn                                = NO_TURN    // 没有拐弯，有些车道
本身是曲线，如路口的左拐弯，右拐弯车道
direction                           = FORWARD   // 前向，反向，或者双
向
speed_limit                         = 30        // 限速30km/h

```

17.1.1. 建图流程

可以看到对比map结构中的lane，graph中的节点和边省去了很多信息，主要关注的是lane之间的关系。在理解了上述数据结构之后，理解建图的过程就轻松多了，下面我们结合代码来分析具体的建图流程。建图的代码目录为”routing/topo_creator”，其文件结构如下：

```

.
├── BUILD
├── edge_creator.cc           // 建边
├── edge_creator.h
├── graph_creator.cc          // 建图
├── graph_creator.h
└── graph_creator_test.cc
├── node_creator.cc           // 建节点
├── node_creator.h
└── topo_creator.cc           // main函数

```

编译生成可执行文件”topo_creator”，地图需要事先通过”topo_creator”把base_map转换为routing_map。其中建图的主流程

在”graph_creator.cc”，并且创建节点和边。建图的主流程在函数”GraphCreator::Create()”中，下面我们具体分析这个函数。

```
bool GraphCreator::Create() {
    // 这里注意，有2种格式，一种是openstreet格式，通过OpendriveAdapter来读取
    // 另外一种是apollo自己定义的格式。
    if (common::util::EndWith(base_map_file_path_, ".xml")) {
        if (!hdmap::adapter::OpendriveAdapter::LoadData(base_map_file_path_,
            &pbmap_))
    {
        ERROR << "Failed to load base map file from " <<
        base_map_file_path_;
        return false;
    }
    } else {
        if (!common::util::GetProtoFromFile(base_map_file_path_,
            &pbmap_)) {
            ERROR << "Failed to load base map file from " <<
            base_map_file_path_;
            return false;
        }
    }

    // graph_为最后保存的图，消息格式在topo_graph.proto中申明
    graph_.set_hdmap_version(pbmap_.header().version());
    graph_.set_hdmap_district(pbmap_.header().district());

    // 从base_map中读取道路和lane对应关系，base_map的消息结构在map.proto和map_road.proto中
    for (const auto& road : pbmap_.road()) {
        for (const auto& section : road.section()) {
            for (const auto& lane_id : section.lane_id()) {
                road_id_map_[lane_id.id()] = road.id().id();
            }
        }
    }

    // 初始化禁止的车道线，从配置文件中读取最小掉头半径
    InitForbiddenLanes();
```

```

const double min_turn_radius =
VehicleConfigHelper::GetConfig().vehicle_param().min_turn_radius();

// 遍历base_map中的lane，并且创建节点。
for (const auto& lane : pbmap_.lane()) {
    const auto& lane_id = lane.id().id();
    // 跳过不是城市道路(CITY_DRIVING)的车道
    if (forbidden_lane_id_set_.find(lane_id) !=
forbidden_lane_id_set_.end()) {
        ADEBUG << "Ignored lane id: " << lane_id
            << " because its type is NOT CITY_DRIVING.";
        continue;
    }
    // 跳过掉头曲率太小的车道
    if (lane.turn() == hdmap::Lane::U_TURN &&
        !IsValidUTurn(lane, min_turn_radius)) {
        ADEBUG << "The u-turn lane radius is too small for the
vehicle to turn";
        continue;
    }

    // 存储图中节点index和lane_id的关系，因为通过node可以找到lane,
    // 而通过lane_id需要遍历节点才能找到节点index。
    node_index_map_[lane_id] = graph_.node_size();

    // 如果从road_id_map_中找到lane_id，则把创建节点的时候指定道路
    id,
    // 如果没有找到那么road_id则为空。
    const auto iter = road_id_map_.find(lane_id);
    if (iter != road_id_map_.end()) {
        node_creator::GetPbNode(lane, iter->second,
routing_conf_,
                                graph_.add_node());
    } else {
        AWARN << "Failed to find road id of lane " << lane_id;
        node_creator::GetPbNode(lane, "", routing_conf_,
graph_.add_node());
    }
}

```

```

    std::string edge_id = "";
    // 遍历base_map中的lane，并且创建边。
    for (const auto& lane : pbmap_.lane()) {
        const auto& lane_id = lane.id().id();
        // 跳过不是城市道路(CITY_DRIVING)的车道
        if (forbidden_lane_id_set_.find(lane_id) !=
            forbidden_lane_id_set_.end()) {
            ADEBUG << "Ignored lane id: " << lane_id
                << " because its type is NOT CITY_DRIVING.";
            continue;
        }

        // 这里就是通过上面所说的通过lane_id找到node的index，得到节点，
        // 如果不保存，则需要遍历所有节点通过lane_id来查找节点，原因为node
        // 中有lane_id，
        // 而lane结构中没有node_id。
        const auto& from_node =
graph_.node(node_index_map_[lane_id]);

        // 添加一条该节点到下一个节点的边，注意这里没有换道，所以方向为前。
        AddEdge(from_node, lane.successor_id(), Edge::FORWARD);
        if (lane.length() < FLAGS_min_length_for_lane_change) {
            continue;
        }
        // 车道有左边界，并且允许变道
        // 添加一条该节点到左边邻居的边
        if (lane.has_left_boundary() &&
            IsAllowedToCross(lane.left_boundary())) {
            AddEdge(from_node,
lane.left_neighbor_forward_lane_id(), Edge::LEFT);
        }
        // 同上
        if (lane.has_right_boundary() &&
            IsAllowedToCross(lane.right_boundary())) {
            AddEdge(from_node,
lane.right_neighbor_forward_lane_id(), Edge::RIGHT);
        }
    }

    ...
    // 保存routing_map文件，有2种格式txt和bin
    if (!common::util::SetProtoToASCIIFile(graph_, txt_file)) {

```

```

        AERROR << "Failed to dump topo data into file " <<
txt_file;
        return false;
    }
    AINFO << "Txt file is dumped successfully. Path: " <<
txt_file;
    if (!common::util::SetProtoToBinaryFile(graph_, bin_file))
{
    AERROR << "Failed to dump topo data into file " <<
bin_file;
    return false;
}
    AINFO << "Bin file is dumped successfully. Path: " <<
bin_file;
    return true;
}

```

小结一下创建的图的流程，首先是从base_map中读取道路信息，之后遍历道路，先创建节点，然后创建节点的边，之后把图(点和边的信息)保存到routing_map中，所以routing_map中就是graph_protobuf格式的固化，后面routing模块会读取创建好的routing_map通过astar算法来进行路径规划。

17.1.2. 创建节点

接下来看下创建节点的过程，在函数”GetPbNode()”中：

```

void GetPbNode(const hdmap::Lane& lane, const std::string&
road_id,
                const RoutingConfig& routingconfig, Node*
const node) {
    // 1. 初始化节点信息
    InitNodeInfo(lane, road_id, node);
    // 2. 初始化节点代价
    InitNodeCost(lane, routingconfig, node);
}

```

1. 初始化哪些节点信息呢？

```

void InitNodeInfo(const Lane& lane, const std::string&
road_id,

```

```

        Node* const node) {
    double lane_length = GetLaneLength(lane);
    node->set_lane_id(lane.id().id());
    node->set_road_id(road_id);
    // 根据lane的边界，添加能够变道的路段
    AddOutBoundary(lane.left_boundary(), lane_length, node-
>mutable_left_out());
    AddOutBoundary(lane.right_boundary(), lane_length, node-
>mutable_right_out());
    node->set_length(lane_length);
    node->mutable_central_curve()-
>CopyFrom(lane.central_curve());
    node->set_is_virtual(true);
    if (!lane.has_junction_id() ||
        lane.left_neighbor_forward_lane_id_size() > 0 ||
        lane.right_neighbor_forward_lane_id_size() > 0) {
        node->set_is_virtual(false);
    }
}
}

```

2. 如何计算节点的代价呢？

```

void InitNodeCost(const Lane& lane, const RoutingConfig&
routing_config,
                  Node* const node) {
    double lane_length = GetLaneLength(lane);
    double speed_limit = (lane.has_speed_limit()) ?
lane.speed_limit()
:
routing_config.base_speed();
    double ratio = (speed_limit >= routing_config.base_speed())
                  ? (1 / sqrt(speed_limit /
routing_config.base_speed()))
                  : 1.0;
    // 1. 根据道路长度和速度限制来计算代价
    double cost = lane_length * ratio;
    if (lane.has_turn()) {
        // 2. 掉头代价 > 左转代价 > 右转的代价
        // left_turn_penalty: 50.0
        // right_turn_penalty: 20.0
        // uturn_penalty: 100.0
        if (lane.turn() == Lane::LEFT_TURN) {
            cost += routing_config.left_turn_penalty();
        }
    }
}

```

```

    } else if (lane.turn() == Lane::RIGHT_TURN) {
        cost += routing_config.right_turn_penalty();
    } else if (lane.turn() == Lane::U_TURN) {
        cost += routing_config.uturn_penalty();
    }
}
node->set_cost(cost);
}

```

17.1.3. 创建边

接下来分析如何创建边，创建边的流程在函数”GetPbEdge()”中

```

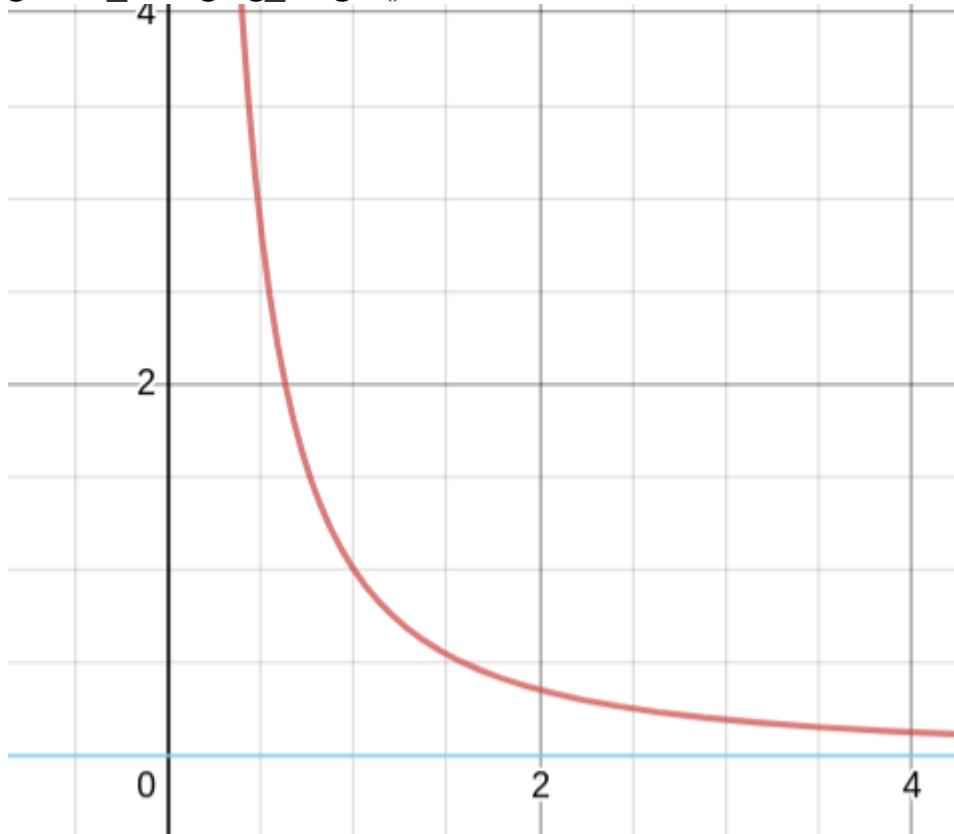
void GetPbEdge(const Node& node_from, const Node& node_to,
               const Edge::DirectionType& type,
               const RoutingConfig& routing_config, Edge*
edge) {
    // 设置起始, 终止车道和类型
    edge->set_from_lane_id(node_from.lane_id());
    edge->set_to_lane_id(node_to.lane_id());
    edge->set_direction_type(type);

    // 默认代价为0, 即直接向前开的代价
    edge->set_cost(0.0);
    if (type == Edge::LEFT || type == Edge::RIGHT) {
        const auto& target_range =
            (type == Edge::LEFT) ? node_from.left_out() :
node_from.right_out();
        double changing_area_length = 0.0;
        for (const auto& range : target_range) {
            changing_area_length += range.end().s() -
range.start().s();
        }
        double ratio = 1.0;
        // 计算代价
        if (changing_area_length <
routing_config.base_changing_length()) {
            ratio = std::pow(
                changing_area_length /
routing_config.base_changing_length(), -1.5);
        }
        edge->set_cost(routing_config.change_penalty() * ratio);
    }
}

```

```
    }  
}
```

我们可以看下edge cost的曲线，因为”changing_area_length / routing_config.base_changing_length() < 1”，这个函数最小值为1，最大

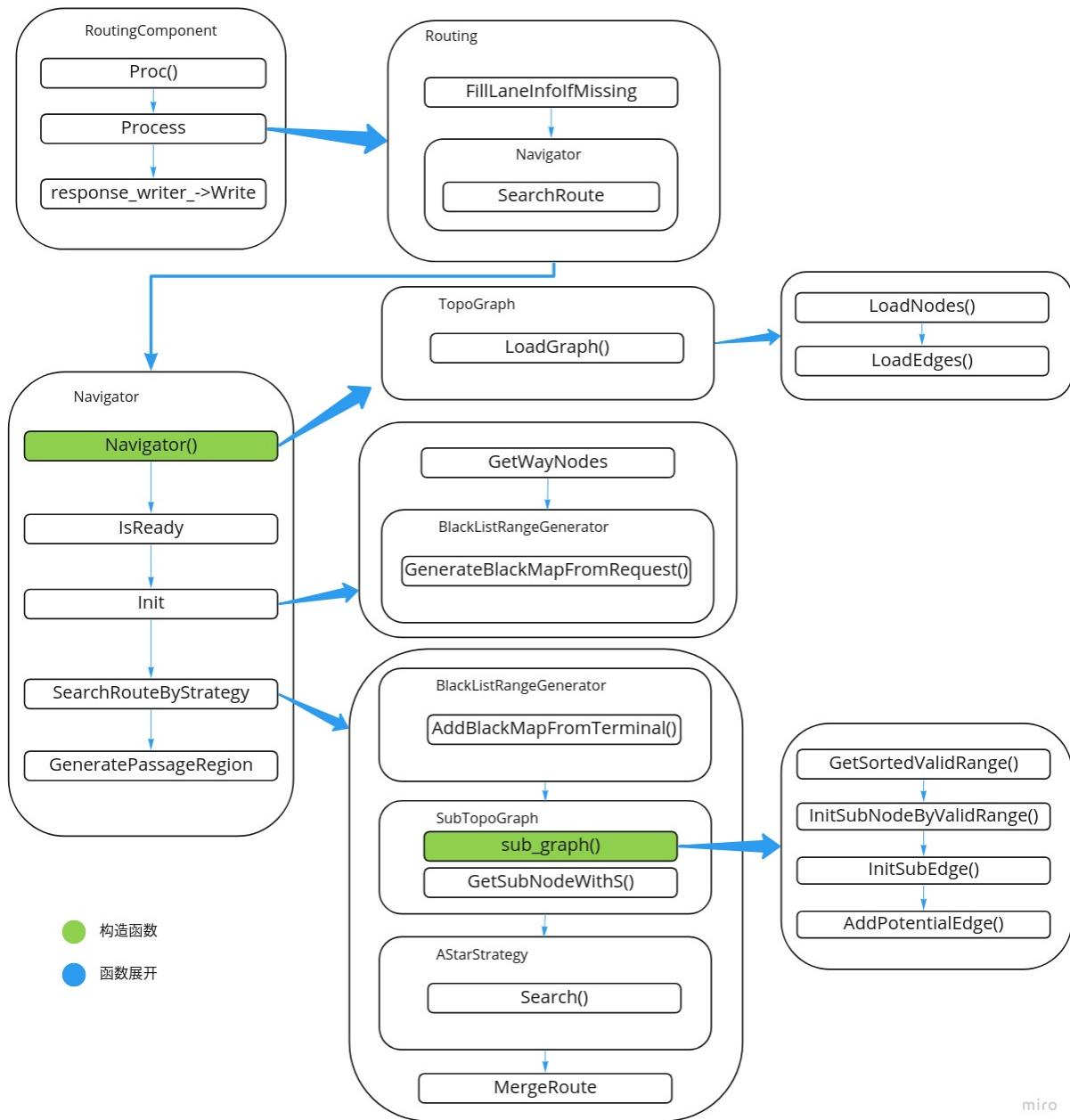


值为无穷。

到这里制作routing_map的流程就结束了，建图的主要目的是把base结构的map转换为graph结构的map，从而利用图结构来查找最佳路径，下面会分析如何通过routing_map得到规划好的路线。

17.2. Routing主流程

Routing模块的流程相对比较简单，主流流程见下图：



把一些主要的流程摘要如下：

1. 在cyber中注册component，接收request请求，响应请求结果 response
2. 读取routing_map并且建图graph
3. 获取request中的routing请求节点
4. 根据black_map生成子图sub_graph
5. 通过astar算法查找最短路径

6. 合并请求结果并且返回

下面在结合具体的流程进行分析，这里主要要弄清楚2点：1.为什么要生成子图？2.如何通过astar算法查找最优路径？

首先我们从”routing_component.h”和”routing_component.cc”开始，apollo的功能被划分为各个模块，启动时候由cyber框架根据模块间的依赖顺序加载(每个模块的dag文件定义了依赖顺序)，每次开始查看一个模块时，都是从**component**文件开始。

```
class RoutingComponent final
    : public ::apollo::cyber::Component<RoutingRequest> {
public:
    // default用来控制默认构造函数的生成。显式地指示编译器生成该函数的默认版本。
    RoutingComponent() = default;
    ~RoutingComponent() = default;

public:
    bool Init() override;
    // 收到routing request的时候触发
    bool Proc(const std::shared_ptr<RoutingRequest>& request)
override;

private:
    // routing消息发布handle
    std::shared_ptr<::apollo::cyber::Writer<RoutingResponse>>
response_writer_ =
    nullptr;
    std::shared_ptr<::apollo::cyber::Writer<RoutingResponse>>
    response_history_writer_ = nullptr;
    // Routing类
    Routing routing_;
    std::shared_ptr<RoutingResponse> response_ = nullptr;
    // 定时器
    std::unique_ptr<::apollo::cyber::Timer> timer_;
    // 锁
    std::mutex mutex_;
};
```

```
// 在cyber框架中注册routing模块  
CYBER_REGISTER_COMPONENT(RoutingComponent)
```

routing模块都按照cyber的模块申明和注册， cyber框架负责调用Init进行初始化，并且收到消息时候触发Proc执行。 我们先看下”Init”函数：

```
bool RoutingComponent::Init() {  
    // 设置消息qos， 控制流量， 创建消息发布response_writer_  
    apollo::cyber::proto::RoleAttributes attr;  
    attr.set_channel_name(FLAGS_routing_response_topic);  
    auto qos = attr.mutable_qos_profile();  
    qos->  
    >set_history(apollo::cyber::proto::QosHistoryPolicy::HISTORY_  
    KEEP_LAST);  
    qos->set_reliability(  
  
        apollo::cyber::proto::QosReliabilityPolicy::RELIABILITY_RELIA  
        BLE);  
    qos->set_durability(  
  
        apollo::cyber::proto::QosDurabilityPolicy::DURABILITY_TRANSIE  
        NT_LOCAL);  
    response_writer_ = node_->CreateWriter<RoutingResponse>  
(attr);  
  
    ...  
    // 历史消息发布， 和response_writer_类似  
    response_history_writer_ = node_-  
>CreateWriter<RoutingResponse>(attr_history);  
  
    // 创建定时器  
    std::weak_ptr<RoutingComponent> self =  
        std::dynamic_pointer_cast<RoutingComponent>  
(shared_from_this());  
    timer_.reset(new ::apollo::cyber::Timer(  
        FLAGS_routing_response_history_interval_ms,  
        [self, this]() {  
            auto ptr = self.lock();  
            if (ptr) {  
                std::lock_guard<std::mutex> guard(this->mutex_);  
                if (this->response_.get() != nullptr) {  
                    auto response = *response_;
```

```

        auto timestamp =
apollo::common::time::Clock::NowInSeconds();
        response.mutable_header()-
>set_timestamp_sec(timestamp);
        this->response_history_writer_->Write(response);
    }
}
},
false));
timer_->Start();

// 执行Routing类
return routing_.Init().ok() && routing_.Start().ok();
}

```

接下来当routing模块收到routing_request时，会触发”Proc()”，返回routing_response：

```

bool RoutingComponent::Proc(const
std::shared_ptr<RoutingRequest>& request) {
auto response = std::make_shared<RoutingResponse>();
// 响应routing_请求
if (!routing_.Process(request, response.get())) {
    return false;
}
// 填充响应头部信息，并且发布
common::util::FillHeader(node_->Name(), response.get());
response_writer_->Write(response);
{
    std::lock_guard<std::mutex> guard(mutex_);
    response_ = std::move(response);
}
return true;
}

```

从上面的分析可以看出，”RoutingComponent”模块实现的主要功能：

1. 实现”Init”和”Proc”函数
2. 接收”RoutingRequest”消息，输出”RoutingResponse”响应。

接下来我们来看routing的具体实现。

17.2.1. Routing类

“Routing”类的实现在“routing.h”和“routing.cc”中，首先看下“Routing”类引用的头文件：

```
#include "modules/common/monitor_log/monitor_log_buffer.h"
#include "modules/common/status/status.h"
#include "modules/map/hdmap/hdmap_util.h"
#include "modules/routing/core/navigator.h"
#include "modules/routing/proto/routing_config.pb.h"
```

看代码之前先看下头文件是个很好的习惯。通过头文件，我们可以知道当前模块的依赖项，从而搞清楚各个模块之间的依赖关系。可以看到”Routing”模块是一个相对比较独立的模块，只依赖于地图。我们先看下Routing的初始化函数。

```
apollo::common::Status Routing::Init() {
    // 读取routing_map, 也就是点和边
    const auto routing_map_file =
        apollo::hdmap::RoutingMapFile();
    navigator_ptr_.reset(new Navigator(routing_map_file));

    // 读取地图, 用来查找routing request请求的点距离最近的lane,
    // 并且返回对应的lane id, 这里很好理解, 比如你在小区里面, 需要打车,
    // 需要找到最近的乘车点, 说直白点, 就是找到最近的路。
    hdmap_ = apollo::hdmap::HDMapUtil::BaseMapPtr();
}
```

之后会执行”Process”主流程，执行的过程如下：

```
bool Routing::Process(const std::shared_ptr<RoutingRequest>&
routing_request,
                      RoutingResponse* const
routing_response) {
    // 找到routing_request节点最近的路
    const auto& fixed_request =
FillLaneInfoIfMissing(*routing_request);
    // 是否能够找到规划路径
    if (!navigator_ptr_->SearchRoute(fixed_request,
routing_response)) {
```

```

        monitor_logger_buffer_.WARN("Routing failed! " +
                                     routing_response-
>status().msg());
        return false;
    }
    monitor_logger_buffer_.INFO("Routing success!");
    return true;
}

```

上述的过程总结一下就是，首先读取routing_map并初始化Navigator类，接着遍历routing_request，因为routing_request请求为一个个的点，所以先查看routing_request的点是否在路上，不在路上则找到最近的路，并且补充信息（不在路上的点则过不去），最后调用”navigator_ptr_->SearchRoute”返回routing响应。

17.2.2. 导航

Navigator初始化

```

bool Navigator::Init(const RoutingRequest& request, const
TopoGraph* graph,
                      std::vector<const TopoNode*>* const
way_nodes,
                      std::vector<double>* const way_s) {
    // 获取routing请求，对应图中的节点
    if (!GetWayNodes(request, graph_.get(), way_nodes, way_s))
{
    AERROR << "Failed to find search terminal point in
graph!";
    return false;
}
    // 根据请求生成对应的黑名单lane
    black_list_generator_->GenerateBlackMapFromRequest(request,
graph_.get(),
&topo_range_manager_);
}

```

在routing请求中可以指定黑名单路和车道，这样routing请求将不会计算这些车道。应用场景是需要避开拥堵路段，这需要能够根据情况实时

请求，在routing_request中可以设置黑名单也刚好可以满足上面的需求，如果直接把黑名单路段固定，则是一个比较蠢的设计。剩下的一些过程比较简单，我们直接看主函数”SearchRouteByStrategy”：

```
bool Navigator::SearchRouteByStrategy(
    const TopoGraph* graph, const std::vector<const
    TopoNode*>& way_nodes,
    const std::vector<double>& way_s,
    std::vector<NodeWithRange>* const result_nodes) const {

    // 通过Astar算法来查找路径
    strategy_ptr.reset(new
        AStarStrategy(FLAGS_enable_change_lane_in_result));

    std::vector<NodeWithRange> node_vec;
    // 编译routing_request节点
    for (size_t i = 1; i < way_nodes.size(); ++i) {
        const auto* way_start = way_nodes[i - 1];
        const auto* way_end = way_nodes[i];
        double way_start_s = way_s[i - 1];
        double way_end_s = way_s[i];

        TopoRangeManager full_range_manager =
topo_range_manager_;
        // 添加黑名单，这里主要是把车道根据起点和终点做分割。
        black_list_generator_->AddBlackMapFromTerminal(
            way_start, way_end, way_start_s, way_end_s,
&full_range_manager);
        // 因为对车道做了分割，这里会创建子图，比如一个车道分成2个子节点，
        // 2个子节点会创建一张子图。
        SubTopoGraph sub_graph(full_range_manager.RangeMap());
    }

    // 获取起点
    const auto* start = sub_graph.GetSubNodeWithS(way_start,
way_start_s);

    // 获取终点
    const auto* end = sub_graph.GetSubNodeWithS(way_end,
way_end_s);

    // 通过Astar查找最优路径
```

```

    std::vector<NodeWithRange> cur_result_nodes;
    if (!strategy_ptr->Search(graph, &sub_graph, start, end,
                               &cur_result_nodes)) {
        return false;
    }
    // 保存结果到node_vec
    node_vec.insert(node_vec.end(), cur_result_nodes.begin(),
                    cur_result_nodes.end());
}

// 合并Route
if (!MergeRoute(node_vec, result_nodes)) {
    return false;
}
return true;
}

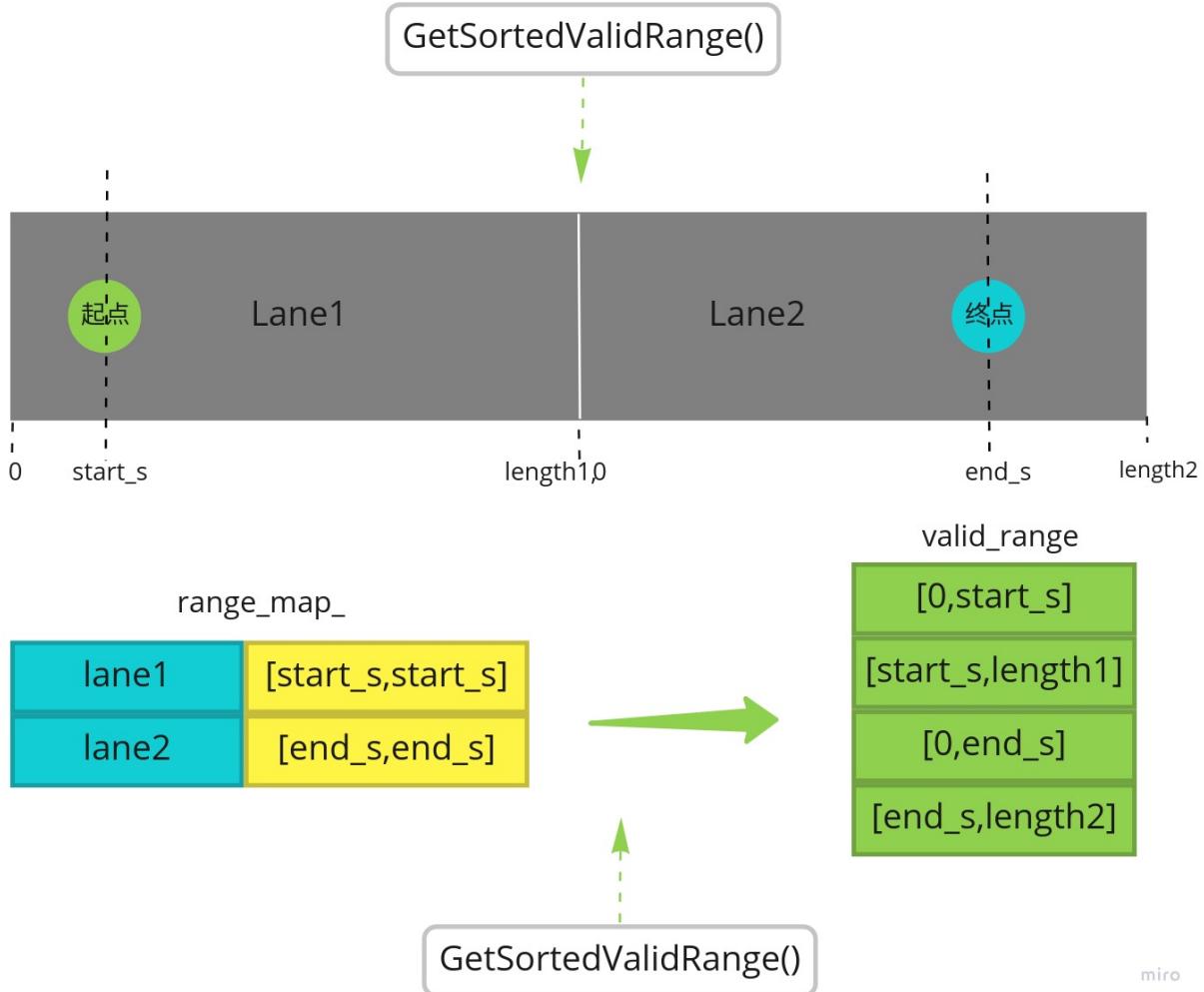
```

17.2.3. 子节点

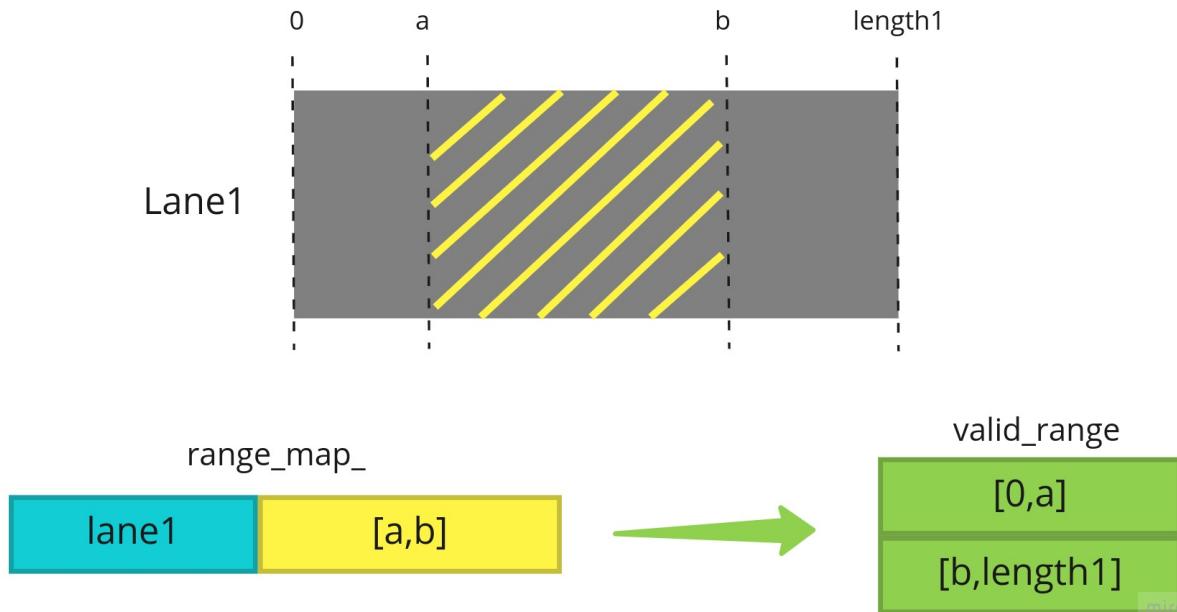
下面我们把子图的概念讲解一下，”AddBlackMapFromTerminal”中会把节点(这里的节点就是lane)切分，切分之后的数据保存在”TopoRangeManager”中，而”SubTopoGraph”会根据”TopoRangeManager”中的数据初始化子图。我们先理解下子节点的概念，节点就是一条lane，而子节点是对lane做了切割，把一条lane根据黑名单区域，生成几个子节点。用图来说明很形象：

17.2.4. 节点切分

节点的切分是根据TopoRangeManager生成好的区间，然后进行切分生成子节点。我们先看下如何生成”TopoRangeManager”，在”AddBlackMapFromTerminal”输入参数为**routing_request**的开始lane，结束的lane，开始位置，结束位置，输出参数为分段好的区间(**range**)，在”range_map_”中保存lane和lane中range的关系，其中key为节点，value为区间(range)。我们还是先看一张图，来描述这个过程：



可以看到上图中有2条lane，lane1为 $[0, \text{length1}]$ ，lan2为 $[0, \text{length2}]$ ，routing_request的起点为`start_s`，终点为`end_s`（注意这里的起点和终点都是在对应lane中的位置，而不是整条路的长度）。之后会生成一个range_map，其中的键为node即对应的lane，值为切分好的range，只是这里的range比较特殊，拿lane1举例子，这里range的起点和终点都是`start_s`。之后再通过”GetSortedValidRange”找到合法的区间，这里就看到找到range为 $[0, \text{start}_s]$, $[\text{start}_s, \text{length1}]$ 。这样这个节点就切分成2个子节点。实际上上述特殊的例子只是为了把routing请求的起点和终点对lane做切分，这样做好处可能是模块的功能能够统一，方便计算节点的代价。实际上真正的功能没用到，我们还是根据图来解释这个过程：



上述的过程是根据一段 $\text{range}[a,b]$, 即 $[a,b]$ 为黑名单, 或者禁止停车区域, 那么最后生成的 range_map , 其中的键为 node 即对应的 lane , 值为黑名单的 $\text{range}[a,b]$, 和上面的例子不一样的地方在于, 这里起点和终点不是一个点, 而是一个 range , 这样生成的 valid_range 则为 $[0,a]$, $[b,\text{length1}]$, 生成的2个子节点就是可以通过的区域, 而这2个节点不是连续的, 不能直接通过。也就是说 routing_request 是这种情况的特例, 即黑名单 range 的起点和终点都是一个点, 生成的2个子节点也就是连续的。apollo的代码做了冗余设计, 但是实际没有用到, 我们在总结下2种添加黑名单的设计:

1. **GenerateBlackMapFromRequest** - 通过 request 请求传入黑名单 lane 和 road , 每次直接屏蔽一整条 road 或者 lane 。
2. **AddBlackMapFromTerminal** - 虽然 range_manager 支持传入 range , 但是这种场景只是针对 routing_request 传入的点对 lane 做切割, 方便计算, 每次切割的区间的起点和终点重合, 是一个特殊场景, 后续应该有用到比如在一条 lane 里, 有某一段不能行驶的功能。
接下来看具体的实现:

```
void BlackListRangeGenerator::AddBlackMapFromTerminal(
    const TopoNode* src_node, const TopoNode* dest_node,
    double start_s,
    double end_s, TopoRangeManager* const range_manager)
const {
```

```

double start_length = src_node->Length();
double end_length = dest_node->Length();
...
double start_cut_s = MoveSBackward(start_s, 0.0);
// 注意这里range的起点和终点是同一个点, 为routing的起点
range_manager->Add(src_node, start_cut_s, start_cut_s);
// 把平行的节点也按照比例做相同的切分
AddBlackMapFromOutParallel(src_node, start_cut_s /
start_length,
                           range_manager);

// 注意这里range的起点和终点是同一个点, 为routing的终点
double end_cut_s = MoveSForward(end_s, end_length);
range_manager->Add(dest_node, end_cut_s, end_cut_s);
AddBlackMapFromInParallel(dest_node, end_cut_s /
end_length, range_manager);

// 排序并且合并
range_manager->SortAndMerge();
}

```

接着就是根据上面的range生成valid_range, 在”GetSortedValidRange”中实现:

```

void GetSortedValidRange(const TopoNode* topo_node,
                        const std::vector<NodeSRange>&
origin_range,
                        std::vector<NodeSRange>*
valid_range) {
    std::vector<NodeSRange> block_range;
    MergeBlockRange(topo_node, origin_range, &block_range);
    double start_s = topo_node->StartS();
    double end_s = topo_node->EndS();
    std::vector<double> all_value;
    // 添加node起点, 边界和node终点
    all_value.push_back(start_s);
    for (const auto& range : block_range) {
        all_value.push_back(range.StartS());
        all_value.push_back(range.EndS());
    }
    all_value.push_back(end_s);
}

```

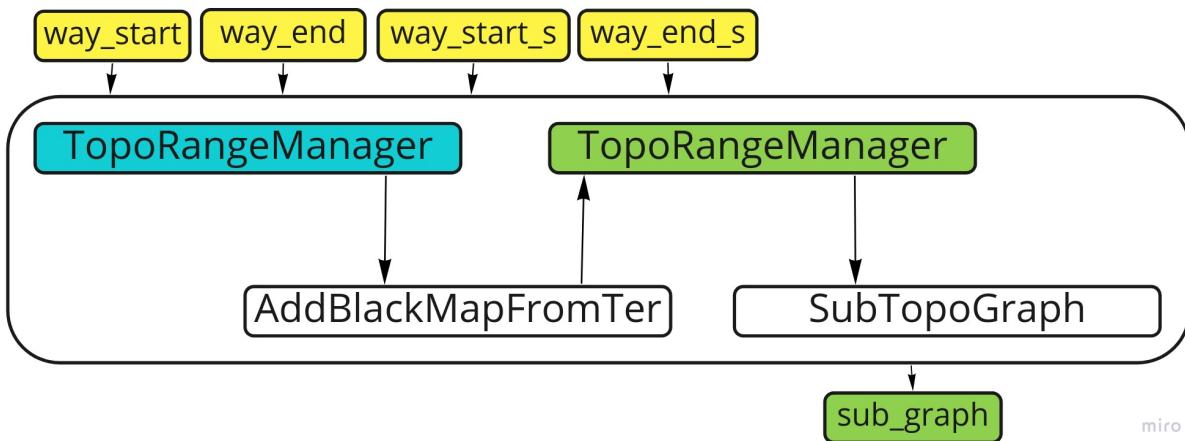
```

// **核心在这里，每次i+2，即跳过balck_range，生成valid_range**
for (size_t i = 0; i < all_value.size(); i += 2) {
    NodeSRange new_range(all_value[i], all_value[i + 1]);
    valid_range->push_back(std::move(new_range));
}
}

```

17.2.5. 生成子图

然后我们再回过头去看下如何生成子图，生成子图的流程如下：



生成子图主要在构造函数中：

```

SubTopoGraph::SubTopoGraph(
    const std::unordered_map<const TopoNode*,
std::vector<NodeSRange>>&
    black_map) {
    std::vector<NodeSRange> valid_range;
    for (const auto& map_iter : black_map) {
        valid_range.clear();
        GetSortedValidRange(map_iter.first, map_iter.second,
&valid_range);
        // 生成子节点
        InitSubNodeByValidRange(map_iter.first, valid_range);
    }

    for (const auto& map_iter : black_map) {
        // 生成子边
        InitSubEdge(map_iter.first);
    }
}

```

```

    }

    for (const auto& map_iter : black_map) {
        AddPotentialEdge(map_iter.first);
    }
}

```

上述过程比较简单，浏览代码即可以理解。我们主要看下如何使用 `subgraph`。

由于Graph节点中已经有边的信息，因此原先的Graph中的边的信息实际上已经保存在节点中了，最后Astar实际上只用到了子图的信息，因为节点有自己边的信息。`subgraph`的核心是通过边找到子边，如果节点不存在子节点，那么返回原先的边，通过该函数可以同时找到边和子边，这样节点和子节点都可以找到了。我们下面重点分析下”`GetSubInEdgesIntoSubGraph`”，另外一个类似：

```

void SubTopoGraph::GetSubInEdgesIntoSubGraph(
    const TopoEdge* edge,
    std::unordered_set<const TopoEdge*>* const sub_edges)
const {
    const auto* from_node = edge->FromNode();
    const auto* to_node = edge->ToNode();
    std::unordered_set<TopoNode*> sub_nodes;
    // 如果起点是子节点，终点是子节点，或者终点没有子节点，返回边
    // TODO: **这里传入的是edge，根本不可能是子节点? **
    if (from_node->IsSubNode() || to_node->IsSubNode() ||
        !GetSubNodes(to_node, &sub_nodes)) {
        sub_edges->insert(edge);
        return;
    }
    // 如果终点有子节点，则返回所有的子边
    for (const auto* sub_node : sub_nodes) {
        for (const auto* in_edge : sub_node->InFromAllEdge()) {
            if (in_edge->FromNode() == from_node) {
                sub_edges->insert(in_edge);
            }
        }
    }
}

```

关于子图的分析就结束了，子图主要是针对一条lane切分为几个子节点的情况，根据切分好的子节点从新生成一张图，比原先根据routing_map建立的图有更细的粒度。

17.2.6. Astar算法

最后根据生成好的子图，通过Astar算法来查找最佳路径，实现
在”routing/strategy”目录。可以看到strategy中实现了一个”Strategy”的基
类，也就是说后面可以扩展其他的查找策略。

```
class Strategy {
public:
    virtual ~Strategy() {}
    // 在派生类中实现查找方法
    virtual bool Search(const TopoGraph* graph, const
SubTopoGraph* sub_graph,
                     const TopoNode* src_node, const
TopoNode* dest_node,
                     std::vector<NodeWithRange>* const
result_nodes) = 0;
};
```

下面先介绍astar算法的原理，astar算法这是一种在图形平面上，有多个节点的路径，求出最低通过成本的算法。在此算法中，如果以 $g(n)$ 表示从起点到任意顶点n的实际距离， $h(n)$ 表示任意顶点 n到目标顶点的估算距离（根据所采用的评估函数的不同而变化），那么A*算法的估算函数为：

$$f(n) = g(n) + h(n)$$

这个公式遵循以下特性：如果 $g(n)$ 为0，即只计算任意顶点n到目标的评估函数 $h(n)$ ，而不计算起点到顶点n的距离，则算法转化为使用贪心策略的最良优先搜索，速度最快，但可能得不出最优解；如果 $h(n)$ 不大于顶点n到目标顶点的实际距离，则一定可以求出最优解，而且 $h(n)$ 越小，需要计算的节点越多，算法效率越低，常见的评估函数有——欧几里得距离、曼哈顿距离、切比雪夫距离；如果 $h(n)$ 为0，即只需要求出起点到任意顶点n的最短路径 $g(n)$ ，而不计算任何评估函数 $h(n)$ ，则

转化为单源最短路径问题，即Dijkstra算法，此时需要计算最多的顶点；

TODO: 具体算法实现可以参考维基百科的伪代码，由于网上已经有大量的astar算法的介绍，这里暂时先跳过，后面再增加这一部分的介绍。

跳过Astar算法找到最优路径之后，发送routing_response，然后planning模块根据生成好的路径，控制车辆行驶。这里routing模块的使命就完成了，除非planning模块需要重新规划，则会重新发送routing_request再进行规划。

17.3. 调试工具

在routing/tools目录实现了如下3个功能：

```
routing_cast.cc // 定时发送routing response响应  
routing_dump.cc // 保存routing请求  
routing_tester.cc // 定时发送routing request请求
```

17.4. 问题

如果是曲线转弯，并且需要变道的情况，是否可以规划？比如在十字路口，左转中途有车挡住，这时候需要变道，就是edge左转，再加上node是曲线的情况，是否能够实现，这应该是planning应该考虑的情况？答：可以实现，lane有直道和弯道的区别，edge有左转和右转的区别，在转弯过程中如果需要左转，继续左转就可以了，这里只描述了道路的信息，不关注是直道还是弯道。

17.5. OSM数据查找

通过下面的链接，替换掉网址最后的id，就可以查找到对应的way，node和relation。

- If it's a polygon, then it's a closed way in the OSM database. You can find ways by id as simple as this:

<http://www.openstreetmap.org/way/305293190>

- If a specific node (the building blocks of ways) is giving a problem, the link would be : <http://www.openstreetmap.org/node/305293190>
- If it is a multipolygon (for example a building with a hole in it), the link would be : <http://www.openstreetmap.org/relation/305293190>

地图可以在下面的链接选择导出， 导出格式为OSM格式：

- You can download the osm from:
<https://www.openstreetmap.org/export#map=15/22.5163/113.9380>

17.6. Reference

- [OSM地图介绍](#) [<https://blog.csdn.net/scy411082514/article/details/7484497>]
- [OSM Routing](#) [<https://wiki.openstreetmap.org/wiki/Routing>]
- [OSM Pathfinding](#) [<https://github.com/daohu527/osm-pathfinding>]
- [OpenstreetMap](#) [<https://www.openstreetmap.org/>]
- [OpenstreetMap Elements](#) [<https://wiki.openstreetmap.org/wiki/Elements>]
- [OpenstreetMap地图渲染](#) [https://wiki.openstreetmap.org/wiki/Zh-hans:Beginners_Guide_1.5]
- [最短路径问题](#) [https://en.wikipedia.org/wiki/Shortest_path_problem]
- [A*搜索算法](#)
[https://zh.wikipedia.org/wiki/A*%E6%90%9C%E5%B0%8B%E6%BC%94%E7%AE%97%E6%B3%95]

18. Tools

君子生非异也，善假于物也。

18.1. mapviewers

mapviewers用来可视化生成好的高精度地图。

1. 首先编译文件，这里可以把python文件编译为可执行文件。

```
bazel build //modules/tools/mapviewers:hdmapviewer  
bazel build //modules/tools/mapviewers:gmapviewer
```

2. 编译好之后，执行命令。

```
./bazel-bin/modules/tools/mapviewers/hdmapviewer -m  
modules/map/data/demo/base_map.txt
```

3. 生成好的可视化地图文件在当前目录的base_map.html。

18.1.1. todo

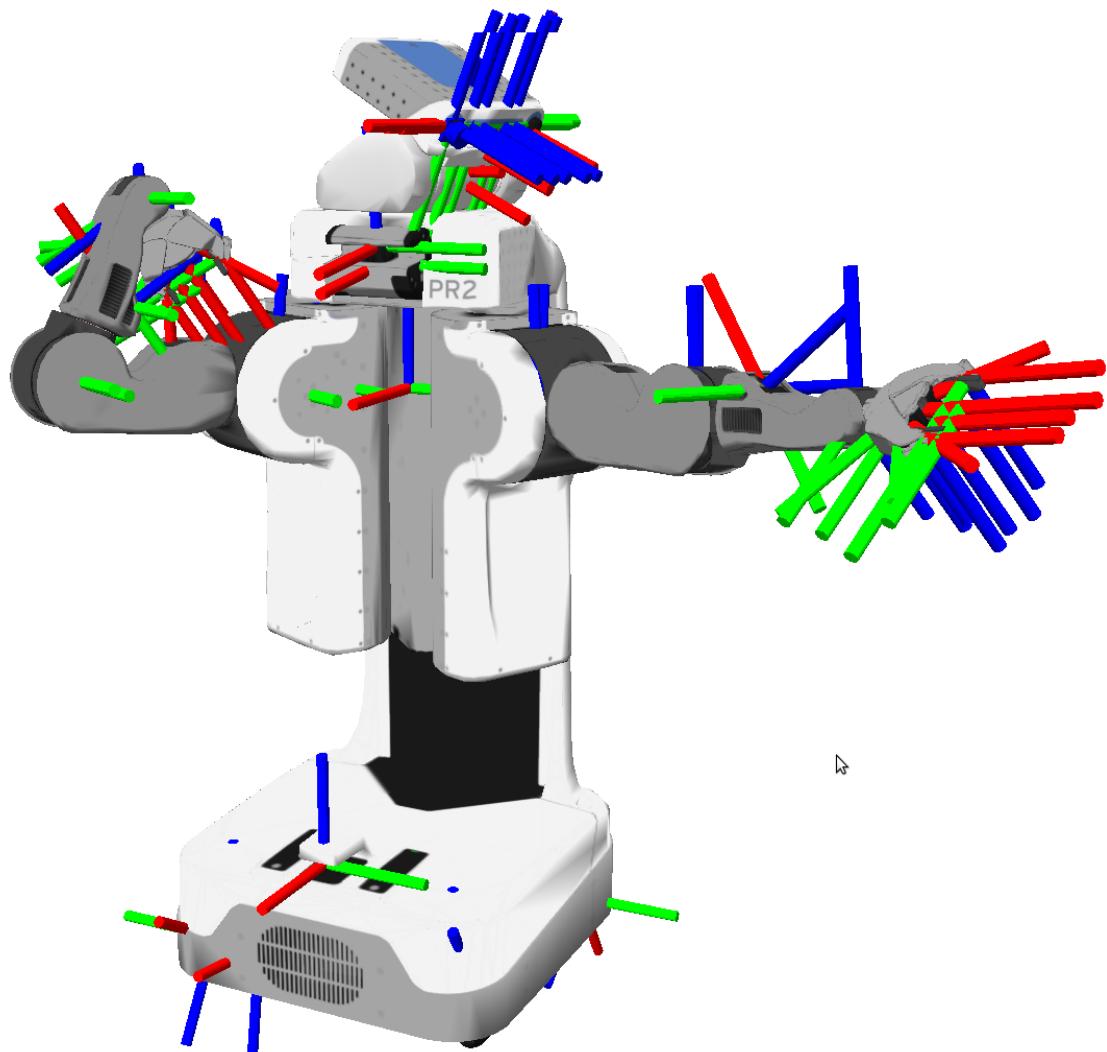
这里gmapviewer和hdmapviewer的区别是什么？

19. Transform

三人行，必有我师。

19.1. Transform模块简介

关于transform模块开始一直不知道是干啥的，一直看到一个”/tf”的TOPIC，还以为是tensorflow的缩写，想着是不是和神经网络有关系，后来才知道tf是transform的缩写，主要的用途是进行坐标转换，原型即是大名鼎鼎的”ros/tf2”库。那么为什么要进行坐标转换呢？



在机器人系统中，经常需要用到坐标转换，比如一个机器人的手臂要去拿一个运动的物体，控制的时候我们需要手臂的当前坐标，同时还需要知道手臂和身体的坐标，这个时候就需要用到坐标转换，把手臂的坐标转换为身体的坐标系，这样控制起来就方便一点。当然这里是动态的情况，也存在静态的情况，比如机器人头顶的摄像头，转换到身体的坐标系，那么位置关系是相对固定的，所以可以一开始就写到固定的位置。这里就引入了以下几个问题：

1. 有固定转换关系的文件放在哪里？如果都是集中放在一个地方，那么这个地方损坏会导致所有的转换关系失效，一个比较好的方法是各个节点自己广播自己的转换关系。而其实静态的转换关系只需要发送一次就可以了，因为不会变化。
2. 有动态转换关系的节点，需要实时动态发布自己的转换关系，这样会涉及到时间戳，以及过时。
3. 转换关系的拓扑结构如何确定？是树型还是网络型的，这涉及到转换关系传递的问题。

19.2. Transform(静态变换)

TransformComponent模块的入口

在”static_transform_component.cc”和”static_transform_component.h”中。实现了”StaticTransformComponent”类，我们接下来看下它的实现。

```
class StaticTransformComponent final : public
apollo::cyber::Component<> {
public:
    StaticTransformComponent() = default; // 构造函数
    ~StaticTransformComponent() = default; // 析构函数

public:
    bool Init() override; // 初始化函数

private:
    void SendTransforms(); // 发送变换
    void SendTransform(const std::vector<TransformStamped>&
msgtf); // 发送变换，参数为数组
    bool ParseFromYaml(const std::string& file_path,
TransformStamped* transform); // 从yaml中解析

    apollo::static_transform::Conf conf_; // 配置文件
    std::shared_ptr<cyber::Writer<TransformStamped>> writer_;
    // cyber node写句柄
    TransformStampeds transform_stampeds_; // 变换关系，在proto
    中定义
};
```

下面我们来分析StaticTransformComponent类具体的实现，首先是Init函数，Init函数做了2件事情，一是读取conf配置，二是发布”/tf_static”消息。

```
bool StaticTransformComponent::Init() {
    // 读取配置
    if (!GetProtoConfig(&conf_)) {
        AERROR << "Parse conf file failed, " << ConfigFilePath();
        return false;
    }
    // 发布消息
    cyber::proto::RoleAttributes attr;
    attr.set_channel_name("/tf_static");
    attr.mutable_qos_profile()->CopyFrom(
        cyber::transport::QosProfileConf::QOS_PROFILE_TF_STATIC);
    // 注意这里的node_继承至apollo::cyber::Component
    writer_ = node_->CreateWriter<TransformStamped>(attr);
    SendTransforms();
    return true;
}
```

接着看SendTransforms()函数，主要就是遍历conf文件，判断extrinsic_file(实际上对应各种传感器的外参)是否使能，如果使能则根据提供的文件路径解析对应的转换关系”ParseFromYaml”，把转换关系添加到数组”transform_stamped_vec”中，然后发送。

```
void StaticTransformComponent::SendTransforms() {
    std::vector<TransformStamped> transform_stamped_vec;
    // 遍历对应的文件，实际上对应各种传感器的外参
    for (auto& extrinsic_file : conf_.extrinsic_file()) {
        // 是否使能
        if (extrinsic_file.enable()) {
            AINFO << "Broadcast static transform, frame id ["
                << extrinsic_file.frame_id() << "], child frame
id ["
                << extrinsic_file.child_frame_id() << "]";
            TransformStamped transform;
            // 解析yaml文件，获取转换，并且添加到数组中
            if (ParseFromYaml(extrinsic_file.file_path(),
&transform)) {
```

```

        transform_stamped_vec.emplace_back(transform);
    }
}
}

// 发送对应的转换
SendTransform(transform_stamped_vec);
}

```

解析yaml需要注意的地方，在conf中的frame_id和child_id实际上没有使用，最后还是以yaml文件中的为准。其中yaml文件的格式为

```

child_frame_id: novatel
transform:
  translation:
    x: 0.0
    y: 0.0
    z: 0.0
  rotation:
    x: 0.0
    y: 0.0
    z: 0.0
    w: 1.0
header:
  frame_id: localization

```

我们在看下如何解析yaml文件：

```

bool StaticTransformComponent::ParseFromYaml(
    const std::string& file_path, TransformStamped*
transform_stamped) {
    ...
    YAML::Node tf = YAML::LoadFile(file_path);
    try {
        // 读取yaml文件中的frame_id和child_frame_id
        transform_stamped->mutable_header()->set_frame_id(
            tf["header"]["frame_id"].as<std::string>());
        transform_stamped->set_child_frame_id(
            tf["child_frame_id"].as<std::string>());
        // translation 位置
        auto translation =
            transform_stamped->mutable_transform()-
        >mutable_translation();
    }
}

```

```

        translation->set_x(tf["transform"]["translation"]
["x"].as<double>());
        translation->set_y(tf["transform"]["translation"]
["y"].as<double>());
        translation->set_z(tf["transform"]["translation"]
["z"].as<double>());
        // rotation 角度
        auto rotation = transform_stamped->mutable_transform()-
>mutable_rotation();
        rotation->set_qx(tf["transform"]["rotation"]
["x"].as<double>());
        rotation->set_qy(tf["transform"]["rotation"]
["y"].as<double>());
        rotation->set_qz(tf["transform"]["rotation"]
["z"].as<double>());
        rotation->set_qw(tf["transform"]["rotation"]
["w"].as<double>());
    } catch (...) {
        AERROR << "Extrinsic yaml file parse failed: " <<
file_path;
        return false;
    }
    return true;
}

```

最后我们再看下如何发送转换关系:

```

void StaticTransformComponent::SendTransform(
    const std::vector<TransformStamped>& msgtf) {
    for (auto it_in = msgtf.begin(); it_in != msgtf.end();
++it_in) {
        bool match_found = false;
        int size = transform_stampeds_.transforms_size();

        // 如果child_frame_id重复, 那么则覆盖对应的配置
        for (int i = 0; i < size; ++i) {
            if (it_in->child_frame_id() ==
                transform_stampeds_.mutable_transforms(i)-
>child_frame_id()) {
                auto it_msg =
transform_stampeds_.mutable_transforms(i);
                *it_msg = *it_in;
                match_found = true;
            }
        }
    }
}

```

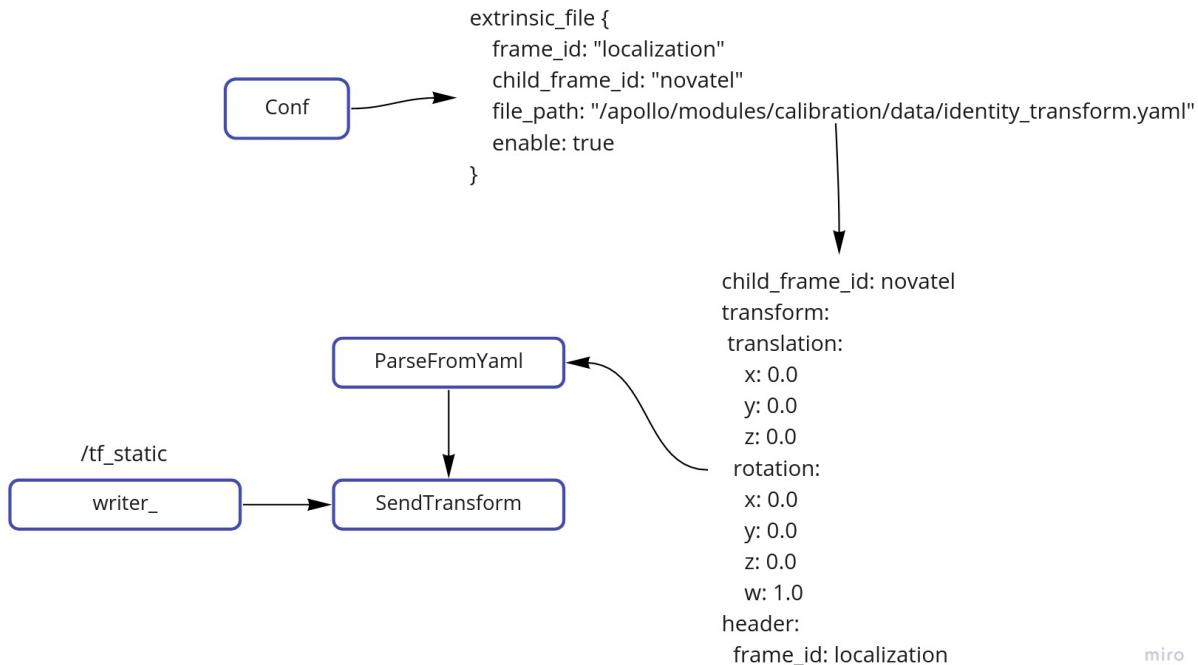
```

        break;
    }
}
if (!match_found) {
    // 获取增加的指针地址，并且赋值
    auto ts = transform_stampeds_.add_transforms();
    *ts = *it_in;
}
writer_->Write(std::make_shared<TransformStamped>
(transform_stampeds_));
}

```

所以这里需要注意**child_frame_id**的值一定不要一样，否则会覆盖之前的配置。

按照流程总结一下就如下图所示，首先遍历conf，获取传感器的外参数文件路径，然后解析对应的yaml文件，并且发布到”/tf_static”。



19.3. transform_broadcaster（广播）

各个模块通过广播的方式来发布动态变换，实际上就是各个模块通过调用**transform_broadcaster**的库函数来实现广播转换消息，我们接

下来看下transform_broadcaster是如何实现的，transform_broadcaster做为一个lib库，入口在”transform_broadcaster.h”和”transform_broadcaster.cc”中。

```
class TransformBroadcaster {
public:
    // 这里注意构造的时候需要传入node
    explicit TransformBroadcaster(const std::shared_ptr<cyber::Node>& node);

    // 发送单个转换关系
    void SendTransform(const TransformStamped& transform);

    // 发送一组转换关系
    void SendTransform(const std::vector<TransformStamped>& transforms);

private:
    std::shared_ptr<cyber::Node> node_;
    std::shared_ptr<cyber::Writer<TransformStamped>> writer_;
};
```

从上面的分析可以看出，构造TransformBroadcaster的时候需要传入node。为什么需要传入node呢，因为cyber的一个module不能同时创建2个node，所以这里谁调用，就用谁的node创建reader和writer。如果是自己创建node，那么其他模块自己的node和引用该模块创建的node就打破了cyber上述的限制，这里的node是否可以理解为一个进程？下面我们分析具体的实现，首先是TransformBroadcaster构造函数：

```
TransformBroadcaster::TransformBroadcaster(
    const std::shared_ptr<cyber::Node>& node)
: node_(node) {
    cyber::proto::RoleAttributes attr;
    // 发布的topic为"/tf"
    attr.set_channel_name("/tf");
    writer_ = node_->CreateWriter<TransformStamped>(attr);
}
```

创建writer并且往”/tf”发消息。这里可以看到，这里存在多个节点往一个topic发消息的情况。发送消息比较简单，直接写对应的消息：

```

void TransformBroadcaster::SendTransform(
    const std::vector<TransformStamped>& transforms) {
    auto message = std::make_shared<TransformStamped>();
    *message->mutable_transforms() = {transforms.begin(), transforms.end()};
    writer_->Write(message);
}

```

19.4. Buffer (接收缓存)

Buffer实际上提供了一个工具类给其它模块，它的主要作用是接收”/tf”和”/tf_static”的消息，并且保持在buffer中，提供给其它节点进行查找和转换到对应的坐标系，我们先看BufferInterface的实现：

19.4.1. 缓存接口

BufferInterface类定义了缓存需要实现的接口：

```

class BufferInterface {
public:
    // 根据frame_id获取2帧的转换关系
    virtual apollo::transform::TransformStamped
lookupTransform(
    const std::string& target_frame, const std::string&
source_frame,
    const cyber::Time& time, const float timeout_second =
0.01f) const = 0;

    // 根据frame_id获取2帧的转换关系，假设固定帧？？
    virtual apollo::transform::TransformStamped
lookupTransform(
    const std::string& target_frame, const cyber::Time&
target_time,
    const std::string& source_frame, const cyber::Time&
source_time,
    const std::string& fixed_frame,
    const float timeout_second = 0.01f) const = 0;

    // 测试转换是否可行
    virtual bool canTransform(const std::string& target_frame,

```

```
        const std::string& source_frame,
        const cyber::Time& time,
        const float timeout_second =
0.01f,
                           std::string* errstr = nullptr)
const = 0;

// 测试转换是否可行
virtual bool canTransform(const std::string& target_frame,
                           const cyber::Time& target_time,
                           const std::string& source_frame,
                           const cyber::Time& source_time,
                           const std::string& fixed_frame,
                           const float timeout_second =
0.01f,
                           std::string* errstr = nullptr)
const = 0;

// Transform, simple api, with pre-allocation
// 预分配内存
template <typename T>
T& transform(const T& in, T& out, const std::string&
target_frame, // NOLINT
            float timeout = 0.0f) const {
    // do the transform
    tf2::doTransform(in, out, lookupTransform(target_frame,
tf2::getFrameId(in),

tf2::getTimestamp(in), timeout));
    return out;
}

// transform, simple api, no pre-allocation
// 没有预分配内存
template <typename T>
T transform(const T& in, const std::string& target_frame,
            float timeout = 0.0f) const {
    T out;
    return transform(in, out, target_frame, timeout);
}

// transform, simple api, different types, pre-allocation
// 不同的类型
```

```

template <typename A, typename B>
B& transform(const A& in, B& out, const std::string&
target_frame, // NOLINT
             float timeout = 0.0f) const {
    A copy = transform(in, target_frame, timeout);
    tf2::convert(copy, out);
    return out;
}

// Transform, advanced api, with pre-allocation
template <typename T>
T& transform(const T& in, T& out, const std::string&
target_frame, // NOLINT
             const cyber::Time& target_time, const
std::string& fixed_frame,
             float timeout = 0.0f) const {
    // do the transform
    tf2::doTransform(
        in, out, lookupTransform(target_frame, target_time,
tf2::getFrameId(in),
                               tf2::getTimestamp(in),
fixed_frame, timeout));
    return out;
}

// transform, advanced api, no pre-allocation
template <typename T>
T transform(const T& in, const std::string& target_frame,
            const cyber::Time& target_time, const
std::string& fixed_frame,
            float timeout = 0.0f) const {
    T out;
    return transform(in, out, target_frame, target_time,
fixed_frame, timeout);
}

// Transform, advanced api, different types, with pre-
allocation
template <typename A, typename B>
B& transform(const A& in, B& out, const std::string&
target_frame, // NOLINT
             const cyber::Time& target_time, const
std::string& fixed_frame,

```

```

        float timeout = 0.0f) const {
    // do the transform
    A copy = transform(in, target_frame, target_time,
fixed_frame, timeout);
    tf2::convert(copy, out);
    return out;
}
};

```

BufferInterface实现的功能主要是查找转换关系，以及查看转换关系是否存在，以及做最后的转换。

19.4.2. 缓存实现

下面我们接着看buffer类的实现，可以看到buffer类继承了”BufferInterface”和”tf2::BufferCore”，其中”tf2::BufferCore”就是大名鼎鼎的ROS中的tf2库。

```

class Buffer : public BufferInterface, public tf2::BufferCore
{
public:
    using tf2::BufferCore::canTransform;
    using tf2::BufferCore::lookupTransform;

    // 构造buffer object
    int Init();

    // 根据frame_id获取2帧的转换关系，继承至BufferInterface
    virtual apollo::transform::TransformStamped
lookupTransform(
        const std::string& target_frame, const std::string&
source_frame,
        const cyber::Time& time, const float timeout_second =
0.01f) const;

    // 继承至BufferInterface
    virtual apollo::transform::TransformStamped
lookupTransform(
        const std::string& target_frame, const cyber::Time&
target_time,
        const std::string& source_frame, const cyber::Time&

```

```
source_time,
    const std::string& fixed_frame, const float
timeout_second = 0.01f) const;

// 继承至BufferInterface
virtual bool canTransform(const std::string& target_frame,
                           const std::string& source_frame,
                           const cyber::Time& target_time,
                           const float timeout_second =
0.01f,
                           std::string* errstr = nullptr)
const;

// 继承至BufferInterface
virtual bool canTransform(const std::string& target_frame,
                           const cyber::Time& target_time,
                           const std::string& source_frame,
                           const cyber::Time& source_time,
                           const std::string& fixed_frame,
                           const float timeout_second =
0.01f,
                           std::string* errstr = nullptr)
const;

private:
    // 转换回调? ? ?
    void SubscriptionCallback(
        const std::shared_ptr<const
apollo::transform::TransformStamped>&
        transform);
    void StaticSubscriptionCallback(
        const std::shared_ptr<const
apollo::transform::TransformStamped>&
        transform);
    void SubscriptionCallbackImpl(
        const std::shared_ptr<const
apollo::transform::TransformStamped>&
        transform,
        bool is_static);
    void AsyncSubscriptionCallbackImpl(
        const std::shared_ptr<const
apollo::transform::TransformStamped>&
        transform,
```

```

        bool is_static);

    // 转换tf2消息为cyber protobuf格式
    void TF2MsgToCyber(
        const geometry_msgs::TransformStamped&
        tf2_trans_stamped,
        apollo::transform::TransformStamped& trans_stamped)
    const; // NOLINT

    std::unique_ptr<cyber::Node> node_;

    std::shared_ptr<cyber::Reader<apollo::transform::TransformStamped>>
        message_subscriber_tf_;

    std::shared_ptr<cyber::Reader<apollo::transform::TransformStamped>>
        message_subscriber_tf_static_;

    cyber::Time last_update_;
    std::vector<geometry_msgs::TransformStamped> static_msgs_;
    // 单例
    DECLARE_SINGLETON(Buffer)
}; // class

```

这里注意buffer为单例模式，即接收转换消息，并且放到buffer中保存。其他模块需要用到转换的时候，则从buffer中查找是否存在转换关系，并且进行对应的转换。

下面我们看buffer类的具体实现，buffer类的初始化在Init函数中：

```

int Buffer::Init() {
    std::string node_name =
        "transform_listener_" +
    std::to_string(cyber::Time::Now().ToNanosecond());
    // 创建node节点
    node_ = cyber::CreateNode(node_name);
    cyber::ReaderConfig tf_reader_config;

    // 读取"/tf"消息
    tf_reader_config.channel_name = "/tf";
    tf_reader_config.pending_queue_size = 5;
}

```

```

    message_subscriber_tf_ =
        node_-
>CreateReader<apollo::transform::TransformStamped>(
            tf_reader_config,
            [&] (const std::shared_ptr<const
apollo::transform::TransformStamped>&
            msg_evt) {
SubscriptionCallbackImpl(msg_evt, false); });

// 读取"/tf_static"消息
apollo::cyber::proto::RoleAttributes attr_static;
attr_static.set_channel_name("/tf_static");
attr_static.mutable_qos_profile()->CopyFrom(
    apollo::cyber::transport::QosProfileConf::QOS_PROFILE_TF_STAT
IC);
    message_subscriber_tf_static_ =
        node_-
>CreateReader<apollo::transform::TransformStamped>(
            attr_static,
            [&] (const
std::shared_ptr<apollo::transform::TransformStamped>&
            msg_evt) {
SubscriptionCallbackImpl(msg_evt, true); });

    return cyber::SUCC;
}

```

可以看到在Init函数中主要实现的功能是创建节点，并且订阅”/tf”和”/tf_static”消息，由于Buffer为单例，在cyber初始化的时候创建的node，不是在模块内部创建的node（关于这块，后面有时间在详细论述下，cyber可以存在多个node，而启动的模块则不能，是不是因为cyber做为调度器，为了方便控制）。

回调函数都是SubscriptionCallbackImpl，我们看下它是如何缓存消息的？

```

void Buffer::SubscriptionCallbackImpl(
    const std::shared_ptr<const
apollo::transform::TransformStamped>& msg_evt,

```

```
    bool is_static) {
cyber::Time now = cyber::Time::Now();
// authority的用途? ? ?
std::string authority =
    "cyber_tf"; // msg_evt.getPublisherName(); // lookup
the authority

// 看起来不可能进入这个条件，除非多线程? ? ?
if (now.ToNanosecond() < last_update_.ToNanosecond()) {
    AINFO << "Detected jump back in time. Clearing TF
buffer.";
    clear();
    // cache static transform stamped again.
    for (auto& msg : static_msgs_) {
        setTransform(msg, authority, true);
    }
}
last_update_ = now;

for (int i = 0; i < msg_evt->transforms_size(); i++) {
    try {
        // 封装消息
        geometry_msgs::TransformStamped trans_stamped;

        // header
        const auto& header = msg_evt->transforms(i).header();
        trans_stamped.header.stamp =
            static_cast<uint64_t>(header.timestamp_sec() *
kSecondToNanoFactor);
        trans_stamped.header.frame_id = header.frame_id();
        trans_stamped.header.seq = header.sequence_num();

        // child_frame_id
        trans_stamped.child_frame_id = msg_evt-
>transforms(i).child_frame_id();

        // translation
        const auto& transform = msg_evt-
>transforms(i).transform();
        trans_stamped.transform.translation.x =
transform.translation().x();
        trans_stamped.transform.translation.y =
transform.translation().y();
    }
}
```

```
    trans_stamped.transform.translation.z =
transform.translation().z();

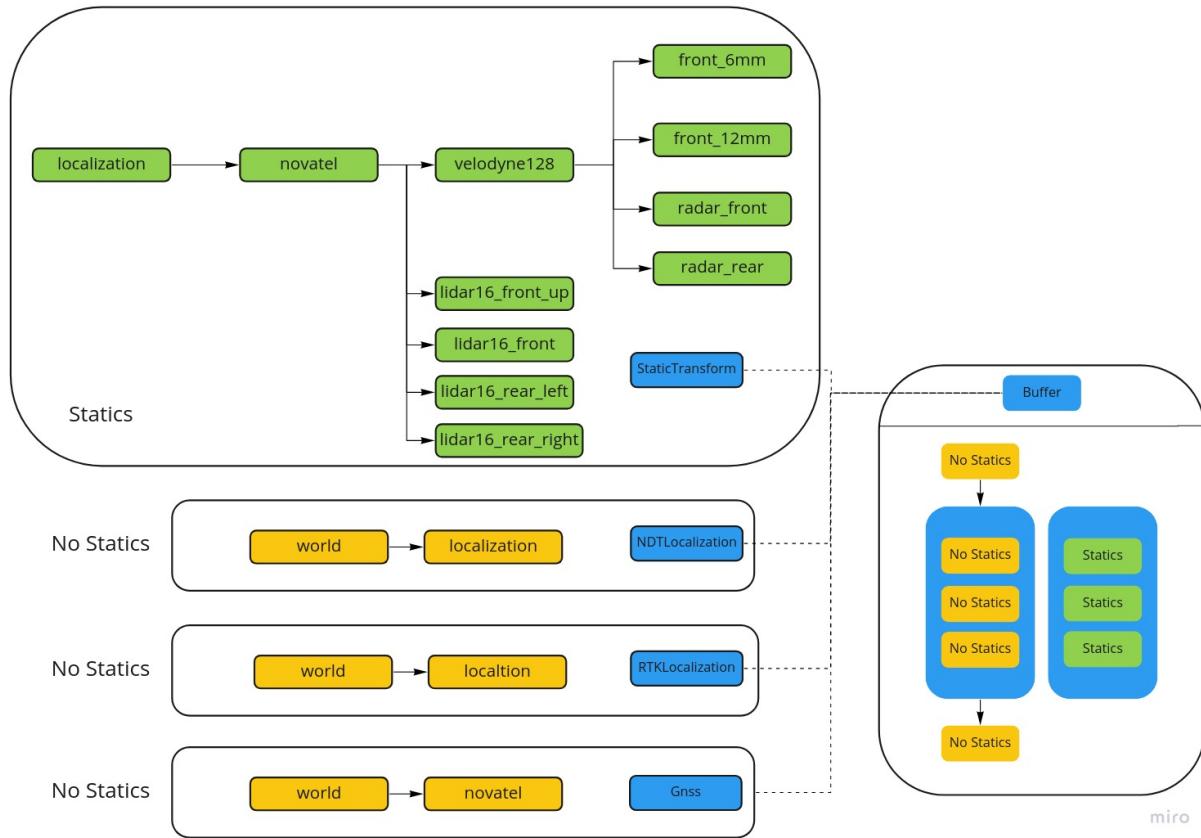
    // rotation
    trans_stamped.transform.rotation.x =
transform.rotation().qx();
    trans_stamped.transform.rotation.y =
transform.rotation().qy();
    trans_stamped.transform.rotation.z =
transform.rotation().qz();
    trans_stamped.transform.rotation.w =
transform.rotation().qw();

    // 保存静态转换，用于上面判断条件的时候，重新设置静态转换
    if (is_static) {
        static_msgs_.push_back(trans_stamped);
    }
    // 调用tf2的函数，保存转换到cache，区分静态和动态的转换
    setTransform(trans_stamped, authority, is_static);
} catch (tf2::TransformException& ex) {
    std::string temp = ex.what();
    AERROR << "Failure to set received transform:" <<
temp.c_str();
}
}
```

接着是lookupTransform和canTransform分别调用tf2的库函数，实现查找转换和判断是否能够转换的实现，由于函数功能比较简单这里就不介绍了。可以看到主要的缓存实现都是在tf2的库函数中，后面有时间再分析下tf2具体的实现。

19.5. 总结

接下来我们用一张图来总结Apollo中的坐标变换关系，即静态坐标转换由”StaticTransform”模块提供，而动态转换由需要发布的模块自行发布如”NDTLocalization”，”RTKLocalization”和””Gnss，可以看到动态变换主要是世界坐标到本地坐标的转换，而静态转换主要是各个传感器之间的转换。最后转换关系统一由Buffer模块接收，并且提供查询。



19.6. Reference

- [tf2](http://wiki.ros.org/tf2) [<http://wiki.ros.org/tf2>]

20. V2X

业精于勤，荒于嬉；行成于思，毁于随。

20.1. v2x目录结构

v2x的目录结构如下。

```
.  
├── BUILD          // 编译  
├── common         // 公共目录  
├── conf           // 配置  
├── launch           
├── proto          // protobuf文件  
└── v2x_proxy      // v2x代理
```

主要的实现在”v2x_proxy”中，无人驾驶车主要是OBU单元，和路侧RSU单元进行交互。

20.2. v2x_proxy

v2x_proxy的目录结构如下。

```
.  
├── app  
├── obu_interface  
└── os_interface
```

20.2.1. app

v2x模块的入口函数在”app/main.cc”中，在主函数中读取参数并且初始化v2x proxy。在”v2x_proxy.h”和”v2x_proxy.cc”中实现了”V2xProxy”类。

1. 初始化 首先我们看”V2xProxy”的初始化过程。

```

V2xProxy::V2xProxy(std::shared_ptr<::apollo::hdmap::HDMap>
hdmap)
    : node_(::apollo::cyber::CreateNode("v2x_proxy")),
exit_(false) {
    internal_ = std::make_shared<InternalData>();
    // 1. 读取高精度地图
    hdmap_ = std::make_shared<::apollo::hdmap::HDMap>();
    const auto hdmap_file = apollo::hdmap::BaseMapFile();
    // 2.
    ::apollo::cyber::TimerOption v2x_car_status_timer_option;
    v2x_car_status_timer_option.period =
        static_cast<uint32_t>((1000 +
FLAGS_v2x_car_status_timer_frequency - 1) /
FLAGS_v2x_car_status_timer_frequency);
    v2x_car_status_timer_option.callback = [this]() {
        this->OnV2xCarStatusTimer();
    };
    v2x_car_status_timer_option.oneshot = false;
    v2x_car_status_timer_.reset(
        new
::apollo::cyber::Timer(v2x_car_status_timer_option));
    os_interface_.reset(new OsInterface());
    obu_interface_.reset(new ObuInterfaceGrpcImpl());
    recv_thread_.reset(new std::thread([this]() {
        while (!exit_.load()) {
            this->RecvTrafficlight();
        }
    }));
    planning_thread_.reset(new std::thread([this]() {
        while (!exit_.load()) {
            this->RecvOsPlanning();
        }
    }));
    obs_thread_.reset(new std::thread([this]() {
        while (!exit_.load()) {
            std::shared_ptr<::apollo::v2x::V2XObstacles> obs =
nullptr;
            this->obu_interface_->GetV2xObstaclesFromObu(&obs); // Blocked
            this->os_interface_->SendV2xObstacles2Sys(obs);
        }
    }));
}

```

```

    } );
v2x_car_status_timer_->Start();
// 从文件获取RSU列表
GetRsuListFromFile(FLAGS_rsu_whitelist_name, &rsu_list_);
init_flag_ = true;
}

```

可以看到”V2xProxy”初始化了2个定时器，一个是RSU发送给车的红绿灯信息，一个是主动上报的车辆状态信息，另外还初始化了os接口和obu接口。

Apollo 6.0中又增加了几种消息，并且通过定时器发布，下面我们来看V2xProxy中包含几种定时器。

1. v2x_car_status_timer_- OnV2xCarStatusTimer
2. obu_status_timer_-
3. rsu_whitelist_timer_-

几个线程

1. recv_thread_RcvTrafficlight
2. planning_thread_RcvOsPlanning
3. rsi_thread_ 目前没有使用
4. obs_thread_GetV2xObstaclesFromObu -> SendV2xObstacles2Sys

20.3. OnV2xCarStatusTimer

发送车的信息到OBU->RSU GetRsuInfo -> SendCarStatusToObu

20.4. RcvTrafficlight

接收OBU的红绿灯消息，并且进行处理

```

void V2xProxy::RcvTrafficlight() {
    // get traffic light from obu
    std::shared_ptr<ObuLight> x2v_traffic_light = nullptr;
    // 1. 从OBU获取红绿灯状态，并且发送给Apollo
    obu_interface-
}

```

```

>GetV2xTrafficLightFromObu(&x2v_traffic_light);
    os_interface_-
>SendV2xObuTrafficLightToOs(x2v_traffic_light);
    auto os_light = std::make_shared<OSLight>();
    std::string junction_id = "";
{
    std::lock_guard<std::mutex> lg(lock_hdmap_junction_id_);
    junction_id = hdmap_junction_id_;
}
bool res_success_ProcTrafficlight = internal_-
>ProcTrafficlight(
    hdmap_, x2v_traffic_light.get(), junction_id, u_turn_,
    FLAGS_traffic_light_distance, FLAGS_check_time,
&os_light);
if (!res_success_ProcTrafficlight) {
    return;
}
utils::UniqueOsLight(os_light.get());
// 3. 发送红绿灯消息到HMI???
os_interface_->SendV2xTrafficLightToOs(os_light);
// save for hmi
std::lock_guard<std::mutex> lock(lock_last_os_light_);
ts_last_os_light_ =
::apollo::cyber::Time::MonoTime().ToMicrosecond();
last_os_light_ = os_light;
}

```

20.5. RecvOsPlanning

获取planning路线,并且根据红绿灯的剩余时间来调整线路?

```

void V2xProxy::RecvOsPlanning() {
    auto adc_trajectory =
std::make_shared<::apollo::planning::ADCTrajectory>();
    auto res_light =

std::make_shared<::apollo::perception::TrafficLightDetection>()
;
    os_interface_->GetPlanningAdcFromOs(adc_trajectory);
    // OK get planning message
    std::shared_ptr<OSLight> last_os_light = nullptr;

```

```

{
    std::lock_guard<std::mutex> lock(lock_last_os_light_);

    auto now_us =
::apollo::cyber::Time::MonoTime().ToMicrosecond();
    if (last_os_light_ == nullptr ||
        2000LL * 1000 * 1000 < now_us - ts_last_os_light_) {
        AWARN << "V2X Traffic Light is too old!";
        last_os_light_ = nullptr;
    } else {
        ADEBUG << "V2X Traffic Light is on time.";
        last_os_light_ = std::make_shared<OSLight>();
        last_os_light->CopyFrom(*last_os_light_);
    }
}
// proc planning message
bool res_proc_planning_msg = internal_-
>ProcPlanningMessage(
    adc_trajectory.get(), last_os_light.get(), &res_light);
if (!res_proc_planning_msg) {
    return;
}
os_interface_->SendV2xTrafficLight4Hmi2Sys(res_light);
}

```

20.6. obs_thread

获取OBU发布的障碍物信息,然后发送到OS

20.6.1. TrafficLightTimer

交通灯的定时器会定时回调”OnX2vTrafficLightTimer”，下面我们看定时回调里面执行了什么？

```

void V2xProxy::OnX2vTrafficLightTimer() {
    x2v_trafficlight_->Clear();
    // 1. 从obu接口中获取红绿灯的状态
    obu_interface_-
>GetV2xTrafficLightFromObu(x2v_trafficlight_);
    if (!x2v_trafficlight_->has_current_lane_trafficlight()) {

```

```

    AERROR << "Error:v2x trafficlight ignore, no traffic
light contained.";
    return;
}
// 2. 当前红绿灯状态
auto current_traff = x2v_trafficlight_-
>mutable_current_lane_trafficlight();
if (current_traff->single_traffic_light().empty()) {
    AERROR << "Error:v2x trafficlight ignore, no traffic
light contained.";
    return;
}
ADEBUG << x2v_trafficlight_->DebugString();
// 3. 执行红绿灯逻辑
if (!TrafficLightProc(current_traff)) {
    return;
}
// 4. 发送红绿灯状态到os接口
os_interface_->SendV2xTrafficLightToOs(x2v_trafficlight_);
}

```

下面是红绿灯的处理过程。先根据接收到的坐标信息查找前面一定距离的所有红绿灯，然后把当前范围内的所有红绿灯改为接收到的颜色。该过程可能较少的考虑到一些逻辑，估计后面会继续完善。

```

bool V2xProxy::TrafficLightProc(CurrentLaneTrafficLight* msg)
{
    // 1. 获取rsu发送的红绿灯坐标
    apollo::common::PointENU point;
    point.set_x(msg->gps_x_m());
    point.set_y(msg->gps_y_m());
    std::vector<apollo::hdmap::SignalInfoConstPtr> signals;
    // 2. 获取当前范围内所有的红绿灯
    if (hdmap_->GetForwardNearestSignalsOnLane(point, 1000.0,
&signals) != 0) {
        AERROR << "Error::v2x trafficlight ignore, hdmap get no
signals";
        AERROR << "traffic light size : " << signals.size();
        return false;
    }
    // 3. 如果只有一个信号灯，则设置id，并且返回
    if (signals.size() == 1) {

```

```

        auto single = msg->mutable_single_traffic_light(0);
        single->set_id(signals[0]->id().id());
        return true;
    }
    // 4. 如果有多个信号灯，则把所有的信号灯设置为发送的颜色
    auto color = msg->single_traffic_light(0).color();
    msg->clear_single_traffic_light();

    for (auto i = signals.begin(); i != signals.end(); i++) {
        auto single = msg->add_single_traffic_light();
        single->set_id((*i)->id().id());
        single->set_color(color);
    }
    return true;
}

```

疑问：

1. 按照代码RSU只发送了一个信号灯状态，并且RSU没有高精度地图信息，不知道无人车中高精度地图的signal_id，所以一方面要填充signal_id信息，一方面找到多个红绿灯时候，需要把所有的红绿灯信息都修改为发送的状态。
2. RSU应该发送多个红绿灯的状态，而不仅仅只发送一个，这样就需要RSU侧也需要高精度地图，并且和车的高精度信息要同步。

20.6.2. OnV2xCarStatusTimer

发送本车的状态到RSU。

```

void V2xProxy::OnV2xCarStatusTimer() {
    v2x_carstatus_->Clear();
    auto localization = std::make_shared<LocalizationEstimate>();
    // 1. 获取本车的位置信息
    os_interface_->GetLocalizationFromOs(localization);
    if (!localization || !localization->has_header() ||
        !localization->has_pose()) {
        AERROR << "Error:localization ignore, no pose or header
in it.";
        return;
    }
}

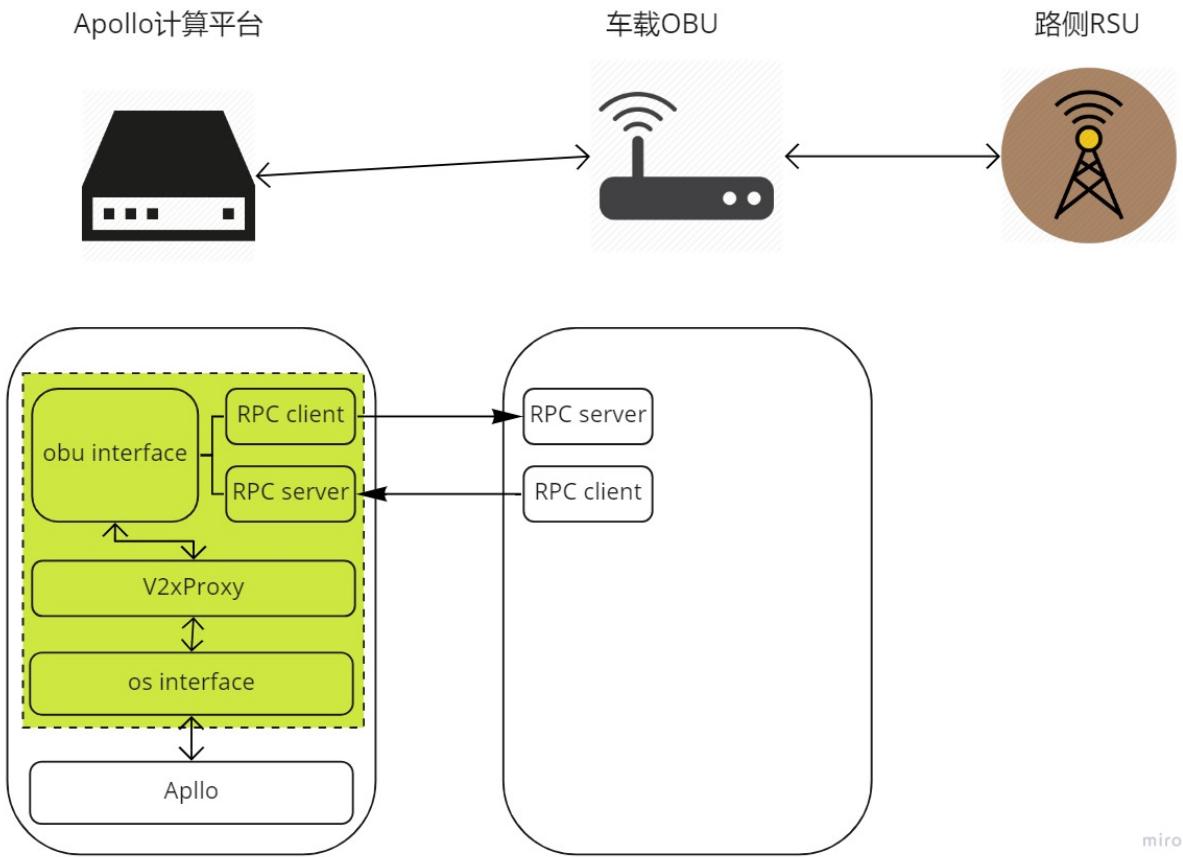
```

```

// 2. 发送当前的位置信息到OBU
v2x_carstatus_->mutable_localization()->CopyFrom(*localization);
obu_interface_->SendCarStatusToObu(v2x_carstatus_);
}

```

通过上述分析，我们可以清晰的了解到”V2xProxy”实际上相当于一个桥梁，通过”os_interface_”获取车的信息，通过”obu_interface_”发送消息。下面是流程图。



20.7. OBU接口(ObuInterfaceGrpcImpl)

OBUs are actually the bridge between cars and RSUs. Currently, OBUs can be separate devices connected via network, so here they are implemented using gRPC. The main function is to send and receive obstacle information and traffic light information. `ObuInterfaceBase` is a pure virtual class, defining the interface for communicating with OBUs.

```
class ObuInterFaceGrpcImpl : public ObuInterFaceBase {
public:
    ObuInterFaceGrpcImpl();
    ~ObuInterFaceGrpcImpl();
    // 1. 初始化grpc服务端
    bool InitialServer() override;
    // 2. 初始化grpc客户端
    bool InitialClient() override;

    // 3. 从OBU获取障碍物信息
    void GetV2xObstaclesFromObu(
        const std::shared_ptr<apollo::perception::PerceptionObstacles>
        &msg)
        override;
    // 4. 从OBU获取红绿灯信息
    void GetV2xTrafficLightFromObu(
        const std::shared_ptr<IntersectionTrafficLightData>
        &msg) override;

    // 5. 发送车的状态到OBU
    void SendCarStatusToObu(const std::shared_ptr<CarStatus>
        &msg) override;

    // 6. 发送障碍物信息到OBU
    void SendObstaclesToObu(
        const std::shared_ptr<apollo::perception::PerceptionObstacles>
        &msg)
        override;
};
```

ObuInterFaceGrpcImpl中创建了一个grpc客户端和服务端，服务端监听OBU发送过来的消息，并且保存。grpc客户端则发送消息到OBU。

20.7.1. 远程调用服务(grpc_interface)

主要实现了grpc的客户端和服务端，后面看下grpc的介绍之后再详细介绍。

1. 其中GrpcServerImpl提供rpc服务，当OBU发送请求获取障碍物信息时候，返回无人车感知到的障碍物信息，反之同理。（OBU提供请求）
2. GrpcClientImpl向OBU发出请求，获取红绿灯和障碍物信息。
(OBU提供grpc服务)

上述过程中无人车客户端会启动2个定时器，通过rpc客户端去获取OBU提供的红绿灯和障碍物信息，这里又回到之前的问题，为什么需要红绿灯做同步？如果无人车和OBU的检测不一致，那么理论上应该听谁的？

20.8. 系统接口(OsInterFace)

OsInterFace中实现了2个模板，分别接收和发布消息给apollo，下面我们主要看下发布和订阅函数。

20.8.1. SendMsgToOs

发布函数非常简单，就是通过writer发送指定的topic，需要注意一定要对RSU发布的消息做融合了之后才能输出，如果不做融合，一个简单的例子如果RSU发布的障碍物apollo没有看到，当RSU发布之后，Apollo的感知如果不做融合，下次发送的是apollo自己的感知结果，会出现一帧的障碍物存在，而下一帧不存在的情况，因此要对结果做融合。另一个疑问是如何保证融合的时间戳一致，因为RSU的频率可能和激光雷达的时间戳不一致。

```
template <typename MessageT>
void SendMsgToOs(cyber::Writer<MessageT> *writer,
                 const std::shared_ptr<MessageT> &msg) {
    if (writer == nullptr) {
        AERROR << "Writer is not valid";
        return;
    }
    // 1. 发送消息
    if (writer->Write(msg) == true) {
        ADEBUG << "Write msg success to: " << writer-
>GetChannelName();
```

```

    } else {
        AERROR << "Write msg failed to: " << writer-
>GetChannelName ();
    }
}

```

20.8.2. GetMsgFromOs

从Apollo系统接收消息，这部分的消息接收没有采用事件驱动的方式，而是采用定时发布的方式。

```

template <typename MessageT>
void GetMsgFromOs(const cyber::Reader<MessageT> *reader,
                  const std::shared_ptr<MessageT> &msg) {
    node_->Observe();
    if (reader->Empty()) {
        AINFO_EVERY(100) << "Has not received any data from "
                           << reader->GetChannelName ();
        return;
    }
    msg->CopyFrom(* (reader->GetLatestObserved()));
}

```

20.9. 感知模块

最后感知模块”trafficlights_perception_component.cc”会订阅”/apollo/v2x/traffic_light”这个TOPIC，然后把V2X获取到的结果放入buffer中再进行处理。

```

int TrafficLightsPerceptionComponent::InitV2XListener() {
    typedef const
    std::shared_ptr<apollo::v2x::IntersectionTrafficLightData>
        V2XTrafficLightsMsgType;
    std::function<void(const V2XTrafficLightsMsgType&)>
    sub_v2x_tl_callback =
        std::bind(&TrafficLightsPerceptionComponent::OnReceiveV2XMsg,
this,
                  std::placeholders::_1);
    auto sub_v2x_reader = node_->CreateReader(

```

```

        v2x_trafficlights_input_channel_name_,
sub_v2x_t1_callback);
    return cyber::SUCC;
}

```

20.10. fusion模块

Apollo6.0在v2x中新增加了fusion模块，fusion模块的输入是”/perception/vehicle/obstacles”，输出也是”/apollo/perception/obstacles”.接收的是感知模块输出的感知信息,输出融合之后的障碍物信息.这个模块的启动也在感知模块的”dag_streaming_perception.dag”中,也就是说V2X模块的感知融合可能会合入感知模块中.

输入: /perception/vehicle/obstacles /apollo/v2x/obstacles
/apollo/localization/pose

输出: /apollo/perception/obstacles

V2XFusionComponent模块的处理过程主要在”V2XMessageFusionProcess”中.

```

bool V2XFusionComponent::V2XMessageFusionProcess(
    const std::shared_ptr<PerceptionObstacles>&
perception_obstacles) {
    // 1. 读取最新的位置
    localization_reader_->Observe();
    auto localization_msg = localization_reader_-
>GetLatestObserved();
    base::Object hv_obj;
    CarstatusPb2Object(*localization_msg, &hv_obj, "VEHICLE");

    v2x_obstacles_reader_->Observe();
    auto v2x_obstacles_msg = v2x_obstacles_reader_-
>GetLatestObserved();
    // 2. 读取v2x的感知信息,如果没有,则直接输出感知模块的结果
    if (v2x_obstacles_msg == nullptr) {
        AERROR << "V2X: cannot receive any v2x obstacles
message.";
        perception_fusion_obstacles_writer_-

```

```

>Write(*perception_obstacles);
} else {
    header_.CopyFrom(perception_obstacles->header());
    std::vector<Object> fused_objects;
    std::vector<Object> v2x_fused_objects;
    std::vector<std::vector<Object>> fusion_result;
    std::vector<Object> v2x_objects;
    // 3. 转换v2x感知消息为对象,转换感知模块消息为对象
    V2xPbs2Objects(*v2x_obstacles_msg, &v2x_objects, "V2X");
    std::vector<Object> perception_objects;
    Pbs2Objects(*perception_obstacles, &perception_objects,
    "VEHICLE");
    perception_objects.push_back(hv_obj);

    // 4. 合并新资源
    fusion_.CombineNewResource(perception_objects,
    &fused_objects,
                           &fusion_result);
    fusion_.CombineNewResource(v2x_objects, &fused_objects,
    &fusion_result);
    // 5. 获取v2x融合对象
    fusion_.GetV2xFusionObjects(fusion_result,
    &v2x_fused_objects);
    // 6. 发送消息
    auto output_msg = std::make_shared<PerceptionObstacles>
();
    SerializeMsg(v2x_fused_objects, output_msg);
    perception_fusion_obstacles_writer_->Write(*output_msg);
}
return true;
}

```

20.11. Fusion类

Fusion类的构造函数.

```

Fusion::Fusion() {
    ft_config_manager_ptr_ = FTConfigManager::Instance();
    // 读取score params, 参数在"fusion_params.pt"中
    score_params_ = ft_config_manager_ptr_-
>fusion_params_.params.score_params();

```

```

switch (score_params_.confidence_level()) {
    case fusion::ConfidenceLevel::C90P:
        m_matched_dis_limit_ = std::sqrt(4.605);
        break;
    case fusion::ConfidenceLevel::C95P:
        m_matched_dis_limit_ = std::sqrt(5.991);
        break;
    case fusion::ConfidenceLevel::C975P:
        m_matched_dis_limit_ = std::sqrt(7.378);
        break;
    case fusion::ConfidenceLevel::C99P:
        m_matched_dis_limit_ = std::sqrt(9.210);
        break;
    default:
        break;
}
}

```

fusion_params.pt中的参数是,可以看到”confidence_level”为C99P则m_matched_dis_limit_为”std::sqrt(9.210)”。

```

score_params {
    prob_scale: 0.125
    max_match_distance: 10
    min_score: 0
    use_mahalanobis_distance: true
    check_type: false
    confidence_level: C99P
}

```

那么我们看下CombineNewResource的实现.

```

bool Fusion::CombineNewResource(
    const std::vector<base::Object> &new_objects,
    std::vector<base::Object> *fused_objects,
    std::vector<std::vector<base::Object>> *fusion_result) {
    // 1. 如果fused_objects为空,则直接添加
    if (fused_objects->size() < 1) {
        fused_objects->assign(new_objects.begin(),
        new_objects.end());
        for (unsigned int j = 0; j < new_objects.size(); ++j) {
            std::vector<base::Object> matched_objects;

```

```

        matched_objects.push_back(new_objects[j]);
        fusion_result->push_back(matched_objects);
    }
    return true;
}
int u_num = fused_objects->size();
int v_num = new_objects.size();
Eigen::MatrixXf association_mat(u_num, v_num);
// 2. 计算关联矩阵
ComputeAssociateMatrix(*fused_objects, new_objects,
&association_mat);
std::vector<std::pair<int, int>> match_cps;
// 3. 采用km_matcher_进行匹配
if (u_num > v_num) {
    km_matcher_.GetKMResult(association_mat.transpose(),
&match_cps, true);
} else {
    km_matcher_.GetKMResult(association_mat, &match_cps,
false);
}
// 4. 融合结果
for (auto it = match_cps.begin(); it != match_cps.end();
it++) {
    if (it->second != -1) {
        if (it->first == -1) {
            fused_objects->push_back(new_objects[it->second]);
            std::vector<base::Object> matched_objects;
            matched_objects.push_back(fused_objects->back());
            fusion_result->push_back(matched_objects);
        } else {
            (*fusion_result)[it->first].push_back(new_objects[it-
>second]);
        }
    }
}
return true;
}

```

计算关联矩阵,先计算距离分数,再计算类型分数

```

bool Fusion::ComputeAssociateMatrix(
    const std::vector<base::Object> &in1_objects, // fused
    const std::vector<base::Object> &in2_objects, // new

```

```

Eigen::MatrixXf *association_mat) {
for (unsigned int i = 0; i < in1_objects.size(); ++i) {
    for (unsigned int j = 0; j < in2_objects.size(); ++j) {
        const base::Object &obj1_ptr = in1_objects[i];
        const base::Object &obj2_ptr = in2_objects[j];
        double score = 0;
        // 1. 计算距离分数
        if (!CheckDissScore(obj1_ptr, obj2_ptr, &score)) {
            AERROR << "V2X Fusion: check dis score failed";
        }
        // 2. 计算类型分数，采用距离分数乘以类型系数
        if (score_params_.check_type() &&
            !CheckTypeScore(obj1_ptr, obj2_ptr, &score)) {
            AERROR << "V2X Fusion: check type failed";
        }
        (*association_mat)(i, j) =
            (score >= score_params_.min_score()) ? score : 0;
    }
}
return true;
}

```

20.12. KMkernal

KM匹配算法.

20.13. trans_tools

“trans_tools.cc”和“trans_tools.h”中主要是一些工具类,用来转换对象到proto和从proto到对象.

```

void Objects2Pbs(const std::vector<base::Object> &objects,
                  std::shared_ptr<PerceptionObstacles>
                  obstacles) {
    obstacles->mutable_perception_obstacle()->Clear();
    if (objects.size() < 1) {
        return;
    }
    // obstacles->mutable_header()->set_frame_id(objects[0].frame_id);

```

```
for (const auto &object : objects) {
    if (object.v2x_type == base::V2xType::HOST_VEHICLE) {
        continue;
    }
    PerceptionObstacle obstacle;
    Object2Pb(object, &obstacle);
    obstacles->add_perception_obstacle()->CopyFrom(obstacle);
}
```

20.14. V2x消息

2017年9月中旬，中国智能网联汽车产业创新联盟正式发布《合作式智能交通系统车用通信系统应用层及应用数据交互标准》。该标准是一个应用层的标准，[下载链接](http://www.sae-china.org/download/1745/%E5%90%88%E4%BD%9C%E5%BC%8F%E6%99%BA%E8%83%BD%E8%BF%90%E8%BE%93%E7%B3%BB%E7%BB%9F%+E8%BD%A6%E7%94%A8%E9%80%9A%E4%BF%A1%E7%B3%BB%E7%BB%9F%E5%BA%94%E7%94%A8%E5%B1%82%E5%8F%8A%E5%BA%94%E7%94%A8%E6%95%B0%E6%8D%AE%E4%BA%A4%E4%BA%92%E6%A0%87%E5%87%86.pdf) [<http://www.sae-china.org/download/1745/%E5%90%88%E4%BD%9C%E5%BC%8F%E6%99%BA%E8%83%BD%E8%BF%90%E8%BE%93%E7%B3%BB%E7%BB%9F%+E8%BD%A6%E7%94%A8%E9%80%9A%E4%BF%A1%E7%B3%BB%E7%BB%9F%E5%BA%94%E7%94%A8%E5%B1%82%E5%8F%8A%E5%BA%94%E7%94%A8%E6%95%B0%E6%8D%AE%E4%BA%A4%E4%BA%92%E6%A0%87%E5%87%86.pdf>].

标准中规定了5大类消息：

- BSM - basic safety message
 - MAP - map data
 - RSM - road side safety message
 - SPAT - signal phase and timing message
 - RSI - road side information

目前Apollo中实现了RSI,SPAT,MAP 3种消息格式, RSM和BSM消息没有定义.Apollo中的BSM可能可以对应到CarStatus消息.每种消息的格式以及意义消息中都进行了明确的定义,并且对一些应用应该发什么消息,消息的频率和交互流程也做了定义.因此可以参考完成一些应用.由于网联车辆还是一个比较新的领域,里面的一些流程可能不一定能够完全照搬,所以应该参考消息的用意,而具体的流程可以适当做一些修改.

21. 性能分析

四方上下曰宇，往古来今曰宙。

我们主要从以下几个方面来优化我们的系统，使得系统更加稳定，节省资源，同时能够又能保证任务的实时性。以下几个技术都是目前Apollo Cyber中采用的技术。

21.1. 线程调度

由于linux操作系统提供了控制线程的API接口，cyber通过系统提供的API对进程的优先级和使用的资源进行调节。首先cyber将要求比较高的进程设置为实时进程，linux操作系统中实时进程的优先级最高，实时进程可以抢占其它线程，好处是能够保证实时进程在一定的时间内返回结果，这在自动驾驶控制系统中非常关键，试想一下如果需要发送一条指令给汽车，系统没有在规定的时间内响应，或者响应有延迟，就有可能导致车祸。linux对实时进程的调度有2种方式：

1. SCHED_FIFO - 先到的进程优先执行，后到的进程需要等之前的进程执行完成之后再开始执行。
2. SCHED_RR - 基于时间片轮转，先到的进程执行完成之后放到队列尾部，在队列中循环执行。

基于FIFO方式的平均等待时间和进程的顺序有关系，如果先到的进程执行时间很长，那么后到的进程等待时间就会变长；如果先到进程的执行时间很短，那么后到进程的等待时间就会变短。当然基于时间片轮转的方式就没有这个缺点，但是先到进程的执行时间会长，因为基于轮转的，需要循环队列执行，那么先到进程需要等待其它进程的执行。所以需要根据不同的场景来选择不同的调度策略。

21.2. Cgroups

cgroups，名称源自控制组群（control groups）的简写，是Linux内核的一个功能，用来限制、控制与分离一个进程组群的资源（如CPU、内存、磁盘输入输出等）。cgroups的一个设计目标是为不同的应用情况提供统一的接口，从控制单一进程（像nice）到操作系统层虚拟化（像OpenVZ, Linux-VServer, LXC）。cgroups提供：

- 资源限制：组可以被设置不超过设定的内存限制；这也包括虚拟内存。
- 优先级：一些组可能会得到大量的CPU或磁盘IO吞吐量。
- 结算：用来衡量系统确实把多少资源用到适合的目的上。
- 控制：冻结组或检查点和重启动。

利用cgroups技术，我们可以设置这一组进程的优先级，并且根据重要程度和进程类型分配不同的资源。例如给重要的进程组分配更多的CPU和内存，限制其他进程组的CPU和内存防止其影响系统性能等。

21.3. CPU亲和性

CPU亲和性又叫Processor affinity或CPU pinning。现在的CPU都是多核心的，比如Apollo推荐的计算单元就是4核8线程，多核心CPU的好处是可以同时执行多个任务。现在假设多核CPU有以下场景，一个核上的任务很多，而另外的核心都是空闲状态，那么就会出现一个核累死，而其他的核都在等待的状态，这时候操作系统就想到了一种技术来解决这个问题，即CPU的负载均衡，当一个CPU核心上的任务很多，而其他CPU是空闲状态的时候，操作系统会把这个核上的任务迁移到其他核心，这样多核CPU的利用率就上来了。对整个系统来说，CPU负载均衡是一个好技术，但是对单个线程来说，就不是那么好了。线程迁移会导致额外的开销，比如当前的CACHE需要重新刷新，而且把重要的任务绑定到单独的核心上，可以保证这个任务的高效执行而不被打断。linux操作系统中通过”sched_setaffinity” API来设置线程的CPU亲和性，通过”sched_getaffinity”来获取线程的CPU亲和性。

```
#define __GNU_SOURCE          /* See feature_test_macros(7)
 */
#include <sched.h>
```

```
int sched_setaffinity(pid_t pid, size_t cpusetsize,
                      cpu_set_t *mask);

int sched_getaffinity(pid_t pid, size_t cpusetsize,
                      cpu_set_t *mask);
```

21.4. 中断绑定

中断绑定又叫smp_affinity，通过”cat /proc/interrupts”可以列出系统中每个I/O设备中每个CPU的中断数，处理的中断数，中断类型，以及注册为接收中断的驱动程序列表。系统通过”smp_affinity”可以指定多核CPU是否会响应这个中断，这在频繁有中断的系统中相当有用，比如CAN总线会频繁通过中断来传递传感器消息，如果没有绑定中断，那么系统中每个核心都可能被打断，如果这个核心上有任务在运行，那么CPU就会打断当前任务的执行，而去处理中断程序，从而带来中断上下文切换开销。如果我们把中断绑定到一个单独的核心上，让这个CPU核心去处理中断，而其它CPU核心则不会被频繁打断。

smp_affinity 的默认值为 f，即可为系统中任意 CPU 提供 IRQ。将这个值设定为 1，如下，即表示只有 CPU 0 可以提供这个中断：

```
# echo 1 >/proc/irq/32/smp_affinity
# cat /proc/irq/32/smp_affinity
1
```

21.5. linux性能优化

linux操作系统的”perf”命令可以采样一段时间内的系统调用，保存成文件之后再结合火焰图，可以查看当前系统各个进程对cpu的使用情况，火焰图中的横轴代表了CPU占用时间的比例，宽度越宽，代表该进程越耗时。火焰图的横轴是当前进程的调用栈，可以逐级查看每个调用栈和具体的耗时。

21.5.1. perf安装

ubuntu下执行如下命令安装perf:

```
apt-get install linux-tools-common linux-tools-generic linux-tools-`uname -r`
```

21.5.2. 火焰图

安装成功之后可以执行”perf”命令来采样系统进程调用：

```
// 先找到需要统计的apollo进程  
sudo ps -ef | grep apollo  
  
// 查看到进行号之后，用进程号替换下面的PID，进行采样，采样频率为99HZ，采  
样时间为120秒  
sudo perf record -F 99 -p PID -g -- sleep 120  
  
// 输出perf文件  
sudo perf script > out.perf
```

上述步骤就完成了对apollo进程的采样，并且输出了采样文件，下面我们通过生成火焰图来分析进程的调用状况。火焰图采用开源工具”FlameGraph”，执行如下命令：

```
// 下载FlameGraph项目  
git clone --depth 1  
https://github.com/brendangregg/FlameGraph.git  
  
// 折叠调用栈  
FlameGraph/stackcollapse-perf.pl out.perf > out.folded  
  
// 生成火焰图  
FlameGraph/flamegraph.pl out.folded > out.svg
```

最后把生成的”out.svg”文件在浏览器中打开，就可以点击并且查看对应的调用时间和调用栈，来分析系统耗时。火焰图如下，源文件在github上下载： [https://github.com/daohu527/Dig-into-Apollo/blob/master/performance/Gregg4.svg]

图片引用自阮一峰《如何读懂火焰图？》

21.6. Reference

[cgroups](https://zh.wikipedia.org/wiki/Cgroups) [https://zh.wikipedia.org/wiki/Cgroups] [Processor affinity](#)
[https://en.wikipedia.org/wiki/Processor_affinity] [sched_setaffinity](#)
[https://linux.die.net/man/2/sched_setaffinity] [SMP IRQ affinity](#)
[https://www.kernel.org/doc/Documentation/IRQ-affinity.txt] [4.3. 中断和 IRQ 调节](#)
[https://access.redhat.com/documentation/zh-cn/red_hat_enterprise_linux/6/html/performance_tuning_guide/s-cpu-irq] [perf](#)
[http://www.brendangregg.com/perf.html] [使用Perf和火焰图分析CPU性能](#)
[http://senlinzhan.github.io/2018/03/18/perf/] [如何读懂火焰图？](#)
[http://www.ruanyifeng.com/blog/2017/09/flame-graph.html]

22. 仿真

古之学者必有师。师者，所以传道受业解惑也。

22.1. 为什么需要仿真



1. 想象一下当你发现了一个新的算法，但还不确认它是否有效，你是否会直接找一辆自动驾驶汽车，更新软件，并且进行测试呢？这样做可能并不安全，你必须把所有的场景测试一遍以保证它足够好，这可需要大量的时间。仿真的好处显而易见，它通过软件模拟来发现和复现问题，而不需要真实的环境和硬件，可以极大的节省成本和时间。
2. 随着现在深度学习的兴起，仿真在自动驾驶领域有了新的用武之地。自动驾驶平台通过仿真采集数据，可以把训练时间大大提高，远远超出路测的时间，加快模型迭代速度。先利用集群训练模型，然后再实际的路测中去检验，采用数据驱动的方式来研究自动驾驶。

自动驾驶的仿真的论文可以参考英伟达的[End to End Learning for Self-Driving Cars](https://arxiv.org/abs/1604.07316) [<https://arxiv.org/abs/1604.07316>]，主要的目的是通过软件来模拟车以及车所在的环境，实现自动驾驶的集成测试，训练模型，模拟事故发生现场等功能。那么我们是如何模拟车所在的环境的呢？

22.2. 如何仿真

要模拟车所在的环境，就得把真实世界投影到虚拟世界，并且需要构造真实世界的物理规律。例如需要模拟真实世界的房子，车，树木，道路，红绿灯，不仅需要大小一致，还需要能够模拟真实世界的物理规律，比如树和云层会遮挡住阳光，房子或者障碍物会阻挡你的前进，车启动和停止的时候会有加减速曲线。总之，这个虚拟世界得满足真实世界的物理规律才足够真实，模拟才足够好。而这些场景恰恰和游戏很像，游戏就是模拟真实世界，并且展示出来，游戏做的越好，模拟的也就越真实。实现这一切的就是游戏引擎，通过游戏引擎模拟自然界的各种物理规律，可以让游戏世界和真实世界差不多。这也是越来越多的人沉迷游戏的原因，因为有的时候根本分不清是真实世界还是游戏世界。现在我们找到了一条捷径，用游戏来模拟自动驾驶，这看起来是一条可行的路，我们把自动驾驶中的场景复制到游戏世界，然后模拟自动驾驶中各种传感器采集游戏世界中的数据，看起来我们就像在真实世界中开着自动驾驶汽车在测试了。

22.2.1. 仿真软件

我们已经知道可以用游戏来模拟自动驾驶，而现在大家也都是这么做的，目前主流的仿真软件都是根据游戏引擎来开发，下面是主要的几个仿真软件：

仿真软件 引擎 介绍

[Udacity](#) Unity 优达学城的自动驾驶仿真平台

[Carla](#) Unreal4 Intel和丰田合作的自动驾驶仿真平台

[AirSim](#) Unreal4 微软的仿真平台，还可以用于无人机

仿真软件 引擎 介绍

[lgsvl](#) Unity LG的自动驾驶仿真平台

[Apollo](#) Dreamview百度的自动驾驶仿真平台

- **Unreal4** - 主要的编程方式是c++, 源码完全开源, 还可以通过蓝图来编程。比较著名的游戏有: 《鬼泣5》 《绝地求生: 刺激战场》
- **Unity** - 主要的编程方式是c#和脚本, 源码不开放, 超过盈利上限收费。比较著名的游戏有: 《王者荣耀》 《炉石传说》

22.2.2. 工作方式

那么仿真软件是如何工作的呢? 大部分的仿真软件分为2部分: server端和client端。

- server端主要就是游戏引擎, 提供模拟真实世界的传感器数据, 并且提供控制车辆, 红绿灯以及行人的接口, 还提供一些辅助接口, 例如改变天气状况, 检测车辆是否有碰撞等。
- client端则根据server端返回的传感器数据进行具体的控制, 调整参数等。

可以认为server就是游戏机, 而client则是游戏手柄, 根据游戏中的情况, 选择适当的控制方式, 直到游戏通关。

22.2.3. 工作原理

我们知道游戏引擎模拟了传感器的数据, 那么游戏引擎是如何实现模拟真实世界中的传感器数据的呢?

- 摄像头深度信息
- 摄像头场景分割

- 摄像头长短焦
- Lidar点云
- radar毫米波
- Gps信息

除了传感器数据，还需要模拟真实世界的物理规律：

- 碰撞检测
- 光线和天气变化
- 汽车动力学模型

TODO: 补充原理

下面分析下carla中如何实现上述的模拟，其实也可以看做Unreal4中如何实现上述功能，carla传感器的实现
在”carla/Unreal/CarlaUE4/Plugins/Carla/Source/Carla/Sensor”中。

1. 其中摄像头深度信息是通过投影”carla/Unreal/CarlaUE4/Plugins/Carla/Content/PostProcessingMaterials”中的材质实现的，这里有点疑惑就是难道深度信息是实现就生成的，还是说材质类似做一层滤镜的操作？
2. 而Lidar是通过Raycast来实现的，即发送射线检测距离。主要的疑问是如何模拟点云的角度，参数等信息？
3. 天气的变化直接是通过”蓝图”实现的，没有找到具体的地方？
4. 汽车动力学模型暂时也没有找到地方？

22.3. 如何使用

22.3.1. 桥接器

如果是单独实现或者测试一个算法，直接拿写好的算法在仿真软件上进行测试就可以了，但是如果是需要测试已经开发好的软件，比如apollo和autoware系统，则需要实现仿真软件和自动驾驶系统的对接。一个简单的想法就是增加一个桥接器，就像手机充电器的转换头一样，通过桥接器来连接仿真软件和自动驾驶系统。目前carla和lgsvl都

实现了通过桥接器和自动驾驶系统的对接，可以直接通过仿真软件来测试自动驾驶系统。

目前carla和lgsvl都是单独把apollo和autoware拉了一个分支，然后在其中集成一个适配器(ROS桥接)，来实现仿真软件和自动驾驶系统的对接。当然apollo3.5切换到cyber框架之后，可以通过cyber桥接来实现。

22.3.2. 制作地图

仿真中另外一个问题经常遇到的问题就是制作地图，以上的仿真软件都提供了地图编辑器来构建自己想要测试的地图。目前地图格式主要采用的是OpenDrive格式的地图，如果是和Apollo集成的化，需要把OpenDrive格式的地图转换为Apollo中能够使用的地图格式。现在的主要问题是地图编辑器不是那么好用，大部分好用的地图编辑软件都需要收费。

22.3.3. 测试场景

根据我们的测试需求，我们可以构建以下几种测试场景：



1.场景复现



2.集成测试



3.训练模型

miro

- **场景复现** - 假如自动驾驶过程中出现了一次接管（自动驾驶遇到突发状况解决不了，被人类驾驶员接管），首先我们需要复现当时的场景，这时候不可能再重新回去构建相同的场景，这时候就需要仿真去模拟当时的场景，找到问题之后，我们也可以通过仿真来看针对上述接管的情况是否解决。仿真通过模拟当时接管的场景，可以复现当时出现的问题，同时判断修改软件之后是否对当时的场景有所改善。

- **集成测试** - 每次开发一个新的功能和迭代之后，通过仿真构造全部场景的测试用例，例如：红绿灯，超车，停车，左拐弯，右拐弯，掉头，十字路口等情况来测试所有场景是否都没有问题，可以在软件真正上车测试之前保证软件的可靠性，检验新开发的功能，提高软件质量，减少测试成本。
- **训练模型** - 通过仿真软件来生成数据训练模型，真实场景的数据采集需要大量的车和时间，而软件可以通过分布式部署就可以实现模拟真实场景的大量数据，特别是针对目前感知的深度学习算法需要大量数据训练的情况，所以通过仿真可以加快模型训练和部署的速度。另外斯坦福大学还通过仿真来模拟汽车失控的情况下，尽量避免碰撞的场景，做一些新的研究和尝试。

22.3.4. 功能多样化

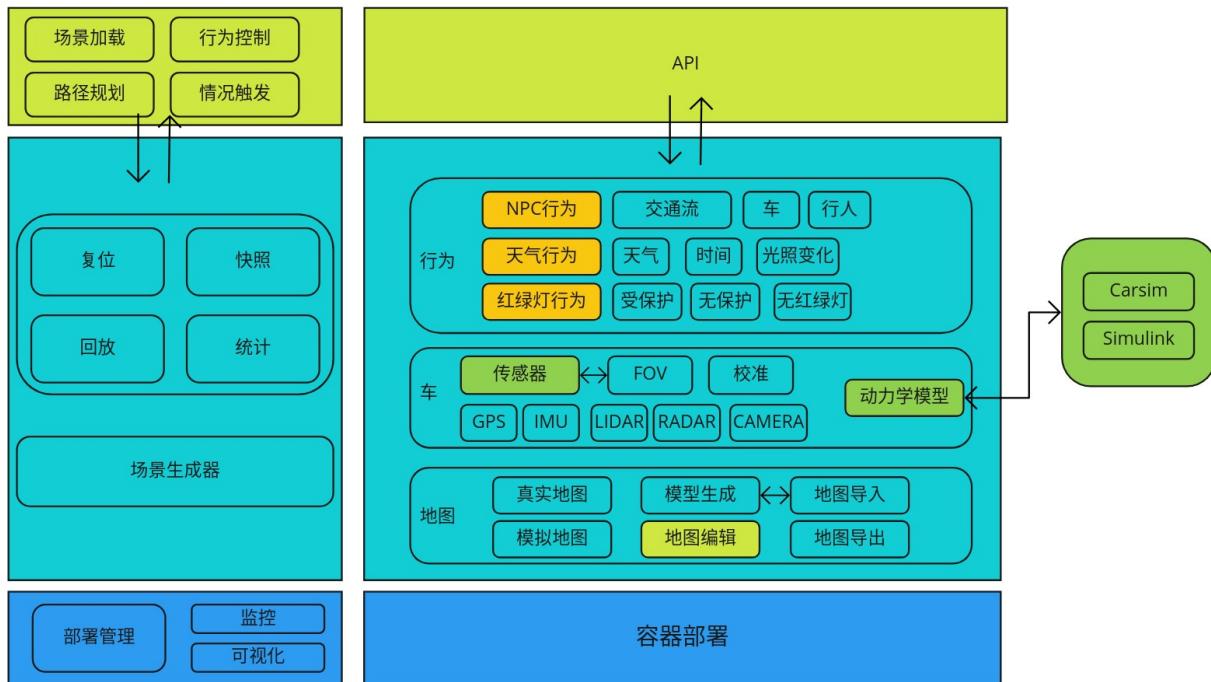
我们需要仿真软件能够适应不同的测试场景，就必须要求仿真软件能够提供灵活和多样化的功能，我们要提供哪些功能呢？

- **多机控制** - 不仅可以控制自己，还可以控制游戏中的其他角色。控制多辆车在一个地图里面跑，好处是可以多辆车竞争，有点类似遗传算法，把一些车放到里面跑，然后其中选出最好的，如此往复，得到最好的模型。同时多机控制还可以帮助我们控制游戏中的其他车辆，构建不同的测试场景，比如：行人横穿马路，超车等情况。
- **传感器参数调整** - 通过调整传感器参数实现不同硬件配置下的自动驾驶模拟，例如调整摄像头的参数，调整激光雷达的位置等，增加了传感器的灵活性。
- **汽车模型** - 根据需要导入不同的汽车模型，包括卡车，三轮车，小汽车的3D模型和动力学模型。
- **地图模型** - 如果纯手工制作模型太难了，是否可以根据3D点云的数据，然后根据软件来虚拟生成道路模型。

22.4. 如何构建自动驾驶仿真系统？

仿真最主要的目的：通过模拟真实环境和构建汽车模型，找出自动驾驶过程中可能出现的问题。那么如何构建自动驾驶仿真系统

呢？目前主流的实现方式是通过游戏引擎来模拟真实环境，通过**CarSim**等软件构建汽车的动力学模型来实现自动驾驶仿真。下面我们先看下自动驾驶仿真系统的整体结构。



注：仿真就是汽车动力学模型和环境模拟

miro

我们需要自动驾驶仿真系统满足：

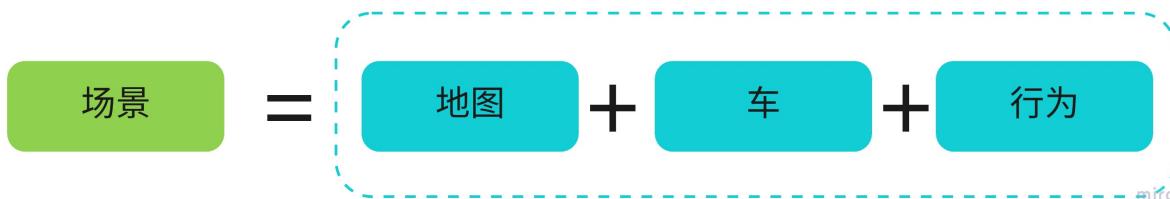
1. 场景丰富
2. 接口灵活
3. 恢复快速
4. 部署方便

首先我们关注仿真器本身，仿真器无非是模拟支持各种场景，其中场景分为：可以定义的场景和随机场景。可以定义的场景又分为：单元场景和真实场景。下面我们分别介绍下这几种场景：

- 可定义的场景 - 主要是针对驾驶过程中遇到的不同情况，比如会车，超车，红绿灯，变道等，这些场景一般都比较简单，类似于单元测试，主要是测试单个场景是否能够满足要求，这一部分业界已经有规范，可以参考[openscenario](http://www.openscenario.org/) [<http://www.openscenario.org/>]。拿超车的场景举例子，可以创建一辆NPC车辆在本车的前面，在不同的速度和距离条件下，测试本车超车是否成功。

- **真实场景** - 复现真实场景中遇到的问题，比如真实路测过程中遇到问题，需要复现当时的情况，并且验证问题是否已经解决，可以回放真实场景的数据来进行测试。
- **随机场景** - 这种场景类似于路测，模拟真实环境中的地图，并且随机生成NPC，天气，交通情况等，模拟汽车在虚拟的环境中进行路测，由于可以大规模部署，可以快速的发现问题。

我们可以看到不管是哪个场景，都是”地图+车+行为”的模式，场景的需求复杂多变，因此能够灵活的加载地图，车和行为就成为仿真器易用的关键。



我们的需求是能够根据不同的要求创建不同的场景，动态的添加地图，车和行为。场景生成器是一个框架，支持通过不同的配置，动态创建不同的场景，来满足我们的要求。除了场景生成器，我们还需要仿真器具备以下几个基本功能：

- **复位** - 在故障发生之后，我们能够复位环境和车辆到初始状态，同时也要求我们能够复位对应的自动驾驶系统。这样再每次故障后，可以不用人工操作，而自动恢复测试。
- **快照** - 能够生成对应帧的信息，保存快照的好处是能够恢复事故现场，同时也可用于自动驾驶数据集的建设。保存的点云和图片由于有groundtruth，可以作为机器学习的输入来训练模型。
- **回放** - 回放功能主要是用于故障定位，在发生碰撞之后，回放信息用于定位问题。
- **统计** - 统计主要是用于作为benchmark，来衡量系统的稳定性。

有了这些基础功能还不够，我们还需要关心具体的场景，下面我们分别对地图、车以及行为来详细描述需要实现的具体功能：

22.5. 地图

地图是场景中第一个需要考虑的，地图包括2部分，其中一部分是游戏中的模型，另外一部分是这些模型的高精度地图。换一种说法就是，首先我们需要在游戏中构建一个1:1的虚拟世界，然后再绘制出这个世界的高精度地图。其实游戏中的模型是游戏引擎的需求，游戏引擎是根据模型来渲染游戏画面的，没有模型也就渲染不出地图。而高精度地图是自动驾驶系统所需要的，高精度地图可以采用根据现场绘制的地图，也可以先得到游戏模型，然后在模型中绘制。下面是游戏中的地图和高精度地图的对应关系。



22.6. 真实场景地图生成

22.6.1. 地图模型制作

游戏中地图模型的制作相对来说是工作量比较大的工作，涉及到以下2点：

- **单个模型制作** - 单个模型包括地图中的建筑物、道路、树木、信号灯、交通牌、以及其他的信息。这些信息如果是要完全模拟真实环境，需要大量的材质和贴图，一般是在maya和3d-max等软件中建模，然后再导入模型到游戏引擎中使用。

- 地图布局 - 有了单个模型，当需要把单个模型组合成地图的时候，首先需要解决的是道路的位置信息，比如这个道路有多长，道路的曲率是多少？比较简单点的方法是直接导入2维地图（百度，高德，OSM），然后对照着2维地图放模型，最后生成整个地图的布局。而实际的问题是2维地图的精度往往达不到要求，国内的地图还加入了GPS偏置，所以生成的地图布局必定会不太准确。

22.6.2. 高精度地图制作

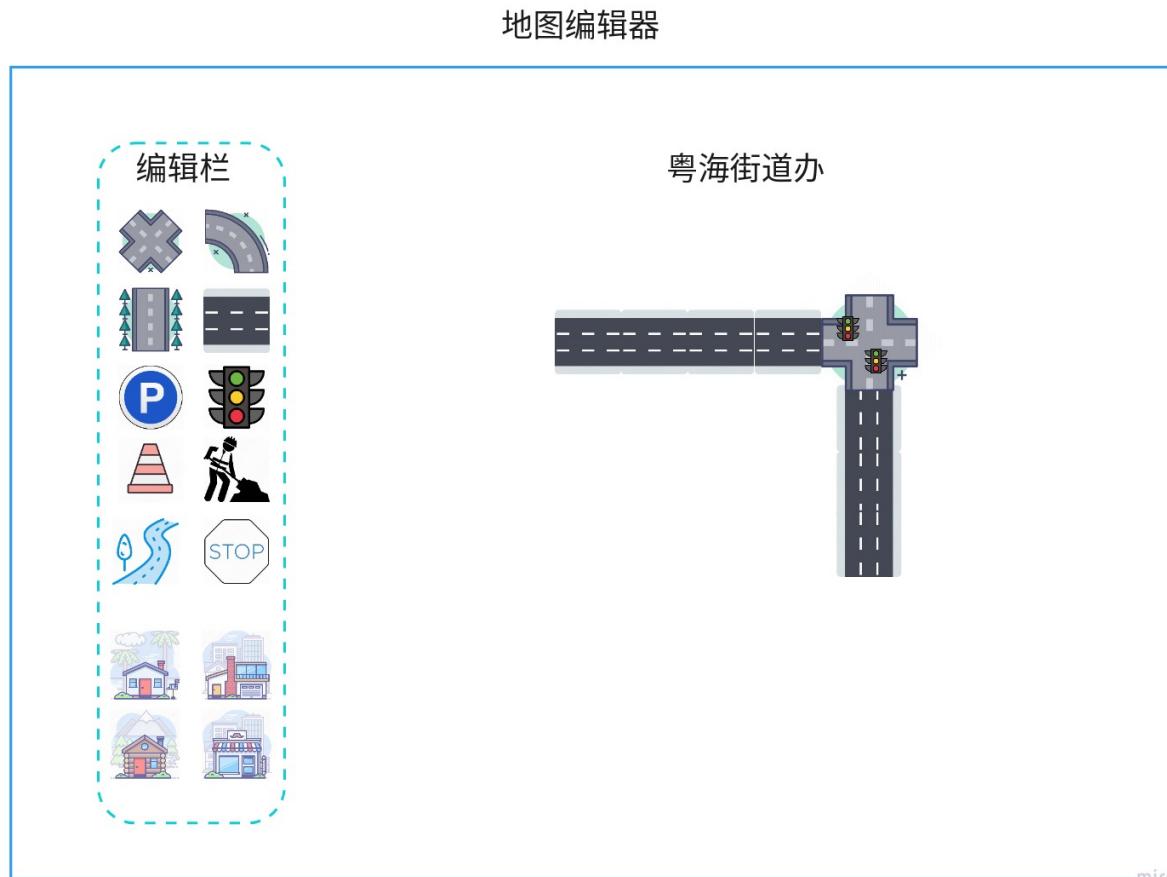
- 根据模型生成地图 - 接着上面的地图布局来讲，虽然得到的地图布局不准确，但是我们再根据游戏中的模型布局，绘制出高精度地图，然后把这个高精度地图给自动驾驶系统使用，基本上也能满足我们的要求。
- 根据地图生成模型 - 上述的问题就是游戏中的真实位置和实际道路的位置有轻微的误差。要解决上面的问题，我们可以反其道而行之，先生成高精度地图，即根据真实环境先绘制出高精度地图，然后再把高精度地图导入游戏引擎，动态的生成模型，这个方案的好处是地图100%是真实场景，而且不需要在游戏引擎中重新绘制高精度地图，坏处是建筑的模型无法生成。

关于真实场景的地图生成，目前还没有一个比较完美的解决方案，都需要大量的工作。下面我们再看下虚拟场景的地图生成。

22.7. 虚拟场景地图生成

虚拟场景的道路生成就比较简单，主要的应用场景是一些园区，或者一些测试场景。这一部分完全可以制作一个地图编辑器，类似游戏中的地图编辑器，玩家可以根据自己的需求创建游戏中的地图，然后再由脚本动态的生成高精度地图。这部分的功能主要是对标Carsim等

仿真软件的地图编辑功能。



说完了地图，接下来看下车

22.8. 车

车主要分为2部分：车的动力学模型，以及传感器。接下来我们详细分析下这2部分：

- **车的动力学模型** - 这一部分是传统仿真软件的强项，由于应用已经非常成熟，游戏中的汽车动力学模型都比较简单，由于CarSim等软件没有开源，所以目前短期内一个比较好的解决方案是，仿真器提供API接口，调用CarSim和Simulink等软件的动力学模型，实现对汽车的模拟。
- **传感器** - 传感器主要是GPS、IMU、LIDAR、RADAR、CAMERA等，涉及到传感器的位置，校准参数等。当然这一部分也可以仿

真传感器视野范围(FOV)，也可以仿真传感器的校准算法。

22.9. 行为

现在我们加载了地图，车辆，接着我们需要定义一些行为来模拟真实世界。

22.9.1. NPC

npc包括行人和车辆。

- 行人 - 目前主要是模拟行人过马路，以及在路边行走，以及更加复杂的场景，例如下雨天打伞的行人，对于这些异常场景，感知模块不一定能够正常识别。
- 车辆 - 车辆的行为可以由一些简单的行为来模拟复杂的行为，例如停车，变道，加速，减速，来组合出超车，会车等复杂行为。也可以通过模拟真实情况的交通流数据，来模拟整个行为。前一种测试的行为比较成熟，后一种需要根据实际的情况提取出行
- 为，再加入补全信息，才能够正常工作。

22.9.2. 天气

天气主要是影响传感器的感知，最主要的就是摄像头。对LIDAR的影响由于目前没有阅读相关平台是否有加入噪声，这里就先不展开了。

- 天气 - 雨、雪、雾、云层 调整不同的比率来模拟不同的天气情况对传感器的影响，云层主要是会影响光照变化，多云投射的阴影对车道线识别等会有影响。
- 时间 - 白天和夜晚不同光照场景下对传感器的影响。

22.9.3. 红绿灯

这一部分可以归纳为交通信号的行为，其中分为：

- 有保护的红绿灯 - 各大城市是最普遍的，即有箭头的红绿灯，根据对应车道的红绿灯直行或者拐弯。

- 无保护的红绿灯 - 即圆形的红绿灯，对面可以直线的同时，你可以拐弯，需要注意对面直行的车辆，选择让车之后再拐弯。
- 无红绿灯 - 这种常见于郊区路口，需要判断有没有车辆经过而让行或者停止，然后再通过路口。

关于仿真器就介绍完毕了，那么我们如何控制仿真器来实现这些呢？

22.10. API

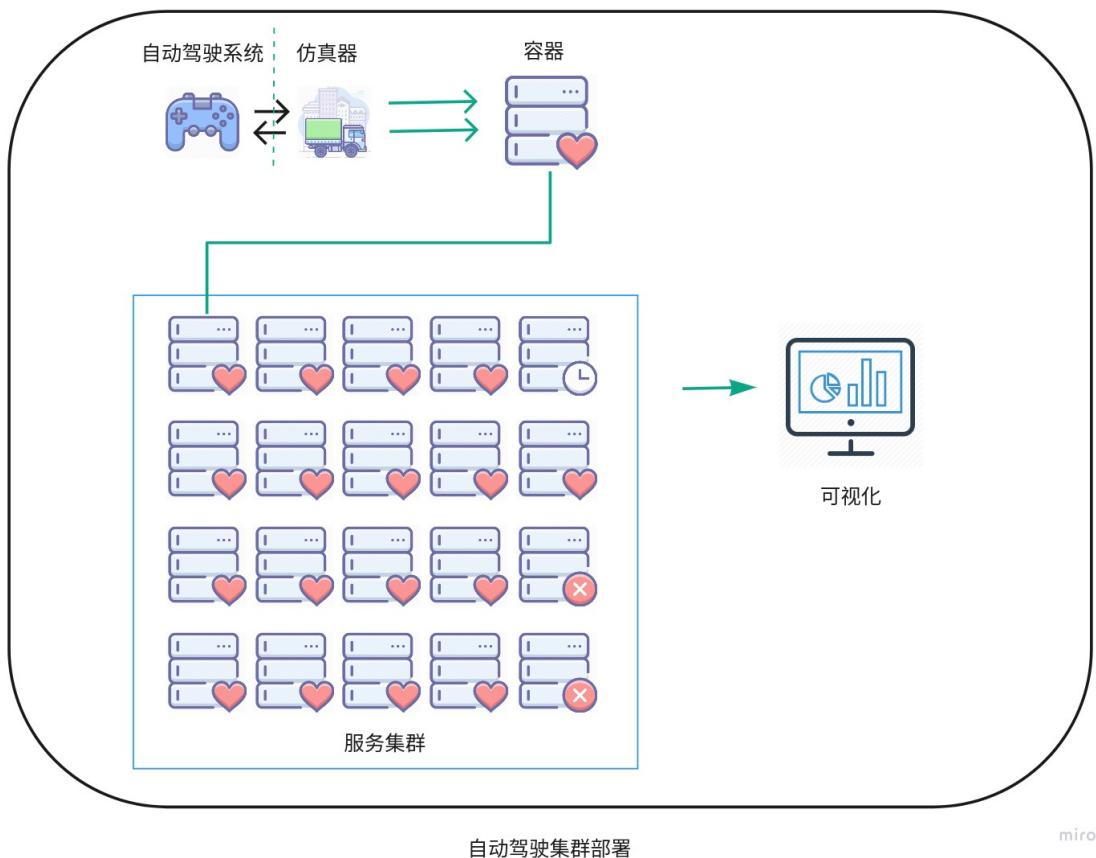
目前主要是通过python API的方式来控制仿真器加载模型，控制仿真器的行为。好处是不用图形界面手工操作，可以实现自动化部署。API 的主要是根据上述所说仿真器的功能实现统一的接口，实现交互。

22.11. 部署

为了提高测试效率，我们还需要大规模部署，一个比较好的方式是通过容器化的方式部署。针对多台机器，一个显而易见的需求就是创建一个管理平台来实现对仿真器的管理。容器部署平台可以监控对应仿真器的状态，并且提供可视化的配置界面，生成和部署不同的场景。

- 监控 - 可以监控仿真器的监控状态，显示正常和有问题的集群，保存日志，维护集群的稳定。
- 可视化 - 首先是配置可视化，可以方便的选择不同的配置（不同的地图，车，行为）来生成不同的场景，其次是通过可视化反馈

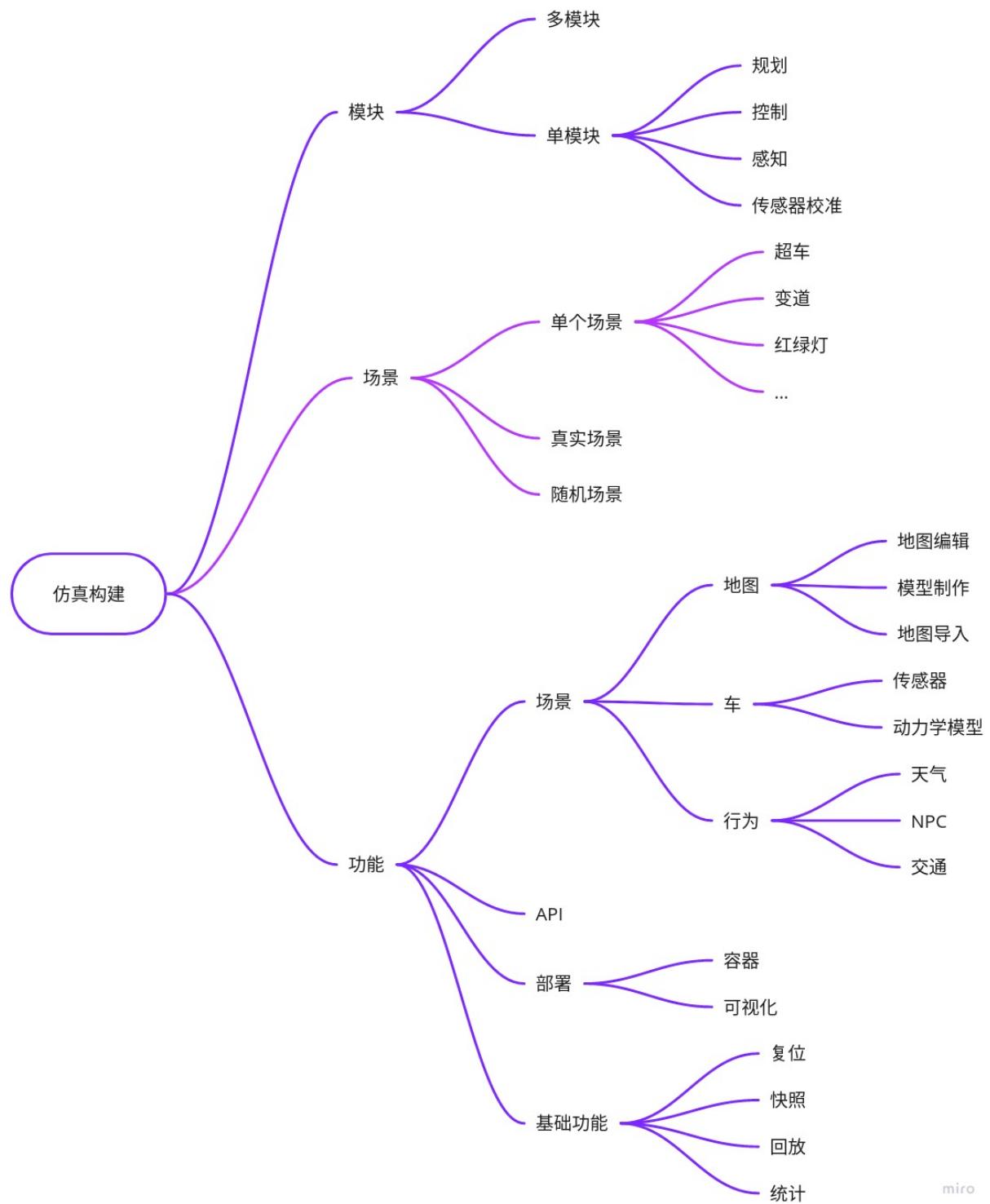
仿真结果，屏蔽仿真集群的细节，使用起来更加直观方便。



22.12. 总结

最后根据功能划分，我们可以单独仿真自动驾驶系统的规划控制模块，也可以单独仿真感知模块，可以仿真传感器校准，也可以端到端的仿真所有模块。可以仿真单个受限的场景，也可以仿真整个地图。总之，仿真系统需要提供灵活的场景生成框架，统一的API接口，以

及大规模部署的能力。



22.13. 参考

虚幻引擎游戏列表

[<https://zh.wikipedia.org/wiki/%E8%99%9A%E5%B9%BB%E5%BC%95%E6%93%8E%E6%B8%B8%E6%88%8F%E5%88%97%E8%A1%A8>] [Unity3D](https://baike.baidu.com/item/Unity3D) [<https://baike.baidu.com/item/Unity3D>]
[Udacity](https://github.com/udacity/self-driving-car-sim) [<https://github.com/udacity/self-driving-car-sim>] [Carla](https://github.com/carla-simulator/carla) [<https://github.com/carla-simulator/carla>] [AirSim](https://github.com/Microsoft/AirSim) [<https://github.com/Microsoft/AirSim>] [lgsyl](https://github.com/lgsyl/simulator)
[<https://github.com/lgsyl/simulator>] [Apollo](https://github.com/ApolloAuto/apollo) [<https://github.com/ApolloAuto/apollo>] [EventSystem](https://docs.unity3d.com/ScriptReference/EventSystems.EventSystem.html)
[<https://docs.unity3d.com/ScriptReference/EventSystems.EventSystem.html>] [openscenario](http://www.openscenario.org/)
[<http://www.openscenario.org/>]

1. 引用的库

读书患不多，思义患不明。

1.1. Table of Contents

- [Abseil](https://github.com/abseil/abseil-cpp) [https://github.com/abseil/abseil-cpp]
- [ADOL-C](https://github.com/coin-or/ADOL-C) [https://github.com/coin-or/ADOL-C]
- [ad-rss-lib](https://github.com/intel/ad-rss-lib) [https://github.com/intel/ad-rss-lib]
- [Benchmark](https://github.com/google/benchmark) [https://github.com/google/benchmark]
- [Bazel](https://github.com/bazelbuild/bazel) [https://github.com/bazelbuild/bazel]
- [Boost](http://www.boost.org) [http://www.boost.org]
- [Bootstrap](https://github.com/twbs/bootstrap) [https://github.com/twbs/bootstrap]
- [Buildifier](https://github.com/bazelbuild/buildtools) [https://github.com/bazelbuild/buildtools]
- [ColPack](https://github.com/CSCsw/ColPack) [https://github.com/CSCsw/ColPack]
- [CivetWeb](https://github.com/civetweb/civetweb) [https://github.com/civetweb/civetweb]
- [Cpplint](https://github.com/cpplint/cpplint) [https://github.com/cpplint/cpplint]
- [cuteci](https://github.com/hasboeuf/cuteci) [https://github.com/hasboeuf/cuteci]
- [Docker](https://www.docker.com) [https://www.docker.com]
- [Eigen](http://eigen.tuxfamily.org/index.php) [http://eigen.tuxfamily.org/index.php]
- [FastCDR](https://github.com/eProsima/Fast-CDR) [https://github.com/eProsima/Fast-CDR]
- [Fast RTPS](https://github.com/eProsima/Fast-RTPS) [https://github.com/eProsima/Fast-RTPS]
- [FFmpeg](https://github.com/FFmpeg/FFmpeg) [https://github.com/FFmpeg/FFmpeg]
- [FFTW](http://www.fftw.org/fftw3_doc/License-and-Copyright.html) [http://www.fftw.org/fftw3_doc/License-and-Copyright.html]
- [Ipopt](https://github.com/coin-or/Ipopt) [https://github.com/coin-or/Ipopt]
- [GFlags](https://github.com/gflags/gflags) [https://github.com/gflags/gflags]
- [GLog](https://github.com/google/glog) [https://github.com/google/glog]
- [gRPC](https://github.com/grpc/grpc) [https://github.com/grpc/grpc]
- [GoogleTest](https://github.com/google/googletest) [https://github.com/google/googletest]
- [jQuery](https://jquery.org/) [https://jquery.org/]
- [JSON for Modern C++](https://github.com/nlohmann/json) [https://github.com/nlohmann/json]
- [OpenCV](https://opencv.org) [https://opencv.org]
- [OSQP](https://github.com/oxfordcontrol/osqp) [https://github.com/oxfordcontrol/osqp]
- [PCL](https://github.com/PointCloudLibrary/pcl) [https://github.com/PointCloudLibrary/pcl]

- [PyTorch](https://github.com/pytorch/pytorch) [https://github.com/pytorch/pytorch]
- [POCO C++ Libraries](https://github.com/pocoproject/poco) [https://github.com/pocoproject/poco]
- [PROJ](https://github.com/OSGeo/PROJ) [https://github.com/OSGeo/PROJ]
- [Prettier](https://github.com/prettier/prettier) [https://github.com/prettier/prettier]
- [Protocol Buffer](https://github.com/protocolbuffers/protobuf) [https://github.com/protocolbuffers/protobuf]
- [python-requests](http://docs.python-requests.org) [http://docs.python-requests.org]
- [Qt5](https://doc.qt.io/qt-5) [https://doc.qt.io/qt-5]
- [shfmt](https://github.com/mvdan/sh) [https://github.com/mvdan/sh]
- [socketio](https://github.com/socketio/socket.io-client) [https://github.com/socketio/socket.io-client]
- [VTK](https://github.com/Kitware/VTK) [https://github.com/Kitware/VTK]
- [YAML-cpp](https://github.com/jbeder/yaml-cpp) [https://github.com/jbeder/yaml-cpp]
- [three.js](https://threejs.org/) [https://threejs.org/]

2. 论文

我见青山多妩媚，料青山见我应如是。

2.1. Localization papers

- [Robust and Precise Vehicle Localization based on Multi-sensor Fusion in Diverse City Scenes](https://arxiv.org/abs/1711.05805) [<https://arxiv.org/abs/1711.05805>]

2.2. Perception papers

[ChauffeurNet: Learning to Drive by Imitating the Best and Synthesizing the Worst](https://arxiv.org/abs/1812.03079) [<https://arxiv.org/abs/1812.03079>]

2.3. Prediction papers

- [Simple Online and Realtime Tracking with a Deep Association Metric](https://arxiv.org/abs/1703.07402) [<https://arxiv.org/abs/1703.07402>] - [code](https://github.com/nwojke/deep_sort) [https://github.com/nwojke/deep_sort]

2.4. Planning papers

- [Baidu Apollo EM Motion Planner](https://arxiv.org/abs/1807.08048) [<https://arxiv.org/abs/1807.08048>]

2.5. Control papers

- [Baidu Apollo Auto-Calibration System - An Industry-Level Data-Driven and Learning based Vehicle Longitude Dynamic Calibrating Algorithm](https://arxiv.org/abs/1808.10134) [<https://arxiv.org/abs/1808.10134>]

2.6. Simulation papers

- [AADS: Augmented Autonomous Driving Simulation using Data-driven Algorithms](#) [<https://arxiv.org/abs/1901.07849>]
- [Augmented LiDAR Simulator for Autonomous Driving](#) [<https://arxiv.org/abs/1811.07112>]

3. 常见问题

非淡漠无以明德，非宁静无以致远。

3.1. build

3.2. map

- How to change map.bin to human-readable map.txt? In apollo docker run below cmds.

```
source scripts/apollo_base.sh  
  
protoc --decode apollo.hdmap.Map modules/map/proto/map.proto  
< modules/map/data/sunnyvale_loop/base_map.bin > base_map.txt
```

- How to create new map with RTK

1. 解压制作好的地图

```
python modules/tools/map_gen/extract_path.py map_file  
data/bag/2021040612554.record.00000
```

查看当前轨迹

```
python modules/tools/map_gen/plot_path.py map_file
```

2. 生成地图

```
python modules/tools/map_gen/map_gen.py map_file
```

3. 查看地图

```
python modules/tools/mapshow/mapshow.py -m map_map_file.txt
```

生成地图

```
python modules/tools/create_map/convert_map_txt2bin.py -i  
map_map_file.txt -o  
/apollo/modules/map/data/your_map_dir/base_map.bin
```

生成sim_map

```
./bazel-bin/modules/map/tools/sim_map_generator -  
map_dir=/apollo/modules/map/data/your_map_dir -  
output_dir=/apollo/modules/map/data/your_map_dir
```

生成routing_map

```
/apollo/bazel-bin/modules/routing/topo_creator/topo_creator -  
map_dir=/apollo/modules/map/data/your_map_dir --  
flagfile=modules/routing/conf/routing.conf
```

测试之前需要修改”vi modules/common/data/global_flagfile.txt”，屏蔽选项”–log_dir/-use_navigation_mode”

```
--map_dir=/apollo/modules/map/data/your_map_dir
```

测试生成的routing_map是否可以联通

```
python modules/tools/routing/debug_topo.py
```

3.3. simulation

3.4. Planning

- How to add decider or optimizer to a planning scenario ?
 1. Add your own decider in “modules/planning/tasks/deciders”
 2. Add config in “modules/planning/conf/scenario/lane_follow_config.pb.txt”
 3. Add TaskType in “modules/planning/proto/planning_config.proto”
 4. Register your task in “TaskFactory::Init”

3.5. misc

- What is the format of the config file in Apollo? The config file base by **protobuf** format, and read by `cyber::common::GetProtoFromFile()` method.
- How to open the debug log?

1. modify the “apollo/cyber/setup.bash”

```
# for DEBUG log  
export GLOG_v=4
```

2. Enable environment variables

```
source cyber/setup.bash
```

3. Export bag data to lidar,camera

```
./bazel-  
bin/modules/localization/msf/local_tool/data_extraction/cyber  
_record_parser --bag_file  
data/bag/20210305145950.record.00000 --out_folder data/ --  
cloud_topic=/apollo/sensor/lidar32/compensator/PointCloud2
```

Index

Dig into Apollo - Cyber

license MIT

Table of Contents

- How do you design cyber?
- 需求分析
- 系统设计
 - 随意的假设
 - 多节点
 - 通信方式
 - 资源调度
 - linux进程调度
 - 无人驾驶线程调度
 - 软件复用
 - 快速测试
- 其他
 - 云平台
- Reference

How do you design cyber?

无人驾驶车借鉴了很多机器人领域的技术，我们可以把无人车看做一个轮式机器人。Apollo的计算平台之前一直采用的是ROS，3.5版本用Cyber替换了这一架构，那么如果让我们来重新设计这一个框架，我们需要支持哪些特性呢，我们如何去实现它呢？

- 我们需要一个什么样的系统？
- 如何保证系统的稳定性和灵活性？
- 如何来调试和维护这样复杂的系统？

需求分析

我们先借鉴下ROS的思路：

分布式计算 现代机器人系统往往需要多个计算机同时运行多个进程，例如：

- 一些机器人搭载多台计算机，每台计算机用于控制机器人的部分驱动器或传感器；
- 即使只有一台计算机，通常仍将程序划分为独立运行且相互协作的小的模块来完成复杂的控制任务，这也是常见的做法；
- 当多个机器人需要协同完成一个任务时，往往需要互相通信来支撑任务的完成；

单计算机或者多计算机不同进程间的通信问题是上述例子中的主要挑战。ROS为实现上述通信提供两种相对简单、完备的机制。

软件复用 随着机器人研究的快速推进，诞生了一批应对导航、路径规划、建图等通用任务的算法。当然，任何一个算法实用的前提是其能够应用于新的领域，且不必重复实现。事实上，如何将现有算法快速移植到不同系统一直是一个挑战，ROS通过以下两种方法解决这个问题。

- ROS 标准包（Standard Packages）提供稳定、可调式的各类重要机器人算法实现。
- ROS通信接口正在成为机器人软件互操作的事实标准，也就是说绝大部分最新的硬件驱动和最前沿的算法实现都可以在ROS中找到。例如，在ROS的官方网页上有着大量的开源软件库，这些软件使用ROS通用接口，从而避免为了集成它们而重新开发新的接口程序。

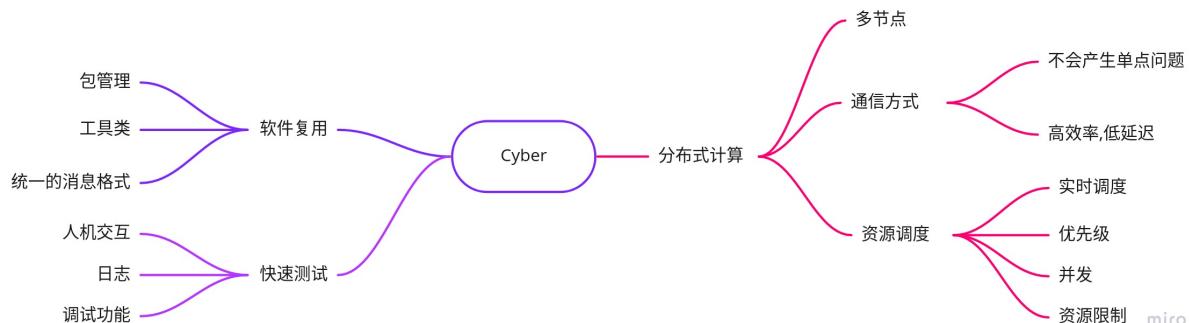
综上所述，开发人员将更多的时间用于新思想和新算法的设计与实现，尽量避免重复实现已有研究结果。

快速测试 为机器人开发软件比其他软件开发更具挑战性，主要是因为调试准备时间长，且调试过程复杂。况且，因为硬件维修、经费有限等因素，不一定随时有机器人可供使用。ROS提供两种策略来解决上述问题。

- 精心设计的ROS系统框架将底层硬件控制模块和顶层数据处理与决策模块分离，从而可以使用模拟器替代底层硬件模块，独立测试顶层部分，提高测试效率。
- ROS另外提供了一种简单的方法可以在调试过程中记录传感器数据及其他类型的消息数据，并在试验后按时间戳回放。通过这种方式，每次运行机器人可以获得更多的测试机会。例如，可以记录传感器的数据，并通过多次回放测试不同的数据处理算法。在ROS术语中，这类记录的数据叫作包（bag），一个被称为rosbag的工具可以用于记录和回放包数据。
- 用户通常通过台式机、笔记本或者移动设备发送指令控制机器人，这种人机交互接口可以认为是机器人软件的一部分。

采用上述方案的一个最大优势是实现代码的“无缝连接”，因为实体机器人、仿真器和回放的包可以提供同样（至少是非常类似）的接口，上层软件不需要修改就可以与它们进行交互，实际上甚至不需要知道操作的对象是不是实体机器人。

参考上述实现，我们可以把需求细化为以下几个方面：

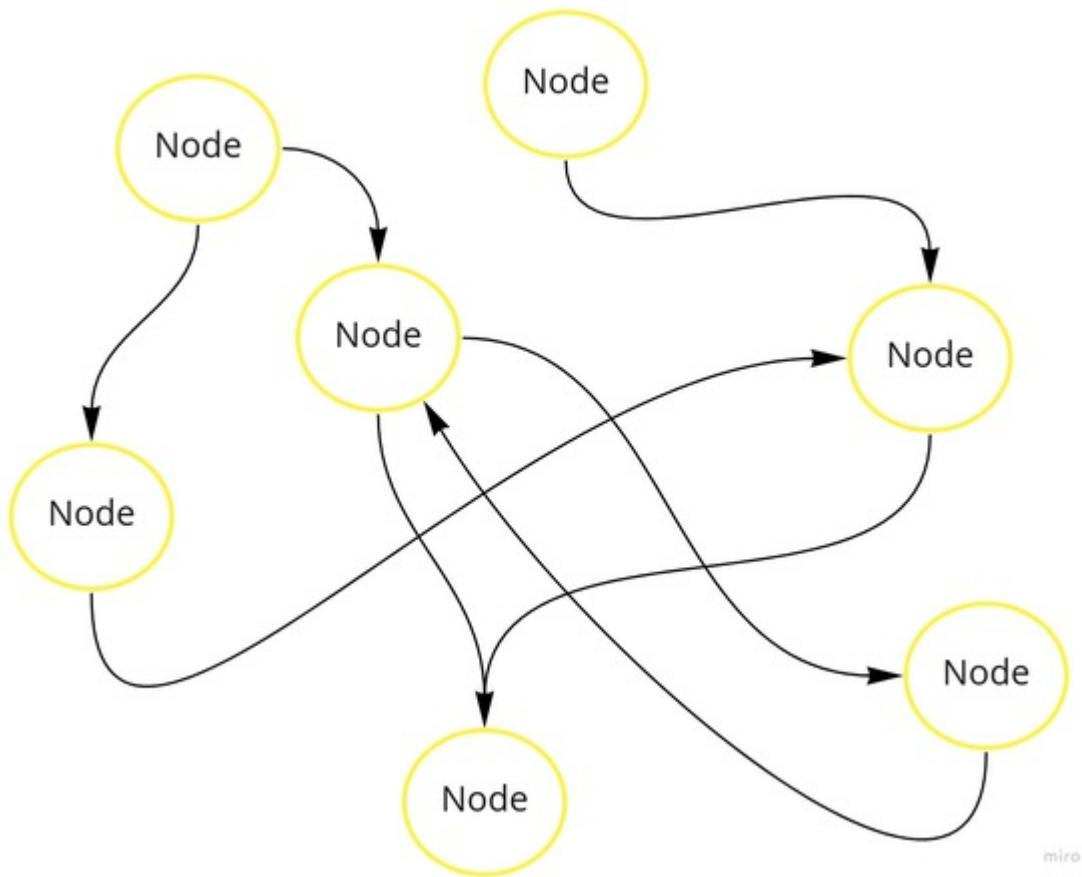


实际上Apollo主要用到了ROS消息通信的功能，同时也用到了录制bag包等一些工具类。所以目前Cyber的首要设计就是替换ROS消息通信的功能。

系统设计

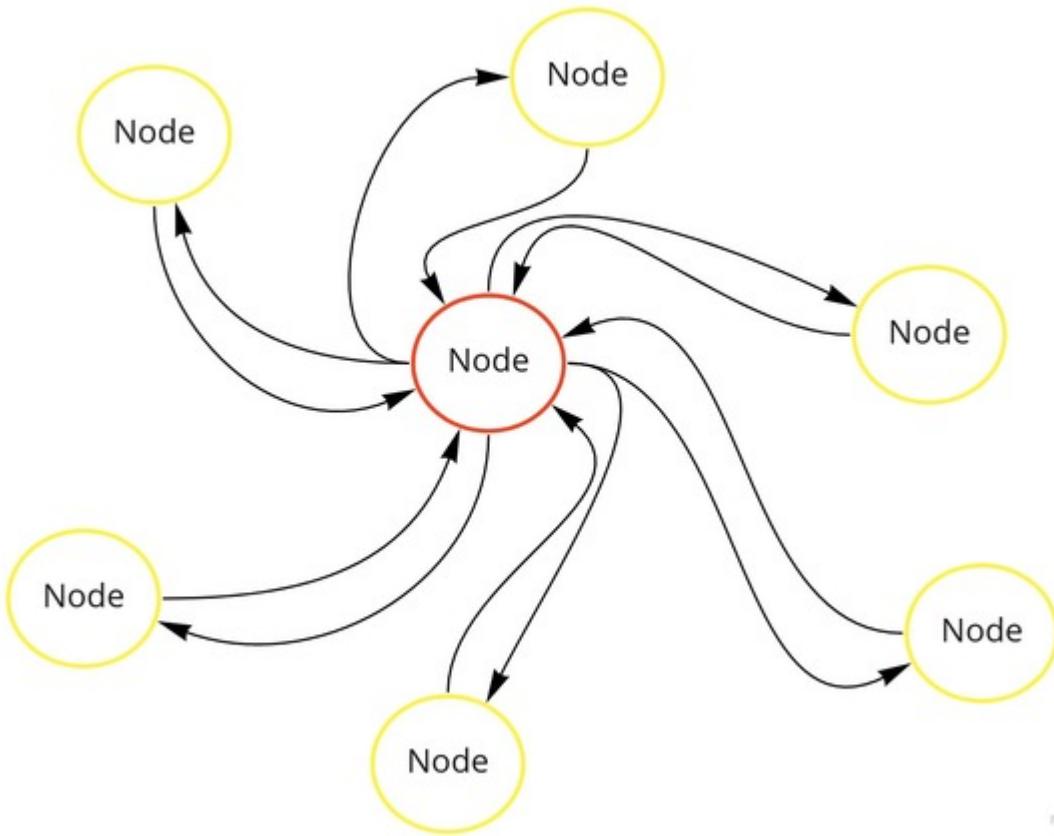
随意的假设

按照上述需求，我们可以随便假想，或者根据自己的理解先画出系统的草图，这里我们要实现一个分布式的系统：



1. 上述的系统是一个分布式系统，每个节点作为一个Node。
2. 上述系统每个节点之间都可以相互通信，一个节点下线，不会导致到整个系统瘫痪。
3. 上述系统可以灵活的增加删除节点。

那么我们再看下其他的设计方式：



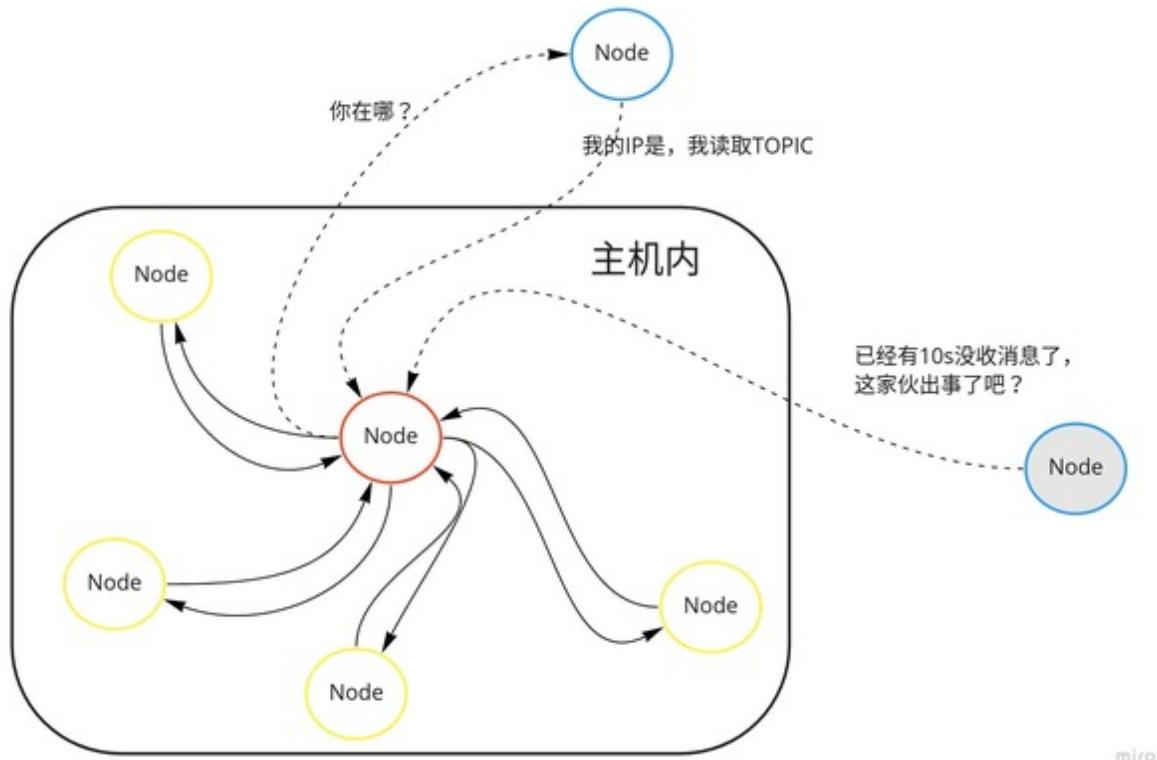
上述系统采用了集中式的消息管理，每个节点之间通讯必须经过主节点来转发对应的消息，如果主节点下线，那么所有的节点都会通信失败，导致系统瘫痪。

1. 上述系统是一个分布式系统，每个节点作为一个Node。
2. 上述系统每个节点通过主节点通信，主节点下线会导致系统崩溃。
3. 上述系统可以灵活的增加删除节点。对上述系统，一个补救措施就是在增加一个主节点，作为备份，当主节点下线时，启动备份主节点。

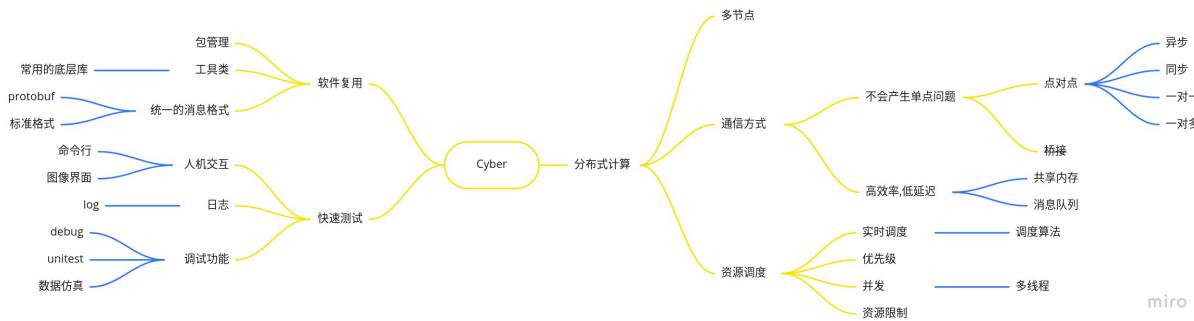
这2种方式的主要区别就是通信方式的区别。

当然集中式的消息管理是否有好处呢？集中式的消息处理天然支持管理节点的功能，而点对点的消息处理不支持。例如：

- 当一个节点有10s没有发送消息，那么集中式的消息可以监控并且知道这个节点是否出故障了；
- 集中式的消息可以知道哪些节点在线去找到这些节点，这在多机网络通信的时候很管用，节点只需要注册自己的IP地址，然后由管理节点告诉你去哪里拿到消息。



上述只是一个初步的想法，那么基于上面的启发，我们针对上述的每项需求，完成我们的系统设计。



我们接下来详细的分析每个需求：

多节点

1. 节点管理
2. 节点依赖

通信方式

1. 点对点
2. 采用共享内存的方式可以提高效率，需要注意并发访问时候的问题

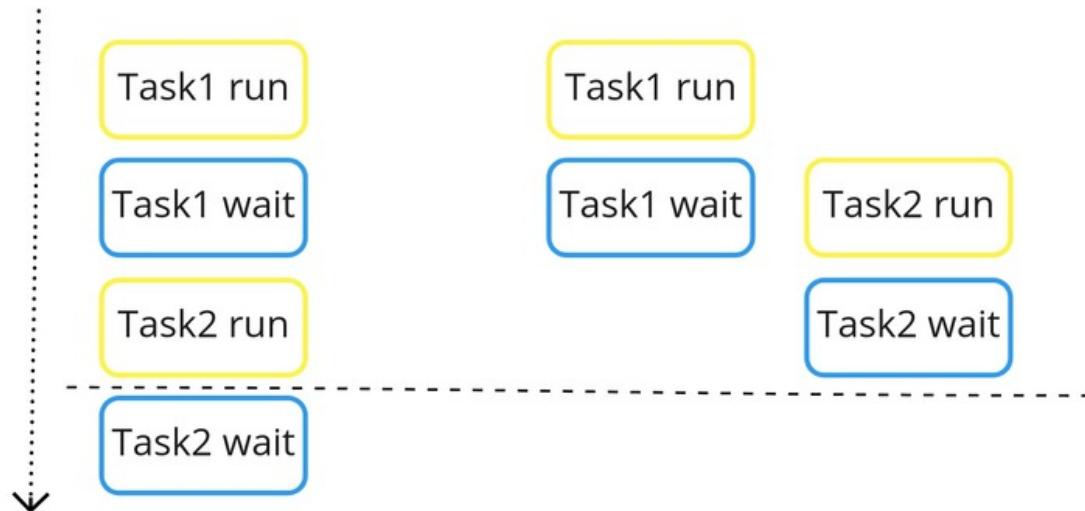
资源调度

1. 进程调度算法改为实时算法
2. 进程有优先级
3. 支持并发
4. 能够限制系统的资源占用

linux进程调度

操作系统最基本的功能就是管理线程，linux的线程调度采用的是CFS(Completely Fair Scheduler)算法，我们先看下没有调度和有调度的情况下差异。

CPU Timeline



miro

上述是单个CPU核心的情况下，左边是没有CPU调度的情况，任务1在进行完计算之后，会读取内存或者IO的数据，这时候CPU会进入等待状态，CPU在等待的时候没有做任何事情。而右边采用了调度策略，在CPU等待的过程中，任务1主动让出CPU，这样下一个任务就可以在当前任务等待IO的过程中执行，可以看到对任务的调度合理的利用了CPU，使得CPU的利用率更高，从而使任务执行的更快。

linux内核又分为可以抢占的和非抢占的，非抢占的内核禁止抢占，即在一个任务执行完成之前，除非他主动让出CPU或者执行完成，CPU会一直被这个任务占据，不能够被更高优先级的任务抢占。而抢占式的内核则支持在一个任务执行的过程中，如果有更高优先级的任务请求，那么内核会暂停现在运行的任务，转而运行优先级更高的任务，显然抢占式的内核的实时性更好。

CPU把任务根据优先级划分，并且划分不同的时间片，通过时间片轮转，使CPU看起来在同一时间能够执行多个任务，就好像一个人同时交叉的做几件事情，看起来多个事情是一起完成的一样。每个进程会分配一段时间片，在当前进程的时间片用完的时候，如果没有其他任

务，那么会继续执行；如果有其他任务，那么当前任务会暂停，切换到其他任务执行。这样带来一个问题就是如何判断进程的优先级。

内核把任务做了区分，分为交互型和脚本型，如果是交互型的进程，对实时性的要求比较高，但是大部分情况下又不会一直运行，典型的情况是，键盘输入的情况，大部分情况下键盘可能没有输入，但是一旦用户输入了，又要求能够立刻响应，否则用户会觉得输入很卡顿。而脚本型因为一直在后台运行，对实时性的要求没那么高，所以不需要立刻响应。linux通过抢占式的方式，对任务的优先级进行排序，交互型进程的优先级要比脚本型进程的优先级要高。从而在交互性进程到来之前能够抢占CPU，优先运行。还有一类是实时进程，这类进程的优先级最高，实时进程必须要保证执行，因此会有限抢占其他进程。

如果单纯的根据优先级，低优先级的任务可能很长一段时间都得不到执行，因此需要更加公平的算法，在一个进程等待时间太长的时候，会动态的提高它的优先级，如果一个进程执行很长的一段时间了，那么会动态降低它的优先级，这样带来的好处是，不会导致低优先级的长期得不到CPU，而高优先级的CPU长期霸占CPU，linux采用的就是CFS(Completely Fair Scheduler)算法，通过该算法可以保证进程能够相对公平的占用CPU。

同时在多CPU和多核场景下，由于每个核心的进程调度队列都是单独的，那么会导致一个问题，如果任务都集中在某一个CPU核心，而其他的CPU核心的队列都是空闲状态，这样也会导致CPU的性能低下，在这种情况下，linux会把任务迁移到其他CPU核心，使得CPU之间的负载均衡，linux引入了Cgroups用来限制，控制与分离一个进程组群的资源（如CPU、内存、磁盘输入输出等）。当然，线程迁移会带来开销，有些时候我们会绑定任务到某一个核心，防止线程迁移。同时如果系统频繁的中断，CPU会频繁停下任务去处理中断，有些场景(网络设备)需要频繁处理网络中断的情况下，通常会绑定中断到某一个CPU核心，这样其他的核心就不会频繁中断，减少了进程切换的开销。

无人驾驶线程调度

参考linux的线程调度，我们也可以思考下无人驾驶线程调度的算法。我们假设有如下线程：定位，感知，规划，控制，传感器读取，日志，地图（这只是对任务的抽象，当然系统的进程不可能只有这么几个）。假设目前的CPU只有2个核心，那么我们如何规划这些任务的优先级呢？

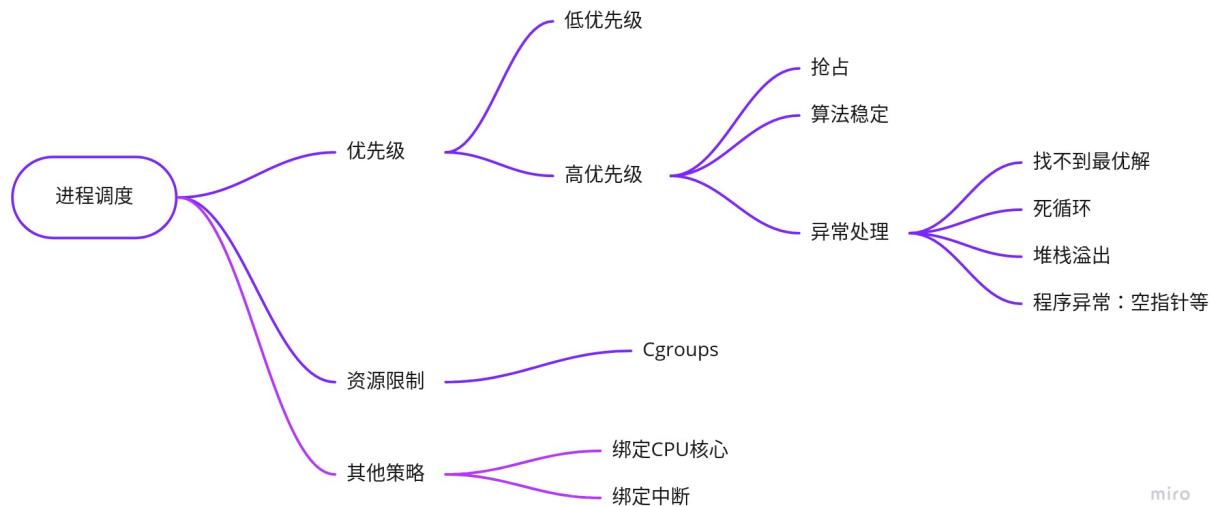
1. 首先，我们假设定位，感知，规划和控制，传感器读取的优先级比日志和地图更高。这也很容易理解，打不打日志和地图读取的慢点对系统的影响不大，而上述的模块如果读取的很慢，则会导致系统故障。
2. 接下来我们再看优先级高的模块，因为目前我们只有2核心，所以不可能同时执行上述所有模块，只能通过时间片轮转来实现。这里就引入了一个问题，如果分配的时间片太长，会导致响应不及时，如果分配的时间片太短，又会导致线程切换开销，需要折中考虑。如果运行过程中感知和规划正在执行，并且分配的时间片还没有用完，那么控制模块不会抢占CPU，直到运行中的模块时间片用完。
3. 对这些模块的算法复杂度有要求，如果感知模块采用了复杂度较高的算法提高准确率，这样导致的结果是感知会占用更多的CPU时间，其他模块每次需要和感知模块竞争CPU，结果就是导致总体的执行时间会变长。比如，规定感知只需要在200ms的时候处理完任务就可以了，之前感知的算法实现是100ms，而控制模块的时间是100ms，CPU的时间片是50ms，那么感知需要2个时间片，控制需要2个时间片，总的需要时间是200ms，控制模块完成的时间由于时间片轮转，可能是150ms。但是如果感知为了提高效果，增加了算法的复杂度，运行时间改为200ms，感知模块照常能够完成自己的任务，因为只要200ms完成任务，感知模块就完成了任务，总的需要的时间可能是300ms，但是引入的另外的问题是由于竞争控制模块可能完成的时间是200ms，这样就会导致控制模块的时延达不到要求。其实这样的情况总的来说一是需要升级硬件，比如增加CPU的核数；另外的办法就是降低系统算法的复杂度，每个模块的任务要尽可能的高效。
4. 通过上面的要求也可以看到，系统进程的算法复杂度要尽可能的稳定，不能一下子是50ms，一下子是200ms，或者直接找不到最优解，这是最坏的情况，如果各个模块的算法都不太稳定，带来的影响就是当遇到极端情况，每个模块需要的时间都变多时候，

系统的负载会一下子变高，导致模块的相应不及时，这对自动驾驶是很致命的问题。

5. 上述是理想情况下，那么我们会遇到哪些情况，系统的进程会奔溃或者一直占用CPU的情况呢？

- 找不到最优解，死循环。大部分情况下程序没有响应是因为找不到最优解，或者死循环，这种状态可以通过代码和算法实现保证。
- 堆栈溢出，内存泄露，空指针。这种情况是由于程序编写错误，也可以通过代码保证。
- 硬件错误。极小概率的情况下，CPU的寄存器会出错，嵌入式(powerpc)的CPU都会有冗余校正，而家用或者服务器(intel)没有这种设计，这种情况下只能重启进程，或者硬件。

我们根据上述的思路，可以得到如下图所示：



- 把控制的优先级设置到最高，规划其次，感知和定位的优先级设置相对较低，因为控制和规划必须马上处理，感知如果当前帧处理不过来，大不了就丢掉，接着处理下一帧。当然这些线程都需要设置为实时进程。而地图，日志，定位等的优先级设置较低，在其他高优先级的进程到来时候会被抢占。
- Canbus等传感器数据，可以绑定到一个CPU核心上处理，这样中断不会影响到其他核心，导致频繁线程切换。
- 对线程设置cgroups，可以控制资源使用，设置优先级等。

- 测试算法的时间复杂度，是否稳定。

软件复用

1. 包管理
2. 工具类

快速测试

1. 人机交互
2. 日志
3. 调试功能
4. 通信接口

其他

云平台

如果需要监控线上无人车的状态，那么需要无人车提供连接到云的能力，即发送消息和接收消息的能力。Cyber需要支持能够发送消息给云端，并且接收来自云端消息的能力。

cyber分析

cyber入口

cyber的入口在”cyber/mainboard”目录，我们先看下目录结构：

```
.  
├── mainboard.cc          // 入口  
├── module_argument.cc    // 模块参数  
├── module_argument.h  
└── module_controller.cc  // 模块控制  
   └── module_controller.h
```

根据文件名称也可以大概猜到cyber主目录的工作，cyber主函数通过模块的参数加载cyber中的所有模块，而cyber模块是有依赖顺序的，每个cyber模块都有一个DAG文件，这个文件声明了各个模块的依赖关系，而module_controller大概率就是控制模块的加载顺序。接下来我们通过看代码验证我们的猜想是否正确。

我们从”mainboard.cc”开始，阅读代码之前的头文件相当关键，头文件可以告诉我们文件之间的依赖关系，引用了哪些模块。我们可以看到主模块引用

了”mainboard/module_argument.h”和”mainboard/module_controller.h”，所以我们先从”mainboard.cc”开始看，剩下的2个文件自然会在”mainboard.cc”中引用。还有一些其它的引用是状态和标志位，可以先略过。

```
#include "cyber/common/global_data.h"
#include "cyber/common/log.h"
#include "cyber/init.h"
#include "cyber/mainboard/module_argument.h"          // 
"mainboard.cc"引用
#include "cyber/mainboard/module_controller.h"        // 
"mainboard.cc"引用
#include "cyber/state.h"

#include "gflags/gflags.h"
```

接下来我们看下函数的主流程：

```
int main(int argc, char** argv) {
    google::SetUsageMessage("we use this program to load dag
and run user apps.");

    // 解析模块参数
    // parse the argument
    ModuleArgument module_args;
    module_args.ParseArgument(argc, argv);

    // 初始化cyber
    // initialize cyber
    apollo::cyber::Init(argv[0]);

    // 启动模块
```

```

// start module
ModuleController controller(module_args);
if (!controller.Init()) {
    controller.Clear();
    AERROR << "module start error.";
    return -1;
}

// 等待cyber关闭
apollo::cyber::WaitForShutdown();
controller.Clear();
AINFO << "exit mainboard.";

return 0;
}

```

接下来我们详细的分析每个过程。

- **解析模块参数** 解析模块参数

在”module_argument.h”和”module_argument.cc”中的”ModuleArgument”类中，具体的实现如下

```

void ModuleArgument::ParseArgument(const int argc, char*
const argv[]) {

    // 解析输入参数
    GetOptions(argc, argv);

    ...

    // 设置执行组，类似linux的cgroups
    GlobalData::Instance()->SetProcessGroup(process_group_);
    // 设置调度器名称
    GlobalData::Instance()->SetSchedName(sched_name_);
    // 打印模块的信息：名称，组，DAG配置
    AINFO << "binary_name_ is " << binary_name_ << ", "
process_group_ is "
        << process_group_ << ", has " <<
dag_conf_list_.size() << " dag conf";
    // 打印所有模块的依赖关系
    for (std::string& dag : dag_conf_list_) {
        AINFO << "dag_conf: " << dag;
    }
}

```

```
}
```

- **初始化cyber** 初始化cyber就是cyber目录下的”init.h”和”init.cc”中，具体的实现如下：

```
bool Init(const char* binary_name) {
    // 获取锁，为了改变state状态而获取锁
    std::lock_guard<std::mutex> lg(g_mutex);
    // 如果已经初始化，则返回失败
    if (GetState() != STATE_UNINITIALIZED) {
        return false;
    }

    // 初始化日志，并且把打印日志线程放入调度器
    InitLogger(binary_name);
    auto thread = const_cast<std::thread*>(async_logger-
>LogThread());
    scheduler::Instance()->SetInnerThreadAttr("async_log",
thread);
    std::signal(SIGINT, OnShutdown);

    // 注册退出句柄ExitHandle，调用Clear()函数执行
    // Register exit handlers
    if (!g_atexit_registered) {
        if (std::atexit(ExitHandle) != 0) {
            AERROR << "Register exit handle failed";
            return false;
        }
        AINFO << "Register exit handle succ.";
        g_atexit_registered = true;
    }
    // 设置状态为已经初始化
    SetState(STATE_INITIALIZED);
    return true;
}
```

- **启动模块** 启动模块功能

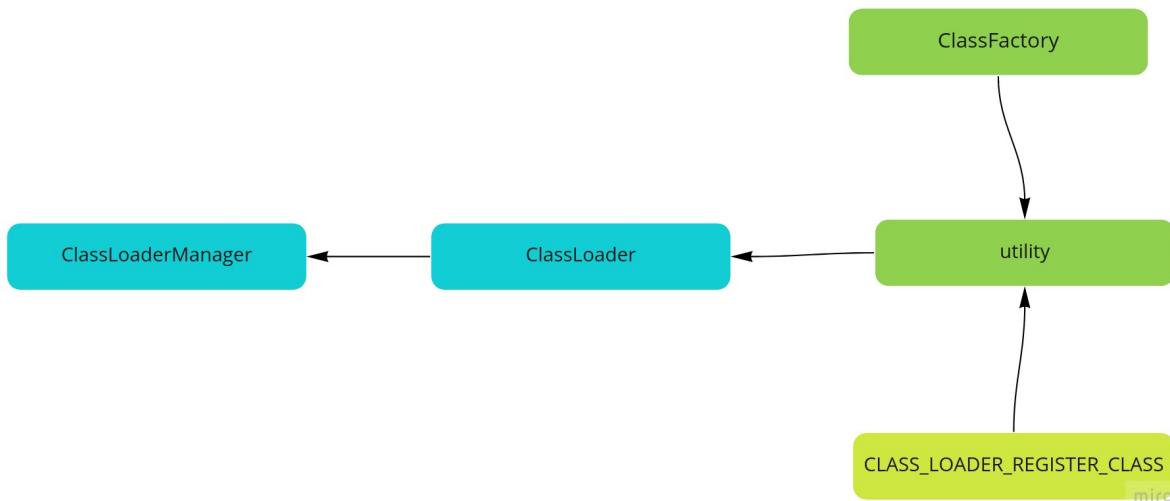
在”module_controller.h”和”module_controller.cc”中实现，具体的流程如下：

```
// 1. 构造ModuleController  
// 2. ModuleController控制器初始化
```

classloader(类动态加载)

首先我们需要搞清楚classloader的作用，classloader动态的加载”.so”文件，从而实现动态的加载和卸载模块。说的直白一点就是cyber通过classloader动态的加载定位，感知，规划，控制等模块。这样的好处是当一个模块奔溃时候，只需要动态的从新加载这个模块就可以了，而不需要从新加载其他模块。

我先看下整体的结构，ClassLoaderManager管理着ClassLoader，而ClassLoader调用utility来实现具体的功能，实际上utility是通过c++的PocoFoundation库来实现加载动态库的。



ClassLoader

首先我们来看一下”ClassLoader”类：

```
// 动态库是否已经加载  
bool IsLibraryLoaded();  
// 加载动态库  
bool LoadLibrary();  
// 卸载动态库  
int UnloadLibrary();
```

```

// 获取动态库路径
const std::string GetLibraryPath() const;
// 获取有效的类名称
std::vector<std::string> GetValidClassNames();
// 创建对象
std::shared_ptr<Base> CreateClassObj(const std::string&
class_name);
// 类是否有效
bool IsClassValid(const std::string& class_name);

```

也就是说classloader提供了一系列的方法来实现类的加载和卸载。下面我们就逐个分析classloader的工作原理：

todo: 获取classloader中加载的类的集合？？？

```

template <typename Base>
std::vector<std::string> ClassLoader::GetValidClassNames() {
    return (utility::GetValidClassNames<Base>(this));
}

```

todo: 查找类是否加载？

```

template <typename Base>
bool ClassLoader::IsClassValid(const std::string& class_name)
{
    std::vector<std::string> valid_classes =
GetValidClassNames<Base>();
    return (std::find(valid_classes.begin(),
valid_classes.end(), class_name) !=
            valid_classes.end());
}

```

todo: 根据类名称创建对象，并且返回对象指针，注意创建对象的过程中classobj_ref_count_加1，释放对象之后减1，通过计数器表明类加载器是否还存在引用关系，而不会释放掉。关于只能指针指定删除器可以[参考](https://zh.cppreference.com/w/cpp/memory/shared_ptr/shared_ptr) [https://zh.cppreference.com/w/cpp/memory/shared_ptr/shared_ptr]

```

template <typename Base>
std::shared_ptr<Base> ClassLoader::CreateClassObj(
    const std::string& class_name) {
    if (!IsLibraryLoaded()) {

```

```

    // 加载动态库
    LoadLibrary();
}

// 创建对象
Base* class_object = utility::CreateClassObj<Base>
(class_name, this);
if (class_object == nullptr) {
    AWARN << "CreateClassObj failed, ensure class has been
registered. "
        << "classname: " << class_name << ", lib: " <<
GetLibraryPath();
return std::shared_ptr<Base>();
}

std::lock_guard<std::mutex> lck(classobj_ref_count_mutex_);
classobj_ref_count_ = classobj_ref_count_ + 1;

// 构造智能指针，并且指定删除器
std::shared_ptr<Base> classObjSharePtr(
    class_object,
std::bind(&ClassLoader::OnClassObjDelete, this,
          std::placeholders::_1));
return classObjSharePtr;
}

template <typename Base>
void ClassLoader::OnClassObjDelete(Base* obj) {
    if (nullptr == obj) {
        return;
    }

    std::lock_guard<std::mutex> lck(classobj_ref_count_mutex_);
    delete obj;
    --classobj_ref_count_;
}

```

接着看下”class_loader.cc”中的构造函数，可以看到一个ClassLoader需要指定动态库路径，初始化引用次数，然后加载对应的动态库。

```

ClassLoader::ClassLoader(const std::string& library_path)
    : library_path_(library_path),

```

```

    loadlib_ref_count_(0),
    classobj_ref_count_(0) {
LoadLibrary();
}

```

动态库是否已经加载

```

bool ClassLoader::IsLibraryLoaded() {
    return utility::IsLibraryLoaded(library_path_, this);
}

```

加载动态库，每次加载动态库的引用计数加1

```

bool ClassLoader::LoadLibrary() {
    std::lock_guard<std::mutex> lck(loadlib_ref_count_mutex_);
    ++loadlib_ref_count_;
    AINFO << "Begin LoadLibrary: " << library_path_;
    return utility::LoadLibrary(library_path_, this);
}

```

卸载动态库，在”classobj_ref_count_ > 0”的时候证明类还有引用，这时候不能卸载，而”loadlib_ref_count_ == 0”的时候才会卸载动态库，返回的loadlib_ref_count_表示当前的加载动态库的计数，加锁是为了多线程访问。

```

int ClassLoader::UnloadLibrary() {
    std::lock_guard<std::mutex>
lckLib(loadlib_ref_count_mutex_);
    std::lock_guard<std::mutex>
lckObj(classobj_ref_count_mutex_);

    if (classobj_ref_count_ > 0) {
        AINFO << "There are still classobjs have not been
deleted, "
                "classobj_ref_count_: "
                << classobj_ref_count_;
    } else {
        --loadlib_ref_count_;
        // 卸载动态库
        if (loadlib_ref_count_ == 0) {
            utility::UnloadLibrary(library_path_, this);
        } else {

```

```

        if (loadlib_ref_count_ < 0) {
            loadlib_ref_count_ = 0;
        }
    }
    return loadlib_ref_count_;
}

```

从上述过程可以看到”ClassLoader”类主要实现了类的加载，卸载，创建对象，而具体的实现主要通过”utility”来实现。

utility

utility通过调用c++的PocoFoundation库来实现加载动态库，下面我们来具体看下：

Cyber通信方式

cyber的通信方式有以下几种：

```

switch (mode) {
    case OptionalMode::INTRA:
        transmitter = std::make_shared<IntraTransmitter<M>>
(modified_attr);
        break;

    case OptionalMode::SHM:
        transmitter = std::make_shared<ShmTransmitter<M>>
(modified_attr);
        break;

    case OptionalMode::RTPS:
        transmitter =
            std::make_shared<RtpsTransmitter<M>>(modified_attr,
participant());
        break;

    default:
        transmitter =
            std::make_shared<HybridTransmitter<M>>

```

```
(modified_attr, participant());
    break;
}
```

我们先看下是根据什么配置来决定通信方式的？

SHM (shared-memory queues) SHM模式的配置可以指定IP和Port

```
message ShmMulticastLocator {
    optional string ip = 1;
    optional uint32 port = 2;
};

message ShmConf {
    optional string notifier_type = 1;
    optional ShmMulticastLocator shm_locator = 2;
};
```

cyber的ip地址：

```
export CYBER_IP=127.0.0.1
```

RTPS (Real-Time Publish Subscribe) 实时发布订阅
<https://tools.ietf.org/html/draft-thiebaut-rtps-wps-00>

RTPS协议是针对视频流新推出的网络协议，增加了控制信息。

Simple Discovery Protocol (SDP). It is divided in the Simple Participant Discovery Protocol (SPDP) and the Endpoint Discovery Protocol (SEDP).

https://en.wikipedia.org/wiki/Simple_Service_Discovery_Protocol

https://community.rti.com/static/documentation/connext-dds/5.2.3/doc/manuals/connext_dds/html_files/RTI_ConnextDDS_CoreLibraries_UsersManual/Content/UsersManual/Ports_Used_for_Discovery.htm
<https://zh.wikipedia.org/wiki/%E7%AE%80%E5%8D%95%E6%9C%8D%E5%8A%A1%E5%8F%91%E7%8E%B0%E5%8D%8F%E8%AE%AE>

1. 首先注册Participant，设置配置，比如广播地址，端口
2. 然后通过创建发布和订阅者来实现服务注册

```

// 1.

void Participant::CreateFastRtpsParticipant(
    const std::string& name, int send_port,
    eprosima::fastrtps::ParticipantListener* listener) {
    uint32_t domain_id = 80;

    const char* val = ::getenv("CYBER_DOMAIN_ID");
    if (val != nullptr) {
        try {
            domain_id = std::stoi(val);
        } catch (const std::exception& e) {
            AERROR << "convert domain_id error " << e.what();
            return;
        }
    }

    auto part_attr_conf =
        std::make_shared<proto::RtpsParticipantAttr>();
    auto& global_conf = common::GlobalData::Instance()-
        >Config();
    if (global_conf.has_transport_conf() &&
        global_conf.transport_conf().has_participant_attr()) {
        part_attr_conf-
        >CopyFrom(global_conf.transport_conf().participant_attr());
    }

    eprosima::fastrtps::ParticipantAttributes attr;
    attr.rtps.defaultSendPort = send_port;
    attr.rtps.port.domainIDGain =
        static_cast<uint16_t>(part_attr_conf-
        >domain_id_gain());
    attr.rtps.port.portBase = static_cast<uint16_t>
        (part_attr_conf->port_base());
    attr.rtps.use_IP6_to_send = false;

    attr.rtps.builtin.use_SIMPLE_RTPSParticipantDiscoveryProtocol
        = true;
    attr.rtps.builtin.use_SIMPLE_EndpointDiscoveryProtocol =
        true;

    attr.rtps.builtin.m_simpleEDP.use_PublicationReaderANDSubscriber =
        true;
}

```

```

attr.rtps.builtin.m_simpleEDP.use_PublicationWriterANDSubscriber =
    true;
attr.rtps.builtin.domainId = domain_id;
attr.rtps.builtin.leaseDuration.seconds = part_attr_conf->lease_duration();
attr.rtps.builtin.leaseDuration_announcementperiod.seconds =
= part_attr_conf->announcement_period();

attr.rtps.setName(name.c_str());

std::string ip_env("127.0.0.1");
const char* ip_val = ::getenv("CYBER_IP");
if (ip_val != nullptr) {
    ip_env = ip_val;
    if (ip_env.size() == 0) {
        AERROR << "invalid CYBER_IP (an empty string)";
        return;
    }
}
ADEBUG << "cyber ip: " << ip_env;

eprosima::fastrtps::rtps::Locator_t locator;
locator.port = 0;
RETURN_IF(!locator.set_IP4_address(ip_env));

locator.kind = LOCATOR_KIND_UDPv4;

attr.rtps.defaultUnicastLocatorList.push_back(locator);
attr.rtps.defaultOutLocatorList.push_back(locator);

attr.rtps.builtin.metatrafficUnicastLocatorList.push_back(locator);

locator.set_IP4_address(239, 255, 0, 1);

attr.rtps.builtin.metatrafficMulticastLocatorList.push_back(locator);

fastrtps_participant_ =
    eprosima::fastrtps::Domain::createParticipant(attr,

```

```

    listener);
    RETURN_IF_NULL(fastrtps_participant_);

eprosima::fastrtps::Domain::registerType(fastrtps_participant_
_, &type_);
}

// 2.

bool Manager::StartDiscovery(RtpsParticipant* participant) {
    if (participant == nullptr) {
        return false;
    }
    if (is_discovery_started_.exchange(true)) {
        return true;
    }
    if (!CreatePublisher(participant) ||
!CreateSubscriber(participant)) {
        AERROR << "create publisher or subscriber failed.";
        StopDiscovery();
        return false;
    }
    return true;
}

```

fast-RTPS

ParticipantImpl::createPublisher

PDPSimple.cpp 普通查找服务

IntraTransmitter 不确定是不是以下内容

<https://www.developershome.com/sms/intraInterInternationalSMS.asp>

广播

multicast_notifier.cc 广播

```

notify_fd_ = socket(AF_INET, SOCK_DGRAM, 0);
ssize_t nbytes =

```

```
    sendto(notify_fd_, info_str.c_str(), info_str.size(),
0,
        (struct sockaddr*)&notify_addr_,
sizeof(notify_addr_));
```

监听

```
listen_fd_ = socket(AF_INET, SOCK_DGRAM, 0);
bind(listen_fd_, (struct sockaddr*)&listen_addr_,
sizeof(listen_addr_))
ssize_t nbytes = recvfrom(listen_fd_, buf, 32, 0, nullptr,
nullptr);
```

node属性

RoleAttributes

module初始化

如果一个module只是需要传输一些节点，而不需要传递其他任何信息？ module的工作流程是如何的？ module和node的关系如何？

cyber

设置日志等级在”cyber/setup.bash”中设置

```
# for DEBUG log
#export GLOG_minloglevel=-1
#export GLOG_v=4
```

cyber创建进程

cyber通过类std::thread表示单个执行线程。

scheduler

[c++内存模型](http://senlinzhan.github.io/2017/12/04/cpp-memory-order/) [http://senlinzhan.github.io/2017/12/04/cpp-memory-order/]

我们来看下如何切换堆栈，下面这段是汇编代码，实现的功能是保存cpu寄存器的值，并且压入堆栈，然后回复croutine的寄存器和堆栈：

```
.globl ctx_swap
.type ctx_swap, @function
ctx_swap:
    pushq %rdi          // rdi寄存器压入堆栈
    pushq %r12          // r12压入堆栈
    pushq %r13
    pushq %r14
    pushq %r15
    pushq %rbx          // rbx寄存器
    pushq %rbp          // 堆栈底部
    movq %rsp, (%rdi)   // rsp的值赋值给rdi

    movq (%rsi), %rsp   // 出栈
    popq %rbp
    popq %rbx
    popq %r15
    popq %r14
    popq %r13
    popq %r12
    popq %rdi
    ret
```

加入有几千个croutine，当主线程要切换到对应的croutine的时候如何知道对应的堆栈地址，如何跳转？？？原来swap的参数是传入的？？？也就是说地址是通过函数传入的。

```
inline void SwapContext(char** src_sp, char** dest_sp) {
    ctx_swap(reinterpret_cast<void**>(src_sp),
    reinterpret_cast<void**>(dest_sp));
}
```

sche

sche中的task又是什么概念？？？如何去唤醒现在的croutine？？？

SetUpdateFlag

NotifyProcessor

在update中实现croutine状态的转换：

```
inline RoutineState CRoutine::UpdateState() {
    // Synchronous Event Mechanism
    if (state_ == RoutineState::SLEEP &&
        std::chrono::steady_clock::now() > wake_time_) {
        state_ = RoutineState::READY;
        return state_;
    }

    // Asynchronous Event Mechanism
    if (!updated_.test_and_set(std::memory_order_release)) {
        if (state_ == RoutineState::DATA_WAIT || state_ ==
            RoutineState::IO_WAIT) {
            state_ = RoutineState::READY;
        }
    }
    return state_;
}
```

其中只需要释放该锁就可以实现”state_”状态由”DATA_WAIT/IO_WAIT”变为”READY”，因此通过设置”SetUpdateFlag”来实现在事件触发时候调用croutine。

Reference

机器人操作系统（ROS）浅析

[<https://www.cse.sc.edu/~jokane/agitr/%E6%9C%BA%E5%99%A8%E4%BA%BA%E6%93%8D%E4%BD%9C%E7%B3%BB%E7%BB%9F%EF%BC%88ROS%EF%BC%89%E6%B5%85%E6%9E%90.pdf>]

线程与进程的区别及其通信方式 [<https://segmentfault.com/a/1190000008732448>]

cgroups [<https://zh.wikipedia.org/wiki/Cgroups>]

Scheduling (computing) [[https://en.wikipedia.org/wiki/Scheduling_\(computing\)](https://en.wikipedia.org/wiki/Scheduling_(computing))]

Scheduling Algorithms

[http://www.math.nsc.ru/LBRT/k5/Scheduling/BruckerSchedulingAlgorithms_Full.pdf]

参考 <https://github.com/lgsvl/apollo-3.5>

容器内部python运行

1. 在apollo目录下

```
source cyber/setup.bash
```

2. 执行

```
python cyber/python/examples/listener.py  
python cyber/python/examples/talker.py
```

容器外部执行

1. python 环境变量

```
export PYTHONPATH=$PYTHONPATH:/media/data/k8s/apollo  
export PYTHONPATH=$PYTHONPATH:/media/data/k8s/apollo/bazel-bin/cyber/py_wrapper
```

3. 环境变量 export

```
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/lib
```

4. 安装Fast-RTPS 直接拷贝so也可以 /usr/local/fast-rtps/lib/libfastcdr.so.1 /usr/local/fast-rtps/lib/libfastrtps.so.1

5. 拷贝so 在容器里面 cp /usr/local/lib/libglog.so.0 . cp /usr/local/lib/libgflags.so.2.2 . cp /usr/lib/libprotobuf.so.17 . cp /usr/lib/libPocoFoundation.so.9 .

拷贝到另外的机器

1. 设置机器IP A机器 在setup.bash中设置 export

```
CYBER_IP=192.168.1.101 B机器 在setup.bash中设置 export
```

CYBER_IP=192.168.1.102

2. 拷贝文件和依赖库 192.168.1.101 在目录
/home/k8s/apollo/PythonClient
3. 设置环境变量 source setup.bash

```
export PYTHONPATH=$PYTHONPATH:/home/k8s/apollo/bazel-bin/cyber/py_wrapper export
PYTHONPATH=$PYTHONPATH:/home/k8s/apollo/py_proto

export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/lib export
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/home/k8s/apollo/bazel-bin

export
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/home/k8s/apollo/third_party/tf2/lib/

export
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/home/k8s/apollo/bazel-bin/cyber/py_wrapper/_cyber_init.so.runfiles/apollo/_solib_k8/ export
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/home/k8s/apollo/bazel-bin/cyber/py_wrapper/_cyber_node.so.runfiles/apollo/_solib_k8

export PYTHONPATH=$PYTHONPATH:/home/k8s/apollo/third_party

sudo pip install protobuf
```

运行并且测试

A机器

```
python client_example.py -host 192.168.1.102
```

B机器

1. 启动carla ./CarlaUE4.sh -carla-server

2. 容器内部执行 python listener.py

TODO

适配carla和apollo的参数，并且对接发布出去。

dreamview

1. bag转record

```
source /your-path-to-apollo-install-dir/cyber/setup.bash
rosbag_to_record input.bag output.record
```

参考 [https://github.com/ApolloAuto/apollo/blob/master/docs/cyber/CyberRT_Developer_Tools.md]

2. record播放

```
./bazel-
bin/modules/dreamview/backend/simulation_command/sim_cmd
cyber_recorder play -f out.record -s 10 -c
/apollo/canbus/chassis /apollo/localization/pose
/apollo/sensor/gnss/odometry
```

3. HMI status HMI status一直保存在内存中？

ClassLoader

每个ClassLoader加载一个路径，每个路径代表一个so库?
load_ref_count_代表加载library引用计数。

plugin_ref_count_代表创建类的引用计数？

loadLibrary

```
void ClassLoader::loadLibrary()
{
    boost::recursive_mutex::scoped_lock
lock(load_ref_count_mutex_);
    // 每次引用计数加1
    load_ref_count_ = load_ref_count_ + 1;
    // 加载库
    class_loader::impl::loadLibrary(getLibraryPath(), this);
}
```

isLibraryLoaded

判断Library是否加载

```
bool ClassLoader::isLibraryLoaded()
{
    return
class_loader::impl::isLibraryLoaded(getLibraryPath(), this);
}
```

isLibraryLoadedByAnyClassloader

判断Library是否被其它的classloader加载

```
bool ClassLoader::isLibraryLoadedByAnyClassloader()
{
    return
class_loader::impl::isLibraryLoadedByAnybody(getLibraryPath()
);
}
```

systemLibraryFormat

获取库的名称？

```
std::string systemLibraryFormat(const std::string &
library_name)
{
    return systemLibraryPrefix() + library_name +
systemLibrarySuffix();
}
```

unloadLibrary

卸载Library，先判断是否plugin_ref_count_，即是否有创建的类没有销毁，如果没有销毁则不能卸载，如果类已经销毁了，需要加载次数清零，保证加载和卸载次数相等。

创建类

```
template<class Base>
std::shared_ptr<Base> createSharedInstance(const
std::string & derived_class_name)
{
    return std::shared_ptr<Base>(
        createRawInstance<Base>(derived_class_name, true),
        boost::bind(&ClassLoader::onPluginDeletion<Base>, this,
-1));
}
```

registerPlugin

申明静态变量g_register_plugin_UniqueId

注册类，这里是注册派生类到factoryMap，通过派生类可以找到对应的MetaObject<Derived, Base>，创建的时候创建派生类。

```
// Create factory
impl::AbstractMetaObject<Base> * new_factory =
    new impl::MetaObject<Derived, Base>(class_name,
base_class_name);
new_factory-
>addOwningClassLoader(getCurrentlyActiveClassLoader());
new_factory-
>setAssociatedLibraryPath(getCurrentlyLoadingLibraryName());

FactoryMap & factoryMap = getFactoryMapForBaseClass<Base>()
();
factoryMap[class_name] = new_factory;
```

总结

1. 首先通过宏定义注册派生类和基类

```
CLASS_LOADER_REGISTER_CLASS(Derived, Base)
```

2. 注册的派生类和基类会注册到哈希表，key为派生类的名称，value为MetaObject<Derived, Base>，创建的时候会创建派生类。

AbstractMetaObjectBase 对象有一个ClassLoaderVector，是一个加载器数组。 AbstractMetaObjectBase抽象类可以对应多个classloader？？？

一个ClassLoader对应一个library_path，同时还有引用计数和实例化计数。创建了多少个对象则有多少个plugin_ref_count_，每个对象析构的时候会引用计数plugin_ref_count_减去1。同一个类可能被不同的

ClassLoader加载，加载的路径当然也不一样，然后把classloader放到一个数组中。

程序链接几种方式

1. 静态链接
2. 动态链接
3. 动态加载

前2种在编译阶段就要指定，后1种是程序运行过程中动态加载到内存。有插件的系统都采用动态加载的方式来设计。

工作原理

linux

dlopen dlsym dlclose

ELF文件

文件格式 INIT FIN ctor 符号表

通过nm命令查看 readelf

加载原理

extern c

这样符号表的函数名称不会变化。

静态变量

静态变量加载的时候会动态初始化

1. elf文件中的init和ctor段，也可以用`_attribute_` 宏来实现启动的时候就执行，然后再把控制权交给main函数

全部类继承一个基类，实现了`new object`方法，然后再注册工厂类到全局变量，这样就可以创建变量了。

注册

1. 注册通过宏定义实现
2. 加载的时候动态构造，并且注册到全局变量
3. 类实现了创建方法

创建

1. 找到对应的类，然后调用创建方法
2. 每次销毁的时候引用计数减1

加载

采用`dlopen`加载，可以反复加载，但是返回同一个指针，卸载的时候次数相等？`classloader`对原生重复加载做了拦截？

卸载

1. 判断是否有引用计数，有对象存活则不能卸载
2. 没有对象存活，则卸载库，这里的引用计数看起来没什么用（采用`bool`型就可以解决）？

总之上述2种方法都是找一个绝对的物理地址去调用，函数在代码段，通过名称去调用，静态变量在数据段，然后去调用。

静态变量作用域

1. file scope只作用在单个文件，全局变量的作用域，需要加上extern
2. 函数中的static变量作用域只在函数可见，并且在函数调用的时候初始化
3. static变量在动态库卸载的时候会清零，再次加载的时候会重新初始化（apollo中和ros中的区别，需要做一些实验去验证是否正确）为什么函数中的全局变量没有被卸载？？
4. static变量的生命周期，创建一直保存到程序结束，对动态加载程序创建的变量好像是卸载的时候删除？
5. static变量线程安全，c++11之后是线程安全的

croutine

libco

libco协程库分析

目录

co_closure.h 定义了一些函数宏，具体的作用是什么？ co_comm.h 可重入锁， lockguard co_epoll.h 网络接口 co_hook_sys_call.cpp hook系统调用 co_routine_inner.h co_routine_specific.h

coctx_swap.S 栈切换 co_routine.h
coctx.h

co_create

创建协程

```
int co_create( stCoRoutine_t **ppco,const stCoRoutineAttr_t
*attr,pfn_co_routine_t pfn,void *arg )
{
    if( !co_get_curr_thread_env() )
    {
        co_init_curr_thread_env();
    }
    stCoRoutine_t *co = co_create_env(
co_get_curr_thread_env(), attr, pfn,arg );
    *ppco = co;
    return 0;
}
```

co_resume

```

void co_resume( stCoRoutine_t *co )
{
    stCoRoutineEnv_t *env = co->env;
    stCoRoutine_t *lpCurrRoutine = env->pCallStack[ env-
>iCallStackSize - 1 ];
    if( !co->cStart )
    {
        coctx_make( &co->ctx,
(coctx_pfn_t)CoRoutineFunc,co,0 );
        co->cStart = 1;
    }
    env->pCallStack[ env->iCallStackSize++ ] = co;
    co_swap( lpCurrRoutine, co );
}

}

```

co_yield

```

void co_yield_env( stCoRoutineEnv_t *env )
{
    stCoRoutine_t *last = env->pCallStack[ env-
>iCallStackSize - 2 ];
    stCoRoutine_t *curr = env->pCallStack[ env-
>iCallStackSize - 1 ];

    env->iCallStackSize--;
    co_swap( curr, last );
}

```

co_release

```

void co_release( stCoRoutine_t *co )
{
    co_free( co );
}

```

DDS协议介绍

主要是理解epoll实现，以及为什么epoll会快？？？顺便了解下select的实现？

Session

Poller

PollHandler

Dig into Apollo - Cyber

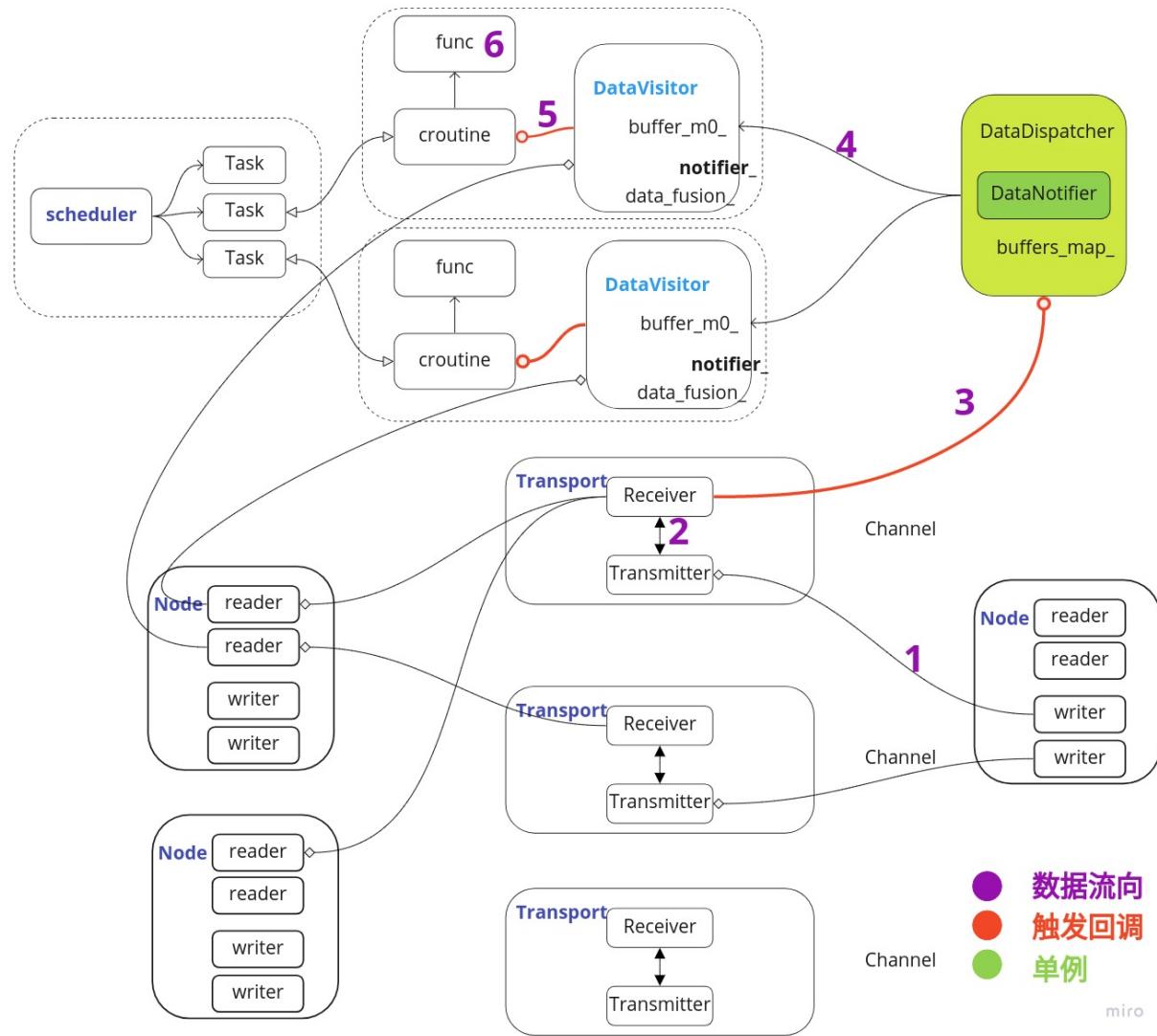
[license](#) [MIT](#)

写在之前，之前的分析都是一些源码级别的分析，发现一开始就深入源码，很容易陷进去，特别是模块非常多的情况下，需要看很多遍才能理解清楚。要写出更容易理解的文档，需要的不是事无巨细的分析代码，更主要的是能够把复杂的东西抽象出来，变为简单的东西。一个很简答的例子是画函数调用流程图很简单，但是要把流程图转换成框图却很难。

数据处理流程

我们先看下cyber中整个的数据处理流程，通过理解数据流程中各个模块如何工作，来搞清楚每个模块的作用，然后我们再接着分析具体的

模块。



如上图所示，cyber的数据流程可以分为6个过程。

1. Node节点中的Writer往通道里面写数据。
2. 通道中的Transmitter发布消息，通道中的Receiver订阅消息。
3. Receiver接收到消息之后，触发回调，触发DataDispatcher进行消息分发。
4. DataDispatcher接收到消息后，把消息放入CacheBuffer，并且触发Notifier，通知对应的DataVisitor处理消息。
5. DataVisitor把数据从CacheBuffer中读出，并且进行融合，然后通过notifier_唤醒对应的协程。

6. 协程执行对应的注册回调函数，进行数据处理，处理完成之后接着进入睡眠状态。

对数据流程有整体的认识之后，下面我们在分析具体的每个模块，我们还是按照功能划分。

整体介绍

首先我们对cyber中各个模块做一个简单的介绍，之后再接着分析。实际上我们只要搞清楚了下面一些概念之间的关系，就基本上理解清楚了整个Cyber的数据流程。

1.Component和Node的关系

Component是cyber中封装好的数据处理流程，对用户来说，对应自动驾驶中的Planning Component, Perception Component等，目的是帮助我们更方便的订阅和处理消息。实际上Component模块在加载之后会执行”Initialize()”函数，这是个隐藏的初始化过程，对用户不可见。

在”Initialize”中，Component会创建一个Node节点，概念上对应ROS的节点，每个**Component**模块只能有一个**Node**节点，也就是说每个Component模块有且只能有一个节点，在Node节点中进行消息订阅和发布。

2.Node和Reader\Writer的关系

在Node节点中可以创建Reader订阅消息，也可以创建Writer发布消息，每个Node节点中可以创建多个Reader和Writer。

3.Reader和Receiver,Writer和Transmitter,Channel的关系

一个Channel对应一个Topic，概念上对应ROS的消息通道，每个Topic都是唯一的。而Channel中包括一个发送器(Transmitter)和接收器(Receiver)，通过Receiver接收消息，通过Transmitter发送消息。

一个Reader只能订阅一个通道的消息，如果一个Node需要订阅多个通道的消息，需要创建多个Reader。同理一个Writer也只能发布一个通道

的消息，如果需要发布多个消息，需要创建多个Writer。Reader中调用Receiver订阅消息，而Writer通过Transmitter发布消息。

4. Receiver, DataDispatcher和DataVisitor的关系

每一个Receiver接收到消息之后，都会触发回调，回调中触发DataDispatcher（消息分发器）发布消息，DataDispatcher是一个单例，所有的数据分发都在数据分发器中进行，DataDispatcher会把数据放到对应的缓存中，然后Notify(通知)对应的协程（实际上这里调用的是DataVisitor中注册的Notify）去处理消息。

DataVisitor（消息访问器）是一个辅助的类，一个数据处理过程对应一个**DataVisitor**，通过在**DataVisitor**中注册**Notify**（唤醒对应的协程，协程执行绑定的回调函数），并且注册对应的**Buffer**到**DataDispatcher**，这样在DataDispatcher的时候会通知对应的DataVisitor去唤醒对应的协程。

也就是说DataDispatcher（消息分发器）发布对应的消息到DataVisitor，DataVisitor（消息访问器）唤醒对应的协程，协程中执行绑定的数据处理回调函数。

5. DataVisitor和Croutine的关系

实际上DataVisitor中的Notify是通过唤醒协程（为了方便理解也可以理解为线程，可以理解为你有一个线程池，通过线程池绑定数据处理函数，数据到来之后就唤醒对应的线程去执行任务），每个协程绑定了一个数据处理函数和一个DataVisitor，数据到达之后，通过DataVisitor中的Notify唤醒对应的协程，执行数据处理回调，执行完成之后协程进入休眠状态。

6. Scheduler, Task和Croutine

通过上述分析，数据处理的过程实际上就是通过协程完成的，每一个协程被称为一个Task，所有的Task(任务)都由Scheduler进行调度。从这里我们可以分析得出实际上Cyber的实时调度由协程去保障，并且可以灵活的通过协程去设置对应的调度策略，当然协程依赖于进程，Apollo在linux中设置进程的优先级为实时轮转，先保障进程的优先级最高，然后内部再通过协程实现对应的调度策略。

协程和线程的优缺点这里就不展开了，这里有一个疑问是协程不能被终止，除非协程主动退出，这里先留一个伏笔，后面我们再分析协程的调度问题。

上述就是各个概念之间的关系，上述介绍对理解数据的流程非常有帮助，希望有时间的时候，大家可以画一下对应的数据流程图和关系。

Component介绍

我们首先需要清楚一点，component实际上是cyber为了帮助我们特意实现的对象，component加载的时候会自动帮我们创建一个node，通过node来订阅和发布对应的消息，每个component有且只能对应一个node。

component对用户提供2个接口”Init()”和”Proc()”，用户在Init中进行初始化，在”Proc”中接收Topic执行具体的算法。对用户隐藏的部分包括component的”Initialize()”初始化，以及”Process()”调用执行。

component还可以动态的加载和卸载，这也可以对应到在dreamviewer上动态的打开关系模块。下面我们先大致介绍下component的工作流程，然后再具体介绍各个模块。

component工作流程

component的工作流程大致如下：

1. 通过继承”cyber::Component”，用户自定义一个模块，并且实现”Init()”和”Proc()”函数。编译生成”.so”文件。
2. 通过classloader加载component模块到内存，创建component对象，调用”Initialize()”初始化。（Initialize中会调用Init）
3. 创建协程任务，并且注册”Process()”回调，当数据到来的时候，唤醒对象的协程任务执行”Process()”处理数据。（Process会调用Proc）综上所述，component帮助用户把初始化和数据收发的流程进行了封装，减少了用户的工作量，component封装了整个数据的收发流程，component本身并不是单独的一个线程执行，模块的初始化都在主线程中执行，而具体的任务则是在协程池中执行。

cyber入口

cyber的入口在”cyber/mainboard/mainboard.cc”中，主函数中先进行cyber的初始化，然后启动cyber模块，然后运行，一直等到系统结束。

```
int main(int argc, char** argv) {
    // 1. 解析参数
    ModuleArgument module_args;
    module_args.ParseArgument(argc, argv);

    // 2. 初始化cyber
    apollo::cyber::Init(argv[0]);

    // 3. 启动cyber模块
    ModuleController controller(module_args);
    if (!controller.Init()) {
        controller.Clear();
        AERROR << "module start error.";
        return -1;
    }

    // 4. 等待直到程序退出
    apollo::cyber::WaitForShutdown();
    controller.Clear();
    return 0;
}
```

Cyber实现的功能

cyber提供的功能概括起来包括2方面：

1. 消息队列 - 主要作用是接收和发送各个节点的消息，涉及到消息的发布、订阅以及消息的buffer缓存等。
2. 实时调度 - 主要作用是调度处理上述消息的算法模块，保证算法模块能够实时调度处理消息。

除了这2方面的工作，cyber还需要提供以下2部分的工作：

1. 用户接口 - 提供灵活的用户接口

2. 工具 - 提供一系列的工具，例如bag包播放，点云可视化，消息监控等



总结起来就是，cyber是一个分布式收发消息，和调度框架，同时对外提供一系列的工具和接口来辅助开发和定位问题。其中cyber对比ROS来说有很多优势，唯一的劣势是cyber相对ROS没有丰富的算法库支持。

下面我们开始分析整个cyber的代码流程。

cyber入口

cyber的入口在”cyber/mainboard”目录中：

```
|-- mainboard.cc          // 主函数
|-- module_argument.cc    // 模块输入参数
|-- module_argument.h
|-- module_controller.cc  // 模块加载，卸载
└-- module_controller.h
```

mainboard中的文件比较少，也很好理解，我们先从”mainboard.cc”中开始分析：

```
int main(int argc, char **argv) {
    google::SetUsageMessage("we use this program to load dag
and run user apps.");

    // 注册信号量，当出现系统错误时，打印堆栈信息
    signal(SIGSEGV, SigProc);
    signal(SIGABRT, SigProc);

    // parse the argument
    // 解析参数
    ModuleArgument module_args;
    module_args.ParseArgument(argc, argv);

    // initialize cyber
    // 初始化cyber
    apollo::cyber::Init(argv[0]);
```

```

// start module
// 加载模块
ModuleController controller(module_args);
if (!controller.Init()) {
    controller.Clear();
    AERROR << "module start error.";
    return -1;
}

// 等待cyber关闭
apollo::cyber::WaitForShutdown();
// 卸载模块
controller.Clear();
AINFO << "exit mainboard.";

return 0;
}

```

上述是”mainboard.cc”的主函数，下面我们重点介绍下具体的过程。

打印堆栈

在主函数中注册了信号量”SIGSEGV”和”SIGABRT”，当系统出现错误的时候（空指针，异常）等，这时候就会触发打印堆栈信息，也就是说系统报错的时候打印出错的堆栈，方便定位问题，[参考](#) [<https://www.runoob.com/cplusplus/cpp-signal-handling.html>]。

```

// 注册信号量，当出现系统错误时，打印堆栈信息
signal(SIGSEGV, SigProc);
signal(SIGABRT, SigProc);

```

打印堆栈的函数在”SigProc”中实现，而打印堆栈的实现是通过”backtrace”实现，[参考](#)

[https://www.gnu.org/software/libc/manual/html_node/Backtraces.html]。

```

// 打印堆栈信息
void ShowStack() {
    int i;
    void *buffer[STACK_BUF_LEN];
    int n = backtrace(buffer, STACK_BUF_LEN);

```

```

char **symbols = backtrace_symbols(buffer, n);
AINFO << "=====call stack begin=====";
for (i = 0; i < n; i++) {
    AINFO << symbols[i];
}
AINFO << "=====call stack end=====";
}

```

解析参数

解析参数是在”ModuleArgument”类中实现的，主要是解析加载DAG文件时候带的参数。

```

void ModuleArgument::ParseArgument(const int argc, char*
const argv[]) {
    // 二进制模块名称
    binary_name_ = std::string(basename(argv[0]));
    // 解析参数
    GetOptions(argc, argv);

    // 如果没有process_group_ 和sched_name_，则赋值为默认值
    if (process_group_.empty()) {
        process_group_ = DEFAULT_process_group_;
    }

    if (sched_name_.empty()) {
        sched_name_ = DEFAULT_sched_name_;
    }

    // 如果有，则设置对应的参数
    GlobalData::Instance()->SetProcessGroup(process_group_);
    GlobalData::Instance()->SetSchedName(sched_name_);
    AINFO << "binary_name_ is " << binary_name_ << ", "
process_group_ is "
        << process_group_ << ", has " <<
dag_conf_list_.size() << " dag conf";

    // 打印dag_conf配置，这里的dag是否可以设置多个？？
    for (std::string& dag : dag_conf_list_) {
        AINFO << "dag_conf: " << dag;
    }
}

```

模块加载

在”ModuleController”实现cyber模块的加载，
在”ModuleController::Init()”中调用”LoadAll()”来加载所有模块，我们
接着看cyber是如何加载模块。

1. 首先是找到模块的路径

```
if (module_config.module_library().front() == '/') {
    load_path = module_config.module_library();
} else {
    load_path =
        common::GetAbsolutePath(work_root,
module_config.module_library());
}
```

2. 通过”class_loader_manager_”加载模块，后面我们会接着分 析”ClassLoaderManager”的具体实现，加载好对应的类之后在创建 对应的对象，并且初始化对象（调用对象的Initialize()方法，也就是说所有的cyber模块都是通过Initialize()方法启动的，后面我们会接着分析Initialize具体干了什么）。

这里的”classloader”其实类似java中的classloader，即java虚拟机在运行时加载对应的类，并且实例化对象。

cyber中其实也是实现了类型通过动态加载并且实例化类的功能，好处是可以动态加载和关闭单个cyber模块(定位，感知，规划等)，也就是在dreamview中的模块开关按钮，实际上就是动态的加载和卸载对应的模块。

```
// 通过类加载器加载load_path下的模块
class_loader_manager_.LoadLibrary(load_path);

// 加载模块
for (auto& component : module_config.components()) {
    const std::string& class_name = component.class_name();
    // 创建对象
    std::shared_ptr<ComponentBase> base =
        class_loader_manager_.CreateClassObj<ComponentBase>
(class_name);
    // 调用对象的Initialize方法
```

```

        if (base == nullptr || !base-
>Initialize(component.config())) {
    return false;
}
component_list_.emplace_back(std::move(base));
}

// 加载定时器模块
for (auto& component : module_config.timer_components())
{
    const std::string& class_name = component.class_name();
    std::shared_ptr<ComponentBase> base =
        class_loader_manager_.CreateClassObj<ComponentBase>
(class_name);
    if (base == nullptr || !base-
>Initialize(component.config())) {
        return false;
}
component_list_.emplace_back(std::move(base));
}

```

上述就是cyber mainboard的整个流程，cyber main函数中先解析dag参数，然后根据解析的参数，通过类加载器动态的加载对应的模块，然后调用Initialize方法初始化模块。

下面我们会接着分析类加载器(ClassLoaderManager)

类加载器(class_loader)

类加载器的作用就是动态的加载动态库，然后实例化对象。我们先来解释下，首先apollo中的各个module都会编译为一个动态库，拿planning模块来举例子，在”planning/dag/planning.dag”中，会加载：

```

module_config {
    module_library : "/apollo/bazel-
bin/modules/planning/libplanning_component.so"

```

也就是说，apollo中的模块都会通过类加载器以动态库的方式加载，然后实例化，之后再调用Initialize方法初始化。也就是说，我们讲清楚下面3个问题，也就是讲清楚了类加载器的原理。

1. cyber如何加载apollo模块?
2. 如何实例化模块?
3. 如何初始化模块?

目录结构

类加载器的实现在”cyber/class_loader”目录中，通过”Poco/SharedLibrary.h”库来实现动态库的加载，关于Poco动态库的加载可以[参考](https://pocoproject.org/docs/Poco.SharedLibrary.html) [<https://pocoproject.org/docs/Poco.SharedLibrary.html>]

```
├── BUILD                  // 编译文件
├── class_loader.cc         // 类加载器
└── class_loader.h
├── class_loader_manager.cc // 类加载器管理
└── class_loader_manager.h
├── class_loader_register_macro.h // 类加载器注册宏定义
└── utility
    ├── class_factory.cc      // 类工厂
    ├── class_factory.h
    ├── class_loader_utility.cc // 类加载器工具类
    └── class_loader_utility.h
```

类加载器(ClassLoader)

我们先从”class_loader.h”开始看起，首先我们分析下”class_loader”实现的具体方法：

```
class ClassLoader {
public:
    explicit ClassLoader(const std::string& library_path);
    virtual ~ClassLoader();

    // 库是否已经加载
    bool IsLibraryLoaded();
    // 加载库
    bool LoadLibrary();
    // 卸载库
    int UnloadLibrary();
    // 获取库的路径
    const std::string GetLibraryPath() const;
```

```

// 获取累名称
template <typename Base>
std::vector<std::string> GetValidClassNames();
// 实例化类对象
template <typename Base>
std::shared_ptr<Base> CreateClassObj(const std::string&
class_name);
// 类是否有效
template <typename Base>
bool IsClassValid(const std::string& class_name);

private:
// 当类删除
template <typename Base>
void OnClassObjDelete(Base* obj);

private:
// 类的路径
std::string library_path_;
// 类加载引用次数
int loadlib_ref_count_;
// 类加载引用次数锁
std::mutex loadlib_ref_count_mutex_;
// 类引用次数
int classobj_ref_count_;
// 类引用次数锁
std::mutex classobj_ref_count_mutex_;
};


```

可以看到类加载器主要是提供了加载类，卸载类和实例化类的接口。实际上加载类和卸载类的实现都比较简单，都是调用”utility”类中的实现，我们暂时先放一边，先看下实例化对象的实现。

```

template <typename Base>
std::shared_ptr<Base> ClassLoader::CreateClassObj(
    const std::string& class_name) {
    // 加载库
    if (!IsLibraryLoaded()) {
        LoadLibrary();
    }

    // 根据类名称创建对象

```

```

Base* class_object = utility::CreateClassObj<Base>
(class_name, this);

// 类引用计数加1
std::lock_guard<std::mutex> lck(classobj_ref_count_mutex_);
classobj_ref_count_ = classobj_ref_count_ + 1;
// 指定类的析构函数
std::shared_ptr<Base> classObjSharePtr(
    class_object,
std::bind(&ClassLoader::OnClassObjDelete, this,
          std::placeholders::_1));
return classObjSharePtr;
}

```

可以看到创建类的时候，类引用计数加1，并且绑定类的析构函数(OnClassObjDelete)，删除对象的时候让类引用计数减1。

```

template <typename Base>
void ClassLoader::OnClassObjDelete(Base* obj) {
    if (nullptr == obj) {
        return;
    }

    std::lock_guard<std::mutex> lck(classobj_ref_count_mutex_);
    delete obj;
    --classobj_ref_count_;
}

```

我们先简单的分析下ClassLoaderManager，最后再分析utility。

ClassLoaderManager

类加载器管理实际上是管理不同的classloader，而不同的libpath对应不同的classloader。ClassLoaderManager主要的数据结构其实如下：

```
std::map<std::string, ClassLoader*> libpath_loader_map_;
```

其中”libpath_loader_map_”为map结构，在”LoadLibrary”的时候赋值，**key为library_path，而value为ClassLoader.**

```

bool ClassLoaderManager::LoadLibrary(const std::string&
library_path) {
    std::lock_guard<std::mutex> lck(libpath_loader_map_mutex_);
    if (!IsLibraryValid(library_path)) {
        // 赋值
        libpath_loader_map_[library_path] =
            new class_loader::ClassLoader(library_path);
    }
    return IsLibraryValid(library_path);
}

```

也就是说”ClassLoaderManager”对ClassLoader进行保存和管理。

最后我们分析下utility具体的实现，utility分为2部分，一部分为ClassFactory，一部分为工具函数（class_loader_utility.cc）

ClassFactory

可以看到有如下继承关系”ClassFactory -> AbstractClassFactory -> AbstractClassFactoryBase”，其

中”ClassFactory”和”AbstractClassFactory”为模板类，主要的实现
在”AbstractClassFactoryBase”中，我们逐个分析：

首先是类初始化，指定了”relative_library_path_”，”base_class_name_”，
”class_name_”

```

AbstractClassFactoryBase::AbstractClassFactoryBase(
    const std::string& class_name, const std::string&
base_class_name)
: relative_library_path_(""),
  base_class_name_(base_class_name),
  class_name_(class_name) {}

```

设置OwnedClassLoader，而”RemoveOwnedClassLoader”同理。

```

void
AbstractClassFactoryBase::AddOwnedClassLoader(ClassLoader*
loader) {
    if (std::find(relative_class_loaders_.begin(),
relative_class_loaders_.end(),
loader) == relative_class_loaders_.end()) {

```

```

        relative_class_loaders_.emplace_back(loader);
    }
}

```

classloader是否属于该classFactory

```

bool AbstractClassFactoryBase::IsOwnedBy(const ClassLoader*
loader) {
    std::vector<ClassLoader*>::iterator itr = std::find(
        relative_class_loaders_.begin(),
relative_class_loaders_.end(), loader);
    return itr != relative_class_loaders_.end();
}

```

也是说ClassFactory能够生产一个路径下的所有类，一个ClassFactory可能有好几个ClassLoader，分为base_class_name和class_name。

工具函数

接下来我们看”class_loader_utility.cc”的实现，文件中实现了很多函数，这个分析如下：

创建对象(CreateClassObj)的具体实现如下，先找到类对应的factory，然后通过factory创建对象。

```

template <typename Base>
Base* CreateClassObj(const std::string& class_name,
ClassLoader* loader) {
    GetClassFactoryMapMutex().lock();
    ClassClassFactoryMap& factoryMap =
        GetClassFactoryMapByBaseClass(typeid(Base).name());
    AbstractClassFactory<Base>* factory = nullptr;
    if (factoryMap.find(class_name) != factoryMap.end()) {
        factory =
dynamic_cast<utility::AbstractClassFactory<Base>*>(
            factoryMap[class_name]);
    }
    GetClassFactoryMapMutex().unlock();

    Base* classobj = nullptr;
    if (factory && factory->IsOwnedBy(loader)) {

```

```

        classobj = factory->CreateObj();
    }

    return classobj;
}

```

注册类到factory

```

template <typename Derived, typename Base>
void RegisterClass(const std::string& class_name,
                    const std::string& base_class_name) {
    AINFO << "registerclass:" << class_name << "," <<
    base_class_name << ","
                    << GetCurLoadingLibraryName();

    utility::AbstractClassFactory<Base>* new_class_factory_obj
    =
        new utility::ClassFactory<Derived, Base>(class_name,
    base_class_name);
    new_class_factory_obj-
    >AddOwnedClassLoader(GetCurActiveClassLoader());
    new_class_factory_obj-
    >SetRelativeLibraryPath(GetCurLoadingLibraryName());

    GetClassFactoryMapMutex().lock();
    ClassClassFactoryMap& factory_map =
        GetClassFactoryMapByBaseClass(typeid(Base).name());
    factory_map[class_name] = new_class_factory_obj;
    GetClassFactoryMapMutex().unlock();
}

```

查找classloader中所有类的名称

```

template <typename Base>
std::vector<std::string> GetValidclassNames(ClassLoader* loader) {
    std::lock_guard<std::recursive_mutex>
lck(GetClassFactoryMapMutex());

    ClassClassFactoryMap& factoryMap =
        GetClassFactoryMapByBaseClass(typeid(Base).name());
    std::vector<std::string> classes;

```

```

        for (auto& class_factory : factoryMap) {
            AbstractClassFactoryBase* factory = class_factory.second;
            if (factory && factory->IsOwnedBy(loader)) {
                classes.emplace_back(class_factory.first);
            }
        }

        return classes;
    }
}

```

加载类，通过指定的classloader加载指定路径下的库。

```

bool LoadLibrary(const std::string& library_path,
ClassLoader* loader) {
    // 类是否已经被加载，如果被加载则对应的class_factory加上依赖的
    class_loader
    if (IsLibraryLoadedByAnybody(library_path)) {
        ClassFactoryVector lib_class_factory_objs =
            GetAllClassFactoryObjectsOfLibrary(library_path);
        for (auto& class_factory_obj : lib_class_factory_objs) {
            class_factory_obj->AddOwnedClassLoader(loader);
        }
        return true;
    }

    PocoLibraryPtr poco_library = nullptr;
    static std::recursive_mutex loader_mutex;
    {
        std::lock_guard<std::recursive_mutex> lck(loader_mutex);

        try {
            // 设置当前激活的classloader，当前加载库路径
            SetCurActiveClassLoader(loader);
            SetCurLoadingLibraryName(library_path);
            // 创建poco_library
            poco_library = PocoLibraryPtr(new
Poco::SharedLibrary(library_path));
        } catch (const Poco::LibraryLoadException& e) {
            SetCurLoadingLibraryName("");
            SetCurActiveClassLoader(nullptr);
            AERROR << "poco LibraryLoadException: " << e.message();
        } catch (const Poco::LibraryAlreadyLoadedException& e) {
            SetCurLoadingLibraryName("");
        }
    }
}

```

```

        SetCurActiveClassLoader(nullptr);
        AERROR << "poco LibraryAlreadyLoadedException: " <<
e.message();
    } catch (const Poco::NotFoundException& e) {
        SetCurLoadingLibraryName("");
        SetCurActiveClassLoader(nullptr);
        AERROR << "poco NotFoundException: " << e.message();
    }

    SetCurLoadingLibraryName("");
    SetCurActiveClassLoader(nullptr);
}

if (poco_library == nullptr) {
    AERROR << "poco shared library failed: " << library_path;
    return false;
}

auto num_lib_objs =
GetAllClassFactoryObjectsOfLibrary(library_path).size();
if (num_lib_objs == 0) {
    AWARN << "Class factory objs counts is 0, maybe
registerclass failed.";
}

std::lock_guard<std::recursive_mutex>
lck(GetLibPathPocoShareLibMutex());
LibpathPocoLibVector& opened_libraries =
GetLibPathPocoShareLibVector();
// 保存加载路径和对应的poco_library
opened_libraries.emplace_back(
    std::pair<std::string, PocoLibraryPtr>(library_path,
poco_library));
return true;
}

// TODO(zero): 下面这2个问题目前还没有想到答案

```

1. 通过Poco::SharedLibrary(path)动态加载类，但是加载的类保存在对应的opened_libraries中，又是如何利用这个opened_libraries的呢？？？

2. 先通过”`poco_library = PocoLibraryPtr(new Poco::SharedLibrary(library_path))`”加载类，但是最后直接通过`new`创建出类”`Base* CreateObj() const { return new ClassObject; }`”是通过如何实现的呢？？？

上面我们分析了classloader动态的加载并且创建类，而在mainboard中通过动态的加载module，并且调用模块的Initialize方法，实现模块的初始化。下面我们看下模块的初始化过程。

component(cyber组件)

我们先看下component的目录结构。

目录结构

可以看到cyber组件分为2类： 普通组件和定时组件，而二者都继承至基础组件。

```
.  
├── BUILD  
├── component_base.h          // 基础组件  
├── component.h               // 组件  
├── component_test.cc  
├── timer_component.cc        // 定时组件  
└── timer_component.h  
    └── timer_component_test.cc
```

基础组件

我们先看下基础组件中实现了什么，也就是”`component_base.h`”中实现了什么？”`component_base.h`”实现了”`ComponentBase`”类，下面我们逐步分析”`ComponentBase`”类的public方法。

Initialize方法

Initialize方法在派生类中重写了，这里有2个Initialize方法，分别对应上述所说的2种类型的组件。

```

    virtual bool Initialize(const ComponentConfig& config) {
        return false;
    }
    virtual bool Initialize(const TimerComponentConfig& config)
    { return false; }

```

Shutdown方法 用于关闭cyber模块。

```

virtual void Shutdown() {
    if (is_shutdown_.exchange(true)) {
        return;
    }

    Clear();
    for (auto& reader : readers_) {
        reader->Shutdown();
    }
    scheduler::Instance()->RemoveTask(node_->Name());
}

```

GetProtoConfig方法

获取protobuf格式的配置

```

template <typename T>
bool GetProtoConfig(T* config) const {
    return common::GetProtoFromFile(config_file_path_,
config);
}

```

看完公有方法，下面我们看下私有方法。有些简单的方法这里就不详细说了，主要看下”LoadConfigFiles方法”，有2个”LoadConfigFiles”方法这里只介绍第一个：

LoadConfigFiles方法

```

void LoadConfigFiles(const ComponentConfig& config) {
    // 获取配置文件路径
    if (!config.config_file_path().empty()) {
        if (config.config_file_path()[0] != '/') {
            config_file_path_ =
common::GetAbsolutePath(common::WorkRoot(),
config.config_file_path());

```

```

    } else {
        config_file_path_ = config.config_file_path();
    }
}

// 设置flag文件路径
if (!config.flag_file_path().empty()) {
    std::string flag_file_path = config.flag_file_path();
    if (flag_file_path[0] != '/') {
        flag_file_path =
            common::GetAbsolutePath(common::WorkRoot(),
flag_file_path);
    }
    google::SetCommandLineOption("flagfile",
flag_file_path.c_str());
}
}

```

私有成员变量

最后我们在分析下私有成员变量，也就是说每个组件(component)会自动创建一个节点(node)，并且可以挂载多个reader。

```

std::atomic<bool> is_shutdown_ = {false};
std::shared_ptr<Node> node_ = nullptr;
std::string config_file_path_ = "";
std::vector<std::shared_ptr<ReaderBase>> readers_;

```

下面我们开始分析component组件，也就是Component类。

Component类

Component类都需要实现”Initialize”和”Process”2个方法，所以planning, routing, perception等模块都需要实现这2个方法。

```

template <typename M0>
class Component<M0, NullType, NullType, NullType> : public
ComponentBase {
public:
    Component() {}
    ~Component() override {}
    bool Initialize(const ComponentConfig& config) override;
}

```

```

    bool Process(const std::shared_ptr<M0>& msg);

private:
    virtual bool Proc(const std::shared_ptr<M0>& msg) = 0;
};

```

我们接着看下这2个方法是如何实现的，先看”Process”方法。

Process方法

可以看到Process方法比较简单，先判断模块是否关闭，然后执行”Proc”方法。

```

template <typename M0, typename M1>
bool Component<M0, M1, NullType, NullType>::Process(
    const std::shared_ptr<M0>& msg0, const
    std::shared_ptr<M1>& msg1) {
    if (is_shutdown_.load()) {
        return true;
    }
    return Proc(msg0, msg1);
}

```

Initialize方法

```

template <typename M0, typename M1>
bool Component<M0, M1, NullType, NullType>::Initialize(
    const ComponentConfig& config) {
    // 创建node节点
    node_.reset(new Node(config.name()));
    // 加载配置
    LoadConfigFiles(config);

    // 订阅消息数和reader个数要匹配
    if (config.readers_size() < 2) {
        AERROR << "Invalid config file: too few readers.";
        return false;
    }

    // 初始化，在基类(ComponentBase)中实现
    if (!Init()) {
        AERROR << "Component Init() failed.";
        return false;
    }
}

```

```
    bool is_reality_mode = GlobalData::Instance()->IsRealityMode();
    // 创建reader1
    ReaderConfig reader_cfg;
    reader_cfg.channel_name = config.readers(1).channel();

    reader_cfg.qos_profile.CopyFrom(config.readers(1).qos_profile());
    reader_cfg.pending_queue_size =
config.readers(1).pending_queue_size();

    auto reader1 = node_->template CreateReader<M1>(reader_cfg);

    // 创建reader0
    reader_cfg.channel_name = config.readers(0).channel();

    reader_cfg.qos_profile.CopyFrom(config.readers(0).qos_profile());
    reader_cfg.pending_queue_size =
config.readers(0).pending_queue_size();

    std::shared_ptr<Reader<M0>> reader0 = nullptr;
    // is_reality_mode模式则直接创建
    if (cyber_likely(is_reality_mode)) {
        reader0 = node_->template CreateReader<M0>(reader_cfg);
    } else {
        // 如果不是则创建回调函数
        std::weak_ptr<Component<M0, M1>> self =
            std::dynamic_pointer_cast<Component<M0, M1>>
(shared_from_this());
    }

    auto blocker1 = blocker::BlockerManager::Instance()->GetBlocker<M1>(
        config.readers(1).channel());

    auto func = [self, blocker1](const std::shared_ptr<M0>& msg0) {
        auto ptr = self.lock();
        if (ptr) {
            if (!blocker1->IsPublishedEmpty()) {
                auto msg1 = blocker1->GetLatestPublishedPtr();
```

```

        ptr->Process(msg0, msg1);
    }
} else {
    AERROR << "Component object has been destroyed.";
}
};

reader0 = node_->template CreateReader<M0>(reader_cfg,
func);
}
if (reader0 == nullptr || reader1 == nullptr) {
    AERROR << "Component create reader failed.";
    return false;
}
// 保存readers
readers_.push_back(std::move(reader0));
readers_.push_back(std::move(reader1));

if (cyber_unlikely(!is_reality_mode)) {
    return true;
}

auto sched = scheduler::Instance();
std::weak_ptr<Component<M0, M1>> self =
    std::dynamic_pointer_cast<Component<M0, M1>>
(shared_from_this());
auto func = [self](const std::shared_ptr<M0>& msg0,
                    const std::shared_ptr<M1>& msg1) {
    auto ptr = self.lock();
    if (ptr) {
        ptr->Process(msg0, msg1);
    } else {
        AERROR << "Component object has been destroyed.";
    }
};

std::vector<data::VisitorConfig> config_list;
for (auto& reader : readers_) {
    config_list.emplace_back(reader->ChannelId(), reader-
>PendingQueueSize());
}
auto dv = std::make_shared<data::DataVisitor<M0, M1>>
(config_list);

```

```

// 创建协程类
croutine::RoutineFactory factory =
    croutine::CreateRoutineFactory<M0, M1>(func, dv);
return sched->CreateTask(factory, node_->Name());
}

```

component动态加载

cyber主函数在”ModuleController::Init()”进行模块的加载，具体的加载过程在”ModuleController::LoadModule”中。

```

bool ModuleController::LoadModule(const DagConfig&
dag_config) {
    const std::string work_root = common::WorkRoot();

    for (auto module_config : dag_config.module_config()) {
        // 1. 加载动态库
        class_loader_manager_.LoadLibrary(load_path);

        // 2. 加载消息触发模块
        for (auto& component : module_config.components()) {
            const std::string& class_name = component.class_name();
            // 3. 创建对象
            std::shared_ptr<ComponentBase> base =
                class_loader_manager_.CreateClassObj<ComponentBase>
(class_name);
            // 4. 调用对象的Initialize方法
            if (base == nullptr || !base-
>Initialize(component.config())) {
                return false;
            }
            component_list_.emplace_back(std::move(base));
        }

        // 5. 加载定时触发模块
        for (auto& component : module_config.timer_components())
{
            // 6. 创建对象
            const std::string& class_name = component.class_name();
            std::shared_ptr<ComponentBase> base =
                class_loader_manager_.CreateClassObj<ComponentBase>
(class_name);

```

```

    // 7. 调用对象的Initialize方法
    if (base == nullptr || !base-
>Initialize(component.config())) {
        return false;
    }
    component_list_.emplace_back(std::move(base));
}
}
return true;
}

```

模块首先通过classloader加载到内存，然后创建对象，并且调用模块的初始化方法。component中每个模块都设计为可以动态加载和卸载，可以实时在线的开启和关闭模块，实现的方式是通过classloader来进行动态的加载动态库。

component初始化

component一共有4个模板类，分别对应接收0-3个消息，（这里有疑问为什么没有4个消息的模板类，是漏掉了吗？）我们这里主要分析2个消息的情况，其它的可以类推。

```

template <typename M0, typename M1>
bool Component<M0, M1, NullType, NullType>::Initialize(
    const ComponentConfig& config) {
    // 1. 创建Node
    node_.reset(new Node(config.name()));
    LoadConfigFiles(config);

    // 2. 调用用户自定义初始化Init()
    if (!Init()) {
        AERROR << "Component Init() failed.";
        return false;
    }

    bool is_reality_mode = GlobalData::Instance()-
>IsRealityMode();

    ReaderConfig reader_cfg;
    reader_cfg.channel_name = config.readers(1).channel();
}

```

```

reader_cfg.qos_profile.CopyFrom(config.readers(1).qos_profile
());
reader_cfg.pending_queue_size =
config.readers(1).pending_queue_size();

// 3. 创建reader1
auto reader1 = node_->template CreateReader<M1>
(reader_cfg);
...
// 4. 创建reader0
if (cyber_likely(is_reality_mode)) {
    reader0 = node_->template CreateReader<M0>(reader_cfg);
} else {
    ...
}

readers_.push_back(std::move(reader0));
readers_.push_back(std::move(reader1));

auto sched = scheduler::Instance();
// 5. 创建回调，回调执行Proc()
std::weak_ptr<Component<M0, M1>> self =
    std::dynamic_pointer_cast<Component<M0, M1>>
(shared_from_this());
auto func = [self](const std::shared_ptr<M0>& msg0,
                    const std::shared_ptr<M1>& msg1) {
    auto ptr = self.lock();
    if (ptr) {
        ptr->Process(msg0, msg1);
    } else {
        AERROR << "Component object has been destroyed.";
    }
};

std::vector<data::VisitorConfig> config_list;
for (auto& reader : readers_) {
    config_list.emplace_back(reader->ChannelId(), reader-
>PendingQueueSize());
}
// 6. 创建数据访问器
auto dv = std::make_shared<data::DataVisitor<M0, M1>>
(config_list);

```

```

// 7. 创建协程，协程绑定回调func（执行proc）。数据访问器dv在收到订阅
数据之后，唤醒绑定的协程执行任务，任务执行完成之后继续休眠。
croutine::RoutineFactory factory =
    croutine::CreateRoutineFactory<M0, M1>(func, dv);
return sched->CreateTask(factory, node_->Name());
}

```

总结以下component的流程。

1. 创建node节点（1个component只能有1个node节点，之后用户可以用node_在init中自己创建reader或writer）。
2. 调用用户自定义的初始化函数Init()（子类的Init方法）
3. 创建reader，订阅几个消息就创建几个reader。
4. 创建回调函数，实际上是执行用户定义算法Proc()函数
5. 创建数据访问器，数据访问器的用途为接收数据（融合多个通道的数据），唤醒对应的协程执行任务。
6. 创建协程任务绑定回调函数，并且绑定数据访问器到对应的协程任务，用于唤醒对应的任务。

因为之前对cyber数据的收发流程有了一个简单的介绍，这里我们会分别介绍如何创建协程、如何在scheduler注册任务并且绑定Notify。也就是说，为了方便理解，你可以认为数据通过DataDispatcher已经分发到了对应的DataVisitor中，接下来我们只分析如何从**DataVisitor**中取数据，并且触发对应的协程执行回调任务。

创建协程

创建协程对应上述代码

```

croutine::RoutineFactory factory =
    croutine::CreateRoutineFactory<M0, M1>(func, dv);

```

接下来我们查看下如何创建协程呢？协程通过工厂模式方法创建，里面包含一个回调函数和一个dv（数据访问器）。

```

template <typename M0, typename M1, typename F>
RoutineFactory CreateRoutineFactory(
    F&& f, const std::shared_ptr<data::DataVisitor<M0, M1>>&
dv) {

```

```
RoutineFactory factory;
// 1. 工厂中设置DataVisitor
factory.SetDataVisitor(dv);
factory.create_routine = [=] () {
    return [=] () {
        std::shared_ptr<M0> msg0;
        std::shared_ptr<M1> msg1;
        for (;;) {
            CRoutine::GetCurrentRoutine() -
>set_state(RoutineState::DATA_WAIT);
            // 2. 从DataVisitor中获取数据
            if (dv->TryFetch(msg0, msg1)) {
                // 3. 执行回调函数
                f(msg0, msg1);
                // 4. 继续休眠
                CRoutine::Yield(RoutineState::READY);
            } else {
                CRoutine::Yield();
            }
        }
    };
};

return factory;
}
```

上述过程总结如下：

1. 工厂中设置DataVisitor
 2. 工厂中创建设置协程执行函数，回调包括3个步骤：从DataVisitor中获取数据，执行回调函数，继续休眠。

创建调度任务

创建调度任务是在过程”Component::Initialize”中完成。

```
sched->CreateTask(factory, node_->Name());
```

我们接着分析如何在Scheduler中创建任务。

```

        std::shared_ptr<DataVisitorBase>
visitor) {
    // 1. 根据名称创建任务ID
    auto task_id = GlobalData::RegisterTaskName(name);

    auto cr = std::make_shared<CRoutine>(func);
    cr->set_id(task_id);
    cr->set_name(name);
    AINFO << "create croutine: " << name;
    // 2. 分发协程任务
    if (!DispatchTask(cr)) {
        return false;
    }

    // 3. 注册Notify唤醒任务
    if (visitor != nullptr) {
        visitor->RegisterNotifyCallback([this, task_id]() {
            if (cyber_unlikely(stop_.load())) {
                return;
            }
            this->NotifyProcessor(task_id);
        });
    }
    return true;
}

```

TimerComponent

实际上Component分为2类：一类是上面介绍的消息驱动的Component，第二类是定时调用的TimerComponent。定时调度模块没有绑定消息收发，需要用户自己创建reader来读取消息，如果需要读取多个消息，可以创建多个reader。

```

bool TimerComponent::Initialize(const TimerComponentConfig&
config) {
    // 1. 创建node
    node_.reset(new Node(config.name()));
    LoadConfigFiles(config);
    // 2. 调用用户自定义初始化函数
    if (!Init()) {
        return false;
    }
}

```

```

    }

    std::shared_ptr<TimerComponent> self =
        std::dynamic_pointer_cast<TimerComponent>
    (shared_from_this());
    // 3. 创建定时器，定时调用"Proc()"函数
    auto func = [self]() { self->Proc(); };
    timer_.reset(new Timer(config.interval(), func, false));
    timer_->Start();
    return true;
}

```

总结一下TimerComponent的执行流程如下。

1. 创建Node
2. 调用用户自定义初始化函数
3. 创建定时器，定时调用”Proc()”函数

上述就是Component模块的调用流程。为了弄清楚消息的调用过程，下面我们分析”DataDispatcher”和”DataVisitor”。

DataVisitor和DataDispatcher

DataDispatcher（消息分发器）发布消息，DataDispatcher是一个单例，所有的数据分发都在数据分发器中进行，DataDispatcher会把数据放到对应的缓存中，然后Notify(通知)对应的协程（实际上这里调用的是DataVisitor中注册的Notify）去处理消息。

DataVisitor（消息访问器）是一个辅助的类，一个数据处理过程对应一个**DataVisitor**，通过在**DataVisitor**中注册**Notify**（唤醒对应的协程，协程执行绑定的回调函数），并且注册对应的**Buffer**到**DataDispatcher**，这样在DataDispatcher的时候会通知对应的DataVisitor去唤醒对应的协程。

也就是说DataDispatcher（消息分发器）发布对应的消息到DataVisitor，DataVisitor（消息访问器）唤醒对应的协程，协程中执行绑定的数据处理回调函数。

DataVisitor数据访问器

DataVisitor继承至DataVisitorBase类，先看DataVisitorBase类的实现。

```
class DataVisitorBase {
public:
    // 1. 初始化的时候创建一个Notifier
    DataVisitorBase() : notifier_(new Notifier()) {}

    // 2. 设置注册回调
    void RegisterNotifyCallback(std::function<void()>&&
callback) {
        notifier_->callback = callback;
    }

protected:
    // 3. 下一次消息的下标
    uint64_t next_msg_index_ = 0;
    // 4. DataNotifier单例
    DataNotifier* data_notifier_ = DataNotifier::Instance();
    std::shared_ptr<Notifier> notifier_;
};
```

可以看到DataVisitorBase创建了一个”Notifier”类，并且提供注册回调的接口。同时还引用了”DataNotifier::Instance()”单例。

接下来看”DataVisitor”类的实现。

```
template <typename M0, typename M1, typename M2>
class DataVisitor<M0, M1, M2, NullType> : public
DataVisitorBase {
public:
    explicit DataVisitor(const std::vector<VisitorConfig>&
configs)
        : buffer_m0_(configs[0].channel_id,
                     new BufferType<M0>
(configs[0].queue_size)),
          buffer_m1_(configs[1].channel_id,
                     new BufferType<M1>
(configs[1].queue_size)),
          buffer_m2_(configs[2].channel_id,
                     new BufferType<M2>(configs[2].queue_size))
{
    // 1. 在DataDispatcher中增加ChannelBuffer
```

```

DataDispatcher<M0>::Instance() ->AddBuffer(buffer_m0_);
DataDispatcher<M1>::Instance() ->AddBuffer(buffer_m1_);
DataDispatcher<M2>::Instance() ->AddBuffer(buffer_m2_);
// 2. 在DataNotifier::Instance()中增加创建好的Notifier
data_notifier_->AddNotifier(buffer_m0_.channel_id(),
notifier_);
// 3. 对接收到的消息进行数据融合
data_fusion_ =
    new fusion::AllLatest<M0, M1, M2>(buffer_m0_,
buffer_m1_, buffer_m2_);
}

bool TryFetch(std::shared_ptr<M0>& m0, std::shared_ptr<M1>&
m1, // NOLINT
              std::shared_ptr<M2>& m2) {
// NOLINT
// 4. 获取融合数据
if (data_fusion_->Fusion(&next_msg_index_, m0, m1, m2)) {
    next_msg_index_++;
    return true;
}
return false;
}

private:
fusion::DataFusion<M0, M1, M2>* data_fusion_ = nullptr;
ChannelBuffer<M0> buffer_m0_;
ChannelBuffer<M1> buffer_m1_;
ChannelBuffer<M2> buffer_m2_;
} ;

```

总结一下DataVisitor中实现的功能。

1. 在DataDispatcher中添加订阅的ChannelBuffer
2. 在DataNotifier中增加对应通道的Notifier
3. 通过DataVisitor获取数据并进行融合

这里注意

- 如果DataVisitor只访问一个消息，则不会对消息进行融合，如果DataVisitor访问2个以上的数据，那么需要进行融合，并且注册融合回调。之后CacheBuffer中会调用融合回调进行数据处理，而不会把数据放入CacheBuffer中。

```
// 1. 只有一个消息的时候直接从Buffer中获取消息
bool TryFetch(std::shared_ptr<M0>& m0) { // NOLINT
    if (buffer_.Fetch(&next_msg_index_, m0)) {
        next_msg_index_++;
        return true;
    }
    return false;
}

// 2. 当有2个消息的时候，从融合buffer中读取消息
bool TryFetch(std::shared_ptr<M0>& m0, std::shared_ptr<M1>& m1) { // NOLINT
    if (data_fusion_->Fusion(&next_msg_index_, m0, m1)) {
        next_msg_index_++;
        return true;
    }
    return false;
}
```

- 实际上如果有多个消息的时候，会以第1个消息为基准，然后把其它消息的最新消息一起放入融合好的buffer_fusion_。

```
AllLatest(const ChannelBuffer<M0>& buffer_0,
          const ChannelBuffer<M1>& buffer_1)
: buffer_m0_(buffer_0),
  buffer_m1_(buffer_1),
  buffer_fusion_(buffer_m0_.channel_id(),
                 new
CacheBuffer<std::shared_ptr<FusionDataType>>(
                buffer_0.Buffer()->Capacity() -
uint64_t(1))) {
    // 1. 注意这里只注册了buffer_m0_的回调，其它的消息还是把消息放入
    CacheBuffer。
    // 2. 而buffer_m0_直接调用回调函数，把消息放入融合的
    CacheBuffer。
    buffer_m0_.Buffer()->SetFusionCallback(
        [this](const std::shared_ptr<M0>& m0) {
```

```

        std::shared_ptr<M1> m1;
        if (!buffer_m1_.Latest(m1)) {
            return;
        }

        auto data = std::make_shared<FusionDataType>(m0,
m1);
        std::lock_guard<std::mutex>
lg(buffer_fusion_.Buffer()->Mutex());
        buffer_fusion_.Buffer()->Fill(data);
    });
}

```

3. DataFusion类是一个虚类，定义了数据融合的接口”Fusion()”，Apollo里只提供了一种数据融合的方式，即以第一个消息的时间为基准，取其它最新的消息，当然也可以在这里实现其它的数据融合方式。

DataDispatcher数据分发器

接下来我们看DataDispatcher的实现。

```

template <typename T>
class DataDispatcher {
public:
    using BufferVector =
std::vector<std::weak_ptr<CacheBuffer<std::shared_ptr<T>>>>;
    ~DataDispatcher() {}
    // 1. 添加ChannelBuffer到buffers_map_
    void AddBuffer(const ChannelBuffer<T>& channel_buffer);

    // 2. 分发通道中的消息
    bool Dispatch(const uint64_t channel_id, const
std::shared_ptr<T>& msg);

private:
    // 3. DataNotifier单例
    DataNotifier* notifier_ = DataNotifier::Instance();
    std::mutex buffers_map_mutex_;
    // 4. 哈希表，key为通道id，value为订阅通道消息的CacheBuffer数组。

```

```

AtomicHashMap<uint64_t, BufferVector> buffers_map_;
// 5. 单例
DECLARE_SINGLETON(DataDispatcher)
};

```

总结一下DataDispatcher的实现。

1. 添加ChannelBuffer到buffers_map_， key为通道id（topic）， value为订阅通道消息的CacheBuffer数组。
2. 分发通道中的消息。根据通道id，把消息放入对应的CacheBuffer。然后通过DataNotifier::Instance()通知对应的通道。

如果一个通道(topic)有3个**CacheBuffer**订阅，那么每次都会往这3个**CacheBuffer**中写入当前消息的指针。因为消息是共享的，消息访问的时候需要加锁。

那么DataNotifier如何通知对应的Channel的呢？理解清楚了DataNotifier的数据结构，那么也就理解了DataNotifier的原理，DataNotifier保存了

```

class DataNotifier {
public:
    using NotifyVector =
std::vector<std::shared_ptr<Notifier>>;
    ~DataNotifier() {}

    void AddNotifier(uint64_t channel_id,
                    const std::shared_ptr<Notifier>&
notifier);

    bool Notify(const uint64_t channel_id);

private:
    std::mutex notifies_map_mutex_;
    // 1. 哈希表，key为通道id，value为Notify数组
    AtomicHashMap<uint64_t, NotifyVector> notifies_map_;

    DECLARE_SINGLETON(DataNotifier)
};

```

DataNotifier中包含一个哈希表，表的key为通道id，表的值为Notify数组，每个DataVisitorBase在初始化的时候会创建一个Notify。

接着我们看下CacheBuffer的实现，CacheBuffer实际上实现了一个缓存队列，主要关注下Fill函数。

```
void Fill(const T& value) {
    if (fusion_callback_) {
        // 1. 融合回调
        fusion_callback_(value);
    } else {
        // 2. 如果Buffer满，实现循环队列
        if (Full()) {
            buffer_[GetIndex(head_)] = value;
            ++head_;
            ++tail_;
        } else {
            buffer_[GetIndex(tail_ + 1)] = value;
            ++tail_;
        }
    }
}
```

ChannelBuffer是CacheBuffer的封装，主要看下获取值。

```
template <typename T>
bool ChannelBuffer<T>::Fetch(uint64_t* index,
                             std::shared_ptr<T>& m) { // NOLINT
    std::lock_guard<std::mutex> lock(buffer_->Mutex());
    if (buffer_->Empty()) {
        return false;
    }

    if (*index == 0) {
        *index = buffer_->Tail();
        // 1. 为什么是判断最新的加1，而不是大于？？
    } else if (*index == buffer_->Tail() + 1) {
        return false;
    } else if (*index < buffer_->Head()) {
        auto interval = buffer_->Tail() - *index;
        AWARN << "channel[" <<
```

```

GlobalData::GetChannelById(channel_id_) << "] "
    << "read buffer overflow, drop_message[" <<
interval << "] pre_index["
    << *index << "] current_index[" << buffer_->Tail()
<< "] ";
    *index = buffer_->Tail();
}
m = buffer_->at(*index);
return true;
}

```

疑问：这里获取的id是消息的累计数量，也就是说从开始到结束发送了多少的消息。如果消息大于最新的id，而不是等于最大值+1，则会返回错误的值？？？

data 目录总结

通过上述的分析，实际上数据的访问都是通过”DataVisitor”来实现，数据的分发通过”DataDispatcher”来实现。reader中也是通过DataVisitor来访问数据，在reader中订阅对应的DataDispatcher。也就是说如果你要订阅一个通道，首先是在reader中注册消息的topic，绑定DataDispatcher，之后对应通道的消息到来之后，触发DataDispatcher分发消息，而DataDispatcher通过DataVisitor中的Notify唤醒协程，从DataVisitor中获取消息，并执行协程中绑定的回调函数，以上就是整个消息的收发过程。

疑问： Reader中还拷贝了一份数据到Blocker中，实际上数据的处理过程并不需要缓存数据，参考”Planning”模块中的实现都是在回调函数中把数据拷贝到指针中。看注释是说Blocker是用来仿真的？？？后面需要确实下。以下是Planning模块中回调函数中拷贝数据的实现。

```

traffic_light_reader_ = node_-
>CreateReader<TrafficLightDetection>(
    FLAGS_traffic_light_detection_topic,
    [this](const std::shared_ptr<TrafficLightDetection>&
traffic_light) {
        ADEBUG << "Received traffic light data: run traffic
light callback.";
        std::lock_guard<std::mutex> lock(mutex_);

```

```
// 1. 拷贝消息到指针  
traffic_light_.CopyFrom(*traffic_light);  
});
```

这里需要注意系统在component中自动帮我们创建了一个DataVisitor，订阅component中的消息，融合获取最新的消息之后，执行Proc回调。需要注意component的第一个消息一定是模块的基准消息来源，也就是模块中最主要的参考消息，不能随便调换顺序。

Node介绍

Node目录中包含了Node对象、Reader对象和Writer对象。Node对象主要对应Ros中的Node节点，在Node节点中可以创建Reader和Writer来订阅和发布消息，需要管理对应的通道注册。

Node对象

Reader对象

Writer对象

CRoutine协程

协程是用户态的线程，由于进程切换需要用户态到内核态的切换，而协程不需要切换到内核态，因此协程的切换开销比线程低。实际上线程切换的过程如下：

1. 保存当前线程的现场到堆栈，寄存器，栈指针
2. 用户态切换到内核态
3. 切换到另外一个线程，跳转到栈指针，恢复现场

一个线程的几种状态：running、sleeping。那么协程需要几种状态呢？

```
enum class RoutineState { READY, FINISHED, SLEEP, IO_WAIT,  
DATA_WAIT };
```

可以看到协程的状态有5种：准备好、结束、睡觉、等待IO、等待数据。

接着看下CRoutine的实现。

```
class CRoutine {
public:
    explicit CRoutine(const RoutineFunc &func);
    virtual ~CRoutine();

    // static interfaces
    static void Yield();
    static void Yield(const RoutineState &state);
    static void SetMainContext(const
std::shared_ptr<RoutineContext> &context);
    static CRoutine *GetCurrentRoutine();
    static char **GetMainStack();

    // public interfaces
    bool Acquire();
    void Release();

    // It is caller's responsibility to check if state_ is
    valid before calling
    // SetUpdateFlag().
    void SetUpdateFlag();

    // acquire && release should be called before Resume
    // when work-steal like mechanism used
    RoutineState Resume();
    RoutineState UpdateState();
    RoutineContext *GetContext();
    char **GetStack();

    void Run();
    void Stop();
    void Wake();
    void HangUp();
    void Sleep(const Duration &sleep_duration);

    // getter and setter
    RoutineState state() const;
```

```
void set_state(const RoutineState &state);

uint64_t id() const;
void set_id(uint64_t id);

const std::string &name() const;
void set_name(const std::string &name);

int processor_id() const;
void set_processor_id(int processor_id);

uint32_t priority() const;
void set_priority(uint32_t priority);

std::chrono::steady_clock::time_point wake_time() const;

void set_group_name(const std::string &group_name) {
    group_name_ = group_name;
}

const std::string &group_name() { return group_name_; }

private:
CRoutine(CRoutine &) = delete;
CRoutine &operator=(CRoutine &) = delete;

std::string name_;
std::chrono::steady_clock::time_point wake_time_ =
    std::chrono::steady_clock::now();

RoutineFunc func_;
RoutineState state_;

std::shared_ptr<RoutineContext> context_;

std::atomic_flag lock_ = ATOMIC_FLAG_INIT;
std::atomic_flag updated_ = ATOMIC_FLAG_INIT;

bool force_stop_ = false;

int processor_id_ = -1;
uint32_t priority_ = 0;
uint64_t id_ = 0;
```

```
    std::string group_name_;  
  
    static thread_local CRoutine *current_routine_;  
    static thread_local char *main_stack_;  
};
```

对上述方法做一些归类。

下面总结一下CRoutine的切换流程。

Scheduler调度

所谓的调度，一定是系统资源和运行任务的矛盾，如果系统资源足够多，那么就不需要调度了，也没有调度的必要。调度的作用就是在资源有限的情况下，合理利用系统资源，使系统的效率最高。

Scheduler* Instance()

在”scheduler_factory.cc”中实现了”Scheduler* Instance()”方法，该方法根据”conf”目录中的配置创建不同的调度策略。Cyber中有2种调度策略”SchedulerClassic”和”SchedulerChoreography”。要理解上述2种模型，可以参考go语言中的GPM模型。

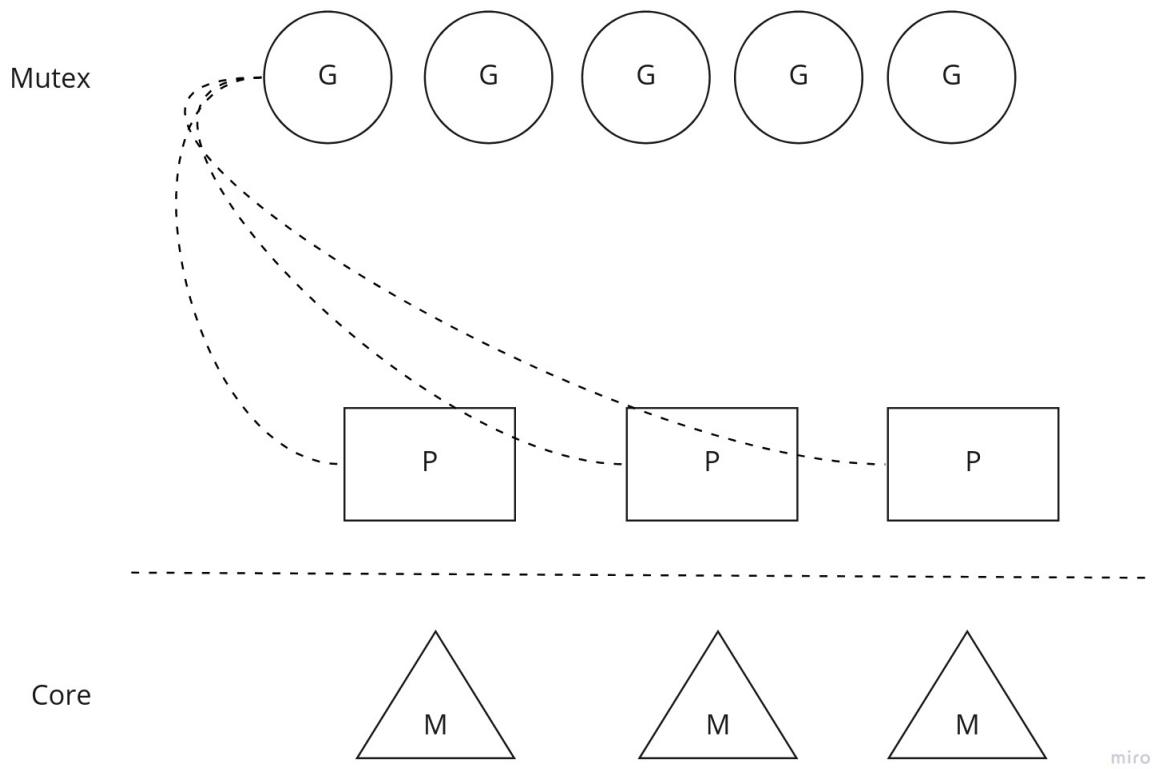
1. SchedulerClassic 采用了协程池的概念，协程没有绑定到具体的 Processor，所有的协程放在全局的优先级队列中，每次从最高优先级的任务开始执行。
2. SchedulerChoreography 采用了本地队列和全局队列相结合的方式，通过”ChoreographyContext”运行本地队列的线程，通过”ClassicContext”来运行全局队列。
疑问：这里的调度对象为原子指针”std::atomic<Scheduler*> instance”，为什么需要设置内存模型，并且加锁呢？

基本概念

1.GPM模型

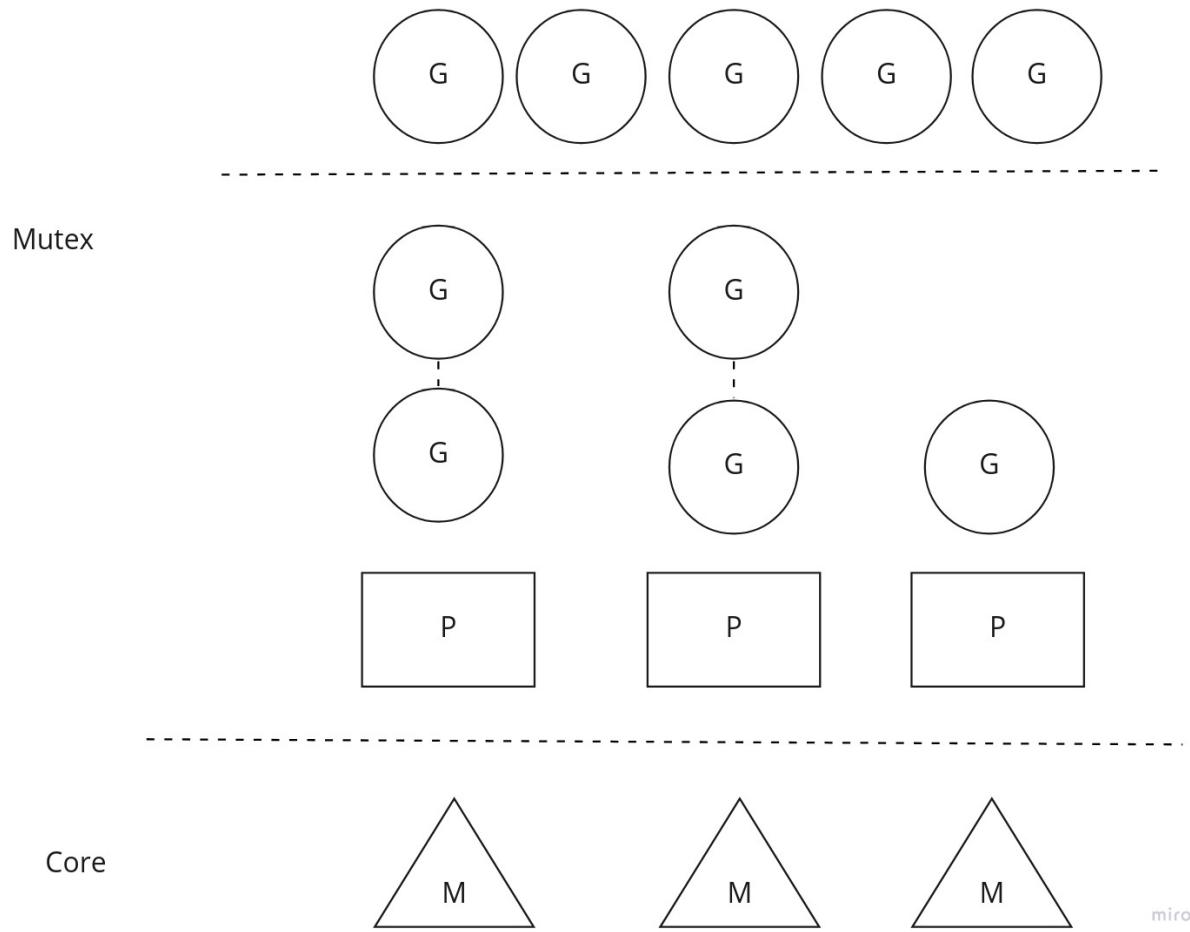
GPM模型 [https://learnku.com/articles/41728]是go语言中的概念，G代表协程，P代表执行器，M代表线程。P代表协程执行的环境，一个P绑定一个线程M，协程G根据不同的上下文环境在P中执行，上述的设计解耦了协程和线程，线程和协程都只知道P中，在go语言中还有一个调度器，把具体的协程分配到不同的P中，这样就可以在用户态实现协程的调度，同时执行多个任务。

在go语言早期，只有一个全局的协程队列，每个P都会从全局队列中取任务，然后执行。因为多个P都会去全局队列中取任务，因此会带来并发访问，需要在全局队列中加锁。全局队列调度模型如图。



上述方法虽然实现了多个任务的调度，但是带来的问题是多个P都会去竞争全局锁，导致效率低下，之后go语言对调度模型做了改善，改善之后每个P都会拥有一个本地队列，P优先从本地队列中取任务执行，如果P中没有任务了，那么P会从全局或者相邻的任务中偷取一半的任务执行，这样带来的好处是不需要全局锁了，每个P都优先执行本地队列。另外调度器还会监视P中的协程，如果协程过长时间阻塞，则会把协程移动到全局队列，以示惩罚。本地队列加全局队列如

图。



理解了GPM模型，下面我们接着看Cyber中的调度器，SchedulerClassic代表着全局队列模型，SchedulerChoreography则代表着本地队列加全局队列模型。SchedulerChoreography模型和go语言的模型稍微有点区别，全局队列和本地队列是隔离的，也就是说本地队列不会去全局队列中取任务。

2.Processor执行器 Processor执行器是协程的载体，对协程来说 Processor就是逻辑的CPU。Processor中有协程执行的上下文信息，还有绑定的线程信息。先看下BindContext的实现。

```
void Processor::BindContext(const  
std::shared_ptr<ProcessorContext>& context) {  
    // 1. 协程执行的上下文信息，会随着协程切换而切换  
    context_ = context;
```

```

// 2. 绑定Processor到具体的线程
std::call_once(thread_flag_,
    [this]() { thread_ =
std::thread(&Processor::Run, this); });
}

```

也就是说协程运行在Processor之上，切换协程的时候传入对应的上下文到Processor，然后Processor开始执行协程任务。下面看Processor是如何执行的。

```

void Processor::Run() {
    // 1. 获取线程的PID，系统内唯一
    tid_.store(static_cast<int>(syscall(SYS_gettid)));

    snap_shot_->processor_id.store(tid_);

    while (cyber_likely(running_.load())) {
        if (cyber_likely(context_ != nullptr)) {
            // 2. 获取优先级最高并且准备就绪的协程
            auto croutine = context_->NextRoutine();
            if (croutine) {
                snap_shot_-
>execute_start_time.store(cyber::Time::Now().ToNanosecond());
                snap_shot_->routine_name = croutine->name();
                // 3. 执行协程任务，完成后释放协程
                croutine->Resume();
                croutine->Release();
            } else {
                snap_shot_->execute_start_time.store(0);
                // 4. 如果协程组中没有空闲的协程，则等待
                context_->Wait();
            }
        } else {
            // 5. 如果上下文为空，则线程阻塞10毫秒
            std::unique_lock<std::mutex> lk(mtx_ctx_);
            cv_ctx_.wait_for(lk, std::chrono::milliseconds(10));
        }
    }
}

```

疑问：

1. 因为Processor每次都会从高优先级的队列开始取任务，假设Processor的数量不够，可能会出现低优先级的协程永远得不到调度的情况。
2. 协程的调度没有抢占，也就是说一个协程在执行的过程中，除非主动让出，否则会一直占用Processor，如果Processor在执行低优先级的任务，来了一个高优先级的任务并不能抢占执行。调度器的抢占还是交给线程模型去实现了，cyber中通过调节cgroup来调节，保证优先级高的任务优先执行。

一共有2种ProcessorContext上下文”ClassicContext”和”ChoreographyContext”上下文，分别对应不同的调度策略。后面分析Scheduler对象的时候，我们会接着分析。

3. conf配置文件

Scheduler调度的配置文件在”conf”目录中，配置文件中可以设置线程线程的CPU亲和性以及调度测量，也可以设置cgroup，还可以设置协程的优先级。下面以”example_sched_classic.conf”文件来举例子。

```

scheduler_conf {
    // 1. 设置调度器策略
    policy: "classic"
    // 2. 设置cpu set
    process_level_cpuset: "0-7,16-23" # all threads in the
process are on the cpuset
    // 3. 设置线程的cpuset，调度策略和优先级
    threads: [
        {
            name: "async_log"
            cpuset: "1"
            policy: "SCHED_OTHER"      # policy:
SCHED_OTHER, SCHED_RR, SCHED_FIFO
            prio: 0
        },
        {
            name: "shm"
            cpuset: "2"
            policy: "SCHED_FIFO"
            prio: 10
        }
    ]
}

```

```
        ]
    classic_conf {
        // 4. 设置分组, 线程组的cpuset, cpu亲和性, 调度测量和优先级。
        // 设置调度器创建"processor"对象的个数, 以及协程的优先级。
        groups: [
            {
                name: "group1"
                processor_num: 16
                affinity: "range"
                cpuset: "0-7,16-23"
                processor_policy: "SCHED_OTHER" # policy:
                SCHED_OTHER, SCHED_RR, SCHED_FIFO
                processor_prio: 0
                tasks: [
                    {
                        name: "E"
                        prio: 0
                    }
                ]
            }, {
                name: "group2"
                processor_num: 16
                affinity: "1to1"
                cpuset: "8-15,24-31"
                processor_policy: "SCHED_OTHER"
                processor_prio: 0
                tasks: [
                    {
                        name: "A"
                        prio: 0
                    }, {
                        name: "B"
                        prio: 1
                    }, {
                        name: "C"
                        prio: 2
                    }, {
                        name: "D"
                        prio: 3
                    }
                ]
            }
        ]
    }
```

```
    }
}
```

下面我们开始看调度器，上述已经简单的介绍了2种调度器，`SchedulerClassic`是基于全局优先队列的调度器。

SchedulerClassic对象

我们知道协程实际上建立在线程之上，线程分时执行多个协程，看上去多个协程就是一起工作的。假如让你设计一个协程池，首先需要设置协程池中协程的个数，当协程超过协程池个数的时候需要把协程放入一个阻塞队列中，如果队列满了，还有协程到来，那么丢弃到来的协程，并且报错。（上述设计借鉴了线程池的思路）

创建Processor

调度器中首先会创建Processor，并且绑定到线程。调度器根据”conf”目录中的cgroup配置初始化线程，根据”processor_num”的个数创建多个Processor，并且绑定到线程。

```
void SchedulerClassic::CreateProcessor() {
    // 读取调度配置文件，参照conf目录
    for (auto& group : classic_conf_.groups()) {
        // 1. 分组名称
        auto& group_name = group.name();
        // 2. 分组执行器(线程)数量 等于协程池大小
        auto proc_num = group.processor_num();
        if (task_pool_size_ == 0) {
            task_pool_size_ = proc_num;
        }
        // 3. 分组CPU亲和性
        auto& affinity = group.affinity();
        // 4. 分组线程调度策略
        auto& processor_policy = group.processor_policy();
        // 5. 分组优先级
        auto processor_prio = group.processor_prio();
        // 7. 分组cpu set
        std::vector<int> cpuset;
        ParseCpuset(group.cpuset(), &cpuset);
    }
}
```

```

        for (uint32_t i = 0; i < proc_num; i++) {
            auto ctx = std::make_shared<ClassicContext>
(group_name);
            pctxs_.emplace_back(ctx);

            auto proc = std::make_shared<Processor>();
            // 8. 绑定上下文
            proc->BindContext(ctx);
            // 9. 设置线程的cpuset和cpu亲和性
            SetSchedAffinity(proc->Thread(), cpuset, affinity, i);
            // 10. 设置线程调度策略和优先级 (proc->Tid()为线程pid)
            SetSchedPolicy(proc->Thread(), processor_policy,
processor_prio,
                            proc->Tid());
            processors_.emplace_back(proc);
        }
    }
}

```

SchedulerClassic是Scheduler的子类，Scheduler中实现了”CreateTask”和”NotifyTask”接口，用于创建任务和唤醒任务。对用户来说只需要关心任务，Scheduler为我们屏蔽了Processor对象的操作。对应的子类中实现了”DispatchTask”，”RemoveTask”和”NotifyProcessor”的操作。我们先看Scheduler如何创建任务。

```

bool Scheduler::CreateTask(std::function<void()>&& func,
                           const std::string& name,
                           std::shared_ptr<DataVisitorBase>
visitor) {

    auto task_id = GlobalData::RegisterTaskName(name);
    // 1. 创建协程，绑定func函数
    auto cr = std::make_shared<CRoutine>(func);
    cr->set_id(task_id);
    cr->set_name(name);
    AINFO << "create croutine: " << name;

    // 2. 分发任务
    if (!DispatchTask(cr)) {
        return false;
    }
}

```

```

// 3. 注册回调, visitor参数为可选的。
if (visitor != nullptr) {
    visitor->RegisterNotifyCallback([this, task_id] () {
        if (cyber_unlikely(stop_.load())) {
            return;
        }
        this->NotifyProcessor(task_id);
    });
}
return true;
}

```

总结：

1. 创建任务的时候只有对应数据访问的DataVisitorBase注册了回调，其它的任务要自己触发回调。
2. DispatchTask中调用子类中的不同策略进行任务分发。

接着我们看SchedulerClassic的DispatchTask调度策略。

```

bool SchedulerClassic::DispatchTask(const
std::shared_ptr<CRoutine>& cr) {
    // 1. 根据协程id, 获取协程的锁
    ...

    // 2. 将协程放入协程map
    {
        WriteLockGuard<AtomicRWLock> lk(id_cr_lock_);
        if (id_cr_.find(cr->id()) != id_cr_.end()) {
            return false;
        }
        id_cr_[cr->id()] = cr;
    }

    // 3. 设置协程的优先级和group
    if (cr_confs_.find(cr->name()) != cr_confs_.end()) {
        ClassicTask task = cr_confs_[cr->name()];
        cr->set_priority(task.prio());
        cr->set_group_name(task.group_name());
    } else {
        // croutine that not exist in conf
    }
}

```

```

        cr->set_group_name(classic_conf_.groups(0).name());
    }

    // 4. 将协程放入对应的优先级队列
    // Enqueue task.
    {
        WriteLockGuard<AtomicRWLock> lk(
            ClassicContext::rq_locks_[cr->group_name()].at(cr-
>priority()));
        ClassicContext::cr_group_[cr->group_name()]
            .at(cr->priority())
            .emplace_back(cr);
    }

    // 5. 唤醒协程组
    ClassicContext::Notify(cr->group_name());
    return true;
}

```

这里NotifyProcessor实际上就是唤醒对应Processor的上下文执行环境。

```

bool SchedulerClassic::NotifyProcessor(uint64_t crid) {
    if (cyber_unlikely(stop_)) {
        return true;
    }

    {
        ReadLockGuard<AtomicRWLock> lk(id_cr_lock_);
        if (id_cr_.find(crid) != id_cr_.end()) {
            auto cr = id_cr_[crid];
            if (cr->state() == RoutineState::DATA_WAIT ||
                cr->state() == RoutineState::IO_WAIT) {
                cr->SetUpdateFlag();
            }
        }

        ClassicContext::Notify(cr->group_name());
        return true;
    }
}

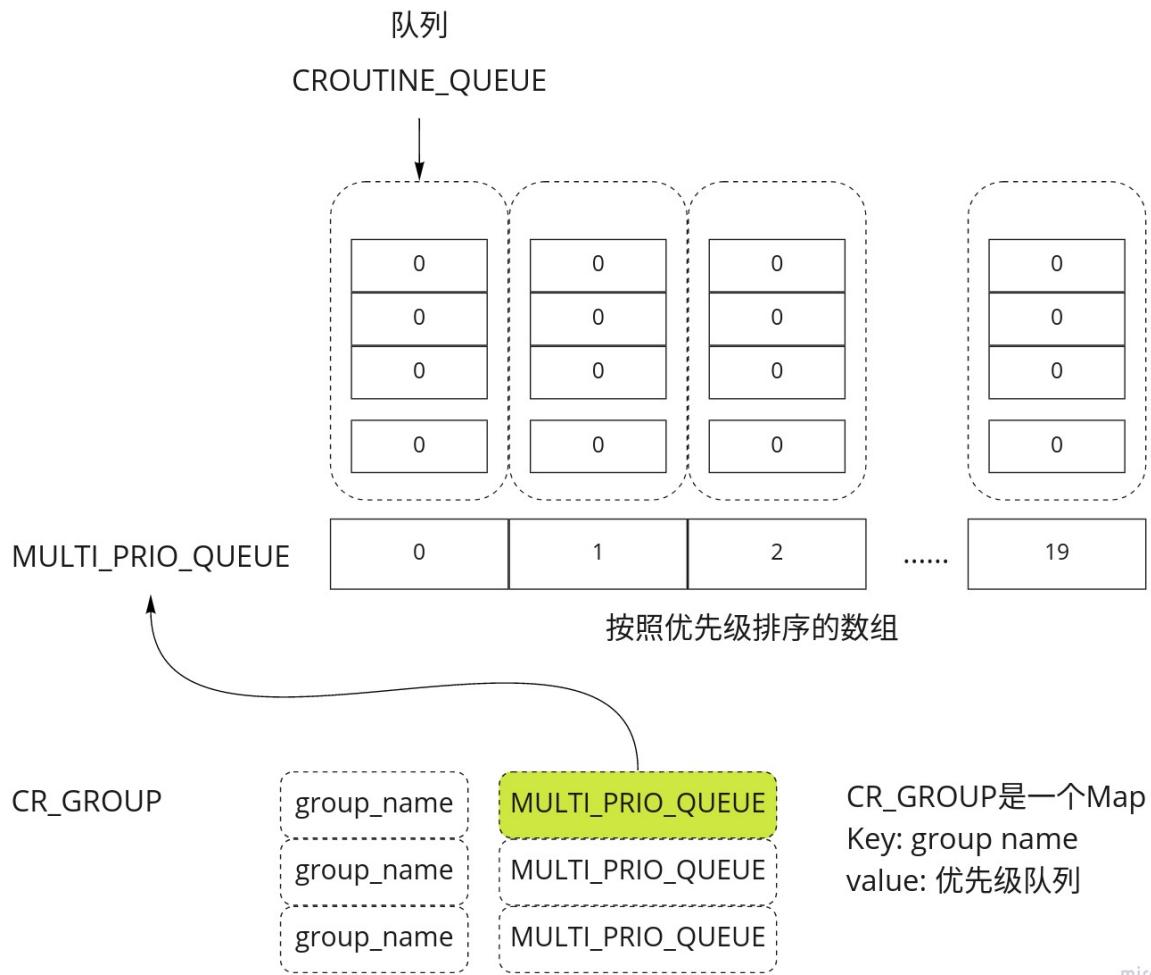
return false;
}

```

疑问：

1. 如果协程在等待IO，系统知道协程在等待io，但还是触发对应的协程组去工作。并没有让协程继续阻塞？？？

再看如何从”ClassicContext”获取Processor执行的协程。下图是全局协程队列的数据结构，为一个2级数组，第一级数组为优先级数组，第二级数组为一个队列。



```
std::shared_ptr<CRoutine> ClassicContext::NextRoutine() {
    if (cyber_unlikely(stop_.load())) {
        return nullptr;
    }

    // 1. 从优先级最高的队列开始遍历
    for (int i = MAX_PRIO - 1; i >= 0; --i) {
```

```

// 2. 获取当前优先级队列的锁
ReadLockGuard<AtomicRWLock> lk(lq_->at(i));
for (auto& cr : multi_pri_rq_->at(i)) {
    if (!cr->Acquire()) {
        continue;
    }
    // 3. 返回状态就绪的协程
    if (cr->UpdateState() == RoutineState::READY) {
        return cr;
    }

    cr->Release();
}
}

return nullptr;
}

```

我们知道线程阻塞的条件有4种：

1. 通过调用sleep(milliseconds)使任务进入休眠状态
2. 通过调用wait()使线程挂起
3. 等待某个输入、输出流
4. 等待锁

而Processor绑定的线程阻塞是通过上下文的等待实现的。在ClassicContext中等待条件（1s的超时时间），等待的时候设置notify_grp_减1。

```

void ClassicContext::Wait() {
    // 1. 获得锁
    std::unique_lock<std::mutex> lk(mtx_wrapper_->Mutex());
    // 2. 等待条件大于0
    cw_->Cv().wait_for(lk, std::chrono::milliseconds(1000),
                         [&]() { return notify_grp_[current_grp] > 0; });
    // 3. 对应协程组的唤醒条件减1
    if (notify_grp_[current_grp] > 0) {
        notify_grp_[current_grp]--;
    }
}

```

Processor的唤醒也是通过上下文实现的。

```
void ClassicContext::Notify(const std::string& group_name) {
    // 1. 加锁
    (&mtx_wq_[group_name]) ->Mutex().lock();
    // 2. 协程唤醒条件加1
    notify_grp_[group_name]++;
    (&mtx_wq_[group_name]) ->Mutex().unlock();
    // 3. 唤醒线程
    cv_wq_[group_name].Cv().notify_one();
}
```

关于SchedulerClassic我们就先介绍到这里，下面我们开始介绍另外一种调度器SchedulerChoreography。

SchedulerClassic调度器

SchedulerChoreography创建Processor，会分2部分创建，一部分是有本地队列的Processor，一部分是协程池的Processor。

```
void SchedulerChoreography::CreateProcessor() {
    for (uint32_t i = 0; i < proc_num_; i++) {
        auto proc = std::make_shared<Processor>();
        // 1. 绑定ChoreographyContext
        auto ctx = std::make_shared<ChoreographyContext>();

        proc->BindContext(ctx);
        SetSchedAffinity(proc->Thread(), choreography_cpuset_,
                          choreography_affinity_, i);
        SetSchedPolicy(proc->Thread(),
                      choreography_processor_policy_,
                      choreography_processor_prio_, proc-
>Tid());
        pctxs_.emplace_back(ctx);
        processors_.emplace_back(proc);
    }

    for (uint32_t i = 0; i < task_pool_size_; i++) {
        auto proc = std::make_shared<Processor>();
        // 2. 绑定ClassicContext
        auto ctx = std::make_shared<ClassicContext>();
```

```

    proc->BindContext(ctx);
    SetSchedAffinity(proc->Thread(), pool_cpuset_,
pool_affinity_, i);
    SetSchedPolicy(proc->Thread(), pool_processor_policy_,
pool_processor_prio_,
                    proc->Tid());
    pctxs_.emplace_back(ctx);
    processors_.emplace_back(proc);
}
}

```

在看SchedulerChoreography如何分发任务。

```

bool SchedulerChoreography::DispatchTask(const
std::shared_ptr<CRoutine>& cr) {
    // 1. 根据协程id，获取协程的锁
    ...

    // 2. 设置优先级和协程绑定的Processor Id
    if (cr_confs_.find(cr->name()) != cr_confs_.end()) {
        ChoreographyTask taskconf = cr_confs_[cr->name()];
        cr->set_priority(taskconf.prio());

        if (taskconf.has_processor()) {
            cr->set_processor_id(taskconf.processor());
        }
    }

    ...
}

uint32_t pid = cr->processor_id();
// 3. 如果Processor Id小于proc_num_，默认Processor Id为-1
if (pid < proc_num_) {
    // 4. 协程放入上下文本地队列中
    static_cast<ChoreographyContext*>(pctxs_[pid].get())-
>Enqueue(cr);
} else {

    cr->set_group_name(DEFAULT_GROUP_NAME);

    // 5. 协程放入classicContext协程池队列中
    {

```

```

        WriteLockGuard<AtomicRwLock> lk(
    ClassicContext::rq_locks_[DEFAULT_GROUP_NAME].at(cr-
>priority()));
    ClassicContext::cr_group_[DEFAULT_GROUP_NAME]
        .at(cr->priority())
        .emplace_back(cr);
}
}
return true;
}

```

疑问：

1. cr->processor_id()的默认值为”-1”，而vector访问越界的时候不会报错，本来应该放入全局队列中的？？？

ChoreographyContext上下文

ChoreographyContext中的调度就非常简单了。

```

std::shared_ptr<CRoutine> ChoreographyContext::NextRoutine()
{
    // 1. 从本地队列中取出协程
    ReadLockGuard<AtomicRwLock> lock(rq_lk_);
    for (auto it : cr_queue_) {
        auto cr = it.second;
        if (!cr->Acquire()) {
            continue;
        }

        if (cr->UpdateState() == RoutineState::READY) {
            return cr;
        }
        cr->Release();
    }
    return nullptr;
}

```

ChoreographyContext中”Wait”和”Notify”方法与ClassicContext类似，这里就不展开了。

总结

1. SchedulerClassic 采用了协程池的概念，协程没有绑定到具体的 Processor，所有的协程放在全局的优先级队列中，每次从最高优先级的任务开始执行。
 2. SchedulerChoreography 采用了本地队列和全局队列相结合的方式，通过”ChoreographyContext”运行本地队列的线程，通过”ClassicContext”来运行全局队列。
 3. Processor对协程来说是一个逻辑上的cpu，ProcessorContext实现 Processor的运行上下文，通过ProcessorContext来获取协程，休眠或者唤醒，Scheduler调度器实现了协程调度算法。

下面介绍下cyber的异步调用接口”cyber::Async”，启动异步执行任务。

异步调用

在“task.h”中定义了异步调用的方法包括“Async”, “Yield”, “SleepFor”, “USleep”方法。下面我们逐个看下上述方法的实现。

Async方法

```
template <typename F, typename... Args>
static auto Async(F&& f, Args&&... args)
    -> std::future<typename std::result_of<F(Args...)>::type>
{
    return GlobalData::Instance()->IsRealityMode()
        ? TaskManager::Instance()-
>Enqueue(std::forward<F>(f),
std::forward<Args>(args)...)
        : std::async(
            std::launch::async,
            std::bind(std::forward<F>(f),
std::forward<Args>(args)...));
}
```

如果为真实模式，则采用”TaskManager”的方法生成协程任务，如果是仿真模式则创建线程。

TaskManager实际上实现了一个任务池，最大执行的任务数为scheduler模块中设置的TaskPoolSize大小。超出的任务会放在大小为1000的有界队列中，如果超出1000，任务会被丢弃。下面我们看TaskManager的具体实现。

```
TaskManager::TaskManager()
// 1. 设置有界队列，长度为1000
    : task_queue_size_(1000),
    task_queue_(new
base::BoundedQueue<std::function<void()>>())
{
    if (!task_queue_->Init(task_queue_size_, new
base::BlockWaitStrategy())) {
        AERROR << "Task queue init failed";
        throw std::runtime_error("Task queue init failed");
    }
// 2. 协程任务，每次从队列中取任务执行，如果没有任务则让出协程，等待数据
auto func = [this]() {
    while (!stop_) {
        std::function<void()> task;
        if (!task_queue_->Dequeue(&task)) {
            auto routine =
croutine::CRoutine::GetCurrentRoutine();
            routine->HangUp();
            continue;
        }
        task();
    }
};

num_threads_ = scheduler::Instance()->TaskPoolSize();
auto factory =
croutine::CreateRoutineFactory(std::move(func));
tasks_.reserve(num_threads_);
// 3. 创建TaskPoolSize个任务并且放入调度器
for (uint32_t i = 0; i < num_threads_; i++) {
    auto task_name = task_prefix + std::to_string(i);

tasks_.push_back(common::GlobalData::RegisterTaskName(task_na
me));
}
```

```

        if (!scheduler::Instance()->CreateTask(factory,
task_name)) {
            AERROR << "CreateTask failed:" << task_name;
        }
    }
}

```

1. 协程承载运行具体的任务，也就是说如果任务队列中有任务，则调用协程去执行，如果队列中没有任务，则让出协程，并且设置协程为等待数据的状态，那么让出协程之后唤醒是谁去触发的呢？
每次在任务队列中添加任务的时候，会唤醒协程执行任务。
2. task_queue_会被多个协程访问，并发数据访问这里没有加锁，需要看下这个队列是如何实现的？？？
3. 上述的任务可以在”conf”文件中设置”/internal/task + index”的优先级来实现。

接着看下Enqueue的实现，加入任务到任务队列。

```

template <typename F, typename... Args>
auto Enqueue(F&& func, Args&&... args)
    // 1. 返回值为future类型
    -> std::future<typename
std::result_of<F(Args...)>::type> {
    using return_type = typename
std::result_of<F(Args...)>::type;
    auto task =
std::make_shared<std::packaged_task<return_type()>>(
        std::bind(std::forward<F>(func), std::forward<Args>
(args)...));
    if (!stop_.load()) {
        // 2. 将函数加入任务队列，注意这里的任务不是调度单元里的任务，可以理解为一个函数
        task_queue_->Enqueue([task] () { (*task)(); });
        // 3. 这里的任务是调度任务，唤醒每个协程执行任务。
        for (auto& task : tasks_) {
            scheduler::Instance()->NotifyTask(task);
        }
    }
    std::future<return_type> res(task->get_future());
}

```

```
    return res;
}
```

每次在任务队列中添加任务的时候，唤醒任务协程中所有的协程。

Yield和Sleep方法

Yield方法和Async类似，如果为协程则让出当前协程，如果为线程则让出线程。SleepFor和USleep方法类似，这里就不展开了。

```
static inline void Yield() {
    if (croutine::CRoutine::GetCurrentRoutine()) {
        croutine::CRoutine::Yield();
    } else {
        std::this_thread::yield();
    }
}
```

SysMo系统监控

SysMo模块的用途主要是监控系统协程的运行状况。

Start

在start中单独启动一个线程去进行系统监控，这里没有设置线程的优先级，因此不能在”conf”文件中设置优先级？？？

```
void SysMo::Start() {
    auto sysmo_start = GetEnv("sysmo_start");
    if (sysmo_start != "" && std::stoi(sysmo_start)) {
        start_ = true;
        sysmo_ = std::thread(&SysMo::Checker, this);
    }
}
```

Checker

每隔100ms调用一次Checker，获取调度信息。

```

void SysMo::Checker() {
    while (cyber_unlikely(!shut_down_.load())) {
        scheduler::Instance()->CheckSchedStatus();
        std::unique_lock<std::mutex> lk(lk_);
        cv_.wait_for(lk,
        std::chrono::milliseconds(sysmo_interval_ms_));
    }
}

```

打印的是协程的调度快照。

```

void Scheduler::CheckSchedStatus() {
    std::string snap_info;
    auto now = Time::Now().ToNanosecond();
    for (auto processor : processors_) {
        auto snap = processor->ProcSnapshot();
        if (snap->execute_start_time.load()) {
            auto execute_time = (now - snap-
>execute_start_time.load()) / 1000000;
            snap_info.append(std::to_string(snap-
>processor_id.load()))
                .append(":")
                .append(snap->routine_name)
                .append(":")
                .append(std::to_string(execute_time));
        } else {
            snap_info.append(std::to_string(snap-
>processor_id.load()))
                .append(":idle");
        }
        snap_info.append(", ");
    }
    snap_info.append("timestamp:");
}).append(std::to_string(now));
AINFO << snap_info;
snap_info.clear();
}

```

Timer定时器

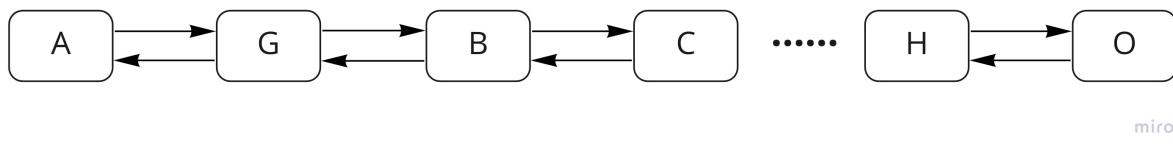
定时器提供在指定的时间触发执行的功能。定时器的应用非常普遍，比如定时触发秒杀活动、定时清理日志、定时发送心跳信息等。实现定时器的方法多种多样，古代有采用水漏或者沙漏的方式，近代有采用机械的方式（各种各样的时钟），数字脉冲，元素衰减等方式。

在计算机领域有2种形式，一种是硬件定时器，一种是软件定时器。硬件定时器的原理是计算时钟脉冲，当规定的时钟脉冲之后由硬件触发中断程序执行，硬件定时器一般是芯片自带的，硬件定时器时间比较精准但是数量有限，因此人们又发明了软件定时器。软件定时器由软件统计计算机时钟个数，然后触发对应的任务执行，由于是纯软件实现，理论上可以创建很多个，下面我们主要看下软件定时器的实现。

定时器的实现

双向链表

首先我们想到的是把定时任务放入一个队列中，每隔固定的时间（一个tick）去检查队列中是否有超时的任务，如果有，则触发执行该任务。这样做好处是实现简单，但是每次都需要轮询整个队列来找到谁需要被触发，当队列的长度很大时，每个固定时间都需要去轮询一次队列，时间开销比较大。当我们需要删除一个任务的时候，也需要轮询一遍队列找到需要删除的任务，实际上我们可以优化一下用双向链表去实现队列，这样删除的任务的时间复杂度就是 $O(1)$ 了。总结一下就是采用双向链表实现队列，插入的时间复杂度是 $O(1)$ ，删除的时间复杂度也是 $O(1)$ ，但是查询的时间复杂度是 $O(n)$ 。

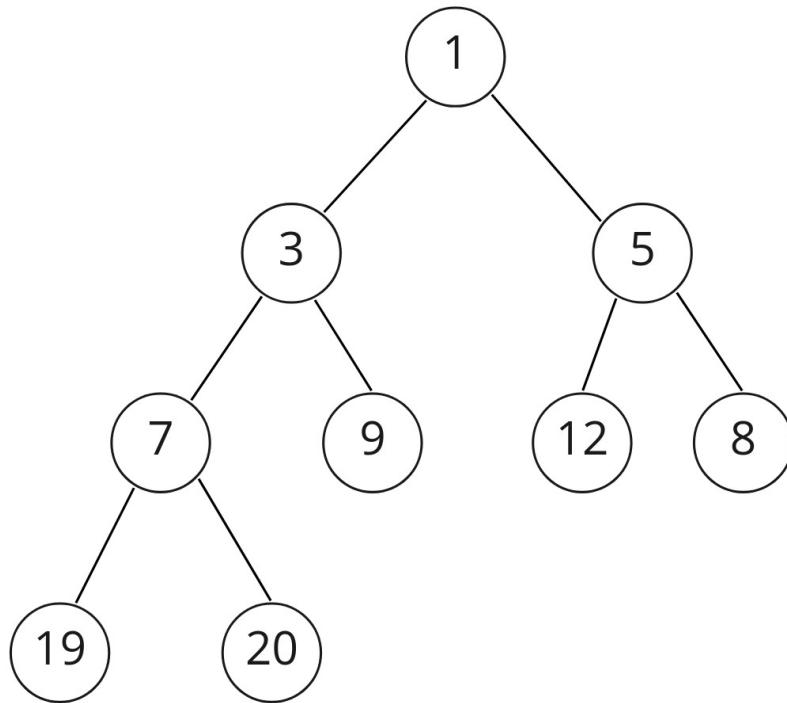


最小堆

最小堆的实现方式是为了解决上述查找的时候需要遍历整个链表的问题，我们知道最小堆中堆顶的元素就是最小的元素，每次我们只需要

检查堆顶的元素是否超时，超时则弹出执行，然后再检查新的堆顶元素是否超时，这样查找可执行任务的时间复杂度约等于 $O(1)$ ，最小堆虽然提高了查找的时间，但是插入和删除任务的时间复杂度为 $O(\log 2n)$ 。下面我们看一个例子。

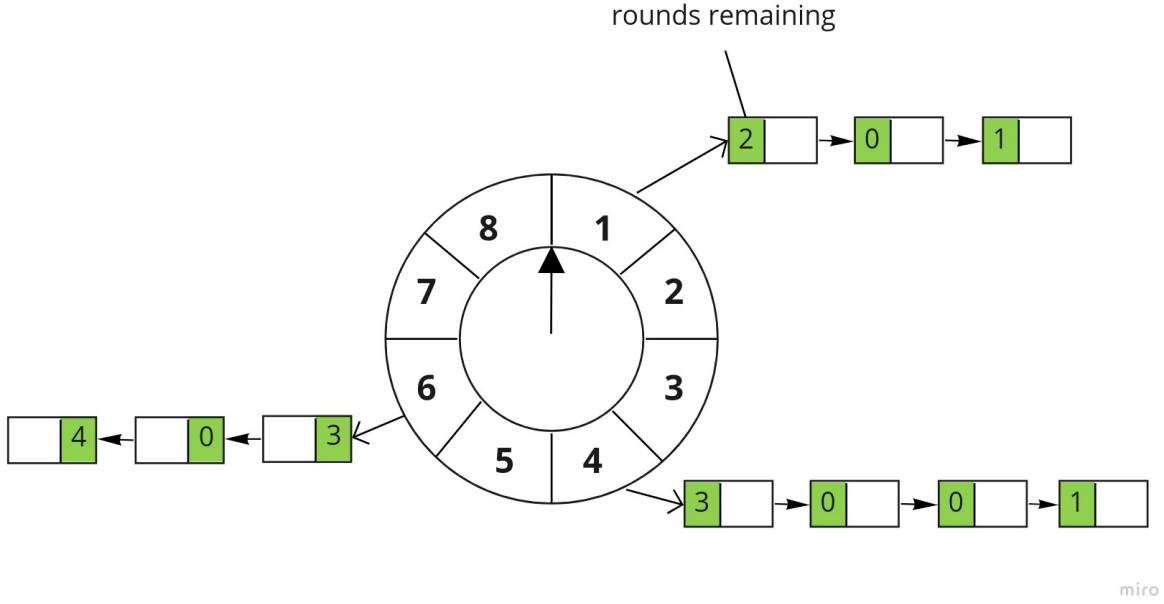
堆中节点的值存放的是任务到期的时间，每隔1分钟判断下是否有任务需要执行，比如任务A是19:01分触发，周期为5分钟，任务B是19:02分触发，周期为10分钟，那么第一次最小堆弹出19:01，执行之后，在堆中重新插入19:06分的任务A，这时候任务B到了堆顶，1分钟之后检测需要执行任务B，执行完成后，在堆中重新插入19:12分的任务B。然后循环执行上述过程。每执行一次任务都需要重新插入任务到堆中，当任务频繁执行的时候，插入任务的开销也不容忽略。



miro

时间轮

最后，我们介绍一种插入、删除和触发执行都是 $O(1)$ 的方法，由计算机科学家”George Varghese”等提出，在NetBSD(一种操作系统)上实现并替代了早期内核中的callout定时器实现。最原始的时间轮如下图。



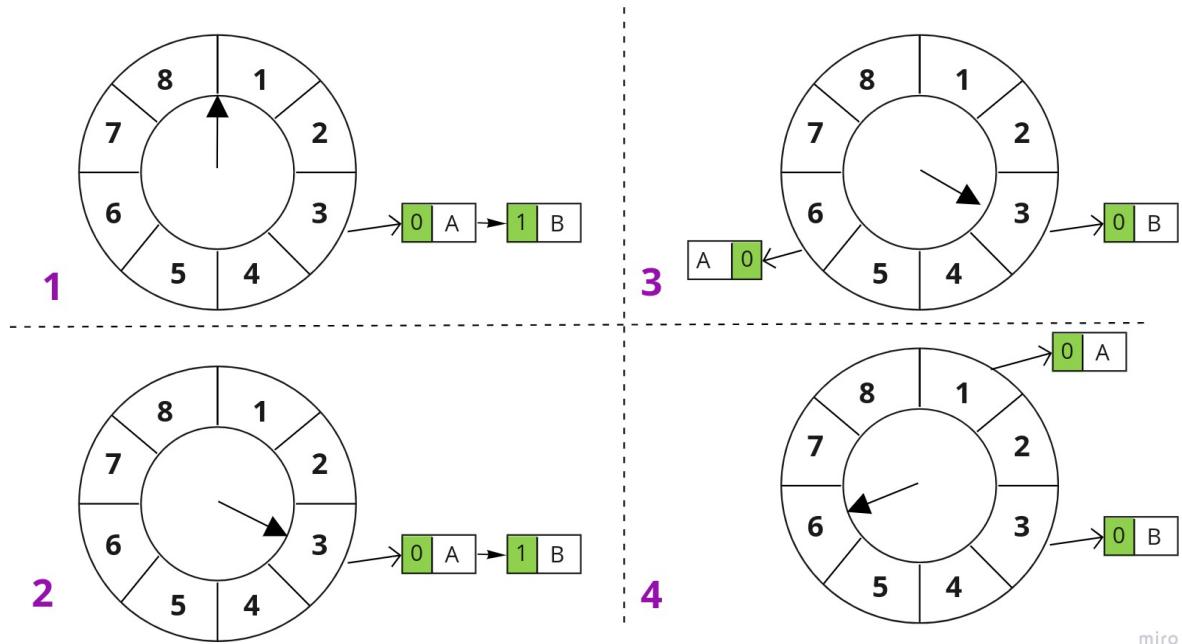
miro

一共有8个bucket，每个bucket代表tick的时间，类似于时钟，每个1秒钟走一格，我们可以定义tick的时间为1秒钟，那么bucket[1]就代表第1秒，而bucket[8]就代表第8秒，然后循环进行上述步骤。一个bucket中可能有多个任务，每个任务采用链表的方式连接起来。下面通过一个例子来说明如何添加、删除和查找任务。

假设时间轮中有8个bucket，每个bucket占用一个tick时间，每个tick为1秒。当前有2个定时任务A、B，分别需要3秒、11秒执行一次。目前指针指在0的位置，3秒钟之后指针将指向bucket[3]的位置，因此我们把任务A放入bucket[3]中，接下来我们再看如何放置任务B，任务B是11秒之后执行，也就是说时间轮转1圈之后，再过3秒种，任务B才执行，那么如何标记任务的圈数呢？这里引入了round的概念，round为1就表示需要1圈，如果round为2则需要2圈，同理推广到其它圈数。我们把B任务也放入bucket[3]，但是设置它的round为1。我们先看下任务A和任务B的执行过程，3秒钟之后时间轮转到bucket[3]，这时候检查bucket[3]中的任务，只执行round为0的任务，这里执行任务A，然后把bucket[3]中所有任务的round减1，这时候任务B的round数为0了，等到时间轮转一圈之后，就会执行任务B了。

这里还有一个疑问就是任务A执行完成之后，下一次触发如何执行，其实在bucket[3]执行完成之后，会把任务A从bucket[3]中删除，然后重新计算 $3+3$ ，放入bucket[6]中，等到bucket[6]执行完成之后，然后再放入 $(6+3)$ 对8取余，放入bucket[1]中。也就是说每次任务执行完成之

后需要重新计算任务在哪个bucket，然后放入对应的bucket中。



可以看到时间轮算法的插入复杂度是 $O(1)$ ，删除的复杂度也是 $O(1)$ ，查找执行的复杂度也是 $O(1)$ ，因此时间轮实现的定时器非常高效。

Cyber定时器实现

用户接口

Timer对象是开放给用户的接口，主要实现了定时器的配置”TimerOption”，启动定时器和关闭定时器3个接口。我们首先看下定时器的配置。

```
TimerOption(uint32_t period, std::function<void()>
callback, bool oneshot)
    : period(period), callback(callback), oneshot(oneshot)
{}
```

包括：定时器周期、回调函数、一次触发还是周期触发（默认为周期触发）。

Timer对象主要的实现都在”Start()”中。

```

void Timer::Start() {
    // 1. 首先判断定时器是否已经启动
    if (!started_.exchange(true)) {
        // 2. 初始化任务
        if (InitTimerTask()) {
            // 3. 在时间轮中增加任务
            timing_wheel_->AddTask(task_);
            AINFO << "start timer [" << task_->timer_id_ << "]";
        }
    }
}

```

Start中的步骤很简单:

1. 判断定时器是否已经启动
2. 如果定时器没有启动，则初始化定时任务
3. 在时间轮中增加任务。

那么初始化任务中做了哪些事情呢？

```

bool Timer::InitTimerTask() {
    // 1. 初始化定时任务
    task_.reset(new TimerTask(timer_id_));
    task_->interval_ms = timer_opt_.period;
    task_->next_fire_duration_ms = task_->interval_ms;
    // 2. 是否单次触发
    if (timer_opt_.oneshot) {
        std::weak_ptr<TimerTask> task_weak_ptr = task_;
        // 3. 注册任务回调
        task_->callback = [callback = this->timer_opt_.callback,
task_weak_ptr]() {
            auto task = task_weak_ptr.lock();
            if (task) {
                std::lock_guard<std::mutex> lg(task->mutex);
                callback();
            }
        };
    } else {
        std::weak_ptr<TimerTask> task_weak_ptr = task_;
        // 注册任务回调
    }
}

```

```

task_->callback = [callback = this->timer_opt_.callback,
task_weak_ptr] () {

    std::lock_guard<std::mutex> lg(task->mutex);
    auto start = Time::MonoTime().ToNanosecond();
    callback();
    auto end = Time::MonoTime().ToNanosecond();
    uint64_t execute_time_ns = end - start;

    if (task->last_execute_time_ns == 0) {
        task->last_execute_time_ns = start;
    } else {
        // start - task->last_execute_time_ns 为2次执行真实间隔
        // time, task->interval_ms是设定的间隔时间
        // 注意误差会修复补偿, 因此这里用的是累计, 2次误差会抵消, 保持绝
        对误差为0
        task->accumulated_error_ns +=
            start - task->last_execute_time_ns - task-
        >interval_ms * 1000000;
    }

    task->last_execute_time_ns = start;
    // 如果执行时间大于任务周期时间, 则下一个tick马上执行
    if (execute_time_ms >= task->interval_ms) {
        task->next_fire_duration_ms = TIMER_RESOLUTION_MS;
    } else {
        int64_t accumulated_error_ms = ::llround(
            static_cast<double>(task->accumulated_error_ns) /
        1e6);
        if (static_cast<int64_t>(task->interval_ms -
execute_time_ms -
                                TIMER_RESOLUTION_MS) >=
        accumulated_error_ms) {
            // 这里会补偿误差
            task->next_fire_duration_ms =
                task->interval_ms - execute_time_ms -
        accumulated_error_ms;
        } else {
            task->next_fire_duration_ms = TIMER_RESOLUTION_MS;
        }
    }
}

TimingWheel::Instance()->AddTask(task);

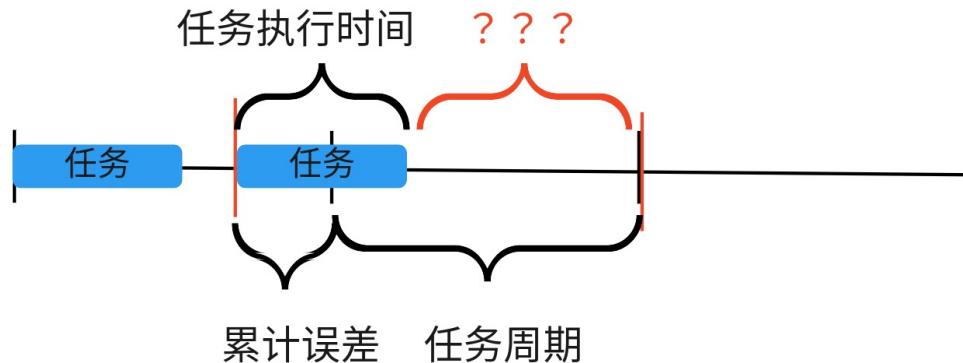
```

```
    };  
}  
return true;  
}
```

下面对Timer中初始化任务的过程做一些解释。

1. 在Timer对象中创建Task任务并注册回调”task_->callback”，任务回调中首先会调用用户传入的”callback()”函数，然后把新的任务放入下一个时间轮bucket中，对应到代码就是”TimingWheel::Instance()->AddTask(task)”。
2. task->next_fire_duration_ms是任务下一次执行的间隔，这个间隔是以task执行完成之后为起始时间的，因为每次插入新任务到时间轮都是在用户”callback”函数执行之后进行的，因此这里的时间起点也是以这个时间为基准。
3. task->accumulated_error_ns是累计时间误差，注意这个误差是累计的，而且每次插入任务的时候都会修复这个误差，因此这个误差不会一直增大，也就是说假设你第一次执行的比较早，那么累计误差为负值，下次执行的时间间隔就会变长，如果第一次执行的时间比较晚，那么累计误差为正值，下次执行的时间间隔就会缩

短。通过动态的调节，保持绝对的时间执行间隔一致。



下一次执行间隔 = 任务周期 + 累计误差 - 程序执行时间

注意：

- 1.上图中累计误差为负值（实际计算应该减累计误差）
- 2.任务间隔起始时间为任务执行回调之后的开始时间

miro

TimingWheel时间轮

接下来看时间轮TimingWheel的实现，TimingWheel时间轮的配置如下：

512个bucket
64个round
tick 为2ms

TimingWheel是通过AddTask调用执行的，下面是具体过程。

```
void TimingWheel::AddTask(const std::shared_ptr<TimerTask>& task,
                           const uint64_t current_work_wheel_index) {
    // 1.不是运行状态则启动时间轮
    if (!running_) {
        // 2.启动Tick线程，并且加入scheduler调度。
    }
}
```

```

        Start();
    }

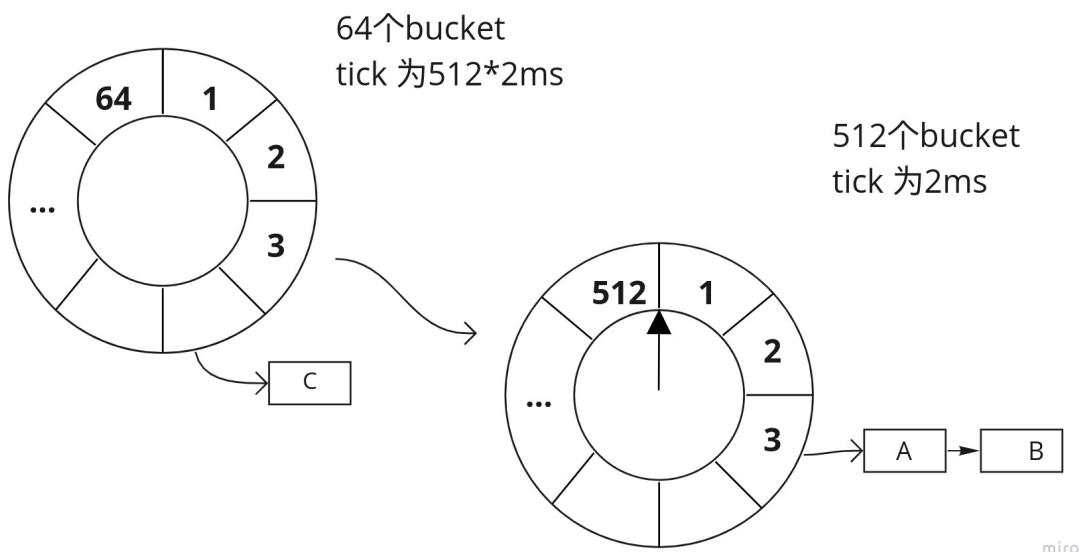
    // 3. 计算一下轮bucket编号
    auto work_wheel_index = current_work_wheel_index +
                           task->next_fire_duration_ms /
    TIMER_RESOLUTION_MS;

    // 4. 如果超过最大的bucket数
    if (work_wheel_index >= WORK_WHEEL_SIZE) {
        auto real_work_wheel_index =
GetWorkWheelIndex(work_wheel_index);
        task->remainder_interval_ms = real_work_wheel_index;
        auto assistant_ticks = work_wheel_index /
WORK_WHEEL_SIZE;
        // 5. 疑问，如果转了一圈之后，为什么直接加入剩余的bucket? ? ?
        if (assistant_ticks == 1 &&
            real_work_wheel_index != current_work_wheel_index_) {
            work_wheel_[real_work_wheel_index].AddTask(task);
            ADEBUG << "add task to work wheel. index :" <<
real_work_wheel_index;
        } else {
            auto assistant_wheel_index = 0;
            {
                // 6. 如果超出，则放入上一级时间轮中
                std::lock_guard<std::mutex>
lock(current_assistant_wheel_index_mutex_);
                assistant_wheel_index = GetAssistantWheelIndex(
                    current_assistant_wheel_index_ +
assistant_ticks);

                assistant_wheel_[assistant_wheel_index].AddTask(task);
            }
            ADEBUG << "add task to assistant wheel. index : "
            << assistant_wheel_index;
        }
    } else {
        // 7. 如果没有超过最大bucket数，则增加到对应的bucket中
        work_wheel_[work_wheel_index].AddTask(task);
        ADEBUG << "add task [" << task->timer_id_
                     << "] to work wheel. index :" << work_wheel_index;
    }
}

```

- 从上述过程可以看出Cyber的时间轮单独采用一个线程调度执行”`std::thread(this { this->TickFunc(); })`”，定时任务则放入协程池中去执行。也就是说主线程单独执行时间计数，而具体的定时任务开多个协程去执行，可以并发执行多个定时任务。定时任务中最好不要引入阻塞的操作，或者执行时间过长。
- Cyber定时器中引入了2级时间轮的方法（消息队列kafka中也是类似实现），类似时钟的小时指针和分钟指针，当一级时间轮触发完成之后，再移动到二级时间轮中执行。第二级时间轮不能超过一圈，因此定时器的最大定时时间为645122ms，最大不超过约65s。



Tick

接下来我们看下时间轮中的Tick是如何工作的。在上述”AddTask”中会调用”Start”函数启动一个线程，线程执行”TickFunc”。

```
void TimingWheel::TickFunc() {
    Rate rate(TIMER_RESOLUTION_MS * 1000000); // ms to ns
    // 1. 循环调用
    while (running_) {
        // 2. 执行bucket中的回调，并且删除当前bucket中的任务(回调中会增加新的任务到bucket)
        Tick();
    }
}
```

```
    tick_count_++;
    // 3. 休眠一个Tick
    rate.Sleep();
}

{
    std::lock_guard<std::mutex>
lock(current_work_wheel_index_mutex_);
    // 4. 获取当前bucket id, 每次加1
    current_work_wheel_index_ =
        GetWorkWheelIndex(current_work_wheel_index_ + 1);
}
// 5. 下一级时间轮已经转了一圈, 上一级时间轮加1
if (current_work_wheel_index_ == 0) {
{
    // 6. 上一级时间轮bucket id加1
    std::lock_guard<std::mutex>
lock(current_assistant_wheel_index_mutex_);
    current_assistant_wheel_index_ =
        GetAssistantWheelIndex(current_assistant_wheel_index_ + 1);
}
// 7.
Cascade(current_assistant_wheel_index_);
}
}
```

这里需要注意假设二级时间轮中有一个任务的时间周期就为512，那么在当前bucket回调中又会在当前bucket中增加一个任务，那么这个任务会执行2次，如何解决这个问题呢？Cyber中采用把这个任务放入上一级时间轮中，然后在触发一个周期之后，放到下一级的时间轮中触发。

总结

经过上述分析，介绍了Cyber中定时器的实现原理，这里还有2个疑问。

1. 一是定时器是否为单线程，任务都是在单线程中的多个协程中执行？？？

答：定时器的计数单独在一个线程中执行，具体的定时任务在协程池中执行，也就是说多个定时任务可以并发执行。

2. 当“TimingWheel::AddTask”中“work_wheel_index >= WORK_WHEEL_SIZE”并且“assistant_ticks == 1”时，假设原始的 current_work_wheel_index_mutex_ = 200，消息触发周期为600个 tick，那么按照上述计算方法得到的work_wheel_index = 800， real_work_wheel_index = 288，assistant_ticks = 1，那么“work_wheel_[real_work_wheel_index].AddTask(task)”会往288增加任务，实际上这个任务在88个tick之后就触发了？？？

data_fusion_

data_fusion_总是以第一个消息为基准，查找融合最新的消息。

Transport

Transport 把消息通过 DataDispatcher 把消息放进buffer 并且触发 DataNotifier::Notify

notify之后会触发 协程执行 而协程会调用DataVisitor::TryFetch 去取数据

取到数据之后，调用process函数执行。

ListenerHandler 和 RtpsDispatcher

是否为通知signal接收发送消息，对应一张线性表？？？

RtpsDispatcher 用来分发消息，同时触发ListenerHandler？？？

Croutine调度

什么时候采用协程，用协程做了哪些工作？？？

scheduler, task和croutine

如果有一个新的任务需要处理，则调度器会创建一个任务，而任务又由协程去处理。创建任务的时候DataVisitorBase在调度器中注册回调，这个回调触发调度器根据任务id进行NotifyProcessor

一个任务就是一个协程，协程负责调用reader enqueue读取消息，平时处于yield状态，等到DataVisitor触发回调之后开始工作。

Reference

[百度Apollo 3.5是如何设计Cyber RT计算框架的？](#)

[[https://t.cj.sina.com.cn/articles/view/6080368657/16a6b101101900fpdw?
sudaref=www.google.com&display=0&retcode=0\]](https://t.cj.sina.com.cn/articles/view/6080368657/16a6b101101900fpdw? sudaref=www.google.com&display=0&retcode=0)]

record主要用来持久化topic数据，下面我们主要分析下如何保存record数据，在分析之前心中有如下几个问题。

1. record的文件格式是什么样？是否可以压缩成更小的格式
2. 订阅多个通道时候，如何找到多个通道，如何保证时间顺序？
3. 持久化数据的时候，数据带宽不够的时候，如何保证写入磁盘呢，采用增加缓存的方式？
4. 如何设置数据的缓冲区大小？？

service介绍

cyber组件启动

cyber_launch主要用来启动cyber模块，其中一个launch文件可以有一个或者多个module，每个module包含一个dag文件，而一个dag文件则对应一个或者多个components。

launch文件中有几个module则会启动几个进程，每个进程有单独的内存空间，比如静态变量等都不会共享。

一个dag文件中可以有一个或者多个components，一个components对应一个协程。协程中的静态变量是共享的，并且全局唯一。

理解了上述原理，我们再详细分析以下2种启动方式。

1. cyber_launch
2. dreamview 实际上上述2种启动方式没有太大区别，都是通过mainboard来启动程序，而一个dag文件对应一个进程，接着根据dag文件中有几个components，生成几个协程，每个components对应一个协程。

下面介绍下cyber_launch的文件结构。

cyber_launch

```
<cyber>
  <module>
    <name>planning</name>    \\ module名称
    <dag_conf>/apollo/modules/planning/dag/planning.dag</dag_conf>
    <process_name>planning</process_name>    \\ 指定调度文件
  </module>
</cyber>
```

- module 用于区分模块
- name 模块名称，主要用来cyber_launch启动的时候显示名称

- `dag_conf` module模块对应的dag文件
- `process_name` 指定module的调度文件，如果找不到则会提示

dag

下面以`velodyne.dag`文件为例来进行说明。

```
module_config {
    module_library : "/apollo/bazel-
bin/modules/drivers/lidar/velodyne/driver/libvelodyne_driver_
component.so"
    // 模块的so文件，用于加载到内存
    # 128
    components { // 第1个组件
        class_name : "VelodyneDriverComponent"
        config {
            name : "velodyne_128_driver" // 名称必须不一样
            // 配置文件
            config_file_path :
"/apollo/modules/drivers/lidar/velodyne/conf/velodyne128_conf
.pb.txt"
        }
    }
    # 16_front_up
    components { // 第2个组件
        class_name : "VelodyneDriverComponent"
        config {
            name : "velodyne_16_front_up_driver"
            config_file_path :
"/apollo/modules/drivers/lidar/velodyne/conf/velodyne16_front
_up_conf.pb.txt"
        }
    }
}

module_config {
    module_library : "/apollo/bazel-
bin/modules/drivers/lidar/velodyne/parser/libvelodyne_convert
_component.so"
    // 模块的so文件，用于加载到内存
```

```

# 128
components {
    class_name : "VelodyneConvertComponent"
    config {
        name : "velodyne_128_convert"
        config_file_path :
"/apollo/modules/drivers/lidar/velodyne/conf/velodyne128_conf
.pb.txt"
        readers {
            channel: "/apollo/sensor/lidar128/Scan"
        }
    }
}
# 16_front_up_center
components {
    class_name : "VelodyneConvertComponent"
    config {
        name : "velodyne_16_front_up_convert"
        config_file_path :
"/apollo/modules/drivers/lidar/velodyne/conf/velodyne16_front
_up_conf.pb.txt"
        readers {
            channel: "/apollo/sensor/lidar16/front/up/Scan"
        }
    }
}
}

```

dag文件有一个或者多个module_config，而每个module_config中对应一个或者多个components。

cyber_launch.py分析

start

在start函数中会遍历module，然后通过

```

for module in root.findall('module'):
    if process_name not in process_list:

```

```

        if process_type == 'binary':
            if len(process_name) == 0:
                logger.error(
                    'Start binary failed. Binary process_name
is null.')
                continue
            pw = ProcessWrapper(
                process_name.split()[0], 0, [
                    ""], process_name, process_type,
                exception_handler)
        # Default is library
    else:
        pw = ProcessWrapper(
            g_binary_name, 0, dag_dict[
                str(process_name)], process_name,
            process_type, sched_name, exception_handler)
    result = pw.start()
    if result != 0:
        logger.error(
            'Start manager [%s] failed. Stop all!' %
process_name)
        stop()
    pmon.register(pw)
    process_list.append(process_name)

```

而ProcessWrapper中通过subprocess.Popen启动新的进程。

```

        self.popen = subprocess.Popen(args_list,
stdout=subprocess.PIPE,
stderr=subprocess.STDOUT)

```

stop

stop实际上是通过找到对应的launch文件，并且杀掉对应的进程来实现。(kill对应的是PID, pkill对应的是command)

```

def stop_launch(launch_file):
    """
    Stop the launch file
    """

```

```

if not launch_file:
    cmd = 'pkill -INT cyber_launch'    # 杀掉对应的进程
else:
    cmd = 'pkill -INT -f ' + launch_file  # 杀掉对应的进程

os.system(cmd)
time.sleep(3)
logger.info('Stop cyber launch finished.')
sys.exit(0)

```

mainboard

mainboard通过LoadModule来加载模块，而module_config来指定加载的so文件，然后启动一个或者多个components，每个components的配置文件可以不一样。

```

bool ModuleController::LoadModule(const DagConfig&
dag_config) {
    const std::string work_root = common::WorkRoot();

    for (auto module_config : dag_config.module_config()) {
        std::string load_path;
        if (module_config.module_library().front() == '/') {
            load_path = module_config.module_library();
        } else {
            load_path =
                common::GetAbsolutePath(work_root,
module_config.module_library());
        }

        if (!common::PathExists(load_path)) {
            AERROR << "Path does not exist: " << load_path;
            return false;
        }
        // 1. 加载模块的so文件
        class_loader_manager_.LoadLibrary(load_path);

        // 2.1 加载触发模块
        for (auto& component : module_config.components()) {
            const std::string& class_name = component.class_name();

```

```

        std::shared_ptr<ComponentBase> base =
            class_loader_manager_.CreateClassObj<ComponentBase>
(class_name);
        if (base == nullptr || !base-
>Initialize(component.config())) {
            return false;
        }
        component_list_.emplace_back(std::move(base));
    }

// 2.2 加载定时模块
for (auto& component : module_config.timer_components())
{
    const std::string& class_name = component.class_name();
    std::shared_ptr<ComponentBase> base =
        class_loader_manager_.CreateClassObj<ComponentBase>
(class_name);
    if (base == nullptr || !base-
>Initialize(component.config())) {
        return false;
    }
    component_list_.emplace_back(std::move(base));
}
}

return true;
}

```

dreamview

dreamview通过界面上的滑动按钮打开和关闭模块，大概的流程是前端通过websocket发送消息到后端，后端通过调用命令行启动模块，后端主要的实现在”hmi_worker.cc”中。

dreamview中的modules通过指定”start_command”和”stop_command”来启动和关闭。

LoadMode

对于so文件，会通过LoadMode来进行生成命令，下面介绍下生成”start_command”和”stop_command”的过程。生成的命令和cyber_launch中调用的命令一致，只是前面增加了nohup。

```
HMIMode HMIWorker::LoadMode(const std::string&
mode_config_path) {
    HMIMode mode;
    ACHECK(cyber::common::GetProtoFromFile(mode_config_path,
&mode))
        << "Unable to parse HMIMode from file " <<
mode_config_path;
    // Translate cyber_modules to regular modules.
    for (const auto& iter : mode.cyber_modules()) {
        const std::string& module_name = iter.first;
        const CyberModule& cyber_module = iter.second;
        // Each cyber module should have at least one dag file.
        ACHECK(!cyber_module.dag_files().empty())
            << "None dag file is provided for " << module_name <<
" module in "
            << mode_config_path;

        Module& module = LookupOrInsert(mode.mutable_modules(),
module_name, {});
        module.set_required_for_safety(cyber_module.required_for_safety());

        // 1. Construct start_command:
        //      nohup mainboard -p <process_group> -d <dag> ... &
        module.set_start_command("nohup mainboard");
        const auto& process_group = cyber_module.process_group();
        if (!process_group.empty()) {
            absl::StrAppend(module.mutable_start_command(), " -p ",
process_group);
        }
        for (const std::string& dag : cyber_module.dag_files()) {
            absl::StrAppend(module.mutable_start_command(), " -d ",
dag);
        }
        absl::StrAppend(module.mutable_start_command(), " &");

        // 2. Construct stop_command: pkill -f '<dag[0]>'
```

```

    const std::string& first_dag = cyber_module.dag_files(0);
    module.set_stop_command(absl::StrCat("pkill -f \"",
    first_dag, "\"));
    // Construct process_monitor_config.
    module.mutable_process_monitor_config()->add_command_keywords("mainboard");
    module.mutable_process_monitor_config()->add_command_keywords(first_dag);
}
mode.clear_cyber_modules();
return mode;
}

```

接着调用void HMIWorker::StartModule(const std::string& module) const来启动模块，启动调用的std::system()来启动命令行，也是一个dag文件对应一个进程。dag中的模块都对应协程。

用途

如何启动2个一模一样的模块

1. 如果你需要启动2个相同的模块，可以在launch文件中增加2个module，并且修改dag文件中模块的名称，这样就可以启动2个一模一样的模块了。上述这种情况在模块中有静态变量，并且你希望静态变量不相互影响的情况下可以采用。
2. 如果需要在同一个进程中启动2个模块，可以在dag中增加components，然后修改config中的名称和配置文件，这样就可以启动2个一模一样的模块了。上述这种情况静态变量会相互影响，A模块改了，B模块的静态变量值也改变了。

perception中的inner消息订阅不到？perception模块中的inner消息只有在同一个进程中才会收到，也就是在同一个dag文件中启动的模块才会接收到，因为reader读取的时候会判断2个节点是否是在同一个进程，如果是同一个进程就直接传递对象，不进行序列化，如果不是同一个进程则先进行序列化，然后再通过共享内存通信。

transport传输

分为Transmitter和Receiver

Transmitter

Transmitter派生了4种类型”ShmTransmitter”， ”RtpsTransmitter”， ”IntraTransmitter”， ”HybridTransmitter”。

SegmentFactory::CreateSegment

创建通道的内存分区？

transceiver

接收发送测试程序。

Contents:

- [2. Audio](#)
 - [2.1. Audio模块介绍](#)
 - [2.2. 目录结构](#)
 - [2.3. 声音识别 \(SirenDetection\)](#)
 - [2.4. 移动检测\(MovingDetection\)](#)
 - [2.5. 方向检测\(DirectionDetection\)](#)
 - [2.6. 工具 \(tools\)](#)
- [3. Bridge](#)
 - [3.1. Table of Contents](#)
- [4. Canbus](#)
 - [4.1. Canbus模块介绍](#)
 - [4.2. Canbus模块主流程](#)
 - [4.3. Canbus\(驱动程序\)](#)
 - [4.4. Reference](#)
- [5. Control](#)
 - [5.1. Control模块简介](#)
 - [5.2. Control模块主流程](#)
 - [5.3. 控制器](#)
 - [5.4. LatController控制器](#)
 - [5.5. MPCCController控制器](#)
 - [5.6. StanleyLatController控制器](#)
 - [5.7. 问题](#)
 - [5.8. Reference](#)
- [6. Data](#)
 - [6.1. data目录结构](#)
 - [6.2. RecordViewer](#)
- [7. Dreamview](#)
- [8. Drivers](#)
 - [8.1. Table of Contents](#)
 - [8.2. Reference](#)
- [9. Guardian](#)
 - [9.1. 简介](#)
 - [9.2. 触发](#)
 - [9.3. TriggerSafetyMode](#)

- [9.4. 问题](#)
- [10. Localization](#)
 - [10.1. Localization模块简介](#)
 - [10.2. 代码目录](#)
 - [10.3. RTK定位流程](#)
 - [10.4. Reference](#)
- [11. Map](#)
 - [11.1. Map模块简介](#)
 - [11.2. Map目录结构](#)
 - [11.3. 地图数据结构](#)
 - [11.4. opendriver地图解析](#)
 - [11.5. 高精度地图API](#)
 - [11.6. tools](#)
 - [11.7. 如何制作高精度地图](#)
 - [11.8. Reference](#)
- [12. Monitor](#)
 - [12.1. 简介](#)
 - [12.2. MonitorManager](#)
 - [12.3. hardware](#)
 - [12.4. software](#)
- [13. Perception](#)
 - [13.1. Perception模块简介](#)
 - [13.2. production目录](#)
 - [13.3. onboard目录](#)
 - [13.4. 子模块介绍](#)
 - [13.5. Reference](#)
- [14. Planning](#)
 - [14.1. Planning模块简介](#)
 - [14.2. Planning模块入口](#)
 - [14.3. OnLanePlanning](#)
 - [14.4. Planner](#)
 - [14.5. Scenario](#)
 - [14.6. Task](#)
 - [14.7. Reference](#)
- [15. Prediction](#)
 - [15.1. 介绍](#)

- [15.2. 目录结构](#)
- [15.3. 预测模块\(PredictionComponent类\)](#)
- [15.4. 消息处理\(MessageProcess\)](#)
- [15.5. 容器\(container\)](#)
- [15.6. 场景\(scenario\)](#)
- [15.7. 评估者\(evaluator\)](#)
- [15.8. 预测器\(predictor\)](#)
- [15.9. Reference](#)
- [16. Routing](#)
 - [16.1. Routing模块简介](#)
 - [16.2. 基础知识](#)
- [17. Routing模块分析](#)
 - [17.1. 创建Routing地图](#)
 - [17.2. Routing主流程](#)
 - [17.3. 调试工具](#)
 - [17.4. 问题](#)
 - [17.5. OSM数据查找](#)
 - [17.6. Reference](#)
- [18. Tools](#)
 - [18.1. mapviewers](#)
- [19. Transform](#)
 - [19.1. Transform模块简介](#)
 - [19.2. Transform\(静态变换\)](#)
 - [19.3. transform broadcaster \(广播\)](#)
 - [19.4. Buffer \(接收缓存\)](#)
 - [19.5. 总结](#)
 - [19.6. Reference](#)
- [20. V2X](#)
 - [20.1. v2x目录结构](#)
 - [20.2. v2x_proxy](#)
 - [20.3. OnV2xCarStatusTimer](#)
 - [20.4. RecvTrafficlight](#)
 - [20.5. RecvOsPlanning](#)
 - [20.6. obs_thread](#)
 - [20.7. OBU接口\(ObuInterFaceGrpcImpl\)](#)
 - [20.8. 系统接口\(OsInterFace\)](#)

- [20.9. 感知模块](#)
- [20.10. fusion模块](#)
- [20.11. Fusion类](#)
- [20.12. KMkernal](#)
- [20.13. trans_tools](#)
- [20.14. V2x消息](#)

1. 在紧急情况下，control模块应该有自己处理汽车当前状况的能力。而什么是紧急状况，紧急状况的定义是什么呢？首先的一点就是规划命令有3秒没有下发（是否一定要触发停车？），或者一些紧急情况，需要立刻停车的场景？

todo

1. 各个控制器的理解和推导

Reference

<https://www.mathworks.com/help/mpc/ug/adaptive-cruise-control-using-model-predictive-controller.html>

[http://ctms.engin.umich.edu/CTMS/index.php?
example=CruiseControl§ion=SystemModeling](http://ctms.engin.umich.edu/CTMS/index.php?example=CruiseControl§ion=SystemModeling)

<https://www.mathworks.com/help/physmod/sdl/ug/about-the-complete-vehicle-model.html>

<https://www.mathworks.com/help/physmod/sdl/ug/control-vehicle-throttle-input-using-a-powertrain-blockset-driver.html>

camera摄像头

Camera模块支持USB UVC类型的摄像头，也就是说只有符合UVC规范的摄像头才支持， Camera的实现参考了linux驱动中V4L2的内核代码。

Dig into Apollo - Drivers/canbus

license MIT

一叶遮目，不见泰山

Table of Contents

- Canbus
 - Linux Canbus
- Reference

Canbus

由于汽车采用的总线是canbus总线，因此汽车采用大部分的器件都是canbus，其中**Radar**，线控系统采用的就是**canbus**总线，当然**canbus**总线也有缺点，就是带宽不够，所以**Camera**采用的就是**USB**总线，个人认为后面汽车可能会有新的总线出现，一满足目前的高带宽，二满足可扩展；另外的一个发展方向就是各个模块的集成化发展，比如以后摄像头和**Radar**一样都有单独的硬件处理数据之后上报。

那么我们再看下**radar**的驱动实际上依赖**canbus**，只有装好**canbus**的驱动，**radar**才可以收发数据。



apollo的**canbus**驱动在”third_party/can_card_library”目录中，主要是引用linux的动态链接库”lib/libntcan.so.4”，我们从linux操作系统开始，把整个**canbus**的调用流程分析一遍。

Linux Canbus

首先下载linux kernel(这里以linux-3.16.64举例)，**Canbus**的驱动主要在”drivers/net/can”目录中，编译内核的时候可以打开对应的开关来指定内核是否编译这些模块。**Canbus**驱动的目录结构如下：

```
.  
|   |-- at91_can.c  
|   |-- bfin_can.c  
|   |-- cc770  
|   |-- c_can  
|   |-- dev.c  
|   |-- flexcan.c  
|   |-- grcan.c  
|   |-- janz-ican3.c  
|   |-- Kconfig  
|   |-- led.c  
|   |-- Makefile  
|   |-- mscan  
|   |-- pch_can.c  
|   |-- rcar_can.c  
|   |-- sja1000  
|   |-- slcan.c  
|   |-- softing  
|   |-- spi  
|   |-- ti_hecc.c  
|   |-- usb  
|   |-- vcan.c  
|   |-- xilinx_can.c
```

Reference

[linux](https://www.kernel.org/) [https://www.kernel.org/]

Dig into Apollo - Drivers/Gnss

license MIT

一叶遮目，不见泰山

Table of Contents

- Gnss
- Reference

Gnss

这里的坐标转换采用的是”proj.4”库来实现。从WGS84坐标转换为UTM坐标。proj.4的参数找了下关于参数的介绍，发现都没有介绍的非常详细。这篇文章翻译的还比较详细，但是英文原版已经找不到了。

[参考](https://www.cnblogs.com/jiangleads/articles/10803464.html) [https://www.cnblogs.com/jiangleads/articles/10803464.html]

Reference

[linux](https://www.kernel.org/) [https://www.kernel.org/]

Dig into Apollo - Microphone

license MIT

遥知兄弟登高处，遍插茱萸少一人。

Table of Contents

Canbus驱动子模块介绍

目录结构

Microphone的主要用途是捕获车当前的声音信息。

Dig into Apollo - Drivers

license MIT

一叶遮目，不见泰山

Table of Contents

- Radar
- Reference

Radar

Radar驱动包含了2种类型的雷达：毫米波雷达和超声波雷达。其中毫米波雷达有博世和理工雷科2种类型，我尝试着在网上找相关资料，暂时没有找到相关的驱动程序，只找到了博世的说明文档，下面根据Apollo中的代码做一个介绍。

```
.  
├── conti_radar  
├── racabit_radar  
└── ultrasonic_radar
```

Reference

[linux](https://www.kernel.org/) [https://www.kernel.org/]

video

video是通过网络UDP来获取录像并发布，看起来是支持网络摄像头，即接口是网口的摄像头。

RTKLocalization

RTKLocalization类的主要功能是实现RTK定位，然后用IMU做插值。

1. 只是通过GPS得到了位置，但是IMU的融合还没有进行计算？？？
2. 姿态解算是如何进行的？？？

InitConfig

```
void RTKLocalization::InitConfig(const rtk_config::Config &config) {
    // imu队列最大长度
    imu_list_max_size_ = config.imu_list_max_size();
    // GPS和IMU最大时间间隔
    gps_imu_time_diff_threshold_ =
    config.gps_imu_time_diff_threshold();
    // 地图偏移
    map_offset_[0] = config.map_offset_x();
    map_offset_[1] = config.map_offset_y();
    map_offset_[2] = config.map_offset_z();
}
```

GpsCallback

```
void RTKLocalization::GpsCallback(
    const std::shared_ptr<localization::Gps> &gps_msg) {
    double time_delay =
        last_received_timestamp_sec_
            ? common::time::Clock::NowInSeconds() -
    last_received_timestamp_sec_
            : last_received_timestamp_sec_;
    // GPS超时
    if (time_delay > gps_time_delay_tolerance_) {
        std::stringstream ss;
        ss << "GPS message time interval: " << time_delay;
        monitor_logger_.WARN(ss.str());
```

```

    }

    {
        std::unique_lock<std::mutex> lock(imu_list_mutex_);
        // 判断IMU消息是否为空，为空则返回
        if (imu_list_.empty()) {
            AERROR << "IMU message buffer is empty.";
            if (service_started_) {
                monitor_logger_.ERROR("IMU message buffer is
empty.");
            }
            return;
        }
    }

    {
        std::unique_lock<std::mutex>
lock(gps_status_list_mutex_);
        // 判断GPS状态是否为空，为空则返回
        if (gps_status_list_.empty()) {
            AERROR << "Gps status message buffer is empty.";
            if (service_started_) {
                monitor_logger_.ERROR("Gps status message buffer is
empty.");
            }
            return;
        }
    }

    // publish localization messages
    // 主要的处理过程都在该函数中
    PrepareLocalizationMsg(*gps_msg,
    &last_localization_result_,
                           &last_localization_status_result_);

    // 开启定位服务，并且记录服务开始时间
    service_started_ = true;
    if (service_started_time == 0.0) {
        service_started_time =
common::time::Clock::NowInSeconds();
    }

    // watch dog

```

```

// 看门狗，主要根据时间来判断? ?
RunWatchDog(gps_msg->header().timestamp_sec());

// 最后一次消息接收时间
last_received_timestamp_sec_ =
common::time::Clock::NowInSeconds();
}

```

开门狗

```

void RTKLocalization::RunWatchDog(double gps_timestamp) {
    if (!enable_watch_dog_) {
        return;
    }

    // 定位服务启动后，GPS超时则设置msg_delay为真
    // check GPS time stamp against system time
    double gps_delay_sec = common::time::Clock::NowInSeconds()
- gps_timestamp;
    double gps_service_delay =
        common::time::Clock::NowInSeconds() -
service_started_time;
    int64_t gps_delay_cycle_cnt =
        static_cast<int64_t>(gps_delay_sec *
localization_publish_freq_);

    bool msg_delay = false;
    if (gps_delay_cycle_cnt > report_threshold_err_num_ &&
        static_cast<int>(gps_service_delay) >
service_delay_threshold) {
        msg_delay = true;
        std::stringstream ss;
        ss << "Raw GPS Message Delay. GPS message is " <<
gps_delay_cycle_cnt
            << " cycle " << gps_delay_sec << " sec behind current
time.";
        monitor_logger_.ERROR(ss.str());
    }

    // 定位服务启动之后，IMU超时则设置msg_delay为真
    // check IMU time stamp against system time
    std::unique_lock<std::mutex> lock(imu_list_mutex_);

```

```

auto imu_msg = imu_list_.back();
lock.unlock();
double imu_delay_sec =
    common::time::Clock::NowInSeconds() -
imu_msg.header().timestamp_sec();
int64_t imu_delay_cycle_cnt =
    static_cast<int64_t>(imu_delay_sec *
localization_publish_freq_);
if (imu_delay_cycle_cnt > report_threshold_err_num_ &&
    static_cast<int>(gps_service_delay) >
service_delay_threshold) {
    msg_delay = true;
    std::stringstream ss;
    ss << "Raw IMU Message Delay. IMU message is " <<
imu_delay_cycle_cnt
        << " cycle " << imu_delay_sec << " sec behind current
time.";
    monitor_logger_.ERROR(ss.str());
}
// 第一次触发之后，然后每隔1s触发一次
// to prevent it from beeping continuously
if (msg_delay && (last_reported_timestamp_sec_ < 1. ||

common::time::Clock::NowInSeconds() >
last_reported_timestamp_sec_ + 1.)) {
    AERROR << "gps/imu frame Delay!";
    last_reported_timestamp_sec_ =
common::time::Clock::NowInSeconds();
}
}

```

定位主流程

```

void RTKLocalization::PrepareLocalizationMsg(
    const localization::Gps &gps_msg, LocalizationEstimate
*localization,
    LocalizationStatus *localization_status) {
// 根据GPS时间寻找匹配的IMU消息
// find the matching gps and imu message
double gps_time_stamp = gps_msg.header().timestamp_sec();
CorrectedImu imu_msg;
FindMatchingIMU(gps_time_stamp, &imu_msg);

```

```

// 构建定位消息
ComposeLocalizationMsg(gps_msg, imu_msg, localization);

drivers::gnss::InsStat gps_status;
// 查找最近的GPS状态信息
FindNearestGpsStatus(gps_time_stamp, &gps_status);
// 填充定位状态信息
FillLocalizationStatusMsg(gps_status, localization_status);
}

```

找到正确的IMU消息

在队列中找到最匹配的IMU，其中区分了队列的第一个，最后一个，以及如果在中间则进行插值。插值的时候根据距离最近的原则进行反比例插值。

```

bool RTKLocalization::FindMatchingIMU(const double
gps_timestamp_sec,
                                         CorrectedImu *imu_msg)
{
    // 1. 判断消息是否为空
    if (imu_msg == nullptr) {
        AERROR << "imu_msg should NOT be nullptr.";
        return false;
    }
    // 加锁，这里有疑问，为什么换个变量就没有锁了呢？
    std::unique_lock<std::mutex> lock(imu_list_mutex_);
    auto imu_list = imu_list_;
    lock.unlock();

    // IMU为空
    if (imu_list.empty()) {
        AERROR << "Cannot find Matching IMU. "
              << "IMU message Queue is empty! GPS timestamp[" <<
gps_timestamp_sec
              << "]";
        return false;
    }
}

```

```

// scan imu buffer, find first imu message that is newer
// than the given
// timestamp
auto imu_it = imu_list.begin();
for (; imu_it != imu_list.end(); ++imu_it) {
    if ((*imu_it).header().timestamp_sec() -
gps_timestamp_sec >
        std::numeric_limits<double>::min()) {
        break;
    }
}

if (imu_it != imu_list.end()) { // found one
    if (imu_it == imu_list.begin()) {
        AERROR << "IMU queue too short or request too old. "
              << "Oldest timestamp[" <<
imu_list.front().header().timestamp_sec()
              << "], Newest timestamp["
              << imu_list.back().header().timestamp_sec() <<
"], GPS timestamp["
              << gps_timestamp_sec << "]";
        *imu_msg = imu_list.front(); // the oldest imu
    } else {
        // here is the normal case
        auto imu_it_1 = imu_it;
        imu_it_1--;
        if (!(*imu_it).has_header() || !
(*imu_it_1).has_header()) {
            AERROR << "imul and imu_it_1 must both have header.";
            return false;
        }
        if (!InterpolateIMU(*imu_it_1, *imu_it,
gps_timestamp_sec, imu_msg)) {
            AERROR << "failed to interpolate IMU";
            return false;
        }
    }
} else {
    // 如果没有找到, 则取最新的20ms以内的消息
    // give the newest imu, without extrapolation
    *imu_msg = imu_list.back();
    if (imu_msg == nullptr) {

```

```

        AERROR << "Fail to get latest observed imu_msg.";
        return false;
    }

    if (!imu_msg->has_header()) {
        AERROR << "imu_msg must have header.";
        return false;
    }

    if (std::fabs(imu_msg->header().timestamp_sec() -
gps_timestamp_sec) >
        gps_imu_time_diff_threshold_) {
        // 20ms threshold to report error
        AERROR << "Cannot find Matching IMU. IMU messages too
old. "
                << "Newest timestamp[" <<
imu_list.back().header().timestamp_sec()
                << "], GPS timestamp[" << gps_timestamp_sec <<
"]";
    }
}

return true;
}

```

IMU插值

```

bool RTKLocalization::InterpolateIMU(const CorrectedImu
&imu1,
                                      const CorrectedImu
&imu2,
                                      const double
timestamp_sec,
                                      CorrectedImu *imu_msg) {
    if (!(imu1.header().has_timestamp_sec() &&
          imu2.header().has_timestamp_sec())))
        AERROR << "imu1 and imu2 has no header or no
timestamp_sec in header";
    return false;
}
if (timestamp_sec - imu1.header().timestamp_sec() <
    std::numeric_limits<double>::min()) {

```

```

    AERROR << "[InterpolateIMU1]: the given time stamp[" <<
timestamp_sec
        << "] is older than the 1st message["
        << imu1.header().timestamp_sec() << "]";
    *imu_msg = imu1;
} else if (timestamp_sec - imu2.header().timestamp_sec() >
           std::numeric_limits<double>::min()) {
    AERROR << "[InterpolateIMU2]: the given time stamp[" <<
timestamp_sec
        << "] is newer than the 2nd message["
        << imu2.header().timestamp_sec() << "]";
    *imu_msg = imu1;
} else {
    // 线性插值
    *imu_msg = imu1;
    imu_msg->mutable_header()-
>set_timestamp_sec(timestamp_sec);

    double time_diff =
        imu2.header().timestamp_sec() -
    imu1.header().timestamp_sec();
    if (fabs(time_diff) >= 0.001) {
        double frac1 =
            (timestamp_sec - imu1.header().timestamp_sec()) /
time_diff;

        if (imu1 imu().has_angular_velocity() &&
            imu2 imu().has_angular_velocity()) {
            auto val =
InterpolateXYZ(imu1 imu().angular_velocity(),
imu2 imu().angular_velocity(), frac1);
            imu_msg->mutable_imu()->mutable_angular_velocity()-
>CopyFrom(val);
        }

        if (imu1 imu().has_linear_acceleration() &&
            imu2 imu().has_linear_acceleration()) {
            auto val =
InterpolateXYZ(imu1 imu().linear_acceleration(),
imu2 imu().linear_acceleration(), frac1);
            imu_msg->mutable_imu()-

```

```

>mutable_linear_acceleration() ->CopyFrom(val);
}

    if (imu1.imu().has_euler_angles() &&
imu2.imu().has_euler_angles()) {
        auto val = InterpolateXYZ(imu1.imu().euler_angles(),
                                imu2.imu().euler_angles(),
frac1);
        imu_msg->mutable_imu()->mutable_euler_angles()-
>CopyFrom(val);
    }
}
return true;
}

```

线性插值算法

根据距离插值，反比例，即frac1越小，则越靠近p1，frac1越大，则越靠近p2

```

template <class T>
T RTKLocalization::InterpolateXYZ(const T &p1, const T &p2,
                                    const double frac1) {
    T p;
    double frac2 = 1.0 - frac1;
    if (p1.has_x() && !std::isnan(p1.x()) && p2.has_x() &&
!std::isnan(p2.x())) {
        p.set_x(p1.x() * frac2 + p2.x() * frac1);
    }
    if (p1.has_y() && !std::isnan(p1.y()) && p2.has_y() &&
!std::isnan(p2.y())) {
        p.set_y(p1.y() * frac2 + p2.y() * frac1);
    }
    if (p1.has_z() && !std::isnan(p1.z()) && p2.has_z() &&
!std::isnan(p2.z())) {
        p.set_z(p1.z() * frac2 + p2.z() * frac1);
    }
    return p;
}

```

填充定位消息

找到最匹配的IMU消息后，和GPS消息做融合。IMU的角度是否不随着物体的旋转而改变？？？涉及到姿态解算

https://blog.csdn.net/MOU_IT/article/details/80369043

<https://zhuanlan.zhihu.com/p/79894982>

<https://zhuanlan.zhihu.com/p/20382236>

位置，航向，线速度是GPS的 加速度，角速度，欧拉角是IMU的

```
void RTKLocalization::ComposeLocalizationMsg(
    const localization::Gps &gps_msg, const
localization::CorrectedImu &imu_msg,
    LocalizationEstimate *localization) {
localization->Clear();

FillLocalizationMsgHeader(localization);

localization-
>set_measurement_time(gps_msg.header().timestamp_sec());

// combine gps and imu
auto mutable_pose = localization->mutable_pose();
// GPS消息包含位置信息
if (gps_msg.has_localization()) {
    const auto &pose = gps_msg.localization();
    // 1. 获取位置
    if (pose.has_position()) {
        // position
        // world frame -> map frame
        mutable_pose->mutable_position()-
>set_x(pose.position().x() -
map_offset_[0]);
        mutable_pose->mutable_position()-
>set_y(pose.position().y() -
map_offset_[1]);
        mutable_pose->mutable_position()-
>set_z(pose.position().z() -
```

```

map_offset_[2]);
}
// 2. 获取方向
// orientation
if (pose.has_orientation()) {
    mutable_pose->mutable_orientation()-
>CopyFrom(pose.orientation());
    double heading = common::math::QuaternionToHeading(
        pose.orientation().qw(), pose.orientation().qx(),
        pose.orientation().qy(), pose.orientation().qz());
    mutable_pose->set_heading(heading);
}
// linear velocity
// 获取线速度
if (pose.has_linear_velocity()) {
    mutable_pose->mutable_linear_velocity()-
>CopyFrom(pose.linear_velocity());
}
}

if (imu_msg.has_imu()) {
    const auto &imu = imu_msg.imu();
    // linear acceleration
    // 获取imu的线性
    if (imu.has_linear_acceleration()) {
        if (localization->pose().has_orientation()) {
            // linear_acceleration:
            // convert from vehicle reference to map reference
            // 为什么需要做旋转? ? ? 转换为车当前方向的速度? ? ?
            Vector3d orig(imu.linear_acceleration().x(),
                          imu.linear_acceleration().y(),
                          imu.linear_acceleration().z());
            Vector3d vec = common::math::QuaternionRotate(
                localization->pose().orientation(), orig);
            mutable_pose->mutable_linear_acceleration()-
>set_x(vec[0]);
            mutable_pose->mutable_linear_acceleration()-
>set_y(vec[1]);
            mutable_pose->mutable_linear_acceleration()-
>set_z(vec[2]);
        }
        // linear_acceleration_vfr
        // 设置线性加速度
    }
}

```

```

        mutable_pose->mutable_linear_acceleration_vrf()-
>CopyFrom(
            imu.linear_acceleration());
    } else {
        AERROR << "[PrepareLocalizationMsg]: "
                << "fail to convert linear_acceleration";
    }
}

//设置角速度，也需要根据航向转换
// angular velocity
if (imu.has_angular_velocity()) {
    if (localization->pose().has_orientation()) {
        // angular_velocity:
        // convert from vehicle reference to map reference
        Vector3d orig(imu.angular_velocity().x(),
imu.angular_velocity().y(),
                           imu.angular_velocity().z());
        Vector3d vec = common::math::QuaternionRotate(
            localization->pose().orientation(), orig);
        mutable_pose->mutable_angular_velocity()-
>set_x(vec[0]);
        mutable_pose->mutable_angular_velocity()-
>set_y(vec[1]);
        mutable_pose->mutable_angular_velocity()-
>set_z(vec[2]);

        // angular_velocity_vf
        mutable_pose->mutable_angular_velocity_vrf()-
>CopyFrom(
            imu.angular_velocity());
    } else {
        AERROR << "[PrepareLocalizationMsg]: fail to convert
angular_velocity";
    }
}

// 设置欧拉角
// euler angle
if (imu.has_euler_angles()) {
    mutable_pose->mutable_euler_angles()-
>CopyFrom(imu.euler_angles());
}

```

```
    }  
}
```

设置localization消息头

填充消息头

```
void RTKLocalization::FillLocalizationMsgHeader(  
    LocalizationEstimate *localization) {  
    auto *header = localization->mutable_header();  
    double timestamp =  
        apollo::common::time::Clock::NowInSeconds();  
    header->set_module_name(module_name_);  
    header->set_timestamp_sec(timestamp);  
    header->set_sequence_num(static_cast<unsigned int>  
        (++localization_seq_num_));  
}
```

查找最新的GPS状态

GPS状态列表可能是乱序的吗？不是按照时间顺序排列的？？？

```
bool RTKLocalization::FindNearestGpsStatus(const double  
    gps_timestamp_sec,  
  
    drivers::gnss::InsStat *status) {  
    CHECK_NONNULL(status);  
  
    std::unique_lock<std::mutex> lock(gps_status_list_mutex_);  
    auto gps_status_list = gps_status_list_;  
    lock.unlock();  
  
    double timestamp_diff_sec = 1e8;  
    auto nearest_itr = gps_status_list.end();  
    for (auto itr = gps_status_list.begin(); itr !=  
        gps_status_list.end();  
        ++itr) {  
        double diff = std::abs(itr->header().timestamp_sec() -  
            gps_timestamp_sec);
```

```

        if (diff < timestamp_diff_sec) {
            timestamp_diff_sec = diff;
            nearest_itr = itr;
        }
    }

    if (nearest_itr == gps_status_list.end()) {
        return false;
    }

    if (timestamp_diff_sec > gps_status_time_diff_threshold_) {
        return false;
    }

    *status = *nearest_itr;
    return true;
}

```

增加位置的状态信息

设置位置的状态，主要是为RTK的状态信息

```

void RTKLocalization::FillLocalizationStatusMsg(
    const drivers::gnss::InsStat &status,
    LocalizationStatus *localization_status) {
    apollo::common::Header *header = localization_status-
>mutable_header();
    double timestamp =
apollo::common::time::Clock::NowInSeconds();
    header->set_timestamp_sec(timestamp);
    localization_status-
>set_measurement_time(status.header().timestamp_sec());

    // 如果没有pose type则返回错误
    if (!status.has_pos_type()) {
        localization_status-
>set_fusion_status(MeasureState::ERROR);
        localization_status->set_state_message(
            "Error: Current Localization Status Is Missing.");
    }
}

```

```

    }

    auto pos_type = static_cast<drivers::gnss::SolutionType>
(status.pos_type());
    switch (pos_type) {
        // RTK FIXED状态
        case drivers::gnss::SolutionType::INS_RTKFIXED:
            localization_status-
>set_fusion_status(MeasureState::OK);
            localization_status->set_state_message("");
            break;
        // RTK FLOAT状态
        case drivers::gnss::SolutionType::INS_RTKFLOAT:
            localization_status-
>set_fusion_status(MeasureState::WARNING);
            localization_status->set_state_message(
                "Warning: Current Localization Is Unstable.");
            break;
        default:
            localization_status-
>set_fusion_status(MeasureState::ERROR);
            localization_status->set_state_message(
                "Error: Current Localization Is Very Unstable.");
            break;
    }
}

```

PublishPoseBroadcastTopic

输出无人车的位置，为什么没有通过IMU解算真实的位置信息？？？

```

void RTKLocalizationComponent::PublishPoseBroadcastTopic(
    const LocalizationEstimate& localization) {
    localization_talker_->Write(localization);
}

```

PublishPoseBroadcastTF

输出坐标转换信息，主要发布位置、航向。

```
void RTKLocalizationComponent::PublishPoseBroadcastTF(
    const LocalizationEstimate& localization) {
    // broadcast tf message
    apollo::transform::TransformStamped tf2_msg;

    auto mutable_head = tf2_msg.mutable_header();
    mutable_head-
>set_timestamp_sec(localization.measurement_time());
    mutable_head->set_frame_id(broadcast_tf_frame_id_);
    tf2_msg.set_child_frame_id(broadcast_tf_child_frame_id_);

    auto mutable_translation = tf2_msg.mutable_transform()-
>mutable_translation();
    mutable_translation-
>set_x(localization.pose().position().x());
    mutable_translation-
>set_y(localization.pose().position().y());
    mutable_translation-
>set_z(localization.pose().position().z());

    auto mutable_rotation = tf2_msg.mutable_transform()-
>mutable_rotation();
    mutable_rotation-
>set_qx(localization.pose().orientation().qx());
    mutable_rotation-
>set_qy(localization.pose().orientation().qy());
    mutable_rotation-
>set_qz(localization.pose().orientation().qz());
    mutable_rotation-
>set_qw(localization.pose().orientation().qw());

    tf2_broadcaster_->SendTransform(tf2_msg);
}
```

PublishLocalizationStatus

发布位置状态信息

```
void RTKLocalizationComponent::PublishLocalizationStatus(
    const LocalizationStatus& localization_status) {
    localization_status_talker_->Write(localization_status);
}
```

NDT

1. NDT的原理？

NDT mapping NDT match

我们主要应用NDT（Normal Distributions Transform，正态分布变换）或者其他SLAM算法来完成稠密点云地图的构建。

<http://xchu.net/2019/09/27/HDMAP%E5%BB%BA%E5%9B%BE%E6%B5%81%E7%A8%8B/>

<http://xchu.net/2019/10/11/autoware%E6%BF%80%E5%85%89%E9%9B%B7%E8%BE%BE%E5%BB%BA%E5%9B%BE%E5%92%8C%E5%AE%9A%E4%BD%8D/>

Dig into Apollo - Localization

license MIT

map_creation

加载pcd pose文件

```
apollo::localization::msf::velodyne::LoadPcdPoses(  
    pose_files[i], &pcd_poses[i], &time_stamps[i],  
&pcd_indices[i]);
```

创建NDT地图

创建ndt地图

```
apollo::localization::msf::pyramid_map::NdtMap  
apollo::localization::msf::pyramid_map::NdtMapNodePool
```

平面提取

```
apollo::localization::msf::FeatureXYPlane
```

下面主要介绍一下local_map中的base_map

base_map

base_map

BaseMap中包括map_node_pool_，也就是说一些map_node_pool_缓存了一些map_node。因为加载map_node需要一些时间，因此采用了提前加载并且缓存的方式。map_node_cache_lvl1_和map_node_cache_lvl2_都是一个LRU的cache。

```
BaseMap::BaseMap(BaseMapConfig* map_config)  
: map_config_(map_config),
```

```

    map_node_cache_lvl1_(nullptr),
    map_node_cache_lvl2_(nullptr),
    map_node_pool_(nullptr) {}

```

先从`map_node_cache_lvl1_`中获取`map node`, 如果没有然后从`map_node_cache_lvl2_`中获取, 并且存放到`map_node_cache_lvl1_`, 如果都没有, 则从磁盘中读取, 磁盘中读取直接放入`map_node_cache_lvl2_`, 然后放入`map_node_cache_lvl1_`并且返回。

```
BaseMapNode* BaseMap::GetMapNodeSafe(const MapNodeIndex& index)
```

提前加载地图节点

```

void BaseMap::PreloadMapNodes(std::set<MapNodeIndex>*& map_ids) {
    DCHECK_LE(static_cast<int>(map_ids->size()),
              map_node_cache_lvl2_->Capacity());
    // check in cacheL2
    typename std::set<MapNodeIndex>::iterator itr = map_ids-
>begin();
    while (itr != map_ids->end()) {
        boost::unique_lock<boost::recursive_mutex>
lock(map_load_mutex_);
        bool is_exist = map_node_cache_lvl2_->IsExist(*itr);
        lock.unlock();
        if (is_exist) {
            itr = map_ids->erase(itr);
        } else {
            ++itr;
        }
    }

    // check whether in already preloading index set
    itr = map_ids->begin();
    auto preloading_itr = map_preloading_task_index_.end();
    while (itr != map_ids->end()) {
        boost::unique_lock<boost::recursive_mutex>
lock(map_load_mutex_);
        preloading_itr = map_preloading_task_index_.find(*itr);
        lock.unlock();
        if (preloading_itr !=

```

```

        map_preloading_task_index_.end()) { // already
    preloading
        itr = map_ids->erase(itr);
    } else {
        ++itr;
    }
}

// load from disk sync
std::vector<std::future<void>> preload_futures;
itr = map_ids->begin();
AINFO << "Preload map node size: " << map_ids->size();
while (itr != map_ids->end()) {
    AINFO << "Preload map node: " << *itr << std::endl;
    boost::unique_lock<boost::recursive_mutex>
lock3(map_load_mutex_);
    map_preloading_task_index_.insert(*itr);
    lock3.unlock();
    preload_futures.emplace_back(
        cyber::Async(&BaseMap::LoadMapNodeThreadSafety, this,
*itr, false));
    ++itr;
}
return;
}

```

加载地图节点

```

void BaseMap::LoadMapNodes(std::set<MapNodeIndex>* map_ids) {

    // 先在cacheL1中查找，如果找到则从map_ids删除找到的节点，最终剩下没有
    找到的节点
    typename std::set<MapNodeIndex>::iterator itr = map_ids-
>begin();
    while (itr != map_ids->end()) {
        if (map_node_cache_lvl1_->IsExist(*itr)) {
            // std::cout << "LoadMapNodes find in L1 cache" <<
std::endl;
            boost::unique_lock<boost::recursive_mutex>
lock(map_load_mutex_);
            map_node_cache_lvl2_->IsExist(*itr); // fresh lru list
            lock.unlock();
            itr = map_ids->erase(itr);
        }
    }
}

```

```

    } else {
        ++itr;
    }
}

// 接着在cacheL2中查找，如果找到则从map_ids删除找到的节点，最终剩下没有找到的节点
itr = map_ids->begin();
BaseMapNode* node = nullptr;
boost::unique_lock<boost::recursive_mutex>
lock(map_load_mutex_);
while (itr != map_ids->end()) {
    if (map_node_cache_lvl2_->Get(*itr, &node)) {
        // std::cout << "LoadMapNodes find in L2 cache" <<
std::endl;
        node->SetIsReserved(true);
        map_node_cache_lvl1_->Put(*itr, node);
        itr = map_ids->erase(itr);
    } else {
        ++itr;
    }
}
lock.unlock();

// 并发从硬盘中加载map_id的节点
std::vector<std::future<void>> load_futures;
itr = map_ids->begin();
while (itr != map_ids->end()) {
    AERROR << "Preload map node failed!";
    load_futures.emplace_back(
        cyber::Async(&BaseMap::LoadMapNodeThreadSafety, this,
*itr, true));
    ++itr;
}

for (auto& future : load_futures) {
    if (future.valid()) {
        future.get();
    }
}
// 然后查看cacheL2是否有对应的map_id的节点，上一步从硬盘中读取的map_node会先放入le_cache。
itr = map_ids->begin();

```

```

node = nullptr;
boost::unique_lock<boost::recursive_mutex>
lock2(map_load_mutex_);
while (itr != map_ids->end()) {
    if (map_node_cache_lvl2_->Get(*itr, &node)) {
        AINFO << "LoadMapNodes: preload missed, load this node
in main thread.\n"
        << *itr;
        node->SetIsReserved(true);
        map_node_cache_lvl1_->Put(*itr, node);
        itr = map_ids->erase(itr);
    } else {
        ++itr;
    }
}
lock2.unlock();

// 检查是否所有节点都已经加载, 如果不是则报错
CHECK(map_ids->empty());
return;
}

```

根据index从硬盘中加载地图, map_node_pool_和map_node_cache_lvl2_ 的关系是什么?

```

void BaseMap::LoadMapNodeThreadSafety(MapNodeIndex index,
bool is_reserved) {
    BaseMapNode* map_node = nullptr;
    while (map_node == nullptr) {
        map_node = map_node_pool_->AllocMapNode();
        if (map_node == nullptr) {
            boost::unique_lock<boost::recursive_mutex>
lock(map_load_mutex_);
            BaseMapNode* node_remove = map_node_cache_lvl2_-
>ClearOne();
            if (node_remove) {
                map_node_pool_->FreeMapNode(node_remove);
            }
        }
    }
    // 初始化map_node, 并且加载
    map_node->Init(map_config_, index, false);
}

```

```

if (!map_node->Load()) {
    AERROR << "Created map node: " << index;
} else {
    AERROR << " Loaded map node: " << index;
}
map_node->SetIsReserved(is_reserved);

// 添加到map_node_cache_lvl2_
boost::unique_lock<boost::recursive_mutex>
lock(map_load_mutex_);
BaseMapNode* node_remove = map_node_cache_lvl2_->Put(index,
map_node);
// if the node already added into cacheL2, erase it from
preloading set
auto itr = map_preloding_task_index_.find(index);
if (itr != map_preloding_task_index_.end()) {
    map_preloding_task_index_.erase(itr);
}
// 在map_node_pool_中释放node_remove
if (node_remove) {
    map_node_pool_->FreeMapNode(node_remove);
}
return;
}

```

提前加载地图区域

```

void BaseMap::PreloadMapArea(const Eigen::Vector3d& location,
                             const Eigen::Vector3d&
trans_diff,
                             unsigned int resolution_id,
unsigned int zone_id) {
    // 四象限中的位置
    int x_direction = trans_diff[0] > 0 ? 1 : -1;
    int y_direction = trans_diff[1] > 0 ? 1 : -1;

    // 获取地图的精度
    std::set<MapNodeIndex> map_ids;
    float map_pixel_resolution =
        this->map_config_->map_resolutions_[resolution_id];

    /// 车的位置平移一个map_node的距离
    // 根据top_left, 查找map_id, 其中resolution_id代表精度数组的编号

```

```

Eigen::Vector3d pt_top_left;
pt_top_left[0] =
    location[0] - (static_cast<float>(this->map_config_-
>map_node_size_x_) *
                    map_pixel_resolution / 2.0f);
pt_top_left[1] =
    location[1] - (static_cast<float>(this->map_config_-
>map_node_size_y_) *
                    map_pixel_resolution / 2.0f);
pt_top_left[2] = 0;

MapNodeIndex map_id = MapNodeIndex::GetMapNodeIndex(
    *(this->map_config_), pt_top_left, resolution_id,
zone_id);
map_ids.insert(map_id);

// 同时根据以下几个坐标，查找map_id
/// top center
/// top right
/// middle left
/// middle center
/// middle right
/// bottom left
/// bottom center
/// bottom right

// 根据x_direction * 1.5, y_direction * 1.5和(x_direction *
1.5, y_direction * 1.5)
// 查找map_id
for (int i = -1; i < 2; ++i) {
    Eigen::Vector3d pt;
    pt[0] = location[0] + x_direction * 1.5 *
        this->map_config_-
>map_node_size_x_ *
                    map_pixel_resolution;
    pt[1] = location[1] + static_cast<double>(i) *
        this->map_config_-
>map_node_size_y_ *
                    map_pixel_resolution;
    pt[2] = 0;
    map_id = MapNodeIndex::GetMapNodeIndex(*(this-
>map_config_), pt,
resolution id,
```

```

    zone_id);
    map_ids.insert(map_id);
}
// y_direction * 1.5
// (x_direction * 1.5, y_direction * 1.5)

this->PreloadMapNodes(&map_ids);
return;
}

```

根据seed_pt3d的位置，加载地图区域

```

bool BaseMap::LoadMapArea(const Eigen::Vector3d& seed_pt3d,
                           unsigned int resolution_id,
                           unsigned int zone_id,
                           int filter_size_x, int
                           filter_size_y) {
    CHECK_NOTNULL(map_node_pool_);
    std::set<MapNodeIndex> map_ids;
    float map_pixel_resolution =
        this->map_config_->map_resolutions_[resolution_id];
    /// top left
    Eigen::Vector3d pt_top_left;
    pt_top_left[0] = seed_pt3d[0] -
        (static_cast<float>(this->map_config_-
>map_node_size_x_) *
         map_pixel_resolution / 2.0f) -
        static_cast<float>(filter_size_x / 2) *
    map_pixel_resolution;
    pt_top_left[1] = seed_pt3d[1] -
        (static_cast<float>(this->map_config_-
>map_node_size_y_) *
         map_pixel_resolution / 2.0f) -
        static_cast<float>(filter_size_y / 2) *
    map_pixel_resolution;
    pt_top_left[2] = 0;
    MapNodeIndex map_id = MapNodeIndex::GetMapNodeIndex(
        *(this->map_config_), pt_top_left, resolution_id,
        zone_id);
    map_ids.insert(map_id);

    /// top center
    /// top right
}

```

```

    /// middle left
    /// middle center
    /// middle right
    /// bottom left
    /// bottom center
    /// bottom right

    this->LoadMapNodes(&map_ids);
    return true;
}

```

NdtMapNode

NdtMapNode是地图的单元格，主要是用来确定无人车的位置，这里的x,y是相对node的位置，然后根据(x,y)的坐标转换为UTM的绝对位置。

```

Eigen::Vector3d NdtMapNode::GetCoordinate3D(unsigned int x,
                                             unsigned int y,
                                             int
                                             altitude_index) const {
    const Eigen::Vector2d& left_top_corner =
    GetLeftTopCorner();
    Eigen::Vector2d coord_2d;
    coord_2d[0] =
        left_top_corner[0] + (static_cast<double>(x) * 
    GetMapResolution());
    coord_2d[1] =
        left_top_corner[1] + (static_cast<double>(y) * 
    GetMapResolution());

    double altitude =
        NdtMapCells::CalAltitude(GetMapResolutionZ(),
    altitude_index);
    Eigen::Vector3d coord_3d;
    coord_3d[0] = coord_2d[0];
    coord_3d[1] = coord_2d[1];
    coord_3d[2] = altitude;

    return coord_3d;
}

```

同时还涉及2个Node节点如何做合并，这里主要是调用了NdtMapMatrix的合并方法。

```
void NdtMapNode::Reduce(NdtMapNode* map_node, const
NdtMapNode& map_node_new) {
    assert(map_node->index_.m_ == map_node_new.index_.m_);
    assert(map_node->index_.n_ == map_node_new.index_.n_);
    assert(map_node->index_.resolution_id_ ==
map_node_new.index_.resolution_id_);
    assert(map_node->index_.zone_id_ ==
map_node_new.index_.zone_id_);
    NdtMapMatrix::Reduce(
        static_cast<NdtMapMatrix*>(map_node->map_matrix_),
        static_cast<const NdtMapMatrix&>
(*map_node_new.map_matrix_));
}
```

NdtMapMatrix包含(m,n)的NdtMapCells矩阵，这里遍历合并2个矩阵。我们主要看下如何合并的逻辑

```
void NdtMapMatrix::Reduce(NdtMapMatrix* cells, const
NdtMapMatrix& cells_new) {
    for (unsigned int y = 0; y < cells->GetRows(); ++y) {
        for (unsigned int x = 0; x < cells->GetCols(); ++x) {
            NdtMapCells& cell = cells->GetMapCell(y, x);
            const NdtMapCells& cell_new = cells_new.GetMapCell(y,
x);
            NdtMapCells::Reduce(&cell, cell_new);
        }
    }
}
```

最后合并2个NdtMapCells。

```
void NdtMapCells::Reduce(NdtMapCells* cell, const
NdtMapCells& cell_new) {
    // Reduce cells
    for (auto it = cell_new.cells_.begin(); it != cell_new.cells_.end(); ++it) {
        int altitude_index = it->first;
        auto got = cell->cells_.find(altitude_index);
        if (got != cell->cells_.end()) {
```

```

        cell->cells_[altitude_index].MergeCell(it->second);
    } else {
        cell->cells_[altitude_index] = NdtMapSingleCell(it-
>second);
    }
}

if (cell_new.max_altitude_index_ > cell-
>max_altitude_index_) {
    cell->max_altitude_index_ = cell_new.max_altitude_index_;
}

if (cell_new.min_altitude_index_ < cell-
>min_altitude_index_) {
    cell->min_altitude_index_ = cell_new.min_altitude_index_;
}

for (auto it_new = cell_new.road_cell_indices_.begin();
     it_new != cell_new.road_cell_indices_.end(); ++it_new)
{
    auto got_it = std::find(cell->road_cell_indices_.begin(),
                           cell->road_cell_indices_.end(),
                           *it_new);
    if (got_it != cell->road_cell_indices_.end()) {
        *got_it += *it_new;
    } else {
        cell->road_cell_indices_.push_back(*it_new);
    }
}
}

```

maptool

异步保存地图

```
map_node_pool_->FreeMapNode(node_remove)
```

介绍完了制作地图的过程，下面开始介绍NDT定位的过程。

NDT定位模块

`NDTLocalizationComponent`进行初始化，并且根据输入的`gps`消息结合点云信息输出定位和坐标`translation`关系。

NDTLocalization

定位的具体功能是在`NDTLocalization`中实现的。NDT的匹配在`lidar_locator_`中，而融合的部分在

```
if (!lidar_locator_.IsInitialized()) {
    lidar_locator_.Init(odometry_pose, resolution_id_,
zone_id_);
    return;
}
lidar_locator_.Update(frame_idx++, odometry_pose,
lidar_frame);
lidar_pose_ = lidar_locator_.GetPose();
pose_buffer_.UpdateLidarPose(time_stamp, lidar_pose_,
odometry_pose);
ComposeLidarResult(time_stamp, lidar_pose_,
&lidar_localization_result_);
ndt_score_ = lidar_locator_.GetFitnessScore();
```

LocalizationPoseBuffer

`LocalizationPoseBuffer`是一个buffer，大小默认为20，会存储历史的lidar姿态和IMU姿态，然后根据当前的IMU姿态推断最新的lidar姿态。

LidarLocatorNdt

NDT定位的主要实现在`LidarLocatorNdt`模块中。根据输入的`pose`然后再进行`ndt`匹配，这样在IMU有偏差的情况下，可以有一定程度的校准，如果长时间GPS和IMU的定位不准确，那么当前的方案可能就不太可行。另外这里没有尝试纯NDT定位的方案，如果需要测试纯NDT定位的效果，也需要修改部分代码才能实现。

主要的处理函数在`LidarLocatorNdt::Update`中实现。

```

int LidarLocatorNdt::Update(unsigned int frame_idx, const
Eigen::Affine3d& pose,
                           const LidarFrame& lidar_frame) {
    // Increasesement from INSPVA
    Eigen::Vector3d trans_diff =
        pose.translation() - pre_input_location_.translation();
    Eigen::Vector3d trans_pre_local =
        pre_estimate_location_.translation() + trans_diff;
    Eigen::Quaterniond quatd(pose.linear());
    Eigen::Translation3d transd(trans_pre_local);
    Eigen::Affine3d center_pose = transd * quatd;

    Eigen::Quaterniond pose_qbn(pose.linear());
    AINFO << "original pose: " << std::setprecision(15) <<
pose.translation()[0]
        << ", " << pose.translation()[1] << ", " <<
pose.translation()[2]
        << ", " << pose_qbn.x() << ", " << pose_qbn.y() << ", "
" << pose_qbn.z()
        << ", " << pose_qbn.w();

    // Get lidar pose Twv = Twb * Tbv
    Eigen::Affine3d transform = center_pose *
velodyne_extrinsic_;
    predict_location_ = center_pose;

    // Pre-load the map nodes
#ifdef USE_PRELOAD_MAP_NODE
    bool map_is_ready =
        map_.LoadMapArea(center_pose.translation(),
resolution_id_, zone_id_,
                           filter_x_, filter_y_);
    map_.PreloadMapArea(center_pose.translation(), trans_diff,
resolution_id_,
                           zone_id_);
#endif

    // Online pointcloud are projected into a ndt map node.
(filtered)
    double lt_x = pose.translation()[0];
    double lt_y = pose.translation()[1];
    double map_resolution =
map_.GetConfig().map_resolutions_[resolution_id_];

```

```

    lt_x -= (map_.GetConfig().map_node_size_x_ * map_resolution
    / 2.0);
    lt_y -= (map_.GetConfig().map_node_size_y_ * map_resolution
    / 2.0);

    // Start Ndt method
    // Convert online points to pcl pointcloud
    apollo::common::time::Timer online_filtered_timer;
    online_filtered_timer.Start();
    pcl::PointCloud<pcl::PointXYZ>::Ptr online_points(
        new pcl::PointCloud<pcl::PointXYZ>());
    for (unsigned int i = 0; i < lidar_frame.pt_xs.size(); ++i)
    {
        pcl::PointXYZ p(lidar_frame.pt_xs[i],
        lidar_frame.pt_ys[i],
                    lidar_frame.pt_zs[i]);
        online_points->push_back(p);
    }

    // Filter online points
    AINFO << "Online point cloud leaf size: " <<
proj_reslution_;
    pcl::PointCloud<pcl::PointXYZ>::Ptr online_points_filtered(
        new pcl::PointCloud<pcl::PointXYZ>());
    pcl::VoxelGrid<pcl::PointXYZ> sor;
    sor.setInputCloud(online_points);
    sor.setLeafSize(proj_reslution_, proj_reslution_,
proj_reslution_);
    sor.filter(*online_points_filtered);
    AINFO << "Online Pointcloud size: " << online_points-
>size() << "/"
    << online_points_filtered->size();
    online_filtered_timer.End("online point calc end.");

    // Obtain map pointcloud
    apollo::common::time::Timer map_timer;
    map_timer.Start();
    Eigen::Vector2d left_top_coord2d(lt_x, lt_y);
    ComposeMapCells(left_top_coord2d, zone_id_, resolution_id_,
map_.GetConfig().map_resolutions_[resolution_id_],
                    transform.inverse());

```

```

// Convert map pointcloud to local coordinate
pcl::PointCloud::Ptr pcl_map_point_cloud(
    new pcl::PointCloud());
for (unsigned int i = 0; i < cell_map_.size(); ++i) {
    Leaf& le = cell_map_[i];
    float mean_0 = static_cast<float>(le.mean_(0));
    float mean_1 = static_cast<float>(le.mean_(1));
    float mean_2 = static_cast<float>(le.mean_(2));
    pcl_map_point_cloud->push_back(pcl::PointXYZ(mean_0,
mean_1, mean_2));
}
map_timer.End("Map create end.");
// Set left top corner for reg
reg_.SetLeftTopCorner(map_left_top_corner_);
// Ndt calculation
reg_.SetInputTarget(cell_map_, pcl_map_point_cloud);
reg_.SetInputSource(online_points_filtered);

apollo::common::time::Timer ndt_timer;
ndt_timer.Start();
Eigen::Matrix3d inv_R = transform.inverse().linear();
Eigen::Matrix4d init_matrix = Eigen::Matrix4d::Identity();
init_matrix.block<3, 3>(0, 0) = inv_R.inverse();

pcl::PointCloud::Ptr output_cloud(
    new pcl::PointCloud());
reg_.Align(output_cloud, init_matrix.cast<float>());
ndt_timer.End("Ndt Align End.");

fitness_score_ = reg_.GetFitnessScore();
bool has_converged = reg_.HasConverged();
int iteration = reg_.GetFinalNumIteration();
Eigen::Matrix4d ndt_pose =
reg_.GetFinalTransformation().cast<double>();
AINFO << "Ndt summary:";
AINFO << "Fitness Score: " << fitness_score_;
AINFO << "Has_converged: " << has_converged;
AINFO << "Iteration: %d: " << iteration;
AINFO << "Relative Ndt pose: " << ndt_pose(0, 3) << ", " <<
ndt_pose(1, 3)
    << ", " << ndt_pose(2, 3);

// Twv

```

```

Eigen::Affine3d lidar_location =
Eigen::Affine3d::Identity();
lidar_location = transform.matrix() * init_matrix.inverse()
* ndt_pose;

// Save results
location_ = lidar_location * velodyne_extrinsic_.inverse();
pre_input_location_ = pose;
pre_estimate_location_ = location_;
pre_imu_height_ = location_.translation()(2);

if (map_is_ready) {
    return 0;
} else {
    return -1;
}
return 0;
}

```

ndt voxel grid covariance

叶子节点中点的个数，以及平局值和协方差。

```

struct Leaf {
    Leaf()
        : nr_points_(0),
          mean_(Eigen::Vector3d::Zero()),
          icov_(Eigen::Matrix3d::Zero()) {}

    /**@brief Get the number of points contained by this voxel.
 */
    int GetPointCount() const { return nr_points_; }

    /**@brief Get the voxel centroid. */
    Eigen::Vector3d GetMean() const { return mean_; }

    /**@brief Get the inverse of the voxel covariance. */
    Eigen::Matrix3d GetInverseCov() const { return icov_; }

    /**@brief Number of points contained by voxel. */
    int nr_points_;
    /**@brief 3D voxel centroid. */

```

```

Eigen::Vector3d mean_;
/**@brief Inverse of voxel covariance matrix. */
Eigen::Matrix3d icov_;
};


```

VoxelGridCovariance

Voxel的协方差计算，VoxelGridCovariance包含了一系列的叶子节点(Leaf)。

初始化voxel结构体，一是SetMap，二是构建kdtree_

```

/**@brief Initializes voxel structure. */
inline void filter(const std::vector<Leaf> &cell_leaf,
                   bool searchable = true) {
    voxel_centroids_ = PointCloudPtr(new PointCloud);
    SetMap(cell_leaf, voxel_centroids_);
    if (voxel_centroids_->size() > 0) {
        kdtree_.setInputCloud(voxel_centroids_);
    }
}
}


```

把map_leaves中的值赋值给leaves_，并且把index保存到voxel_centroids_leaf_indices_中

```

template <typename PointT>
void VoxelGridCovariance<PointT>::SetMap(const
std::vector<Leaf>& map_leaves,
                                         PointCloudPtr
output) {
    voxel_centroids_leaf_indices_.clear();

    // input_输入点云不为空

    // Copy the header + allocate enough space for points
    output->height = 1;
    output->is_dense = true;
    output->points.clear();

    // 获取最大和最小的点云坐标
    Eigen::Vector4f min_p, max_p;
}


```

```

pcl::getMinMax3D<PointT>(*input_, min_p, max_p);

Eigen::Vector4f left_top = Eigen::Vector4f::Zero();
left_top.block<3, 1>(0, 0) =
map_left_top_corner_.cast<float>();
min_p -= left_top;
max_p -= left_top;

// Compute the minimum and maximum bounding box values
min_b_[0] = static_cast<int>(min_p[0] *
inverse_leaf_size_[0]);
max_b_[0] = static_cast<int>(max_p[0] *
inverse_leaf_size_[0]);
min_b_[1] = static_cast<int>(min_p[1] *
inverse_leaf_size_[1]);
max_b_[1] = static_cast<int>(max_p[1] *
inverse_leaf_size_[1]);
min_b_[2] = static_cast<int>(min_p[2] *
inverse_leaf_size_[2]);
max_b_[2] = static_cast<int>(max_p[2] *
inverse_leaf_size_[2]);

// Compute the number of divisions needed along all axis
div_b_ = max_b_ - min_b_ + Eigen::Vector4i::Ones();
div_b_[3] = 0;

// Set up the division multiplier
divb_mul_ = Eigen::Vector4i(1, div_b_[0], div_b_[0] *
div_b_[1], 0);

// Clear the leaves
leaves_.clear();

output->points.reserve(map_leaves.size());
voxel_centroids_leaf_indices_.reserve(leaves_.size());

for (unsigned int i = 0; i < map_leaves.size(); ++i) {
    const Leaf& cell_leaf = map_leaves[i];
    Eigen::Vector3d local_mean = cell_leaf.mean_ -
map_left_top_corner_;
    int ijk0 =
        static_cast<int>(local_mean(0) *
inverse_leaf_size_[0]) - min_b_[0];

```

```

int ijk1 =
    static_cast<int>(local_mean(1) *
inverse_leaf_size_[1]) - min_b_[1];
int ijk2 =
    static_cast<int>(local_mean(2) *
inverse_leaf_size_[2]) - min_b_[2];

// Compute the centroid leaf index
int idx = ijk0 * divb_mul_[0] + ijk1 * divb_mul_[1] +
ijk2 * divb_mul_[2];

Leaf& leaf = leaves_[idx];
leaf = cell_leaf;

if (cell_leaf.nr_points_ >= min_points_per_voxel_) {
    output->push_back(PointT());
    output->points.back().x = static_cast<float>
(leaf.mean_[0]);
    output->points.back().y = static_cast<float>
(leaf.mean_[1]);
    output->points.back().z = static_cast<float>
(leaf.mean_[2]);
    voxel_centroids_leaf_indices_.push_back(idx);
}
}
output->width = static_cast<uint32_t>(output-
>points.size());
}

```

这里为什么可以根据index找到k_leaves?

```

template <typename PointT>
int VoxelGridCovariance<PointT>::RadiusSearch(
    const PointT& point, double radius,
    std::vector<LeafConstPtr>*& k_leaves,
    std::vector<float>*& k_sqr_distances, unsigned int max_nn)
{
    k_leaves->clear();

    // Find neighbors within radius in the occupied voxel
    centroid cloud
    std::vector<int> k_indices;
    int k =

```

```

    kdrtree_.radiusSearch(point, radius, k_indices,
*k_sqr_distances, max_nn);

    // Find leaves corresponding to neighbors
    k_leaves->reserve(k);
    for (std::vector<int>::iterator iter = k_indices.begin();
        iter != k_indices.end(); iter++) {
        k_leaves-
>push_back(&leaves_[voxel_centroids_leaf_indices_[*iter]]);
    }
    return k;
}

```

通过Leaf的质心随机生成100个点进行点云的显示。

```

template <typename PointT>
void VoxelGridCovariance<PointT>::GetDisplayCloud(
    pcl::PointCloud<pcl::PointXYZ>* cell_cloud)

```

NormalDistributionsTransform

设置目标点云

```

    inline void SetInputTarget(const std::vector<Leaf>
&cell_leaf,
                           const PointCloudTargetConstPtr
&cloud) {
    if (cell_leaf.empty()) {
        AWARN << "Input leaf is empty.";
        return;
    }
    if (cloud->points.empty()) {
        AWARN << "Input target is empty.";
        return;
    }

    // 设置目标点云，其中target_cells_为VoxelGridCovariance
    target_ = cloud;
    target_cells_.SetVoxelGridResolution(resolution_,
resolution_, resolution_);
    target_cells_.SetInputCloud(cloud);
}

```

```
    target_cells_.filter(cell_leaf, true);  
}
```

resolution_ 代表体素网格分辨率

step_size_ 牛顿线性查找最大步长

trans_probability_ 注册概率

nr_iterations_ 迭代次数

final_transformation_ 最后的转移矩阵

```
template <typename PointSource, typename PointTarget>  
void NormalDistributionsTransform<PointSource, PointTarget>::  
    ComputeTransformation(PointCloudSourcePtr output,  
                          const Eigen::Matrix4f &guess) {  
    apollo::common::time::Timer timer;  
    timer.Start();  
  
    nr_iterations_ = 0;  
    converged_ = false;  
    double gauss_c1, gauss_c2, gauss_d3;  
  
    // Initializes the guassian fitting parameters (eq. 6.8)  
    [Magnusson 2009]  
    gauss_c1 = 10 * (1 - outlier_ratio_);  
    gauss_c2 = outlier_ratio_ / pow(resolution_, 3);  
    gauss_d3 = -log(gauss_c2);  
    gauss_d1_ = -log(gauss_c1 + gauss_c2) - gauss_d3;  
    gauss_d2_ =  
        -2 * log((-log(gauss_c1 * exp(-0.5) + gauss_c2) -  
                  gauss_d3) / gauss_d1_);  
  
    if (guess != Eigen::Matrix4f::Identity()) {  
        // Initialise final transformation to the guessed one  
        final_transformation_ = guess;  
        // Apply guessed transformation prior to search for  
        neighbours  
        transformPointCloud(*output, *output, guess);  
    }
```

```

// Initialize Point Gradient and Hessian
point_gradient_.setZero();
point_gradient_.block<3, 3>(0, 0).setIdentity();
point_hessian_.setZero();

Eigen::Transform<float, 3, Eigen::Affine, Eigen::ColMajor>
eig_transformation;
eig_transformation.matrix() = final_transformation_;

// Convert initial guess matrix to 6 element transformation
vector
Eigen::Matrix<double, 6, 1> p, delta_p, score_gradient;
Eigen::Vector3f init_translation =
eig_transformation.translation();
Eigen::Vector3f init_rotation =
    eig_transformation.rotation().eulerAngles(0, 1, 2);
p << init_translation(0), init_translation(1),
init_translation(2),
    init_rotation(0), init_rotation(1), init_rotation(2);

Eigen::Matrix<double, 6, 6> hessian;
double score = 0;
double delta_p_norm;

// Calculate derivates of initial transform vector,
// subsequent derivative
// calculations are done in the step length determination.
score = ComputeDerivatives(&score_gradient, &hessian,
output, &p);
timer.End("Ndt ComputeDerivatives");

apollo::common::time::Timer loop_timer;
loop_timer.Start();
while (!converged_) {
    // Store previous transformation
    previous_transformation_ = transformation_;

    // Solve for decent direction using newton method, line
23 in Algorithm
    // 2 [Magnusson 2009]
    Eigen::JacobiSVD<Eigen::Matrix<double, 6, 6>> sv(
        hessian, Eigen::ComputeFullU | Eigen::ComputeFullV);
    // Negative for maximization as opposed to minimization

```

```

delta_p = sv.solve(-score_gradient);

// Calculate step length with guaranteed sufficient
decrease [More,
// Thuente 1994]
delta_p_norm = delta_p.norm();

if (delta_p_norm == 0 || delta_p_norm != delta_p_norm) {
    trans_probability_ = score / static_cast<double>
(input_->points.size());
    converged_ = delta_p_norm == delta_p_norm;
    return;
}

delta_p.normalize();
delta_p_norm = ComputeStepLengthMt(p, &delta_p,
delta_p_norm, step_size_,
transformation_epsilon_ / 2, &score,
&score_gradient,
&hessian, output);
delta_p *= delta_p_norm;

transformation_ =
(Eigen::Translation<float, 3>(static_cast<float>
(delta_p(0)),
static_cast<float>
(delta_p(1)),
static_cast<float>
(delta_p(2))) *
Eigen::AngleAxis<float>(static_cast<float>
(delta_p(3)),
Eigen::Vector3f::UnitX()) *
Eigen::AngleAxis<float>(static_cast<float>
(delta_p(4)),
Eigen::Vector3f::UnitY()) *
Eigen::AngleAxis<float>(static_cast<float>
(delta_p(5)),
Eigen::Vector3f::UnitZ()))
.matrix();

p = p + delta_p;

```

```

        if (nr_iterations_ > max_iterations_ ||
            (nr_iterations_ &&
             (std::fabs(delta_p_norm) <
transformation_epsilon_))) {
    converged_ = true;
}

nr_iterations_++;
}
loop_timer.End("Ndt loop.");

// Store transformation probability. The relative
differences within each
// scan registration are accurate but the normalization
constants need to be
// modified for it to be globally accurate
trans_probability_ = score / static_cast<double>(input_-
>points.size());
}

```

其中的公式推导需要接着分析ndt_solver.hpp。

ndt_map

ndt_map为ndt地图的主要说明目录，

BaseMapNodePool

BaseMapNodePool是一个map node对象池，其中is_fixed_size_表示大小固定，不会新增加大小。free_list_中存放还没有使用的节点，busy_nodes_中存放正在使用的节点，当需要释放的时候可以用ResetMapNode来进行资源的释放。

其中比较有意思的一点在于，BaseMapNodePool在析构函数中进行资源释放，同时结合ndt_map_node->SetIsChanged(true)来保存node节点到硬盘，从而实现地图的制作过程。

poses_interpolation.cc

对pose进行插值，主要的处理在函数中

```
void PosesInterpolation::DoInterpolation() {
    // 读取pose信息
    std::vector<Eigen::Vector3d> input_stds;
    velodyne::LoadPosesAndStds(input_poses_path_,
&input_poses_, &input_stds,
                                &input_poses_timestamps_);

    // 读取pcd的index和时间戳
    LoadPCDTimestamp();

    // Interpolation
    PoseInterpolationByTime(input_poses_,
input_poses_timestamps_,
                                ref_timestamps_, ref_ids_,
&out_indexes_,
                                &out_timestamps_, &out_poses_);

    // 保存pcd文件
    WritePCDPoses();
}
```

插值的主要逻辑在如下函数中，主要是根据lidar的时间戳，查找位姿，并且做插值，最后保存到文件。

```
in_poses // 定位的pose列表
in_timestamps // 定位的时间戳
ref_timestamps // 点云的时间戳
ref_indexes // 点云的index
out_indexes // 输出的index
out_timestamps // 输出的时间戳
out_poses // 输出的姿态

void PosesInterpolation::PoseInterpolationByTime(
    const std::vector<Eigen::Affine3d> &in_poses,
    const std::vector<double> &in_timestamps,
    const std::vector<double> &ref_timestamps,
    const std::vector<unsigned int> &ref_indexes,
    std::vector<unsigned int> *out_indexes,
    std::vector<double> *out_timestamps,
    std::vector<Eigen::Affine3d> *out_poses) {
```

```

out_indexes->clear();
out_timestamps->clear();
out_poses->clear();

unsigned int index = 0;
for (size_t i = 0; i < ref_timestamps.size(); i++) {
    double ref_timestamp = ref_timestamps[i];
    unsigned int ref_index = ref_indexes[i];

    while (index < in_timestamps.size() &&
           in_timestamps.at(index) < ref_timestamp) {
        ++index;
    }

    if (index < in_timestamps.size()) {
        if (index >= 1) {
            double cur_timestamp = in_timestamps[index];
            double pre_timestamp = in_timestamps[index - 1];
            assert(cur_timestamp != pre_timestamp);

            double t =
                (cur_timestamp - ref_timestamp) / (cur_timestamp
- pre_timestamp);
            assert(t >= 0.0);
            assert(t <= 1.0);

            Eigen::Affine3d pre_pose = in_poses[index - 1];
            Eigen::Affine3d cur_pose = in_poses[index];
            Eigen::Quaterniond pre_quatd(pre_pose.linear());
            Eigen::Translation3d
pre_transd(pre_pose.translation());
            Eigen::Quaterniond cur_quatd(cur_pose.linear());
            Eigen::Translation3d
cur_transd(cur_pose.translation());

            Eigen::Quaterniond res_quatd = pre_quatd.slerp(1 - t,
cur_quatd);

            Eigen::Translation3d re_transd;
            re_transd.x() = pre_transd.x() * t + cur_transd.x() *
(1 - t);
            re_transd.y() = pre_transd.y() * t + cur_transd.y() *
(1 - t);
        }
    }
}

```

```
    re_transd.z() = pre_transd.z() * t + cur_transd.z() *  
    (1 - t);  
  
    out_poses->push_back(re_transd * res_quatd);  
    out_indexes->push_back(ref_index);  
    out_timestamps->push_back(ref_timestamp);  
}  
}  
else {  
    AWARN << "[WARN] No more poses. Exit now.";  
    break;  
}  
ADEBUG << "Frame_id: " << i;  
}  
}
```


DetectionComponent

DetectionComponent主要的目的是用来做物体识别。接收点云信息，最后输出感知到的结果。输入TOPIC: drivers::PointCloud 输出TOPIC: LidarFrameMessage

实际上在BUILD文件中，DetectionComponent等几个模块编译为一个模块，最后的可执行文件为”libperception_component_lidar”。也就是说 DetectionComponent的配置文件在DAG中查找 libperception_component_lidar中对应的”DetectionComponent”的配置。
疑问：

目前并没有在dag中找到对应的配置文件，看起来这个模块是给第三方雷达感知算法提供的接口？？？

初始化(Init)

```
bool DetectionComponent::Init() {
    LidarDetectionComponentConfig comp_config;
    // 1. 读取配置文件
    if (!GetProtoConfig(&comp_config)) {
        return false;
    }
    ADEBUG << "Lidar Component Configs: " <<
    comp_config.DebugString();
    output_channel_name_ = comp_config.output_channel_name();
    sensor_name_ = comp_config.sensor_name();
    lidar2novatel_tf2_child_frame_id_ =
        comp_config.lidar2novatel_tf2_child_frame_id();
    lidar_query_tf_offset_ =
        static_cast<float>
    (comp_config.lidar_query_tf_offset());
    enable_hdmap_ = comp_config.enable_hdmap();
    // 2. 注册发布消息
    writer_ = node_->CreateWriter<LidarFrameMessage>
    (output_channel_name_);
    // 3. 初始化算法插件
```

```

    if (!InitAlgorithmPlugin()) {
        AERROR << "Failed to init detection component algorithm
plugin.";
        return false;
    }
    return true;
}

```

InitAlgorithmPlugin

```

bool DetectionComponent::InitAlgorithmPlugin() {
    ACHECK(common::SensorManager::Instance()-
>GetSensorInfo(sensor_name_,
&sensor_info_));
    // 1. 设置雷达障碍物识别器
    detector_.reset(new lidar::LidarObstacleDetection);

    lidar::LidarObstacleDetectionInitOptions init_options;
    init_options.sensor_name = sensor_name_;
    init_options.enable_hdmap_input =
        FLAGS_obs_enable_hdmap_input && enable_hdmap_;
    // 2. 初始化识别器，调用LidarObstacleDetection的Init方法
    if (!detector_->Init(init_options)) {
        AINFO << "sensor_name_ "
            << "Failed to init detection.";
        return false;
    }
    // 3. 初始化坐标转换关系
    lidar2world_trans_.Init(lidar2novatel_tf2_child_frame_id_);
    return true;
}

```

执行(Proc)

执行主要在Proc，而Proc主要是调用内部的实现”InternalProc”。

```

bool DetectionComponent::InternalProc(
    const std::shared_ptr<const drivers::PointCloud>&
in_message,

```

```

    const std::shared_ptr<LidarFrameMessage>& out_message) {
// 1. 为什么要加锁，难道有多个雷达的情况？？
PERCEPTION_PERF_FUNCTION_WITH_INDICATOR(sensor_name_);
{
    std::unique_lock<std::mutex> lock(s_mutex_);
    s_seq_num_++;
}

// 2. 初始化信息帧
out_message->timestamp_ = timestamp;
out_message->lidar_timestamp_ = in_message-
>header().lidar_timestamp();
out_message->seq_num_ = s_seq_num_;
out_message->process_stage_ =
ProcessStage::LIDAR_DETECTION;
out_message->error_code_ = apollo::common::ErrorCode::OK;

auto& frame = out_message->lidar_frame_;
frame = lidar::LidarFramePool::Instance().Get();
frame->cloud = base::PointCloudPool::Instance().Get();
frame->timestamp = timestamp;
frame->sensor_info = sensor_info_;

PERCEPTION_PERF_BLOCK_START();
Eigen::Affine3d pose = Eigen::Affine3d::Identity();
const double lidar_query_tf_timestamp =
    timestamp - lidar_query_tf_offset_ * 0.001;
// 3. 获取车的姿态
if
(!lidar2world_trans_.GetSensor2worldTrans(lidar_query_tf_time
stamp,
                                            &pose)) {
    out_message->error_code_ =
apollo::common::ErrorCode::PERCEPTION_ERROR_TF;
    AERROR << "Failed to get pose at time: " <<
lidar_query_tf_timestamp;
    return false;
}
PERCEPTION_PERF_BLOCK_END_WITH_INDICATOR(
    sensor_name_, "detection_1::get_lidar_to_world_pose");

frame->lidar2world_pose = pose;

```

```

lidar::LidarObstacleDetectionOptions detect_opts;
detect_opts.sensor_name = sensor_name_;

lidar2world_trans_.GetExtrinsics(&detect_opts.sensor2novatel_
extrinsics);

// 4. 开始物体识别，调用LidarObstacleDetection的Process方法，下文
会分析具体的实现
lidar::LidarProcessResult ret =
    detector_->Process(detect_opts, in_message,
frame.get());
if (ret.error_code != lidar::LidarErrorCode::Succeed) {
    out_message->error_code_ =
        apollo::common::ErrorCode::PERCEPTION_ERROR_PROCESS;
    return false;
}
PERCEPTION_PERF_BLOCK_END_WITH_INDICATOR(sensor_name_,
"detection_2::detect_obstacle");

return true;
}

```

LidarObstacleDetection

在DetectionComponent中会初始化LidarObstacleDetection，并且调用类的”Process()”方法，那么LidarObstacleDetection中究竟实现了哪些功能？

下面先分析Init方法

初始化Init

```

bool LidarObstacleDetection::Init(
    const LidarObstacleDetectionInitOptions& options) {
// 1. 根据传感器名称获取模型配置
auto& sensor_name = options.sensor_name;
auto config_manager = lib::ConfigManager::Instance();
const lib::ModelConfig* model_config = nullptr;

```

```

ACHECK(config_manager->GetModelConfig(Name(),
&model_config));

// 2. 获取配置文件
const std::string work_root = config_manager->work_root();
std::string config_file;
std::string root_path;
ACHECK(model_config->get_value("root_path", &root_path));
config_file = cyber::common::GetAbsolutePath(work_root,
root_path);
config_file = cyber::common::GetAbsolutePath(config_file,
sensor_name);
config_file = cyber::common::GetAbsolutePath(
config_file, "lidar_obstacle_detection.conf");

// 3. 从配置文件中获取，探测器的名称，是否采用地图管理，过滤目录参数
LidarObstacleDetectionConfig config;
ACHECK(cyber::common::GetProtoFromFile(config_file,
&config));
detector_name_ = config.detector();
use_map_manager_ = config.use_map_manager();
use_object_filter_bank_ = config.use_object_filter_bank();

use_map_manager_ = use_map_manager_ &&
options.enable_hdmap_input;

// 4. 初始化场景管理
SceneManagerInitOptions scene_manager_init_options;

ACHECK(SceneManager::Instance().Init(scene_manager_init_options));

// 5. 点云预处理
PointCloudPreprocessorInitOptions
preprocessor_init_options;
preprocessor_init_options.sensor_name = sensor_name;

ACHECK(cloud_preprocessor_.Init(preprocessor_init_options));

// 6. 是否采用地图管理
if (use_map_manager_) {
    MapManagerInitOptions map_manager_init_options;
    if (!map_manager_.Init(map_manager_init_options)) {

```

```

        AINFO << "Failed to init map manager.";
        use_map_manager_ = false;
    }
}

// 7. 初始化探测器为PointPillarsDetection
detector_.reset(new PointPillarsDetection);
// detector_.reset(
// BaseSegmentationRegisterer::GetInstanceByName(segmentor_name_));
CHECK_NOTNULL(detector_.get());
DetectionInitOptions detection_init_options;
// segmentation_init_options.sensor_name = sensor_name;
ACHECK(detector_->Init(detection_init_options));

return true;
}

```

接着看LidarObstacleDetection如何进行物体识别

执行Process

Process有多态实现，主要的区别为是否传入点云信息message。

```

LidarProcessResult LidarObstacleDetection::Process(
    const LidarObstacleDetectionOptions& options,
    const std::shared_ptr<apollo::drivers::PointCloud const>&
message,
    LidarFrame* frame) {
    const auto& sensor_name = options.sensor_name;

PERCEPTION_PERF_FUNCTION_WITH_INDICATOR(options.sensor_name);

PERCEPTION_PERF_BLOCK_START();
PointCloudPreprocessorOptions preprocessor_options;
preprocessor_options.sensor2novatel_extrinsics =
    options.sensor2novatel_extrinsics;
PERCEPTION_PERF_BLOCK_END_WITH_INDICATOR(sensor_name,
"preprocess");

```

```

// 1. 点云预处理
if (cloud_preprocessor_.Preprocess(preprocessor_options,
message, frame)) {
    // 2. 点云计算
    return ProcessCommon(options, frame);
}
return LidarProcessResult(LidarErrorCode::PointCloudPreprocessorError,
                           "Failed to preprocess point cloud.");
}

```

点云计算

```

LidarProcessResult LidarObstacleDetection::ProcessCommon(
    const LidarObstacleDetectionOptions& options, LidarFrame* frame) {
    const auto& sensor_name = options.sensor_name;

    if (use_map_manager_) {
        MapManagerOptions map_manager_options;
        // 1. 更新地图选项
        if (!map_manager_.Update(map_manager_options, frame)) {
            return LidarProcessResult(LidarErrorCode::MapManagerError,
                                      "Failed to update map structure.");
        }
    }

    // 2. 进行物体识别
    DetectionOptions detection_options;
    if (!detector_->Detect(detection_options, frame)) {
        return LidarProcessResult(LidarErrorCode::DetectionError,
                                  "Failed to detect.");
    }

    return LidarProcessResult(LidarErrorCode::Succeed);
}

```

上述的识别过程实际上在Init中设置为PointPillarsDetection，也就是说点云识别在PointPillarsDetection中实现。

PointPillarsDetection

PointPillarsDetection主要的实现在Init和Proc中。

初始化

在PointPillarsDetection中初始化”PointPillars”类

```
bool PointPillarsDetection::Init(const DetectionInitOptions&
options) {
    point_pillars_ptr_.reset(
        new PointPillars(reproduce_result_mode_,
score_threshold_,
                           nms_overlap_threshold_,
FLAGS_pfe_onnx_file,
                           FLAGS_rpn_onnx_file));
    return true;
}
```

实现(detect)

```
bool PointPillarsDetection::Detect(const DetectionOptions&
options,
                                     LidarFrame* frame) {

    // record input cloud and lidar frame
    original_cloud_ = frame->cloud;
    original_world_cloud_ = frame->world_cloud;
    lidar_frame_ref_ = frame;

    // check output
    frame->segmented_objects.clear();

    Timer timer;
```

```

// 1. 设置GPU id
if (cudaSetDevice(FLAGS_gpu_id) != cudaSuccess) {
    AERROR << "Failed to set device to gpu " << FLAGS_gpu_id;
    return false;
}

// 2. 转化点云为数组
float* points_array = new float[original_cloud_->size() *
4];
PclToArray(original_cloud_, points_array,
kNormalizingFactor);

// 3. 进行推理
std::vector<float> out_detections;
point_pillars_ptr_->doInference(points_array,
original_cloud_->size(),
&out_detections);
inference_time_ = timer.toc(true);

// 4. 输出障碍物的boundbox
GetObjects(&frame->segmented_objects, frame-
>lidar2world_pose,
&out_detections);

AINFO << "PointPillars: inference: " << inference_time_ <<
"\t"
    << "collect: " << collect_time_;
return true;
}

```

输出障碍物

获取目标做分类，点云模型目前只输出了车的分类，因此无法做其他分类

```

void PointPillarsDetection::GetObjects(
    std::vector<std::shared_ptr<Object>>* objects, const
Eigen::Affine3d& pose,
    std::vector<float>* detections) {
Timer timer;
int num_objects = detections->size() /

```

```

kOutputNumBoxFeature;

objects->clear();
base::ObjectPool::Instance().BatchGet(num_objects,
objects);

for (int i = 0; i < num_objects; ++i) {
    auto& object = objects->at(i);
    object->id = i;

    // read params of bounding box
    float x = detections->at(i * kOutputNumBoxFeature + 0);
    float y = detections->at(i * kOutputNumBoxFeature + 1);
    float z = detections->at(i * kOutputNumBoxFeature + 2);
    float dx = detections->at(i * kOutputNumBoxFeature + 4);
    float dy = detections->at(i * kOutputNumBoxFeature + 3);
    float dz = detections->at(i * kOutputNumBoxFeature + 5);
    float yaw = detections->at(i * kOutputNumBoxFeature + 6);
    yaw += M_PI / 2;
    yaw = std::atan2(sinf(yaw), cosf(yaw));
    yaw = -yaw;

    // directions
    object->theta = yaw;
    object->direction[0] = cosf(yaw);
    object->direction[1] = sinf(yaw);
    object->direction[2] = 0;
    object->lidar_supplement.is_orientation_ready = true;

    // compute vertexes of bounding box and transform to
    world coordinate
    float dx2cos = dx * cosf(yaw) / 2;
    float dy2sin = dy * sinf(yaw) / 2;
    float dx2sin = dx * sinf(yaw) / 2;
    float dy2cos = dy * cosf(yaw) / 2;
    object->lidar_supplement.num_points_in_roi = 8;
    object->lidar_supplement.on_use = true;
    object->lidar_supplement.is_background = false;
    for (int j = 0; j < 2; ++j) {
        PointF point0, point1, point2, point3;
        float vz = z + (j == 0 ? 0 : dz);
        point0.x = x + dx2cos + dy2sin;
        point0.y = y + dx2sin - dy2cos;

```

```

    point0.z = vz;
    point1.x = x + dx2cos - dy2sin;
    point1.y = y + dx2sin + dy2cos;
    point1.z = vz;
    point2.x = x - dx2cos - dy2sin;
    point2.y = y - dx2sin + dy2cos;
    point2.z = vz;
    point3.x = x - dx2cos + dy2sin;
    point3.y = y - dx2sin - dy2cos;
    point3.z = vz;
    object->lidar_supplement.cloud.push_back(point0);
    object->lidar_supplement.cloud.push_back(point1);
    object->lidar_supplement.cloud.push_back(point2);
    object->lidar_supplement.cloud.push_back(point3);
}
for (auto& pt : object->lidar_supplement.cloud) {
    Eigen::Vector3d trans_point(pt.x, pt.y, pt.z);
    trans_point = pose * trans_point;
    PointD world_point;
    world_point.x = trans_point(0);
    world_point.y = trans_point(1);
    world_point.z = trans_point(2);
    object-
>lidar_supplement.cloud_world.push_back(world_point);
}

// classification (only detect vehicles so far)
// TODO(chenjiahao): Fill object types completely
object-
>lidar_supplement.raw_probs.push_back(std::vector<float>(
    static_cast<int>(base::ObjectType::MAX_OBJECT_TYPE),
0.1f));
    object-
>lidar_supplement.raw_classification_methods.push_back(Name())
);
    object->lidar_supplement.raw_probs
        .back() [static_cast<int>(base::ObjectType::VEHICLE)] =
1.0f;
    // copy to type
    object->type_probs.assign(object-
>lidar_supplement.raw_probs.back().begin(),
object-
>lidar_supplement.raw_probs.back().end());

```

```

object->type = static_cast<base::ObjectType>(
    std::distance(object->type_probs.begin(),
                  std::max_element(object-
>type_probs.begin(),
                           object-
>type_probs.end())));
}

collect_time_ = timer.toc(true);
}

```

PointPillars

PointPillars，一种新颖的编码器，它利用PointNets来学习在垂直列柱体组织中的点云的特征。Apollo中的PointPillars是基于autoware。

疑问：

1. 具体的实现是如何实现的，大量用到了cuda

```

PointPillars::PointPillars(const bool reproduce_result_mode,
                           const float score_threshold,
                           const float nms_overlap_threshold,
                           const std::string pfe_onnx_file,
                           const std::string rpn_onnx_file)

```

其中网络在”pfe_onnx_file”和”rpn_onnx_file”中，那么大概判断PointPillars是否是自己实现了cuda对神经网络的加速计算？？？

综上所述DetectionComponent主要实现了雷达的识别，识别的具体功能实现在”perception/lidar”中，同时”perception/lidar”还实现了雷达的分割和追踪，后面我们会接着分析这个模块。

FusionCameraDetectionComponent

初始化Init

```
bool FusionCameraDetectionComponent::Init() {
    // 1. 初始化配置
    if (InitConfig() != cyber::SUCC) {
        AERROR << "InitConfig() failed.";
        return false;
    }
    // 2. 感知结果
    writer_ =
        node_->CreateWriter<PerceptionObstacles>
    (output_obstacles_channel_name_);
    // 3. 提前融合传感器消息帧
    sensorframe_writer_ =
        node_->CreateWriter<SensorFrameMessage>
    (prefused_channel_name_);
    // 4. 相机可视化消息
    camera_viz_writer_ = node_-
    >CreateWriter<CameraPerceptionVizMessage>(
        camera_perception_viz_message_channel_name_);
    // 5. 相机调试消息
    camera_debug_writer_ =
        node_-
    >CreateWriter<apollo::perception::camera::CameraDebug>(
        camera_debug_channel_name_);
    // 6. 初始化场景
    if (InitSensorInfo() != cyber::SUCC) {
        AERROR << "InitSensorInfo() failed.";
        return false;
    }
    // 7. 初始化算法
    if (InitAlgorithmPlugin() != cyber::SUCC) {
        AERROR << "InitAlgorithmPlugin() failed.";
        return false;
    }
    // 8. 初始化相机帧
    if (InitCameraFrames() != cyber::SUCC) {
        AERROR << "InitCameraFrames() failed.";
        return false;
    }
    // 9. 初始化矩阵
    if (InitProjectMatrix() != cyber::SUCC) {
        AERROR << "InitProjectMatrix() failed.";
        return false;
    }
}
```

```

// 10. 初始化相机监听
if (InitCameraListeners() != cyber::SUCC) {
    AERROR << "InitCameraListeners() failed.";
    return false;
}
// 11. 初始化移动服务? ?
if (InitMotionService() != cyber::SUCC) {
    AERROR << "InitMotionService() failed.";
    return false;
}

// 12. 设置相机高度和角度
SetCameraHeightAndPitch();

// Init visualizer
// TODO(techoe, yg13): homography from image to ground
should be
// computed from camera height and pitch.
// Apply online calibration to adjust pitch/height
automatically
// Temporary code is used here for test

double pitch_adj_degree = 0.0;
double yaw_adj_degree = 0.0;
double roll_adj_degree = 0.0;
// load in lidar to imu extrinsic
Eigen::Matrix4d ex_lidar2imu;
LoadExtrinsics(FLAGS_obs_sensor_intrinsic_path + "/" +
                "velodyne128_novatel_extrinsics.yaml",
                &ex_lidar2imu);
AINFO << "velodyne128_novatel_extrinsics: " <<
ex_lidar2imu;

// 13. 可视化
ACHECK(visualize_.Init_all_info_single_camera(
    camera_names_, visual_camera_, intrinsic_map_,
    extrinsic_map_,
    ex_lidar2imu, pitch_adj_degree, yaw_adj_degree,
    roll_adj_degree,
    image_height_, image_width_));
homography_im2car_ =
visualize_.homography_im2car(visual_camera_);

```

```

    camera_obstacle_pipeline_-
>SetIm2CarHomography(homography_im2car_);

    // 14. 车道前方最危险目标检测
    if (enable_cipv_) {
        cipv_.Init(homography_im2car_,
min_laneline_length_for_cipv_,
                    average_lane_width_in_meter_,
max_vehicle_width_in_meter_,
                    average_frame_rate_, image_based_cipv_,
debug_level_);
    }

    // 15. 使能可视化
    if (enable_visualization_) {
        if (write_visual_img_) {
            visualize_.write_out_img_ = true;
            visualize_.SetDirectory(visual_debug_folder_);
        }
    }

    return true;
}

```

算法部分的实现在”InitAlgorithmPlugin()”中初始化，具体的实现
在”ObstacleCameraPerception”中

ObstacleCameraPerception

初始化Init

相机检测的初始化比较复杂，分为了几个方面。

其中读取的配置在文

件”perception\production\conf\perception\camera\obstacle.pt”中，这里的”.pt”文件是否是pytorch传统概念上的pt文件？？？如果打开实际上是一个配置文件，并且指定了具体的pt文件在哪个目录。

```

bool ObstacleCameraPerception::Init(
    const CameraPerceptionInitOptions &options) {

    // 1. 初始化探测器
    base::BaseCameraModelPtr model;
    for (int i = 0; i <
perception_param_.detector_param_size(); ++i) {
        ObstacleDetectorInitOptions detector_init_options;
        app::DetectorParam detector_param =
perception_param_.detector_param(i);
        auto plugin_param = detector_param.plugin_param();
        detector_init_options.root_dir =
            GetAbsolutePath(work_root, plugin_param.root_dir());
        detector_init_options.conf_file =
plugin_param.config_file();
        detector_init_options.gpu_id =
perception_param_.gpu_id();

    // 1.1 获取模型
    model = common::SensorManager::Instance()-
>GetUndistortCameraModel(
        detector_param.camera_name());
    auto pinhole = static_cast<base::PinholeCameraModel *>
(model.get());
    name_intrinsic_map_.insert(std::pair<std::string,
Eigen::Matrix3f>(
        detector_param.camera_name(), pinhole-
>get_intrinsic_params()));
    detector_init_options.base_camera_model = model;
    std::shared_ptr<BaseObstacleDetector> detector_ptr(
BaseObstacleDetectorRegisterer::GetInstanceByName(plugin_para
m.name()));

    // 1.2 探测器
    name_detector_map_.insert(
        std::pair<std::string,
std::shared_ptr<BaseObstacleDetector>>(
            detector_param.camera_name(), detector_ptr));
    // 1.3 初始化探测器

ACHECK(name_detector_map_.at(detector_param.camera_name())-
->Init(detector_init_options))
<< "Failed to init: " << plugin_param.name();
}

```

```

}

// 2. 初始化追踪器
ACHECK(perception_param_.has_tracker_param()) << "Failed to
init tracker.";
{
    ObstacleTrackerInitOptions tracker_init_options;
    tracker_init_options.image_width = static_cast<float>
(model->get_width());
    tracker_init_options.image_height = static_cast<float>
(model->get_height());
    tracker_init_options.gpu_id = perception_param_.gpu_id();
    auto plugin_param =
perception_param_.tracker_param().plugin_param();
    tracker_init_options.root_dir =
        GetAbsolutePath(work_root, plugin_param.root_dir());
    tracker_init_options.conf_file =
plugin_param.config_file();
    tracker_.reset(
        BaseObstacleTrackerRegisterer::GetInstanceByName(plugin_param
.name()));

    ACHECK(tracker_ != nullptr);
    ACHECK(tracker_->Init(tracker_init_options))
        << "Failed to init: " << plugin_param.name();
}

// 3. 初始化转换器
ACHECK(perception_param_.has_transformer_param())
    << "Failed to init transformer.";
{
    ObstacleTransformerInitOptions transformer_init_options;
    auto plugin_param =
perception_param_.transformer_param().plugin_param();
    transformer_init_options.root_dir =
        GetAbsolutePath(work_root, plugin_param.root_dir());
    transformer_init_options.conf_file =
plugin_param.config_file();

    transformer_.reset(BaseObstacleTransformerRegisterer::GetInstanceByName(
        plugin_param.name()));
    ACHECK(transformer_ != nullptr);
}

```

```

    ACHECK(transformer_->Init(transformer_init_options))
        << "Failed to init: " << plugin_param.name();
}

// 4. 初始化障碍物后处理
ACHECK(perception_param_.has_postprocessor_param())
    << "Failed to init obstacle postprocessor.";
{
    ObstaclePostprocessorInitOptions
obstacle_postprocessor_init_options;
    auto plugin_param =
perception_param_.postprocessor_param().plugin_param();
    obstacle_postprocessor_init_options.root_dir =
        GetAbsolutePath(work_root, plugin_param.root_dir());
    obstacle_postprocessor_init_options.conf_file =
plugin_param.config_file();
    obstacle_postprocessor_.reset()

BaseObstaclePostprocessorRegisterer::GetInstanceByName(
    plugin_param.name()));
    ACHECK(obstacle_postprocessor_ != nullptr);
    ACHECK(obstacle_postprocessor_-
>Init(obstacle_postprocessor_init_options))
        << "Failed to init: " << plugin_param.name();
}

// 5. 初始化特征展开
if (!perception_param_.has_feature_param()) {
    AINFO << "No feature config found.";
    extractor_ = nullptr;
} else {
    FeatureExtractorInitOptions init_options;
    auto plugin_param =
perception_param_.feature_param().plugin_param();
    init_options.root_dir = GetAbsolutePath(work_root,
plugin_param.root_dir());
    init_options.conf_file = plugin_param.config_file();
    extractor_.reset()

BaseFeatureExtractorRegisterer::GetInstanceByName(plugin_para
m.name()));
    ACHECK(extractor_ != nullptr);
    ACHECK(extractor_->Init(init_options));
}

```

```

        << "Failed to init: " << plugin_param.name();
    }

lane_calibration_working_sensor_name_ =
    options.lane_calibration_working_sensor_name;

// 6. 初始化车道
InitLane(work_root, perception_param_);

// 7. 初始化校准服务
InitCalibrationService(work_root, model,
perception_param_);

// 8. 初始化调试信息
if (perception_param_.has_debug_param()) {
    // Init debug info
    if (perception_param_.debug_param().has_track_out_file())
{

out_track_.open(perception_param_.debug_param().track_out_file(),
                 std::ofstream::out);
}
    if
(perception_param_.debug_param().has_camera2world_out_file())
{

out_pose_.open(perception_param_.debug_param().camera2world_out_file(),
               std::ofstream::out);
}
}

// 9. 初始化对象模板
if (perception_param_.has_object_template_param()) {
    ObjectTemplateManagerInitOptions init_options;
    auto plugin_param =

perception_param_.object_template_param().plugin_param();
    init_options.root_dir = GetAbsolutePath(work_root,
plugin_param.root_dir());
    init_options.conf_file = plugin_param.config_file();
    ACHECK(ObjectTemplateManager::Instance()-

```

```

>Init(init_options));
}
return true;
}

```

相机感知(Perception)

相机的感知包括车道线和障碍物的感知，障碍物追踪几个功能。

```

bool ObstacleCameraPerception::Perception(
    const CameraPerceptionOptions &options, CameraFrame
    *frame) {

inference::CudaUtil::set_device_id(perception_param_.gpu_id())
);
ObstacleDetectorOptions detector_options;
ObstacleTransformerOptions transformer_options;
ObstaclePostprocessorOptions
obstacle_postprocessor_options;
ObstacleTrackerOptions tracker_options;
FeatureExtractorOptions extractor_options;

frame->camera_k_matrix =
    name_intrinsic_map_.at(frame->data_provider-
>sensor_name());
if (frame->calibration_service == nullptr) {
    AERROR << "Calibraion service is not available";
    return false;
}

// 1. 车道线识别
LaneDetectorOptions lane_detetor_options;
LanePostprocessorOptions lane_postprocessor_options;
if (!lane_detector_->Detect(lane_detetor_options, frame)) {
    AERROR << "Failed to detect lane.";
    return false;
}

// 2. 车道线后处理
if (!lane_postprocessor_-

```

```

>Process2D(lane_postprocessor_options, frame)) {
    AERROR << "Failed to postprocess lane 2D.";
    return false;
}

// 3. 校准服务
frame->calibration_service->Update(frame);
PERCEPTION_PERF_BLOCK_END_WITH_INDICATOR(frame-
>data_provider->sensor_name(),
"CalibrationService");

if (!lane_postprocessor_-
>Process3D(lane_postprocessor_options, frame)) {
    AERROR << "Failed to postprocess lane 3D.";
    return false;
}

// 4. 输出车道信息到文件
if (write_out_lane_file_) {
    std::string lane_file_path =
        absl::StrCat(out_lane_dir_, "/", frame->frame_id,
".txt");
    WriteLanelines(write_out_lane_file_, lane_file_path,
frame->lane_objects);
}

// 5. 输出校准信息到文件
if (write_out_calib_file_) {
    std::string calib_file_path =
        absl::StrCat(out_calib_dir_, "/", frame->frame_id,
".txt");
    WriteCalibrationOutput(write_out_calib_file_,
calib_file_path, frame);
}

// 6. 障碍物追踪
if (!tracker_->Predict(tracker_options, frame)) {
    AERROR << "Failed to predict.";
    return false;
}

std::shared_ptr<BaseObstacleDetector> detector =

```

```

        name_detector_map_.at(frame->data_provider-
>sensor_name()));

// 7. 障碍物识别
if (!detector->Detect(detector_options, frame)) {
    AERROR << "Failed to detect.";
    return false;
}

// 8. 保存障碍物信息为KITTI格式
WriteDetections(
    perception_param_.debug_param().has_detection_out_dir(),
    absl::StrCat(perception_param_.debug_param().detection_out_di-
r(), "/",
                  frame->frame_id, ".txt"),
    frame->detected_objects);
if (extractor_ && !extractor_->Extract(extractor_options,
frame)) {
    AERROR << "Failed to extractor";
    return false;
}

// Save detection results with bbox, detection_feature
WriteDetections(
    perception_param_.debug_param().has_detect_feature_dir(),
    absl::StrCat(perception_param_.debug_param().detect_feature_d-
ir(), "/",
                  frame->frame_id, ".txt"),
    frame);
// Set the sensor name of each object
for (size_t i = 0; i < frame->detected_objects.size(); ++i)
{
    frame->detected_objects[i]->camera_supplement.sensor_name =
        frame->data_provider->sensor_name();
}
if (!tracker_->Associate2D(tracker_options, frame)) {
    AERROR << "Failed to associate2d.";
    return false;
}

```

```

}

// 9. 进行坐标转换
if (!transformer_->Transform(transformer_options, frame)) {
    AERROR << "Failed to transform.";
    return false;
}

// 10. 障碍物后处理

obstacle_postprocessor_options.do_refinement_with_calibration_
_service =
    frame->calibration_service != nullptr;
if (!obstacle_postprocessor_-
>Process(obstacle_postprocessor_options,
            frame)) {
    AERROR << "Failed to post process obstacles.";
    return false;
}

if (!tracker_->Associate3D(tracker_options, frame)) {
    AERROR << "Failed to Associate3D.";
    return false;
}

// 11. 追踪障碍物信息
if (!tracker_->Track(tracker_options, frame)) {
    AERROR << "Failed to track.";
    return false;
}

if (perception_param_.has_debug_param()) {
    if
(perception_param_.debug_param().has_camera2world_out_file())
{
    WriteCamera2World(out_pose_, frame->frame_id, frame-
>camera2world_pose);
    }
    if (perception_param_.debug_param().has_track_out_file())
{
    WriteTracking(out_track_, frame->frame_id, frame-
>tracked_objects);
    }
}

```

```

    }

    // 12. 保存障碍物追踪信息为KITTI格式
    WriteDetections(
        perception_param_.debug_param().has_tracked_detection_out_dir(),
        absl::StrCat(perception_param_.debug_param().tracked_detection_out_dir(),
                     "/",
                     frame->frame_id, ".txt"),
        frame->tracked_objects);

    // 13. 保存结果? ?
    for (auto &obj : frame->tracked_objects) {
        FillObjectPolygonFromBBox3D(obj.get());
        obj->anchor_point = obj->center;
    }

    return true;
}

```

前方最危险目标检测(Cipv)

Cipv在”perception/camera”中，

FusionComponent

融合感知模块采用的是”ObstacleMultiSensorFusion”中的实现。当消息到来的时候会执行Proc，而Proc是调用内部实现”InternalProc”。

```

bool FusionComponent::InternalProc(
    const std::shared_ptr<SensorFrameMessage const>&
    in_message,
    std::shared_ptr<PerceptionObstacles> out_message,
    std::shared_ptr<SensorFrameMessage> viz_message) {
{
    std::unique_lock<std::mutex> lock(s_mutex_);
    s_seq_num_++;

```

```

}

PERCEPTION_PERF_BLOCK_START();
const double timestamp = in_message->timestamp_;
const uint64_t lidar_timestamp = in_message-
>lidar_timestamp_;
std::vector<base::ObjectPtr> valid_objects;
if (in_message->error_code_ != apollo::common::ErrorCode::OK) {
    if (!MsgSerializer::SerializeMsg(
        timestamp, lidar_timestamp, in_message->seq_num_,
        valid_objects,
        in_message->error_code_, out_message.get())) {
        AERROR << "Failed to gen PerceptionObstacles object.";
        return false;
    }
    if (FLAGS_obs_enable_visualization) {
        viz_message->process_stage_ =
ProcessStage::SENSOR_FUSION;
        viz_message->error_code_ = in_message->error_code_;
    }
    AERROR << "Fusion receive message with error code, skip it.";
    return true;
}
base::FramePtr frame = in_message->frame_;
frame->timestamp = in_message->timestamp_;

std::vector<base::ObjectPtr> fused_objects;
if (!fusion_->Process(frame, &fused_objects)) {
    AERROR << "Failed to call fusion plugin.";
    return false;
}

PERCEPTION_PERF_BLOCK_END_WITH_INDICATOR(std::string("fusion_
process"),
                                         in_message-
>sensor_id_);

if (in_message->sensor_id_ != fusion_main_sensor_) {
    return true;
}

```

```

Eigen::Matrix4d sensor2world_pose =
    in_message->frame_->sensor2world_pose.matrix();
if (object_in_roi_check_ && FLAGS_obs_enable_hdmap_input) {
    // get hdmap
    base::HdmapStructPtr hdmap(new base::HdmapStruct());
    if (hdmap_input_) {
        base::PointD position;
        position.x = sensor2world_pose(0, 3);
        position.y = sensor2world_pose(1, 3);
        position.z = sensor2world_pose(2, 3);
        hdmap_input_->GetRoiHDMapStruct(position,
radius_for_roi_object_check_,
                                         hdmap);
        // TODO(use check)
        // ObjectInRoiSlackCheck(hdmap, fused_objects,
&valid_objects);
        valid_objects.assign(fused_objects.begin(),
fused_objects.end());
    } else {
        valid_objects.assign(fused_objects.begin(),
fused_objects.end());
    }
} else {
    valid_objects.assign(fused_objects.begin(),
fused_objects.end());
}
PERCEPTION_PERF_BLOCK_END_WITH_INDICATOR(std::string("fusion_
roi_check"),
                                         in_message-
>sensor_id_);

// produce visualization msg
if (FLAGS_obs_enable_visualization) {
    viz_message->timestamp_ = in_message->timestamp_;
    viz_message->seq_num_ = in_message->seq_num_;
    viz_message->frame_ = base::FramePool::Instance().Get();
    viz_message->frame_->sensor2world_pose =
        in_message->frame_->sensor2world_pose;
    viz_message->sensor_id_ = in_message->sensor_id_;
    viz_message->hdmap_ = in_message->hdmap_;
    viz_message->process_stage_ =
ProcessStage::SENSOR_FUSION;
}

```

```

    viz_message->error_code_ = in_message->error_code_;
    viz_message->frame_->objects = fused_objects;
}
// produce pb output msg
apollo::common::ErrorCode error_code =
apollo::common::ErrorCode::OK;
if (!MsgSerializer::SerializeMsg(timestamp,
lidar_timestamp,
in_message->seq_num_,
valid_objects,
error_code,
out_message.get())) {
    AERROR << "Failed to gen PerceptionObstacles object.";
    return false;
}
PERCEPTION_PERF_BLOCK_END_WITH_INDICATOR(
    std::string("fusion_serialize_message"), in_message-
>sensor_id_);

const double cur_time =
apollo::common::time::Clock::NowInSeconds();
const double latency = (cur_time - timestamp) * 1e3;
AINFO << "FRAME_STATISTICS:Obstacle:End:msg_time[" <<
timestamp
    << "]":cur_time[" << cur_time << "]":cur_latency[" <<
latency
    << "]":obj_cnt[" << valid_objects.size() << "]";
AINFO << "publish_number: " << valid_objects.size() << "
obj";
return true;
}

```

ObstacleMultiSensorFusion

ObstacleMultiSensorFusion实现了障碍物的融合，在目录”modules\perception\fusion”中。

LaneDetectionComponent

LidarOutputComponent

RecognitionComponent

RadarDetectionComponent

SegmentationComponent

TrafficLightsPerceptionComponent

至此，整个感知模块的分析就完成了，可以看到感知模块主要是负责获取障碍物的类型、以及车道等信息反馈给车。

URL

<https://blog.csdn.net/jinzhuojun/article/details/80875264>

<https://blog.csdn.net/jinzhuojun/article/details/83038279>

<https://zhuanlan.zhihu.com/p/33416142>

https://github.com/ApolloAuto/apollo/blob/master/docs/specs/3d_obstacle_perception_cn.md

感知综述

<https://zhuanlan.zhihu.com/p/33416142>

Table of Contents

- Caffe2环境准备
- 安装显卡驱动
- 安装CUDA
 - 选择CUDA版本
 - 安装CUDA
 - 设置环境变量
 - 检验安装
- 安装cuDNN
- 安装Caffe2
- 参考

因为Apollo中的深度学习框架采用的是Caffe2框架，我们需要安装好Caffe2的环境才能进一步学习，下面主要介绍了如何安装Caffe2。安装完成之后，就可以训练深度学习模型了。

Caffe2环境准备

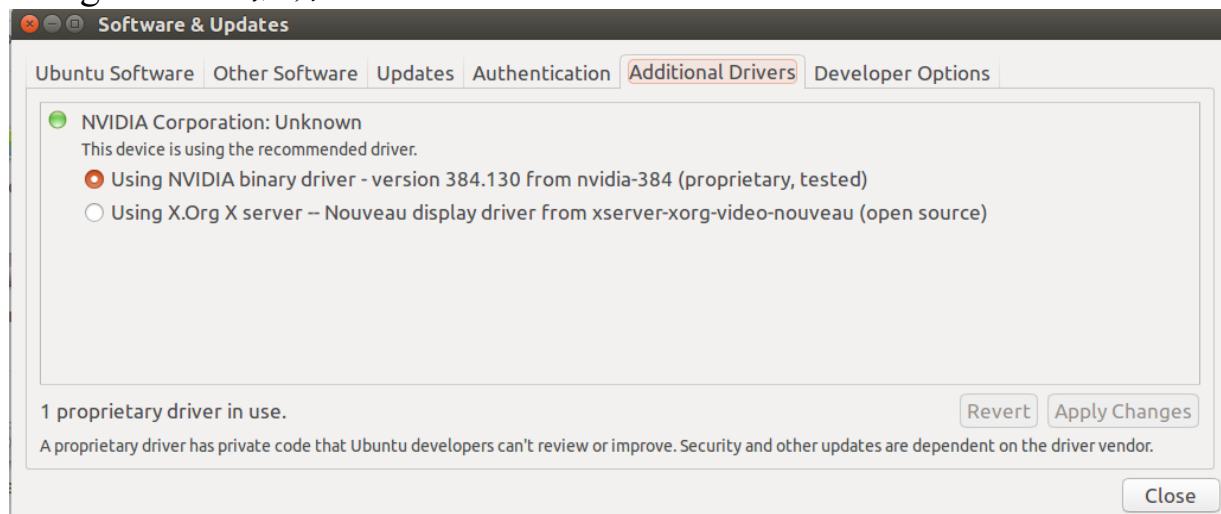
我们以有显卡的情况为例，来安装caffe2环境，安装caffe2之前需要先安装英伟达的显卡驱动，之后还要安装cuda toolkit和cuDNN，最后安装Caffe2。

我们先把需要安装的软件列出来，再告诉如何选择对应的版本：

- 操作系统: Ubuntu 16.04.5 LTS
- 显卡驱动版本: 384.130
- CUDA Toolkit版本: CUDA Toolkit 9.0
- cuDNN 版本: cuDNN v7.6.0
- pytorch 版本: pytorch-nightly-1.2.0

安装显卡驱动

ubuntu 16.04可以在设置”System Settings - Software & Updates”中选择Using NVIDIA驱动。



安装好驱动后可以通过下面的命令来验证驱动是否安装成功：

```
root@root:~/cuda/$ nvidia-smi
Wed May 29 12:05:01 2019
+-----
|-----+
| NVIDIA-SMI 384.130                    Driver Version: 384.130
| |
|-----+-----+-----+
| GPU  Name      Persistence-M| Bus-Id      Disp.A  |
Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|           Memory-Usage | GPU-
Util  Compute M. |
|=====+=====+=====+=====+=====+=====+=====|
|     0  GeForce MX130        Off  | 00000000:02:00.0 Off |
N/A   |
| N/A    73C     P0      N/A /  N/A | 1240MiB / 2002MiB |
67%    Default |
+-----+-----+-----+
+-----+
| Processes:
GPU Memory |
```

```

| GPU      PID  Type   Process name
Usage
=====
=====
| 0      1682    G   /usr/lib/xorg/Xorg
445MiB |
| 0      2744    G   compiz
192MiB |
| 0      3101    G   ...quest-channel-
token=7501548652516259525  599MiB |
+-----+
-----+

```



安装CUDA

选择CUDA版本

安装好显卡驱动后，就可以安装CUDA了，那么我们如何选择CUDA版本呢？首先查看显卡驱动版本，就是上面”nvidia-smi”显示的”Driver Version: 384.130”，然后查看下表，[官网地址](https://docs.nvidia.com/cuda/cuda-toolkit-release-notes/index.html) [<https://docs.nvidia.com/cuda/cuda-toolkit-release-notes/index.html>]：

Table 1. CUDA Toolkit and Compatible Driver Versions

CUDA Toolkit	Linux x86_64 Driver Version	Windows x86_64 Driver Version
CUDA 10.1.105	≥ 418.39	≥ 418.96
CUDA 10.0.130	≥ 410.48	≥ 411.31
CUDA 9.2 (9.2.148 Update 1)	≥ 396.37	≥ 398.26
CUDA 9.2 (9.2.88)	≥ 396.26	≥ 397.44
CUDA 9.1 (9.1.85)	≥ 390.46	≥ 391.29
CUDA 9.0 (9.0.76)	≥ 384.81	≥ 385.54
CUDA 8.0 (8.0.61 GA2)	≥ 375.26	≥ 376.51
CUDA 8.0 (8.0.44)	≥ 367.48	≥ 369.30
CUDA 7.5 (7.5.16)	≥ 352.31	≥ 353.66
CUDA 7.0 (7.0.28)	≥ 346.46	≥ 347.62

可以看到驱动版本” ≥ 390.46 ”选择CUDA 9.0，当然还需要查看下内

核，GCC,GLIBC版本是否支持，参照下表，[官网地址](#)

[<https://docs.nvidia.com/cuda/cuda-installation-guide-linux/index.html>]:

Table 1. Native Linux Distribution Support in CUDA 10.1 Update 1

Distribution	Kernel*	GCC	GLIBC	ICC	PGI	XLC	CLANG
x86_64							
RHEL 7.6	3.10	4.8.5	2.17	19.0	18.x, 19.x	NO	7.0.0
RHEL 6.10	2.6.32	4.4.7	2.12				
CentOS 7.6	3.10	4.8.5	2.17				
CentOS 6.10	2.6.32	4.4.7	2.12				
Fedora 29	4.16	8.0.1	2.27				
OpenSUSE Leap 15.0	4.15.0	7.3.1	2.26				
SLES 15.0	4.12.14	7.2.1	2.26				
SLES 12.4	4.12.14	4.8.5	2.22				
Ubuntu 18.10	4.18.0	8.2.0	2.28				
Ubuntu 18.04.2 (**)	4.15.0	7.3.0	2.27				
Ubuntu 16.04.6 (**)	4.4	5.4.0	2.23				
Ubuntu 14.04.6 (**)	3.13	4.8.4	2.19				
POWER8(***)							
RHEL 7.6	3.10	4.8.5	2.17	NO	18.x, 19.x	13.1.x, 16.1.x	7.0.0
Ubuntu 18.04.1	4.15.0	7.3.0	2.27	NO	18.x, 19.x	13.1.x, 16.1.x	7.0.0
POWER9(****)							
Ubuntu 18.04.1	4.15.0	7.3.0	2.27	NO	18.x, 19.x	13.1.x, 16.1.x	7.0.0
RHEL 7.6 IBM Power LE	4.14.0	4.8.5	2.17	NO	18.x, 19.x	13.1.x, 16.1.x	7.0.0

查看linux内核版本:

```
root:~/cuda$ cat /proc/version
Linux version 4.15.0-50-generic (buildd@lgw01-amd64-029) (gcc
version 5.4.0 20160609 (Ubuntu 5.4.0-6ubuntu1~16.04.10))
#54~16.04.1-Ubuntu SMP Wed May 8 15:55:19 UTC 2019
```

这里提供的是支持CUDA10.1需要的配置，我目前的ubuntu 16.04已经都支持了，所以9.0肯定没有问题。

安装CUDA

选择好版本后，就可以下载安装CUDA了，CUDA官方的[下载地址](https://developer.nvidia.com/cuda-toolkit-archive) [<https://developer.nvidia.com/cuda-toolkit-archive>]，选择对应的版本，下载之后运行：

```
sudo sh cuda_9.0.176_384.81_linux.run
```

记住前面已经安装了驱动，所以安装**CUDA**的时候第一步需要跳过安装驱动，只安装**CUDA Toolkit**，接着按照提示安装就可以了。

设置环境变量

安装完成之后，我们需要设置CUDA环境变量才能运行：

```
sudo vi ~/.bashrc
```

在文件最后增加2行：

```
export LD_LIBRARY_PATH="$LD_LIBRARY_PATH:/usr/local/cuda-9.0/lib64:/usr/local/cuda-9.0/extras/CUPTI/lib64"  
export CUDA_HOME=/usr/local/cuda-9.0
```

将环境变量生效：

```
source ~/.bashrc
```

检验安装

我们可以通过samples的deviceQuery来检验CUDA是否安装成功。

```
cd /usr/local/cuda/samples/1_Utils/deviceQuery  
sudo make  
.deviceQuery
```

如果提示”Result = PASS”，则表示安装成功，如果提示”Result = FAIL”，则表示安装失败：

```

$ ./deviceQuery
./deviceQuery Starting...

CUDA Device Query (Runtime API) version (CUDART static
linking)

Detected 1 CUDA Capable device(s)

Device 0: "GeForce MX130"
  CUDA Driver Version / Runtime Version          9.0 / 9.0
  CUDA Capability Major/Minor version number:    5.0
  Total amount of global memory:                2003 MBytes
  (2100232192 bytes)
    ( 3) Multiprocessors, (128) CUDA Cores/MP:   384 CUDA
  Cores
    GPU Max Clock rate:                      1189 MHz
  (1.19 GHz)
    Memory Clock rate:                      2505 Mhz
    Memory Bus Width:                       64-bit
    L2 Cache Size:                          1048576
  bytes
    Maximum Texture Dimension Size (x,y,z)     1D=(65536),
  2D=(65536, 65536), 3D=(4096, 4096, 4096)
    Maximum Layered 1D Texture Size, (num) layers 1D=(16384),
  2048 layers
    Maximum Layered 2D Texture Size, (num) layers 2D=(16384,
  16384), 2048 layers
    Total amount of constant memory:           65536 bytes
    Total amount of shared memory per block:    49152 bytes
    Total number of registers available per block: 65536
    Warp size:                                32
    Maximum number of threads per multiprocessor: 2048
    Maximum number of threads per block:        1024
    Max dimension size of a thread block (x,y,z): (1024, 1024,
  64)
    Max dimension size of a grid size      (x,y,z): (2147483647,
  65535, 65535)
    Maximum memory pitch:                   2147483647
  bytes
    Texture alignment:                      512 bytes
    Concurrent copy and kernel execution: Yes with 1
  copy engine(s)
    Run time limit on kernels:             Yes

```

```
Integrated GPU sharing Host Memory: No
Support host page-locked memory mapping: Yes
Alignment requirement for Surfaces: Yes
Device has ECC support: Disabled
Device supports Unified Addressing (UVA): Yes
Supports Cooperative Kernel Launch: No
Supports MultiDevice Co-op Kernel Launch: No
Device PCI Domain ID / Bus ID / location ID: 0 / 2 / 0
Compute Mode:
    < Default (multiple host threads can use
::cudaSetDevice() with device simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 9.0,
CUDA Runtime Version = 9.0, NumDevs = 1
Result = PASS
```

安装cuDNN

我们安装好了CUDA之后，如果希望深度学习训练的时候能够加速，则需要安装”cuDNN”，这一步很简单，只需要根据CUDA的版本选择对应的cuDNN版本就可以了。

可以在[官网](https://developer.nvidia.com/rdp/cudnn-download) [https://developer.nvidia.com/rdp/cudnn-download] 下载，CUDA 9.0对应的cuDNN版本为v7.6.0，下载选择linux版本。

下载完成之后解压文件：

```
tar -xzvf cudnn-9.0-linux-x64-v7.6.0.64.tgz
```

解压之后的文件如下：

```
|- cuda
  |-- include
  |   '-- cudnn.h
  '-- lib64
      |-- libcudnn.so -> libcudnn.so.7
      |   '-- libcudnn.so.7 -> libcudnn.so.7.6.0
      '-- libcudnn.so.7.6.0
          '-- libcudnn_static.a
      '-- NVIDIA_SLA_cuDNN_Support.txt
```

然后把文件拷贝到CUDA的目录下就可以了：

```
sudo cp cuda/include/cudnn.h /usr/local/cuda-9.0/include  
sudo cp cuda/lib64/libcudnn* /usr/local/cuda-9.0/lib64
```

经过上面的步骤，我们就安装好了整个CUDA环境，主要是确认好CUDA版本，如果版本不对，对应的检测就不通过。需要仔细确认系统支持的CUDA版本。

安装Caffe2

直接安装（二进制文件安装）

caffe2选择直接用anaconda安装，因为anaconda集成了大部分的工具，安装起来也很简单：

```
conda install pytorch-nightly -c pytorch
```

这是官网的[安装说明](https://caffe2.ai/docs/getting-started.html?platform=ubuntu&configuration=prebuilt) [<https://caffe2.ai/docs/getting-started.html?platform=ubuntu&configuration=prebuilt>]

至此所有的安装就已经完成了，可以开始深度学习的尝试了。

源码安装

上面的方法是直接安装编译好的caffe2，而有些选项默认为关闭，想要打开这些选项（例如USE_LMDB），就需要从源码安装caffe2。

从源码安装caffe2可以参考[官网教程](https://github.com/pytorch/pytorch#from-source) [<https://github.com/pytorch/pytorch#from-source>]。

1. 安装依赖

```
conda install numpy ninja pyyaml mkl mkl-include setuptools  
cmake cffi typing
```

2. 安装magma

```
# Add LAPACK support for the GPU if needed  
conda install -c pytorch magma-cuda90 # or [magma-cuda92 |  
magma-cuda100 ] depending on your cuda version
```

3. 下载PyTorch

```
git clone --recursive https://github.com/pytorch/pytorch  
cd pytorch  
# if you are updating an existing checkout  
git submodule sync  
git submodule update --init --recursive
```

4. 安装PyTorch

```
export CMAKE_PREFIX_PATH=${CONDA_PREFIX:-$(dirname $(which  
conda))/..}"  
python setup.py install
```

如果需要运行MNIST.ipynb的例子，需要同时安装LMDB，参考[issue](https://github.com/pytorch/pytorch/issues/21117)
[<https://github.com/pytorch/pytorch/issues/21117>]

```
1. install lmdb  
conda install lmdb leveldb  
  
2. rebuild caffe2 from source  
USE_LMDB=ON python setup.py install --cmake
```

这之后整个安装过程就结束了，现在你可以试一试在jupyter-notebook中运行MNIST.ipynb，开始学习caffe2！

参考

[Ubuntu16.04+Anaconda3+CUDA+CUDNN+Tensorflow-gpu配置教程](#)

[<https://zhuanlan.zhihu.com/p/37569310>]

[ubuntu16.04下安装CUDA, cuDNN及tensorflow-gpu版本过程](#)

[<https://blog.csdn.net/u014595019/article/details/53732015>]

camera

camera模块的结构和radar类似， 目录如下：

```
.  
├── app           \\\ 主程序  
├── common        \\\ 公共程序  
├── lib           \\\ 库，用来做红绿灯、障碍物检测等功能  
├── test          \\\ 测试用例  
└── tools         \\\ 工具，用来做车道线和红绿灯识别结果展示
```

lib目录

traffic_light

红绿灯的识别分为3个阶段，先预处理，然后识别，然后跟踪？

preprocessor // 预处理 detector - detection // 检测 - recognition // 识别，
通过上面的detection检测？？？ tracker // 追踪红绿灯

preprocessor预处理

首先来看红绿灯预处理，主要有个文件”pose.cc”，”multi_camera_projection.cc”和”tl_preprocessor.cc”，先看”pose.cc”的实现。

在”pose.cc”中实现了”CarPose”类，主要作用是获取车的位置和姿态，从而获取到相机的位置和姿态。在”CarPose”中定义了如下数据结构。

```
Eigen::Matrix4d pose_;    // car(novatel) to world pose  
std::map<std::string, Eigen::Matrix4d> c2w_poses_;    //  
camera to world poses  
double timestamp_;
```

其中”c2w_poses_”存放了不同焦距的相机到世界坐标的转换矩阵。

获取车的位置和姿态的实现如下，这里是默认车的坐标为0？

```
// 初始化车到世界坐标的转换矩阵
bool CarPose::Init(double ts, const Eigen::Matrix4d &pose) {
    timestamp_ = ts;
    pose_ = pose;
    return true;
}

const Eigen::Matrix4d CarPose::getCarPose() const { return
pose_; }

const Eigen::Vector3d CarPose::getCarPosition() const {
    Eigen::Vector3d p;
    p[0] = pose_(0, 3);
    p[1] = pose_(1, 3);
    p[2] = pose_(2, 3);

    return p;
}
```

获取相机到世界坐标的转换矩阵。

```
void CarPose::SetCameraPose(const std::string &camera_name,
                           const Eigen::Matrix4d &c2w_pose)
{
    c2w_poses_[camera_name] = c2w_pose;
}

bool CarPose::GetCameraPose(const std::string &camera_name,
                           Eigen::Matrix4d *c2w_pose) const
{
    if (c2w_pose == nullptr) {
        AERROR << "c2w_pose is not available";
        return false;
    }
    if (c2w_poses_.find(camera_name) == c2w_poses_.end()) {
        return false;
    }
    *c2w_pose = c2w_poses_.at(camera_name);
```

```
    return true;
}
```

删除某个相机到世界坐标的转换矩阵。

```
void CarPose::ClearCameraPose(const std::string &camera_name)
{
    auto it = c2w_poses_.find(camera_name);
    if (it != c2w_poses_.end()) {
        c2w_poses_.erase(it);
    }
}
```

这里还提供了重载的输入输出流”<<”，有个疑问这里的”pose_”类型为”Eigen::Matrix4d”打印的时候会直接展开吗？

```
std::ostream &operator<<(std::ostream &os, const CarPose
&pose) {
    os << pose.pose_;
    return os;
}
```

“multi_camera_projection.cc”中实现了”MultiCamerasProjection”类，先看数据结构。

```
// sorted by focal length in descending order
// 1. 相机名称，按照焦距升序排列
std::vector<std::string> camera_names_;
// camera_name -> camera_model
// 2. 根据相机名称，获取对应的相机模型
std::map<std::string, base::BrownCameraDistortionModelPtr>
camera_models_;
```

首先看”MultiCamerasProjection”类的初始化流程。

```
bool MultiCamerasProjection::Init(const
MultiCamerasInitOption& options) {
    // 1. 初始化sensor_manager
    common::SensorManager* sensor_manager =
common::SensorManager::Instance();
    if (!sensor_manager->Init()) {
```

```

        AERROR << "sensor_manager init failed";
    }

    // 2. 遍历相机列表
    for (size_t i = 0; i < options.camera_names.size(); ++i) {
        const std::string& cur_camera_name =
options.camera_names.at(i);

        // 3. 查看相机是否存在
        if (!sensor_manager->IsSensorExist(cur_camera_name)) {
            AERROR << "sensor " << cur_camera_name << " do not
exist";
            return false;
        }

        // 4. 从sensor_manager中获取相机模型
        camera_models_[cur_camera_name] =

std::dynamic_pointer_cast<base::BrownCameraDistortionModel>(
            sensor_manager-
>GetDistortCameraModel(cur_camera_name));

        camera_names_.push_back(cur_camera_name);
    }
    // sort camera_names_ by focal lengths (descending order)
    std::sort(camera_names_.begin(), camera_names_.end(),
              [&](const std::string& lhs, const std::string&
rhs) {
        const auto lhs_cam_intrinsics =
            camera_models_[lhs]-
>get_intrinsic_params();
        const auto rhs_cam_intrinsics =
            camera_models_[rhs]-
>get_intrinsic_params();
        // 5. 计算焦点长度，并且比较大小
        auto lhs_focal_length =
            0.5 * (lhs_cam_intrinsics(0, 0) +
lhs_cam_intrinsics(1, 1));
        auto rhs_focal_length =
            0.5 * (rhs_cam_intrinsics(0, 0) +
rhs_cam_intrinsics(1, 1));
        return lhs_focal_length > rhs_focal_length;
    });
}

```

```

// 6. 打印相机名称数组，用空格隔开
AINFO << "camera_names sorted by descending focal lengths:
"
    << std::accumulate(camera_names_.begin(),
camera_names_.end(),
                     std::string("") ,
[](std::string& sum, const
std::string& s) {
                     return sum + s + " ";
} );

return true;
}

```

投影

```

bool MultiCamerasProjection::Project(const CarPose& pose,
                                      const ProjectOption&
option,
                                      base::TrafficLight*
light) const {
    Eigen::Matrix4d c2w_pose;
    // 1. 通过相机名称获取相机到世界坐标的转换矩阵
    c2w_pose = pose.c2w_poses_.at(option.camera_name);

    bool ret = false;

    // 2. 根据 红绿灯区域坐标点 获取 相机上红绿灯投影点
    ret =
BoundaryBasedProject(camera_models_.at(option.camera_name),
c2w_pose,
                           light->region.points, light);

    if (!ret) {
        AWARN << "Projection failed projection the traffic light.
"
        << "camera_name: " << option.camera_name;
        return false;
    }
    return true;
}

```

再接着看”BoundaryBasedProject”的实现。

```
bool MultiCamerasProjection::BoundaryBasedProject(
    const base::BrownCameraDistortionModelPtr camera_model,
    const Eigen::Matrix4d& c2w_pose,
    const std::vector<base::PointXYZID>& points,
    base::TrafficLight* light) const {
    // 1. 获取照片的宽度和高度
    int width = static_cast<int>(camera_model->get_width());
    int height = static_cast<int>(camera_model->get_height());
    // 2. 获取红绿灯框，必须大于等于4个点
    int bound_size = static_cast<int>(points.size());

    std::vector<Eigen::Vector2i> pts2d(bound_size);
    // 3. 相机到世界坐标的逆矩阵
    auto c2w_pose_inverse = c2w_pose.inverse();

    for (int i = 0; i < bound_size; ++i) {
        const auto& pt3d_world = points.at(i);
        // 4. 红绿灯世界坐标到相机坐标
        Eigen::Vector3d pt3d_cam =
            (c2w_pose_inverse *
             Eigen::Vector4d(pt3d_world.x, pt3d_world.y,
pt3d_world.z, 1.0))
            .head(3);
        // 5. 如果距离小于0，则表示已经经过了红绿灯（红绿灯在车后面）
        if (std::islesseqaul(pt3d_cam[2], 0.0)) {
            AWARN << "light bound point behind the car: " <<
pt3d_cam;
            return false;
        }
        // 6. 从3D投影到2D
        pts2d[i] = camera_model->Project(pt3d_cam.cast<float>()
                                         .cast<int>());
    }

    // 7. 在一系列点中找到左下角和右上角的点，刚好能够包围投影出来的点
    int min_x = std::numeric_limits<int>::max();
    int max_x = std::numeric_limits<int>::min();
    int min_y = std::numeric_limits<int>::max();
    int max_y = std::numeric_limits<int>::min();
    for (const auto& pt : pts2d) {
        min_x = std::min(pt[0], min_x);
        max_x = std::max(pt[0], max_x);
        min_y = std::min(pt[1], min_y);
        max_y = std::max(pt[1], max_y);
    }
}
```

```

    max_x = std::max(pt[0], max_x);
    min_y = std::min(pt[1], min_y);
    max_y = std::max(pt[1], max_y);
}

// 8. 构造图片中感兴趣的区域位置，也就是红绿灯的位置
base::BBox2DI roi(min_x, min_y, max_x, max_y);
// 9. 感兴趣区域为0，或者超过图片的范围，图片的坐标起点为(0, 0)
if (OutOfValidRegion(roi, width, height) || roi.Area() == 0) {
    AWARN << "Projection get ROI outside the image. ";
    return false;
}
light->region.projection_roi = base::RectI(roi);
return true;
}

```

疑问：

1. 这里的图片的坐标系是如何的？坐标起点为图片左上角，向下为y轴，向右为x轴？？？

“tl_preprocessor.cc”中实现了”TLPreprocessor”类。

```

bool TLPreprocessor::Init(const
TrafficLightPreprocessorInitOptions &options) {
camera::MultiCamerasInitOption projection_init_option;
projection_init_option.camera_names = options.camera_names;
// 1. 初始化 MultiCamerasProjection projection_
if (!projection_.Init(projection_init_option)) {
    AERROR << "init multi_camera_projection failed.";
    return false;
}

num_cameras_ =
projection_.getCameraNamesByDescendingFocalLen().size();
// 2. 这里申明2个数组，分别代表红绿灯在图片内，和红绿灯不在图片内
lights_on_image_array_.resize(num_cameras_);
lights_outside_image_array_.resize(num_cameras_);
// 3. 同步间隔
sync_interval_seconds_ = options.sync_interval_seconds;

```

```
    return true;  
}
```

更新选择的相机

```
bool TLPreprocessor::UpdateCameraSelection(  
    const CarPose &pose, const TLPreprocessorOption &option,  
    std::vector<base::TrafficLightPtr> *lights) {  
const double &timestep = pose.getTimestamp();  
// 1. 选择当前时间, 最大焦距  
selected_camera_name_.first = timestep;  
selected_camera_name_.second =  
GetMaxFocalLenWorkingCameraName();  
  
// 2.  
if (!ProjectLightsAndSelectCamera(pose, option,  
&  
(selected_camera_name_.second), lights)) {  
    AERROR << "project_lights_and_select_camera failed, ts: "  
<< timestep;  
}  
  
return true;  
}
```

接下来我们接着看”ProjectLightsAndSelectCamera”。

```
bool TLPreprocessor::ProjectLightsAndSelectCamera(  
    const CarPose &pose, const TLPreprocessorOption &option,  
    std::string *selected_camera_name,  
    std::vector<base::TrafficLightPtr> *lights) {  
// 1. 清除数组  
for (auto &light_ptrs : lights_on_image_array_) {  
    light_ptrs.clear();  
}  
for (auto &light_ptrs : lights_outside_image_array_) {  
    light_ptrs.clear();  
}  
  
// 2. 根据焦距获取相机名称  
const auto &camera_names =  
projection_.getCameraNamesByDescendingFocalLen();
```

```

for (size_t cam_id = 0; cam_id < num_cameras_; ++cam_id) {
    const std::string &camera_name = camera_names[cam_id];
    // 3. 获取投影
    if (!ProjectLights(pose, camera_name, lights,
                        &(lights_on_image_array_[cam_id]),
                        &
                        (lights_outside_image_array_[cam_id]))) {
        AERROR << "select_camera_by_lights_projection project
lights on "
                << camera_name << " image failed";
        return false;
    }
}

projections_outside_all_images_ = !lights->empty();
for (size_t cam_id = 0; cam_id < num_cameras_; ++cam_id) {
    projections_outside_all_images_ =
        projections_outside_all_images_ &&
        (lights_on_image_array_[cam_id].size() < lights-
>size());
}
if (projections_outside_all_images_) {
    AWARN << "lights projections outside all images";
}

// 4. 选择相机
SelectCamera(&lights_on_image_array_,
&lights_outside_image_array_, option,
            selected_camera_name);

return true;
}

```

投影红绿灯

```

bool TLPreprocessor::ProjectLights(
    const CarPose &pose, const std::string &camera_name,
    std::vector<base::TrafficLightPtr> *lights,
    base::TrafficLightPtrs *lights_on_image,
    base::TrafficLightPtrs *lights_outside_image) {

// 1. 查看相机是否工作, 通过查找"camera_is_working_flags_"
bool is_working = false;

```

```

    if (!GetCameraWorkingFlag(camera_name, &is_working) || !is_working) {
        AWARN << "TLPreprocessor::project_lights not project lights, "
              << "camera is not working, camera_name: " << camera_name;
        return true;
    }

    // 2. 遍历红绿灯，并且找到红绿灯在相机上的投影
    for (size_t i = 0; i < lights->size(); ++i) {
        base::TrafficLightPtr light_proj(new base::TrafficLight);
        auto light = lights->at(i);
        if (!projection_.Project(pose,
ProjectOption(camera_name), light.get())) {
            // 3. 没有红绿灯投影在相机上，放入 lights_outside_image
            light->region.outside_image = true;
            *light_proj = *light;
            lights_outside_image->push_back(light_proj);
        } else {
            // 4. 有红绿灯投影在相机上，放入 lights_on_image
            light->region.outside_image = false;
            *light_proj = *light;
            lights_on_image->push_back(light_proj);
        }
    }
    return true;
}

```

疑问：

1. 智能指针先指向一个对象，后面重新指向另外一个对象，前面这个对象会自动释放内存？？？

选择相机，这里如何选择的？？？

```

void TLPreprocessor::SelectCamera(
    std::vector<base::TrafficLightPtrs>
*lights_on_image_array,
    std::vector<base::TrafficLightPtrs>
*lights_outside_image_array,
    const TLPreprocessorOption &option, std::string

```

```

*selected_camera_name) {
    // 1. 获取焦距最小的相机名称
    auto min_focal_len_working_camera =
GetMinFocalLenWorkingCameraName();

    const auto &camera_names =
projection_.getCameraNamesByDescendingFocalLen();
    for (size_t cam_id = 0; cam_id < lights_on_image_array-
>size(); ++cam_id) {
        const auto &camera_name = camera_names[cam_id];
        bool is_working = false;
        // 2. 如果相机没有工作，则跳过
        if (!GetCameraWorkingFlag(camera_name, &is_working) ||
!is_working) {
            AINFO << "camera " << camera_name << "is not working";
            continue;
        }

        bool ok = true;
        if (camera_name != min_focal_len_working_camera) {
            // 如果投影在外的相机大于0
            if (lights_outside_image_array->at(cam_id).size() > 0)
{
                continue;
            }
            auto lights = lights_on_image_array->at(cam_id);
            for (const auto light : lights) {
                // 如果红绿灯超出范围
                if (OutOfValidRegion(light->region.projection_roi,
projection_.getImageWidth(camera_name),
projection_.getImageHeight(camera_name),
option.image_borders_size-
>at(camera_name))) {
                    ok = false;
                    break;
                }
            }
        } else {
            // 如果是最小的焦距，红绿灯的个数是否大于0
            ok = (lights_on_image_array->at(cam_id).size() > 0);
        }
    }
}

```

```

    if (ok) {
        *selected_camera_name = camera_name;
        break;
    }
}
AINFO << "select_camera selection: " <<
*selected_camera_name;
}

```

更新红绿灯投影

```

bool TLPreprocessor::UpdateLightsProjection(
    const CarPose &pose, const TLPreprocessorOption &option,
    const std::string &camera_name,
    std::vector<base::TrafficLightPtr> *lights) {
    lights_on_image_.clear();
    lights_outside_image_.clear();

    AINFO << "clear lights_outside_image_ " <<
    lights_outside_image_.size();

    if (lights->empty()) {
        AINFO << "No lights to be projected";
        return true;
    }

    if (!ProjectLights(pose, camera_name, lights,
&lights_on_image_,
                           &lights_outside_image_)) {
        AERROR << "update_lights_projection project lights on "
        << camera_name
                           << " image failed";
        return false;
    }

    if (lights_outside_image_.size() > 0) {
        AERROR << "update_lights_projection failed,"
                           << "lights_outside_image->size() " <<
        lights_outside_image_.size()
                           << " ts: " << pose.getTimestamp();
        return false;
    }
}

```

```

    auto min_focal_len_working_camera =
GetMinFocalLenWorkingCameraName();
    if (camera_name == min_focal_len_working_camera) {
        return lights_on_image_.size() > 0;
    }
    for (const base::TrafficLightPtr &light : lights_on_image_)
{
    if (OutOfValidRegion(light->region.projection_roi,
projection_.getImageWidth(camera_name),
projection_.getImageHeight(camera_name),
option.image_borders_size-
>at(camera_name))) {
        AINFO << "update_lights_projection light project out of
image region. "
            << "camera_name: " << camera_name;
        return false;
    }
}
AINFO << "UpdateLightsProjection success";
return true;
}

```

更新信息

```

bool TLPreprocessor::SyncInformation(const double
image_timestamp,
                                         const std::string
&cam_name) {
    const double &proj_ts = selected_camera_name_.first;
    const std::string &proj_camera_name =
selected_camera_name_.second;

    if (!projection_.HasCamera(cam_name)) {
        AERROR << "sync_image failed, "
            << "get invalid camera_name: " << cam_name;
        return false;
    }

    if (image_timestamp < last_pub_img_ts_) {

```

```

        AWARN << "TLPreprocessor reject the image pub ts:" <<
image_timestamp
                << " which is earlier than last output ts:" <<
last_pub_img_ts_
                << ", image_camera_name: " << cam_name;
        return false;
    }

    if (proj_camera_name != cam_name) {
        AWARN << "sync_image failed - find close enough
projection,"
                << "but camera_name not match.";
        return false;
    }

    last_pub_img_ts_ = image_timestamp;
    return true;
}

```

detector 检测

红绿灯的检测分为2部分，一部分为检测，一部分为识别，分别在”detection”和”recognition”2个目录中。我们先看”detection”的实现。

在”cropbox.h”和”cropbox.cc”中获取裁剪框。首先是初始化裁切尺度”crop_scale”和最小裁切大小”min_crop_size”

```

void CropBox::Init(float crop_scale, int min_crop_size) {
    crop_scale_ = crop_scale;
    min_crop_size_ = min_crop_size;
}
CropBox::CropBox(float crop_scale, int min_crop_size) {
    Init(crop_scale, min_crop_size);
}

```

CropBoxWholeImage根据指定的长、宽裁切图片，如果红绿灯在指定的长、宽之内，则返回裁切框。

```

void CropBoxWholeImage::getCropBox(const int width, const int
height,

```

```

        const
base::TrafficLightPtr &light,
                                base::RectI *crop_box) {
    // 1. 红绿灯在裁剪范围（给定长、宽）内
    if (!OutOfValidRegion(light->region.projection_roi, width,
height) &&
        light->region.projection_roi.Area() > 0) {
        crop_box->x = crop_box->y = 0;
        crop_box->width = width;
        crop_box->height = height;
        return;
    }
    // 2. 否则返回全0
    crop_box->x = 0;
    crop_box->y = 0;
    crop_box->width = 0;
    crop_box->height = 0;
}

```

裁切图片。

```

void CropBox::getCropBox(const int width, const int height,
                           const base::TrafficLightPtr &light,
                           base::RectI *crop_box) {
    int rows = height;
    int cols = width;
    // 1. 如果超出范围，则返回0
    if (OutOfValidRegion(light->region.projection_roi, width,
height) ||
        light->region.projection_roi.Area() <= 0) {
        crop_box->x = 0;
        crop_box->y = 0;
        crop_box->width = 0;
        crop_box->height = 0;
        return;
    }
    // 2. 获取灯的感兴趣区域
    int xl = light->region.projection_roi.x;
    int yt = light->region.projection_roi.y;
    int xr = xl + light->region.projection_roi.width - 1;
    int yb = yt + light->region.projection_roi.height - 1;

    // scale

```

```

int center_x = (xr + xl) / 2;
int center_y = (yb + yt) / 2;
// 3. 感兴趣区域的2.5倍, crop_scale_为裁切比例
int resize =
    static_cast<int>(crop_scale_ * static_cast<float>
(std::max(
                           light-
>region.projection_roi.width,
                           light-
>region.projection_roi.height)));

// 4. 根据裁切比例和长、宽, 取最小值
resize = std::max(resize, min_crop_size_);
resize = std::min(resize, width);
resize = std::min(resize, height);

// 5. 计算并且赋值
xl = center_x - resize / 2 + 1;
xl = (xl < 0) ? 0 : xl;
yt = center_y - resize / 2 + 1;
yt = (yt < 0) ? 0 : yt;
xr = xl + resize - 1;
yb = yt + resize - 1;
if (xr >= cols - 1) {
    xl -= xr - cols + 1;
    xr = cols - 1;
}

if (yb >= rows - 1) {
    yt -= yb - rows + 1;
    yb = rows - 1;
}

crop_box->x = xl;
crop_box->y = yt;
crop_box->width = xr - xl + 1;
crop_box->height = yb - yt + 1;
}

```

疑问

1. 第5步中的实现逻辑是什么样？？？

“select.h”和“select.cc”选择红绿灯。

首先看Select类的数据结构，其中用到了匈牙利优化器？

```
common::HungarianOptimizer<float> munkres_;
```

初始化

```
bool Select::Init(int rows, int cols) {
    if (rows < 0 || cols < 0) {
        return false;
    }

    munkres_.costs() -> Reserve(rows, cols);

    return true;
}
```

疑问

1. 优化器的cost为什么需要行和列数据？

计算高斯分数？？

```
double Select::Calc2dGaussianScore(base::Point2DI p1,
base::Point2DI p2,
                                    float sigma1, float
sigma2) {
    return std::exp(-0.5 * (static_cast<float>((p1.x - p2.x) *
(p1.x - p2.x)) /
                           (sigma1 * sigma1) +
                           (static_cast<float>((p1.y - p2.y) *
(p1.y - p2.y)) /
                           (sigma2 * sigma2))));
}
```

选择红绿灯。

```
void Select::SelectTrafficLights(
    const std::vector<base::TrafficLightPtr> &refined_bboxes,
    std::vector<base::TrafficLightPtr> *hdmap_bboxes) {
    std::vector<std::pair<size_t, size_t> > assignments;
```

```

munkres_.costs()->Resize(hdmap_bboxes->size(),
refined_bboxes.size());

for (size_t row = 0; row < hdmap_bboxes->size(); ++row) {
    auto center_hd = (*hdmap_bboxes)[row]-
>region.detection_roi.Center();
    if ((*hdmap_bboxes)[row]->region.outside_image) {
        AINFO << "projection_roi outside image, set score to
0.";
        for (size_t col = 0; col < refined_bboxes.size(); ++col) {
            (*munkres_.costs())(row, col) = 0.0;
        }
        continue;
    }
    for (size_t col = 0; col < refined_bboxes.size(); ++col)
{
        float gaussian_score = 100.0f;
        auto center_refine = refined_bboxes[col]-
>region.detection_roi.Center();
        // use gaussian score as metrics of distance and width
        double distance_score = Calc2dGaussianScore(
            center_hd, center_refine, gaussian_score,
gaussian_score);

        double max_score = 0.9;
        auto detect_score = refined_bboxes[col]-
>region.detect_score;
        double detection_score =
            detect_score > max_score ? max_score :
detect_score;

        double distance_weight = 0.7;
        double detection_weight = 1 - distance_weight;
        (*munkres_.costs())(row, col) =
            static_cast<float>(detection_weight *
detection_score +
            distance_weight *
distance_score);
        const auto &crop_roi = (*hdmap_bboxes)[row]-
>region.crop_roi;
        const auto &detection_roi = refined_bboxes[col]-
>region.detection_roi;

```

```

// 1. crop roi在detection roi之外
if ((detection_roi & crop_roi) != detection_roi) {
    (*munkres_.costs())(row, col) = 0.0;
}
AINFO << "score " << (*munkres_.costs())(row, col);
}

munkres_.Maximize(&assignments);

for (size_t i = 0; i < hdmap_bboxes->size(); ++i) {
    (*hdmap_bboxes)[i]->region.is_selected = false;
    (*hdmap_bboxes)[i]->region.is_detected = false;
}

// 2. 把结果放入hdmap_bbox_region
for (size_t i = 0; i < assignments.size(); ++i) {
    if (static_cast<size_t>(assignments[i].first) >=
hdmap_bboxes->size() ||
        static_cast<size_t>(
            assignments[i].second) >= refined_bboxes.size() ||
        (*hdmap_bboxes)[assignments[i].first]-
>region.is_selected ||
            refined_bboxes[assignments[i].second]-
>region.is_selected)) {
        } else {
            auto &refined_bbox_region =
refined_bboxes[assignments[i].second]->region;
            auto &hdmap_bbox_region = (*hdmap_bboxes)
[assignments[i].first]->region;
            refined_bbox_region.is_selected = true;
            hdmap_bbox_region.is_selected = true;

            const auto &crop_roi = hdmap_bbox_region.crop_roi;
            const auto &detection_roi =
refined_bbox_region.detection_roi;
            bool outside_crop_roi = ((crop_roi & detection_roi) !=
detection_roi);
            if (hdmap_bbox_region.outside_image ||
outside_crop_roi) {
                hdmap_bbox_region.is_detected = false;
            } else {
                hdmap_bbox_region.detection_roi =

```

```
    refined_bbox_region.detection_roi;
        hdmap_bbox_region.detect_class_id =
refined_bbox_region.detect_class_id;
            hdmap_bbox_region.detect_score =
refined_bbox_region.detect_score;
                hdmap_bbox_region.is_detected =
refined_bbox_region.is_detected;
                    hdmap_bbox_region.is_selected =
refined_bbox_region.is_selected;
                }
            }
        }
    }
```

疑问

功能如何实现的？有什么作用？

接下来我们看”detection.h”和”detection.cc”。首先看一下”TrafficLightDetection”类的参数。

```
traffic_light::detection::DetectionParam detection_param_;
DataProvider::ImageOptions data_provider_image_option_;
std::shared_ptr<inference::Inference> rt_net_ = nullptr;
std::shared_ptr<base::Image8U> image_ = nullptr;
std::shared_ptr<base::Blob<float>> param_blob_;
std::shared_ptr<base::Blob<float>> mean_buffer_;
std::shared_ptr<IGetBox> crop_;
std::vector<base::TrafficLightPtr> detected_bboxes_;
std::vector<base::TrafficLightPtr> selected_bboxes_;
std::vector<std::string> net_inputs_;
std::vector<std::string> net_outputs_;
Select select_;
int max_batch_size_ = 4;
int param_blob_length_ = 6;
float mean_[3];
std::vector<base::RectI> crop_box_list_;
std::vector<float> resize_scale_list_;
int gpu_id_ = 0;
```

Init初始化。

```
bool TrafficLightDetection::Init(
    const camera::TrafficLightDetectorInitOptions &options) {
    std::string proto_path = GetAbsolutePath(options.root_dir,
options.conf_file);
    // 1. 获取检测参数
    if (!cyber::common::GetProtoFromFile(proto_path,
&detection_param_)) {
        AINFO << "load proto param failed, root dir: " <<
options.root_dir;
        return false;
    }

    std::string param_str;

    google::protobuf::TextFormat::PrintToString(detection_param_,
&param_str);
    AINFO << "TL detection param: " << param_str;

    std::string model_root =
        GetAbsolutePath(options.root_dir,
detection_param_.model_name());
    AINFO << "model_root " << model_root;

    std::string proto_file =
        GetAbsolutePath(model_root,
detection_param_.proto_file());
    AINFO << "proto_file " << proto_file;

    std::string weight_file =
        GetAbsolutePath(model_root,
detection_param_.weight_file());
    AINFO << "weight_file " << weight_file;

    if (detection_param_.is_bgr()) {
        data_provider_image_option_.target_color =
base::Color::BGR;
        mean_[0] = detection_param_.mean_b();
        mean_[1] = detection_param_.mean_g();
        mean_[2] = detection_param_.mean_r();
    } else {
        data_provider_image_option_.target_color =
base::Color::RGB;
        mean_[0] = detection_param_.mean_r();
```

```

        mean_[1] = detection_param_.mean_g();
        mean_[2] = detection_param_.mean_b();
    }

    net_inputs_.push_back(detection_param_.input_blob_name());
    net_inputs_.push_back(detection_param_.im_param_blob_name());
    net_outputs_.push_back(detection_param_.output_blob_name());

    AINFO << "net input blobs: "
        << std::accumulate(net_inputs_.begin(),
    net_inputs_.end(),
                    std::string(""),
                    [] (std::string &sum, const
    std::string &s) {
                    return sum + "\n" + s;
                });
    AINFO << "net output blobs: "
        << std::accumulate(net_outputs_.begin(),
    net_outputs_.end(),
                    std::string(""),
                    [] (std::string &sum, const
    std::string &s) {
                    return sum + "\n" + s;
                });
}

const auto &model_type = detection_param_.model_type();
AINFO << "model_type: " << model_type;

rt_net_.reset(inference::CreateInferenceByName(model_type,
proto_file,
weight_file,
net_outputs_,
net_inputs_,
model_root));

AINFO << "rt_net_ create succeed";
rt_net_->set_gpu_id(options.gpu_id);
AINFO << "set gpu id " << options.gpu_id;
gpu_id_ = options.gpu_id;

int resize_height = detection_param_.min_crop_size();

```

```

int resize_width = detection_param_.min_crop_size();
max_batch_size_ = detection_param_.max_batch_size();
param_blob_length_ = 6;

CHECK_GT(resize_height, 0);
CHECK_GT(resize_width, 0);
CHECK_GT(max_batch_size_, 0);

std::vector<int> shape_input = {max_batch_size_,
resize_height, resize_width,
3};

std::vector<int> shape_param = {max_batch_size_, 1,
param_blob_length_, 1};

std::map<std::string, std::vector<int>> input_reshape;
input_reshape.insert(
    (std::pair<std::string, std::vector<int>>
(net_inputs_[0], shape_input)));
input_reshape.insert(
    (std::pair<std::string, std::vector<int>>
(net_inputs_[1], shape_param)));

if (!rt_net_->Init(input_reshape)) {
    AINFO << "net init fail.";
    return false;
}
AINFO << "net init success.";

mean_buffer_.reset(new base::Blob<float>(1, resize_height,
resize_height, 3));

param_blob_ = rt_net_->get_blob(net_inputs_[1]);
float *param_data = param_blob_->mutable_cpu_data();
for (int i = 0; i < max_batch_size_; ++i) {
    auto offset = i * param_blob_length_;
    param_data[offset + 0] = static_cast<float>
(resize_width);
    param_data[offset + 1] = static_cast<float>
(resize_height);
    param_data[offset + 2] = 1;
    param_data[offset + 3] = 1;
    param_data[offset + 4] = 0;
    param_data[offset + 5] = 0;
}

```

```

    }

switch (detection_param_.crop_method()) {
default:
case 0:
    crop_.reset(new CropBox(detection_param_.crop_scale(),
detection_param_.min_crop_size()));
    break;
case 1:
    crop_.reset(new CropBoxWholeImage());
    break;
}

select_.Init(resize_width, resize_height);
image_.reset(
    new base::Image8U(resize_height, resize_width,
base::Color::BGR));
return true;
}

```

tracker 追踪

追踪功能在”SemanticReviser”类中实现的。先看下”SemanticReviser”类中的参数。

```

traffic_light::tracker::SemanticReviseParam
semantic_param_;
// 1. 修订时间
float revise_time_s_ = 1.5f;
// 2. 闪烁阈值
float blink_threshold_s_ = 0.4f;
// 3. 不闪烁阈值
float non_blink_threshold_s_ = 0.8f;
// 4. 滞后阈值
int hysteretic_threshold_ = 1;
// 5. 历史语义
std::vector<SemanticTable> history_semantic_;

```

接着看下”SemanticTable”结构，后面会用到。

```

struct SemanticTable {
    double time_stamp = 0.0;
    double last_bright_time_stamp = 0.0;
    double last_dark_time_stamp = 0.0;
    bool blink = false;
    std::string semantic;
    std::vector<int> light_ids;
    base::TLColor color;
    HystereticWindow hysteretic_window;
};

```

Init初始化

从配置文件中读取参数，其中“non_blink_threshold_s_”是“blink_threshold_s_”参数的2倍。

```

bool SemanticReviser::Init(const
TrafficLightTrackerInitOptions &options) {
    std::string proto_path =
        cyber::common::GetAbsolutePath(options.root_dir,
options.conf_file);
    if (!cyber::common::GetProtoFromFile(proto_path,
&semantic_param_)) {
        AERROR << "load proto param failed, root dir: " <<
options.root_dir;
        return false;
    }

    int non_blink_coef = 2;
    revise_time_s_ = semantic_param_.revise_time_second();
    blink_threshold_s_ =
semantic_param_.blink_threshold_second();
    hysteretic_threshold_ =
semantic_param_.hysteretic_threshold_count();
    non_blink_threshold_s_ =
        blink_threshold_s_ * static_cast<float>
(non_blink_coef);

    return true;
}

```

Track 追踪

```
bool SemanticReviser::Track(const TrafficLightTrackerOptions
&options,
                               CameraFrame *frame) {
    double time_stamp = frame->timestamp;
    std::vector<base::TrafficLightPtr> &lights_ref = frame-
>traffic_lights;
    std::vector<SemanticTable> semantic_table;

    // 1. 如果没有红绿灯，则退出修订
    if (lights_ref.empty()) {
        history_semantic_.clear();
        ADEBUG << "no lights to revise, return";
        return true;
    }

    // 2. 遍历红绿灯，找到发生变化的表
    for (size_t i = 0; i < lights_ref.size(); i++) {
        base::TrafficLightPtr light = lights_ref.at(i);
        int cur_semantic = light->semantic;

        SemanticTable tmp;
        std::stringstream ss;

        if (cur_semantic > 0) {
            ss << "Semantic_" << cur_semantic;
        } else {
            ss << "No_semantic_light_" << light->id;
        }

        tmp.semantic = ss.str();
        tmp.light_ids.push_back(static_cast<int>(i));
        tmp.color = light->status.color;
        tmp.time_stamp = time_stamp;
        tmp.blink = false;
        auto iter =
            std::find_if(std::begin(semantic_table),
std::end(semantic_table),
                         boost::bind(compare, _1, tmp));

        if (iter != semantic_table.end()) {
```

```

        iter->light_ids.push_back(static_cast<int>(i));
    } else {
        semantic_table.push_back(tmp);
    }
}

// 3. 更新红绿灯序列
for (size_t i = 0; i < semantic_table.size(); ++i) {
    SemanticTable cur_semantic_table = semantic_table.at(i);
    ReviseByTimeSeries(time_stamp, cur_semantic_table,
&lights_ref);
}

return true;
}

```

根据时间序列更新红绿灯的状态。

```

void SemanticReviser::ReviseByTimeSeries(
    double time_stamp, SemanticTable semantic_table,
    std::vector<base::TrafficLightPtr> *lights) {

    std::vector<base::TrafficLightPtr> &lights_ref = *lights;
    base::TLColor cur_color = ReviseBySemantic(semantic_table,
lights);
    base::TLColor pre_color = base::TLColor::TL_UNKNOWN_COLOR;
    semantic_table.color = cur_color;
    semantic_table.time_stamp = time_stamp;
    ADEBUG << "revise same semantic lights";
    ReviseLights(lights, semantic_table.light_ids, cur_color);

    std::vector<SemanticTable>::iterator iter =
        std::find_if(std::begin(history_semantic_),
std::end(history_semantic_),
            boost::bind(compare, _1, semantic_table));

    if (iter != history_semantic_.end()) {
        pre_color = iter->color;
        if (time_stamp - iter->time_stamp < revise_time_s_) {
            ADEBUG << "revise by time series";
            switch (cur_color) {
                case base::TLColor::TL_YELLOW:
                    if (iter->color == base::TLColor::TL_RED) {

```

```

        ReviseLights(lights, semantic_table.light_ids,
iter->color);
        iter->time_stamp = time_stamp;
        iter->hystertic_window.hysteretic_count = 0;
    } else {
        UpdateHistoryAndLights(semantic_table, lights,
&iter);
        ADEBUG << "High confidence color " <<
s_color_strs[cur_color];
    }
    break;
case base::TLColor::TL_RED:
case base::TLColor::TL_GREEN:
    UpdateHistoryAndLights(semantic_table, lights,
&iter);
    if (time_stamp - iter->last_bright_time_stamp >
blink_threshold_s_ &&
        iter->last_dark_time_stamp > iter-
>last_bright_time_stamp) {
        iter->blink = true;
    }
    iter->last_bright_time_stamp = time_stamp;
    ADEBUG << "High confidence color " <<
s_color_strs[cur_color];
    break;
case base::TLColor::TL_BLACK:
    iter->last_dark_time_stamp = time_stamp;
    iter->hystertic_window.hysteretic_count = 0;
    if (iter->color == base::TLColor::TL_UNKNOWN_COLOR
||

        iter->color == base::TLColor::TL_BLACK) {
        iter->time_stamp = time_stamp;
        UpdateHistoryAndLights(semantic_table, lights,
&iter);
    } else {
        ReviseLights(lights, semantic_table.light_ids,
iter->color);
    }
    break;
case base::TLColor::TL_UNKNOWN_COLOR:
default:
    ReviseLights(lights, semantic_table.light_ids,
iter->color);

```

```

        break;
    }
} else {
    iter->time_stamp = time_stamp;
    iter->color = cur_color;
}

// set blink status
if (pre_color != iter->color ||
    fabs(iter->last_dark_time_stamp - iter-
>last_bright_time_stamp) >
        non_blink_threshold_s_) {
    iter->blink = false;
}

for (auto index : semantic_table.light_ids) {
    lights_ref[index]->status.blink =
        (iter->blink && iter->color ==
base::TLColor::TL_GREEN);
}

} else {
    semantic_table.last_dark_time_stamp =
semantic_table.time_stamp;
    semantic_table.last_bright_time_stamp =
semantic_table.time_stamp;
    history_semantic_.push_back(semantic_table);
}
}
}

```

修改红绿灯为目标颜色。

```

void
SemanticReviser::ReviseLights(std::vector<base::TrafficLightP
tr> *lights,
                               const std::vector<int>
&light_ids,
                               base::TLColor dst_color) {
    // 1. 遍历红绿灯数组，并且修改颜色为dst_color
    for (auto index : light_ids) {
        lights->at(index)->status.color = dst_color;
    }
}

```

通过语义判别红绿灯颜色。

```
base::TLColor SemanticReviser::ReviseBySemantic(
    SemanticTable semantic_table,
    std::vector<base::TrafficLightPtr> *lights) {
    std::vector<int> vote(static_cast<int>
(base::TLColor::TL_TOTAL_COLOR_NUM), 0);
    std::vector<base::TrafficLightPtr> &lights_ref = *lights;
    base::TLColor max_color = base::TLColor::TL_UNKNOWN_COLOR;

    // 1. 遍历红绿灯，把出现的颜色放入投票桶
    for (size_t i = 0; i < semantic_table.light_ids.size();
++i) {
        int index = semantic_table.light_ids.at(i);
        base::TrafficLightPtr light = lights_ref[index];
        auto color = light->status.color;
        vote.at(static_cast<int>(color))++;
    }

    // 2. 如果红绿灯红、黄、绿的颜色为0
    if ((vote.at(static_cast<size_t>(base::TLColor::TL_RED)) ==
0) &&
        (vote.at(static_cast<size_t>(base::TLColor::TL_GREEN)) ==
0) &&
        (vote.at(static_cast<size_t>(base::TLColor::TL_YELLOW)) ==
0)) {
        // 3. 红绿灯黑色的次数大于0
        if (vote.at(static_cast<size_t>(base::TLColor::TL_BLACK)) >
0) {
            return base::TLColor::TL_BLACK;
        } else {
            return base::TLColor::TL_UNKNOWN_COLOR;
        }
    }

    vote.at(static_cast<size_t>(base::TLColor::TL_BLACK)) = 0;
    vote.at(static_cast<size_t>
(base::TLColor::TL_UNKNOWN_COLOR)) = 0;
    // 4. 找到最多出现的次数
    auto biggest = std::max_element(std::begin(vote),
std::end(vote));

    int max_color_num = *biggest;
```

```

    max_color = base::TLColor(std::distance(std::begin(vote),
biggest));

    vote.erase(biggest);
    // 5. 找到出现第二多的次数
    auto second_biggest = std::max_element(std::begin(vote),
std::end(vote));

    // 6. 如果第一多和第二多的相等，则返回UNKNOWN
    if (max_color_num == *second_biggest) {
        return base::TLColor::TL_UNKNOWN_COLOR;
    } else {
        // 7. 返回出现最多的颜色
        return max_color;
    }
}

```

更新历史记录。

```

void SemanticReviser::UpdateHistoryAndLights(
    const SemanticTable &cur,
    std::vector<base::TrafficLightPtr> *lights,
    std::vector<SemanticTable>::iterator *history) {
    (*history)->time_stamp = cur.time_stamp;
    if ((*history)->color == base::TLColor::TL_BLACK) {
        if ((*history)->hystertic_window.hysteretic_color ==
cur.color) {
            (*history)->hystertic_window.hysteretic_count++;
        } else {
            (*history)->hystertic_window.hysteretic_color =
cur.color;
            (*history)->hystertic_window.hysteretic_count = 1;
        }
    }

    if ((*history)->hystertic_window.hysteretic_count >
hysteretic_threshold_) {
        (*history)->color = cur.color;
        (*history)->hystertic_window.hysteretic_count = 0;
        ADEBUG << "Black lights hysteretic change to " <<
s_color_strs[cur.color];
    } else {
        ReviseLights(lights, cur.light_ids, (*history)->color);
    }
}

```

```

    } else {
        (*history)->color = cur.color;
    }
}

```

obstacle障碍物

首先obstacle的目录结构如下

```
.
├── detector      // 检测
└── postprocessor // 后处理? 做了什么处理???
├── tracker       // 追踪
└── transformer   // 坐标转换
```

下面我们分别分析下每个模块具体的实现。

detector检测

障碍物检测用到的是YOLO深度学习模型，下面我们看下YoloObstacleDetector类的实现。

初始化Yolo网络

```

bool YoloObstacleDetector::Init(const
ObstacleDetectorInitOptions &options) {
    gpu_id_ = options.gpu_id;
    BASE_CUDA_CHECK(cudaSetDevice(gpu_id_));
    BASE_CUDA_CHECK(cudaStreamCreate(&stream_));

    base_camera_model_ = options.base_camera_model;
    ACHECK(base_camera_model_ != nullptr) << "base_camera_model
is nullptr!";
    std::string config_path =
        GetAbsolutePath(options.root_dir, options.conf_file);
    if (!cyber::common::GetProtoFromFile(config_path,
    &yolo_param_)) {
        AERROR << "read proto_config fail";
        return false;
    }
}

```

```

    }

const auto &model_param = yolo_param_.model_param();
std::string model_root =
    GetAbsolutePath(options.root_dir,
model_param.model_name());
std::string anchors_file =
    GetAbsolutePath(model_root,
model_param.anchors_file());
std::string types_file =
    GetAbsolutePath(model_root, model_param.types_file());
std::string expand_file =
    GetAbsolutePath(model_root, model_param.expand_file());
LoadInputShape(model_param);
LoadParam(yolo_param_);
min_dims_.min_2d_height /= static_cast<float>(height_);

if (!LoadAnchors(anchors_file, &anchors_)) {
    return false;
}
if (!LoadTypes(types_file, &types_)) {
    return false;
}
if (!LoadExpand(expand_file, &expands_)) {
    return false;
}
ACHECK(expands_.size() == types_.size());
if (!InitNet(yolo_param_, model_root)) {
    return false;
}
InitYoloBlob(yolo_param_.net_param());
if (!InitFeatureExtractor(model_root)) {
    return false;
}
return true;
}

```

特征提取采用的是”TrackingFeatureExtractor”

```

bool YoloObstacleDetector::InitFeatureExtractor(const
std::string &root_dir) {
    FeatureExtractorInitOptions feat_options;
    feat_options.conf_file =
yolo_param_.model_param().feature_file();

```

```

feat_options.root_dir = root_dir;
feat_options.gpu_id = gpu_id_;
auto feat_blob_name = yolo_param_.net_param().feat_blob();
feat_options.feat_blob = inference_-
>get_blob(feat_blob_name);
feat_options.input_height = height_;
feat_options.input_width = width_;

feature_extractor_.reset(BaseFeatureExtractorRegisterer::GetInstanceByName(
    "TrackingFeatureExtractor"));
if (!feature_extractor_->Init(feat_options)) {
    return false;
}
return true;
}

```

下面我们再看下如何用YOLO做目标检测。

```

bool YoloObstacleDetector::Detect(const
ObstacleDetectorOptions &options,
                                    CameraFrame *frame) {

Timer timer;
// 1. 设置GPU设备
if (cudaSetDevice(gpu_id_) != cudaSuccess) {
    return false;
}

auto input_blob = inference_-
>get_blob(yolo_param_.net_param().input_blob());

DataProvider::ImageOptions image_options;
image_options.target_color = base::Color::BGR;
image_options.crop_roi = base::RectI(
    0, offset_y_, static_cast<int>(base_camera_model_-
>get_width()),
    static_cast<int>(base_camera_model_->get_height()) -
offset_y_);
image_options.do_crop = true;
frame->data_provider->GetImage(image_options,
image_.get());

```

```

inference::ResizeGPU(*image_, input_blob, frame-
>data_provider->src_width(),
                     0);

// 2. 检测
inference_->Infer();
get_objects_gpu(yolo_blobs_, stream_, types_, nms_,
yolo_param_.model_param(),
               light_vis_conf_threshold_,
light_swt_conf_threshold_,
               overlapped_.get(), idx_sm_.get(), &(frame-
>detected_objects));

filter_bbox(min_dims_, &(frame->detected_objects));
FeatureExtractorOptions feat_options;
feat_options.normalized = true;

feature_extractor_->Extract(feat_options, frame);

recover_bbox(frame->data_provider->src_width(),
             frame->data_provider->src_height() -
offset_y_, offset_y_,
             &frame->detected_objects);

// 3. 后处理
int left_boundary =
    static_cast<int>(border_ratio_ * static_cast<float>
(image_->cols()));
int right_boundary = static_cast<int>((1.0f -
border_ratio_) *
                                         static_cast<float>
(image_->cols()));
for (auto &obj : frame->detected_objects) {
    // recover alpha
    obj->camera_supplement.alpha /= ori_cycle_;
    // get area_id from visible_ratios
    if (yolo_param_.model_param().num_areas() == 0) {
        obj->camera_supplement.area_id =
            get_area_id(obj->camera_supplement.visible_ratios);
    }
    // clear cut off ratios
}

```

```

        auto &box = obj->camera_supplement.box;
        if (box.xmin >= left_boundary) {
            obj->camera_supplement.cut_off_ratios[2] = 0;
        }
        if (box.xmax <= right_boundary) {
            obj->camera_supplement.cut_off_ratios[3] = 0;
        }
    }

    return true;
}

```

其中推理过程有GPU实现也有CPU实现。通过”get_objects_gpu”获取物体的信息。

tracker追踪

tracker追踪在”OMTObstacleTracker”类中实现。

初始化Init

```

bool OMTObstacleTracker::Init(const
ObstacleTrackerInitOptions &options) {
    std::string omt_config = GetAbsolutePath(options.root_dir,
options.conf_file);
    // 1. 加载omt配置
    if (!cyber::common::GetProtoFromFile(omt_config,
&omt_param_)) {
        AERROR << "Read config failed: " << omt_config;
        return false;
    }

    AINFO << "load omt parameters from " << omt_config
        << "\nimg_capability: " <<
omt_param_.img_capability()
        << "\nlost_age: " << omt_param_.lost_age()
        << "\nreserve_age: " << omt_param_.reserve_age()
        << "\nborder: " << omt_param_.border()
        << "\ntarget_thresh: " << omt_param_.target_thresh()
        << "\ncorrect_type: " << omt_param_.correct_type();
}

```

```

// 2. 初始化参数
track_id_ = 0;
frame_num_ = 0;
frame_list_.Init(omt_param_.img_capability());
gpu_id_ = options.gpu_id;
similar_map_.Init(omt_param_.img_capability(), gpu_id_);
similar_.reset(new GPUSimilar);
width_ = options.image_width;
height_ = options.image_height;
reference_.Init(omt_param_.reference(), width_, height_);
std::string type_change_cost =
    GetAbsolutePath(options.root_dir,
omt_param_.type_change_cost());
std::ifstream fin(type_change_cost);
ACHECK(fin.is_open());
kTypeAssociatedCost_.clear();
int n_type = static_cast<int>
(base::ObjectSubType::MAX_OBJECT_TYPE);
for (int i = 0; i < n_type; ++i) {
    kTypeAssociatedCost_.emplace_back(std::vector<float>
(n_type, 0));
    for (int j = 0; j < n_type; ++j) {
        fin >> kTypeAssociatedCost_[i][j];
    }
}
targets_.clear();
used_.clear();

// Init object template
object_template_manager_ =
ObjectTemplateManager::Instance();
return true;
}

```

计算运动相似度

```

float OMTObstacleTracker::ScoreMotion(const Target &target,
                                         TrackObjectPtr
track_obj) {
    Eigen::Vector4d x = target.image_center.get_state();
    float target_centerx = static_cast<float>(x[0]);
    float target_centery = static_cast<float>(x[1]);
    base::Point2DF center = track_obj->projected_box.Center();

```

```

base::RectF rect(track_obj->projected_box);
float s = gaussian(center.x, target_centerx, rect.width) *
          gaussian(center.y, target_centery, rect.height);
return s;
}

```

计算形状相似度

```

float OMTObstacleTracker::ScoreShape(const Target &target,
                                      TrackObjectPtr
track_obj) {
    Eigen::Vector2d shape = target.image_wh.get_state();
    base::RectF rect(track_obj->projected_box);
    float s = static_cast<float>((shape[1] - rect.height) *
                                   (shape[0] - rect.width) /
                                   (shape[1] * shape[0]));
    return -std::abs(s);
}

```

计算显示相似度

```

float OMTObstacleTracker::ScoreAppearance(const Target
&target,
                                         TrackObjectPtr
track_obj) {
    float energy = 0.0f;
    int count = 0;
    auto sensor_name = track_obj->indicator.sensor_name;
    for (int i = target.Size() - 1; i >= 0; --i) {
        if (target[i]->indicator.sensor_name != sensor_name) {
            continue;
        }
        PatchIndicator p1 = target[i]->indicator;
        PatchIndicator p2 = track_obj->indicator;

        energy += similar_map_.sim(p1, p2);
        count += 1;
    }

    return energy / (0.1f + static_cast<float>(count) * 0.9f);
}

```

计算重叠相似度

```
float OMTObstacleTracker::ScoreOverlap(const Target &target,
                                         TrackObjectPtr
track_obj) {
    Eigen::Vector4d center = target.image_center.get_state();
    Eigen::VectorXd wh = target.image_wh.get_state();
    base::BBox2DF box_target;
    box_target.xmin = static_cast<float>(center[0] - wh[0] * 0.5);
    box_target.xmax = static_cast<float>(center[0] + wh[0] * 0.5);
    box_target.ymin = static_cast<float>(center[1] - wh[1] * 0.5);
    box_target.ymax = static_cast<float>(center[1] + wh[1] * 0.5);

    auto box_obj = track_obj->projected_box;

    float iou = common::CalculateIOUBBox(box_target, box_obj);
    return iou;
}
```

找到最匹配的障碍物。

```
void OMTObstacleTracker::GenerateHypothesis(const
TrackObjectPtrs &objects) {
    std::vector<Hypothesis> score_list;
    Hypothesis hypo;
    for (size_t i = 0; i < targets_.size(); ++i) {
        ADEBUG << "Target " << targets_[i].id;
        for (size_t j = 0; j < objects.size(); ++j) {
            hypo.target = static_cast<int>(i);
            hypo.object = static_cast<int>(j);
            float sa = ScoreAppearance(targets_[i], objects[j]);
            float sm = ScoreMotion(targets_[i], objects[j]);
            float ss = ScoreShape(targets_[i], objects[j]);
            float so = ScoreOverlap(targets_[i], objects[j]);
            if (sa == 0) {
                hypo.score = omt_param_.weight_diff_camera().motion()
* sm +
                    omt_param_.weight_diff_camera().shape()
```

```

* ss +
omt_param_.weight_diff_camera().overlap() * so;
} else {
    hypo.score =
(omt_param_.weight_same_camera().appearance() * sa +
omt_param_.weight_same_camera().motion() * sm +
omt_param_.weight_same_camera().shape())
* ss +
omt_param_.weight_same_camera().overlap() * so;
}
int change_from_type = static_cast<int>(targets_[i].type);
int change_to_type = static_cast<int>(objects[j]->object->sub_type);
hypo.score += -kTypeAssociatedCost_[change_from_type]
[change_to_type];
ADEBUG << "Detection " << objects[j]->indicator.frame_id << "(" << j
<< ") sa:" << sa << " sm: " << sm << " ss: " <<
ss << " so: " << so
<< " score: " << hypo.score;

// 95.44% area is range [mu - sigma*2, mu + sigma*2]
// don't match if motion is beyond the range
if (sm < 0.045 || hypo.score <
omt_param_.target_thresh()) {
    continue;
}
score_list.push_back(hypo);
}
}

sort(score_list.begin(), score_list.end(),
std::greater<Hypothesis>());
std::vector<bool> used_target(targets_.size(), false);
for (auto &pair : score_list) {
    if (used_target[pair.target] || used_[pair.object]) {
        continue;
    }
    Target &target = targets_[pair.target];

```

```

        auto det_obj = objects[pair.object];
        target.Add(det_obj);
        used_[pair.object] = true;
        used_target[pair.target] = true;
        AINFO << "Target " << target.id << " match " << det_obj-
>indicator.frame_id
            << " (" << pair.object << ")"
            << "at " << pair.score << " size: " <<
target.Size();
    }
}

```

合并重复的障碍物

```

bool OMTObstacleTracker::CombineDuplicateTargets() {
    std::vector<Hypothesis> score_list;
    Hypothesis hypo;
    // 1. targets_ 为数组， 数组元素为Target类
    for (size_t i = 0; i < targets_.size(); ++i) {
        if (targets_[i].Size() == 0) {
            continue;
        }
        for (size_t j = i + 1; j < targets_.size(); ++j) {
            if (targets_[j].Size() == 0) {
                continue;
            }
            int count = 0;
            float score = 0.0f;
            int index1 = 0;
            int index2 = 0;
            while (index1 < targets_[i].Size() && index2 <
targets_[j].Size()) {
                auto p1 = targets_[i][index1];
                auto p2 = targets_[j][index2];
                if (std::abs(p1->timestamp - p2->timestamp) <
omt_param_.same_ts_eps()) {
                    if (p1->indicator.sensor_name != p2-
>indicator.sensor_name) {
                        auto box1 = p1->projected_box;
                        auto box2 = p2->projected_box;
                        score += common::CalculateIOUBBox(box1, box2);
                        base::RectF rect1(box1);
                        base::RectF rect2(box2);

```

```

        score -= std::abs((rect1.width - rect2.width) *
                            (rect1.height - rect2.height) /
                            (rect1.width * rect1.height));
        count += 1;
    }
    ++index1;
    ++index2;
} else {
    if (p1->timestamp > p2->timestamp) {
        ++index2;
    } else {
        ++index1;
    }
}
ADEBUG << "Overlap: (" << targets_[i].id << "," <<
targets_[j].id
            << ") score " << score << " count " << count;
hypo.target = static_cast<int>(i);
hypo.object = static_cast<int>(j);
hypo.score = (count > 0) ? score / static_cast<float>
(count) : 0;
if (hypo.score <
omt_param_.target_combine_iou_threshold()) {
    continue;
}
score_list.push_back(hypo);
}
}
sort(score_list.begin(), score_list.end(),
std::greater<Hypothesis>());
std::vector<bool> used_target(targets_.size(), false);
for (auto &pair : score_list) {
    if (used_target[pair.target] || used_target[pair.object])
{
        continue;
    }
    int index1 = pair.target;
    int index2 = pair.object;
    if (targets_[pair.target].id > targets_[pair.object].id)
{
        index1 = pair.object;
        index2 = pair.target;
    }
}
```

```

    }
    Target &target_save = targets_[index1];
    Target &target_del = targets_[index2];
    for (int i = 0; i < target_del.Size(); i++) {
        // no need to change track_id of all objects in
        target_del
        target_save.Add(target_del[i]);
    }
    std::sort(
        target_save.tracked_objects.begin(),
        target_save.tracked_objects.end(),
        [](const TrackObjectPtr object1, const TrackObjectPtr
            object2) -> bool {
            return object1->indicator.frame_id < object2-
            >indicator.frame_id;
        });
    target_save.latest_object = target_save.get_object(-1);
    base::ObjectPtr object = target_del.latest_object-
    >object;
    target_del.Clear();
    AINFO << "Target " << target_del.id << " is merged into
    Target "
        << target_save.id << " with iou " << pair.score;
    used_target[pair.object] = true;
    used_target[pair.target] = true;
}
return true;
}

```

预测

```

bool OMTObstacleTracker::Predict(const ObstacleTrackerOptions
&options,
                                CameraFrame *frame) {
    for (auto &target : targets_) {
        target.Predict(frame);
        auto obj = target.latest_object;
        frame->proposed_objects.push_back(obj->object);
    }
    return true;
}

```

创建新目标

```
int OMTObstacleTracker::CreateNewTarget(const TrackObjectPtrs &objects) {
    const TemplateMap &kMinTemplateHWL =
        object_template_manager_->MinTemplateHWL();
    std::vector<base::RectF> target_rects;
    for (auto &&target : targets_) {
        if (!target.isTracked() || target.isLost()) {
            continue;
        }
        base::RectF target_rect(target[-1]->object->camera_supplement.box);
        target_rects.push_back(target_rect);
    }
    int created_count = 0;
    for (size_t i = 0; i < objects.size(); ++i) {
        if (!used_[i]) {
            bool is_covered = false;
            const auto &sub_type = objects[i]->object->sub_type;
            base::RectF rect(objects[i]->object->camera_supplement.box);
            auto &min_tmplt = kMinTemplateHWL.at(sub_type);
            if (OutOfValidRegion(rect, width_, height_,
omt_param_.border())) {
                continue;
            }
            for (auto &&target_rect : target_rects) {
                if (IsCovered(rect, target_rect, 0.4f) ||
                    IsCoveredHorizon(rect, target_rect, 0.5f)) {
                    is_covered = true;
                    break;
                }
            }
            if (is_covered) {
                continue;
            }
            if (min_tmplt.empty() // unknown type
                || rect.height > min_tmplt[0] *
omt_param_.min_init_height_ratio()) {
                Target target(omt_param_.target_param());
                target.Add(objects[i]);
                targets_.push_back(target);
            }
        }
    }
}
```

```

        AINFO << "Target " << target.id << " is created by "
            << objects[i]->indicator.frame_id << " ("
            << objects[i]->indicator.patch_id << ")");
        created_count += 1;
    }
}
}

return created_count;
}

```

postprocessor后处理

后处理的入口在”LocationRefinerObstaclePostprocessor”中，下面我们分析下具体实现。
是否在ROI区域

```

bool is_in_roi(const float pt[2], float img_w, float img_h,
float v,
                float h_down) const {
    float x = pt[0];
    float y = pt[1];
    if (y < v) {
        return false;
    } else if (y > (img_h - h_down)) {
        return true;
    }
    float img_w_half = img_w / 2.0f;
    float slope = img_w_half * common::IRec(img_h - h_down -
v);
    float left = img_w_half - slope * (y - v);
    float right = img_w_half + slope * (y - h_down);
    return x > left && x < right;
}

```

LocationRefinerObstaclePostprocessor类，初始化Init

```

bool LocationRefinerObstaclePostprocessor::Init(
    const ObstaclePostprocessorInitOptions &options) {
    std::string postprocessor_config =
        cyber::common::GetAbsolutePath(options.root_dir,

```

```

options.conf_file);
// 1. 读取配置
if (!cyber::common::GetProtoFromFile(postprocessor_config,
&location_refiner_param_)) {
    AERROR << "Read config failed: " << postprocessor_config;
    return false;
}

// 2. 打印配置
AINFO << "Load postprocessor parameters from " <<
postprocessor_config
    << "\nmin_dist_to_camera: "
    << location_refiner_param_.min_dist_to_camera()
    << "\nroi_h2bottom_scale: "
    << location_refiner_param_.roi_h2bottom_scale();
return true;
}

```

处理过程

```

bool LocationRefinerObstaclePostprocessor::Process(
    const ObstaclePostprocessorOptions &options, CameraFrame
*frame) {

    Eigen::Vector4d plane;
    // 1. 校准服务是否有地平面
    if (options.do_refinement_with_calibration_service &&
        !frame->calibration_
>QueryGroundPlaneInCameraFrame(&plane)) {
        AINFO << "No valid ground plane in the service.";
    }

    // 2.
    float query_plane[4] = {
        static_cast<float>(plane(0)), static_cast<float>
(plane(1)),
        static_cast<float>(plane(2)), static_cast<float>
(plane(3)));
    const auto &camera_k_matrix = frame->camera_k_matrix;
    float k_mat[9] = {0};
    for (size_t i = 0; i < 3; i++) {
        size_t i3 = i * 3;

```

```

        for (size_t j = 0; j < 3; j++) {
            k_mat[i3 + j] = camera_k_matrix(i, j);
        }
    }

    // 3. 相机K矩阵
    AINFO << "Camera k matrix input to obstacle postprocessor:
\n"
    << k_mat[0] << ", " << k_mat[1] << ", " << k_mat[2]
<< "\n"
    << k_mat[3] << ", " << k_mat[4] << ", " << k_mat[5]
<< "\n"
    << k_mat[6] << ", " << k_mat[7] << ", " << k_mat[8]
<< "\n";
}

const int width_image = frame->data_provider->src_width();
const int height_image = frame->data_provider-
>src_height();
// 4. 初始化后处理过程
postprocessor_->Init(k_mat, width_image, height_image);
ObjPostProcessorOptions obj_postprocessor_options;

int nr_valid_obj = 0;
// 5. 遍历检测到的障碍物
for (auto &obj : frame->detected_objects) {
    ++nr_valid_obj;
    // 6. 获取物体的中心
    float object_center[3] = {obj-
>camera_supplement.local_center(0),
                           obj-
>camera_supplement.local_center(1),
                           obj-
>camera_supplement.local_center(2)};
    float bbox2d[4] = {
        obj->camera_supplement.box.xmin, obj-
>camera_supplement.box.ymin,
        obj->camera_supplement.box.xmax, obj-
>camera_supplement.box.ymax};

    float bottom_center[2] = {(bbox2d[0] + bbox2d[2]) / 2,
bbox2d[3]};
    float h_down = (static_cast<float>(height_image) -
k_mat[5]) *

```

```

location_refiner_param_.roi_h2bottom_scale();
    // 7. 是否在ROI中
    bool is_in_rule_roi =
        is_in_roi(bottom_center, static_cast<float>
(width_image),
                    static_cast<float>(height_image), k_mat[5],
h_down);
    float dist2camera =
common::ISqr(common::ISqr(object_center[0]) +
common::ISqr(object_center[2]));

    if (dist2camera >
location_refiner_param_.min_dist_to_camera() ||
        !is_in_rule_roi) {
        ADEBUG << "Pass for obstacle postprocessor.";
        continue;
    }

    float dimension_hw1[3] = {obj->size(2), obj->size(1),
obj->size(0)};
    float box_cent_x = (bbox2d[0] + bbox2d[2]) / 2;
    Eigen::Vector3f image_point_low_center(box_cent_x,
bbox2d[3], 1);
    Eigen::Vector3f point_in_camera =
        static_cast<Eigen::Matrix<float, 3, 1, 0, 3, 1>>(
            camera_k_matrix.inverse() *
image_point_low_center);
    float theta_ray =
        static_cast<float>(atan2(point_in_camera.x(),
point_in_camera.z()));
    float rotation_y =
        theta_ray + static_cast<float>(obj-
>camera_supplement.alpha);

    // enforce the ry to be in the range [-pi, pi)
    // 8. 射线的属性? ? ?
    const float PI = common::Constant<float>::PI();
    if (rotation_y < -PI) {
        rotation_y += 2 * PI;
    } else if (rotation_y >= PI) {
        rotation_y -= 2 * PI;
    }
}

```

```

}

// process
memcpy(obj_postprocessor_options.bbox, bbox2d,
sizeof(float) * 4);
obj_postprocessor_options.check_lowerbound = true;
camera::LineSegment2D<float> line_seg(bbox2d[0],
bbox2d[3], bbox2d[2],
bbox2d[3]);
obj_postprocessor_options.line_segs.push_back(line_seg);
memcpy(obj_postprocessor_options.hwl, dimension_hwl,
sizeof(float) * 3);
obj_postprocessor_options.ry = rotation_y;
// refine with calibration service, support ground plane
model currently
// {0.0f, cos(tilt), -sin(tilt), -camera_ground_height}
memcpy(obj_postprocessor_options.plane, query_plane,
sizeof(float) * 4);

// changed to touching-ground center
object_center[1] += dimension_hwl[0] / 2;
// 9. 后处理包括地面
postprocessor_->PostProcessObjWithGround(
    obj_postprocessor_options, object_center,
dimension_hwl, &rotation_y);
object_center[1] -= dimension_hwl[0] / 2;

float z_diff_camera =
    object_center[2] - obj-
>camera_supplement.local_center(2);

// fill back results
obj->camera_supplement.local_center(0) =
object_center[0];
obj->camera_supplement.local_center(1) =
object_center[1];
obj->camera_supplement.local_center(2) =
object_center[2];

obj->center(0) = static_cast<double>(object_center[0]);
obj->center(1) = static_cast<double>(object_center[1]);
obj->center(2) = static_cast<double>(object_center[2]);
obj->center = frame->camera2world_pose * obj->center;

```

```

    AINFO << "diff on camera z: " << z_diff_camera;
    AINFO << "Obj center from postprocessor: " << obj-
>center.transpose();
}
return true;
}

```

对象后处理过程在”ObjPostProcessor”类中，下面我们看下它的具体实现。

设置默认参数。

```

void ObjPostProcessorParams::set_default() {
    max_nr_iter = 5;
    sampling_ratio_low = 0.1f;
    weight_iou = 3.0f;
    learning_r = 0.2f;
    learning_r_decay = 0.9f;
    dist_far = 15.0f;
    shrink_ratio_iou = 0.9f;
    iou_good = 0.5f;
}

```

后处理包括地面

```

bool ObjPostProcessor::PostProcessObjWithGround(
    const ObjPostProcessorOptions &options, float center[3],
    float hwl[3],
    float *ry) {
    memcpy(hwl, options.hwl, sizeof(float) * 3);
    float bbox[4] = {0};
    memcpy(bbox, options.bbox, sizeof(float) * 4);
    *ry = options.ry;

    // 软约束
    bool adjust_soft =
        AdjustCenterWithGround(bbox, hwl, *ry, options.plane,
        center);
    if (center[2] > params_.dist_far) {
        return adjust_soft;
    }
}

```

```

// 硬约束
bool adjust_hard = PostRefineCenterWithGroundBoundary(
    bbox, hwl, *ry, options.plane, options.line_segs,
center,
    options.check_lowerbound);

return adjust_soft || adjust_hard;
}

```

通过地面调整中心？？？

```

bool ObjPostProcessor::AdjustCenterWithGround(const float
*bbox,
                                              const float
*hwl, float ry,
                                              const float
*plane,
                                              float *center)

const {
    float iou_ini = GetProjectionScore(ry, bbox, hwl, center);
    if (iou_ini < params_.iou_good) { // ini pos is not good
enough
        return false;
    }
    const float MIN_COST = hwl[2] * params_.sampling_ratio_low;
    const float EPS_COST_DELTA = 1e-1f;
    const float WEIGHT_IOU = params_.weight_iou;
    const int MAX_ITERATION = params_.max_nr_iter;

    float lr = params_.learning_r;
    float cost_pre = FLT_MAX;
    float cost_delta = 0.0f;
    float center_input[3] = {center[0], center[1], center[2]};
    float center_test[3] = {0};
    float x[3] = {0};
    int iter = 1;
    bool stop = false;

    // std::cout << "start to update the center..." <<
std::endl;
    while (!stop) {
        common::IProjectThroughIntrinsic(k_mat_, center, x);
        x[0] *= common::IRec(x[2]);

```

```

x[1] *= common::IRec(x[2]);
bool in_front =
common::IBackprojectPlaneIntersectionCanonical(
    x, k_mat_, plane, center_test);
if (!in_front) {
    memcpy(center, center_input, sizeof(float) * 3);
    return false;
}
float iou_cur = GetProjectionScore(ry, bbox, hwl,
center);
float iou_test = GetProjectionScore(ry, bbox, hwl,
center_test);
float dist = common::ISqrt(common::ISqr(center[0] -
center_test[0]) +
                           common::ISqr(center[2] -
center_test[2]));
float cost_cur = dist + WEIGHT_IOU * (1.0f - (iou_cur +
iou_test) / 2);
// std::cout << "cost___ " << cost_cur << "@" << iter <<
std::endl;
if (cost_cur >= cost_pre) {
    stop = true;
} else {
    cost_delta = (cost_pre - cost_cur) / cost_pre;
    cost_pre = cost_cur;
    center[0] += (center_test[0] - center[0]) * lr;
    center[2] += (center_test[2] - center[2]) * lr;
    ++iter;
    stop = iter >= MAX_ITERATION || cost_delta <
EPS_COST_DELTA ||
                           cost_pre < MIN_COST;
}
lr *= params_.learning_r_decay;
}
float iou_res = GetProjectionScore(ry, bbox, hwl, center);
if (iou_res < iou_ini * params_.shrink_ratio_iou) {
    memcpy(center, center_input, sizeof(float) * 3);
    return false;
}
return true;
}

```

根据地面边界获取细化的中心？？？

```

bool ObjPostProcessor::PostRefineCenterWithGroundBoundary(
    const float *bbox, const float *hwl, float ry, const
    float *plane,
    const std::vector<LineSegment2D<float>> &line_seg_limits,
    float *center,
    bool check_lowerbound) const {
    bool truncated_on_bottom =
        bbox[3] >= static_cast<float>(height_) -
            (bbox[3] - bbox[1]) *
    params_.sampling_ratio_low;
    if (truncated_on_bottom) {
        return false;
    }

    float iou_before = GetProjectionScore(ry, bbox, hwl,
    center);
    int nr_line_segs = static_cast<int>
    (line_seg_limits.size());
    float depth_pts[4] = {0};
    float pts_c[12] = {0};
    int x_pts[4] = {0};

    GetDepthXPair(bbox, hwl, center, ry, depth_pts, x_pts,
    &pts_c[0]);

    float dx_dz_acc[2] = {0};
    float ratio_x_over_z = center[0] * common::IRec(center[2]);
    for (int i = 0; i < nr_line_segs; ++i) {
        float dx_dz[2] = {0};
        GetDxDzForCenterFromGroundLineSeg(line_seg_limits[i],
    plane, pts_c, k_mat_,
                                width_, height_,
    ratio_x_over_z, dx_dz,
                                check_lowerbound);
        dx_dz_acc[0] += dx_dz[0];
        dx_dz_acc[1] += dx_dz[1];
    }
    center[0] += dx_dz_acc[0];
    center[2] += dx_dz_acc[1];

    float iou_after = GetProjectionScore(ry, bbox, hwl,
    center);
    if (iou_after < iou_before * params_.shrink_ratio_iou) {

```

```

        center[0] -= dxdz_acc[0];
        center[2] -= dxdz_acc[1];
        return false;
    }
    return true;
}

```

获取深度信息？？？

```

int ObjPostProcessor::GetDepthXPair(const float *bbox, const
float *hw1,
                                    const float *center,
float ry,
                                    float *depth_pts, int
*x_pts,
                                    float *pts_c) const {
    int y_min = height_;
    float w_half = hw1[1] / 2;
    float l_half = hw1[2] / 2;
    float x_cor[4] = {l_half, l_half, -l_half, -l_half};
    float z_cor[4] = {w_half, -w_half, -w_half, w_half};
    float pts[12] = {x_cor[0], 0.0f, z_cor[0], x_cor[1], 0.0f,
z_cor[1],
                    x_cor[2], 0.0f, z_cor[2], x_cor[3], 0.0f,
z_cor[3]};
    float rot[9] = {0};
    GenRotMatrix(ry, rot);
    float pt_proj[3] = {0};
    float pt_c[3] = {0};
    float *pt = pts;
    bool save_pts_c = pts_c != nullptr;
    for (int i = 0; i < 4; ++i) {
        common::IProjectThroughExtrinsic(rot, center, pt, pt_c);
        common::IProjectThroughIntrinsic(k_mat_, pt_c, pt_proj);
        depth_pts[i] = pt_c[2];
        x_pts[i] = common::IRound(pt_proj[0] *
common::IRec(pt_proj[2]));
        int y_proj = common::IRound(pt_proj[1] *
common::IRec(pt_proj[2]));
        if (y_proj < y_min) {
            y_min = y_proj;
        }
        if (save_pts_c) {

```

```

        int i3 = i * 3;
        pts_c[i3] = pt_c[0];
        pts_c[i3 + 1] = pt_c[1];
        pts_c[i3 + 2] = pt_c[2];
    }
    pt += 3;
}
return y_min;
}

```

transformer转换

MultiCueObstacleTransformer类，实现对障碍物做转换。

初始化Init

```

bool MultiCueObstacleTransformer::Init(
    const ObstacleTransformerInitOptions &options) {
    std::string transformer_config =
        cyber::common::GetAbsolutePath(options.root_dir,
options.conf_file);
    // 1. 获取参数
    if (!cyber::common::GetProtoFromFile(transformer_config,
&multicue_param_)) {
        AERROR << "Read config failed: " << transformer_config;
        return false;
    }
    AINFO << "Load transformer parameters from " <<
transformer_config
        << "\nmin dimension: " <<
multicue_param_.min_dimension_val()
        << "\ndo template search: " <<
multicue_param_.check_dimension();

    // 2. 初始化对象模板
    object_template_manager_ =
ObjectTemplateManager::Instance();

    return true;
}

```

SetObjMapperOptions设置对象地图属性

```
void MultiCueObstacleTransformer::SetObjMapperOptions(
    base::ObjectPtr obj, Eigen::Matrix3f camera_k_matrix, int
width_image,
    int height_image, ObjMapperOptions *obj_mapper_options,
float *theta_ray) {
    // prepare bbox2d
    float bbox2d[4] = {
        obj->camera_supplement.box.xmin, obj-
>camera_supplement.box.ymin,
        obj->camera_supplement.box.xmax, obj-
>camera_supplement.box.ymax};
    // input insanity check
    bbox2d[0] = std::max(0.0f, bbox2d[0]);
    bbox2d[1] = std::max(0.0f, bbox2d[1]);
    bbox2d[2] = std::min(static_cast<float>(width_image) -
1.0f, bbox2d[2]);
    bbox2d[3] = std::min(static_cast<float>(height_image) -
1.0f, bbox2d[3]);
    bbox2d[2] = std::max(bbox2d[0] + 1.0f, bbox2d[2]);
    bbox2d[3] = std::max(bbox2d[1] + 1.0f, bbox2d[3]);

    // prepare dimension_hw
    float dimension_hw[3] = {obj->size(2), obj->size(1), obj-
>size(0)};

    // prepare rotation_y
    float box_center_x = (bbox2d[0] + bbox2d[2]) / 2;
    Eigen::Vector3f image_point_low_center(box_center_x,
bbox2d[3], 1);
    Eigen::Vector3f point_in_camera =
        static_cast<Eigen::Matrix<float, 3, 1, 0, 3, 1>>(
            camera_k_matrix.inverse() *
image_point_low_center);
    *theta_ray =
        static_cast<float>(atan2(point_in_camera.x(),
point_in_camera.z())));
    float rotation_y =
        *theta_ray + static_cast<float>(obj-
>camera_supplement.alpha);
    base::ObjectSubType sub_type = obj->sub_type;
```

```

// enforce rotation_y to be in the range [-pi, pi)
const float PI = common::Constant<float>::PI();
if (rotation_y < -PI) {
    rotation_y += 2 * PI;
} else if (rotation_y >= PI) {
    rotation_y -= 2 * PI;
}

memcpy(obj_mapper_options->bbox, bbox2d, sizeof(float) *
4);
memcpy(obj_mapper_options->hw1, dimension_hw1,
sizeof(float) * 3);
obj_mapper_options->ry = rotation_y;
obj_mapper_options->is_veh = (obj->type ==
base::ObjectType::VEHICLE);
obj_mapper_options->check_dimension =
multicue_param_.check_dimension();
obj_mapper_options->type_min_vol_index =
    MatchTemplates(sub_type, dimension_hw1);

// 2D到3D转换
ADEBUG << "#2D-to-3D for one obj:";
ADEBUG << "Obj pred ry:" << rotation_y;
ADEBUG << "Obj pred type: " << static_cast<int>(sub_type);
ADEBUG << "Bbox: " << bbox2d[0] << ", " << bbox2d[1] << ", "
<< bbox2d[2]
        << ", " << bbox2d[3];
}

```

匹配模板，返回值type_min_vol_index的作用是什么？？？

```

int
MultiCueObstacleTransformer::MatchTemplates(base::ObjectSubType
sub_type,
                                              float
*dimension_hw1) {
    const TemplateMap &kMinTemplateHWL =
        object_template_manager_->MinTemplateHWL();
    const TemplateMap &kMidTemplateHWL =
        object_template_manager_->MidTemplateHWL();
    const TemplateMap &kMaxTemplateHWL =
        object_template_manager_->MaxTemplateHWL();
}

```

```

    int type_min_vol_index = 0;
    float min_dimension_val =
        std::min(std::min(dimension_hw1[0], dimension_hw1[1]),
dimension_hw1[2]);

    const std::map<TemplateIndex, int>
&kLookUpTableMinVolumeIndex =
    object_template_manager_->LookUpTableMinVolumeIndex();

    switch (sub_type) {
        // 1. 交通锥
        case base::ObjectSubType::TRAFFICCONE: {
            const float *min_tmplt_cur_type =
kMinTemplateHWL.at(sub_type).data();
            const float *max_tmplt_cur_type =
kMaxTemplateHWL.at(sub_type).data();
            dimension_hw1[0] = std::min(dimension_hw1[0],
max_tmplt_cur_type[0]);
            dimension_hw1[0] = std::max(dimension_hw1[0],
min_tmplt_cur_type[0]);
            break;
        }
        // 2. 行人、自行车、摩托车
        case base::ObjectSubType::PEDESTRIAN:
        case base::ObjectSubType::CYCLIST:
        case base::ObjectSubType::MOTORCYCLIST: {
            const float *min_tmplt_cur_type =
kMinTemplateHWL.at(sub_type).data();
            const float *mid_tmplt_cur_type =
kMidTemplateHWL.at(sub_type).data();
            const float *max_tmplt_cur_type =
kMaxTemplateHWL.at(sub_type).data();
            float dh_min = fabsf(dimension_hw1[0] -
min_tmplt_cur_type[0]);
            float dh_mid = fabsf(dimension_hw1[0] -
mid_tmplt_cur_type[0]);
            float dh_max = fabsf(dimension_hw1[0] -
max_tmplt_cur_type[0]);
            std::vector<std::pair<float, float>> diff_hs;
            diff_hs.push_back(std::make_pair(dh_min,
min_tmplt_cur_type[0]));
            diff_hs.push_back(std::make_pair(dh_mid,
mid_tmplt_cur_type[0]));
    }
}

```

```

        diff_hs.push_back(std::make_pair(dimension_hwls[0], max));
    }
    sort(diff_hs.begin(), diff_hs.end(),
        [] (const std::pair<float, float> &a,
            const std::pair<float, float> &b) -> bool {
        return a.first < b.first;
    });
    dimension_hwls[0] = diff_hs[0].second;
    break;
}
// 3. 汽车
case base::ObjectSubType::CAR:
    type_min_vol_index =
kLookUpTableMinVolumeIndex.at(TemplateIndex::CAR_MIN_VOLUME_INDEX);
    break;
case base::ObjectSubType::VAN:
    type_min_vol_index =
kLookUpTableMinVolumeIndex.at(TemplateIndex::VAN_MIN_VOLUME_INDEX);
    break;
// 5. 卡车
case base::ObjectSubType::TRUCK:
    type_min_vol_index =
kLookUpTableMinVolumeIndex.at(TemplateIndex::TRUCK_MIN_VOLUME_INDEX);
    break;
case base::ObjectSubType::BUS:
    type_min_vol_index =
kLookUpTableMinVolumeIndex.at(TemplateIndex::BUS_MIN_VOLUME_INDEX);
    break;
default:
    if (min_dimension_val <
multicue_param_.min_dimension_val()) {
        common::IScale3(dimension_hwls,
multicue_param_.min_dimension_val() *
common::IRec(min_dimension_val));

```

```

        }
        break;
    }
    return type_min_vol_index;
}

```

填充结果，设置物体的大小，朝向等信息。

```

void MultiCueObstacleTransformer::FillResults(
    float object_center[3], float dimension_hw1[3], float
rotation_y,
    Eigen::Affine3d camera2world_pose, float theta_ray,
base::ObjectPtr obj) {
    if (obj == nullptr) {
        return;
    }
    object_center[1] -= dimension_hw1[0] / 2;
    obj->camera_supplement.local_center(0) = object_center[0];
    obj->camera_supplement.local_center(1) = object_center[1];
    obj->camera_supplement.local_center(2) = object_center[2];
    ADEBUG << "Obj id: " << obj->track_id;
    ADEBUG << "Obj type: " << static_cast<int>(obj->sub_type);
    ADEBUG << "Obj ori dimension: " << obj->size[2] << ", " <<
obj->size[1]
                << ", " << obj->size[0];
    obj->center(0) = static_cast<double>(object_center[0]);
    obj->center(1) = static_cast<double>(object_center[1]);
    obj->center(2) = static_cast<double>(object_center[2]);
    obj->center = camera2world_pose * obj->center;

    obj->size(2) = dimension_hw1[0];
    obj->size(1) = dimension_hw1[1];
    obj->size(0) = dimension_hw1[2];

    Eigen::Matrix3d pos_var = mapper_-
>get_position_uncertainty();
    obj->center_uncertainty(0) = static_cast<float>
(pos_var(0));
    obj->center_uncertainty(1) = static_cast<float>
(pos_var(1));
    obj->center_uncertainty(2) = static_cast<float>
(pos_var(2));
}

```

```

    float theta = rotation_y;
    Eigen::Vector3d dir = (camera2world_pose.matrix()).block(0,
0, 3, 3) *
                                Eigen::Vector3d(cos(theta), 0, -
sin(theta));
    obj->direction[0] = static_cast<float>(dir[0]);
    obj->direction[1] = static_cast<float>(dir[1]);
    obj->direction[2] = static_cast<float>(dir[2]);
    obj->theta = static_cast<float>(atan2(dir[1], dir[0]));
    obj->theta_variance = static_cast<float>((mapper_-
>get_orientation_var())(0));

    obj->camera_supplement.alpha = rotation_y - theta_ray;

    ADEBUG << "Dimension hwl: " << dimension_hwl[0] << ", " <<
dimension_hwl[1]
                << ", " << dimension_hwl[2];
    ADEBUG << "Obj ry:" << rotation_y;
    ADEBUG << "Obj theta: " << obj->theta;
    ADEBUG << "Obj center from transformer: " << obj-
>center.transpose();
}

```

转换

```

bool MultiCueObstacleTransformer::Transform(
    const ObstacleTransformerOptions &options, CameraFrame
*frame) {
    // 1. 相机k矩阵内参矩阵
    const auto &camera_k_matrix = frame->camera_k_matrix;
    float k_mat[9] = {0};
    for (size_t i = 0; i < 3; i++) {
        size_t i3 = i * 3;
        for (size_t j = 0; j < 3; j++) {
            k_mat[i3 + j] = camera_k_matrix(i, j);
        }
    }

    const int width_image = frame->data_provider->src_width();
    const int height_image = frame->data_provider-
>src_height();
    const auto &camera2world_pose = frame->camera2world_pose;
    mapper_->Init(k_mat, width_image, height_image);
}

```

```

ObjMapperOptions obj_mapper_options;
float object_center[3] = {0};
float dimension_hw1[3] = {0};
float rotation_y = 0.0f;

int nr_transformed_obj = 0;
for (auto &obj : frame->detected_objects) {

    // 1. 设置对象字典属性
    float theta_ray = 0.0f;
    SetObjMapperOptions(obj, camera_k_matrix, width_image,
height_image,
                        &obj_mapper_options, &theta_ray);

    // 2. 执行
    mapper_->Solve3dBbox(obj_mapper_options, object_center,
dimension_hw1,
                           &rotation_y);

    // 3. 填充结果
    FillResults(object_center, dimension_hw1, rotation_y,
camera2world_pose,
               theta_ray, obj);

    ++nr_transformed_obj;
}
return nr_transformed_obj > 0;
}

```

ObjMapper

ObjMapper找到3D框

```

bool ObjMapper::Solve3dBbox(const ObjMapperOptions &options,
float center[3],
                           float hwl[3], float *ry) {
// set default value for variance
set_default_variance();
float var_yaw = 0.0f;
float var_z = 0.0f;

```

```

// get input from options
memcpy(hwl, options.hwl, sizeof(float) * 3);
float bbox[4] = {0};
memcpy(bbox, options.bbox, sizeof(float) * 4);
*ry = options.ry;
bool check_dimension = options.check_dimension;
int type_min_vol_index = options.type_min_vol_index;

// check input hwl insanity
if (options.is_veh && check_dimension) {
    assert(type_min_vol_index >= 0);
    const std::vector<float> &kVehHwl =
object_template_manager_->VehHwl();
    const float *tmplt_with_min_vol =
&kVehHwl[type_min_vol_index];
    float min_tmplt_vol =
        tmplt_with_min_vol[0] * tmplt_with_min_vol[1] *
tmplt_with_min_vol[2];
    float shrink_ratio_vol =
common::ISqr(sqrtf(params_.iou_high));
    shrink_ratio_vol *= shrink_ratio_vol;
    // float shrink_ratio_vol = sqrt(params_.iou_high);
    if (hwl[0] < params_.abnormal_h_veh ||
        hwl[0] * hwl[1] * hwl[2] < min_tmplt_vol *
shrink_ratio_vol) {
        memcpy(hwl, tmplt_with_min_vol, sizeof(float) * 3);
    } else {
        float hwl_tmplt[3] = {hwl[0], hwl[1], hwl[2]};
        int tmplt_index = -1;
        float score = object_template_manager_-
>VehObjHwlBySearchTemplates(
            hwl_tmplt, &tmplt_index);
        float thres_min_score = shrink_ratio_vol;

        const int kNrDimPerTmplt = object_template_manager_-
>NrDimPerTmplt();
        bool search_success = score > thres_min_score;
        bool is_same_type = (type_min_vol_index /
kNrDimPerTmplt) == tmplt_index;
        const std::map<TemplateIndex, int>
&kLookUpTableMinVolumeIndex =
            object_template_manager_-

```

```

>LookUpTableMinVolumeIndex();
    bool is_car_pred =
        type_min_vol_index ==

kLookUpTableMinVolumeIndex.at(TemplateIndex::CAR_MIN_VOLUME_I
NDEX);

    bool hwl_is_reliable = search_success && is_same_type;
    if (hwl_is_reliable) {
        memcpy(hwl, hwl_tmplt, sizeof(float) * 3);
    } else if (is_car_pred) {
        const float *tmplt_with_median_vol =
            tmplt_with_min_vol + kNrDimPerTmplt;
        memcpy(hwl, tmplt_with_median_vol, sizeof(float) *
3);
    }
}

// call 3d solver
bool success =
    Solve3dBboxGivenOneFullBboxDimensionOrientation(bbox,
hwl, ry, center);

// calculate variance for yaw & z
float yaw_score_mean =
    common::IMean(ry_score_.data(), static_cast<int>
(ry_score_.size()));
    float yaw_score_sdv = common::ISdv(ry_score_.data(),
yaw_score_mean,
                                static_cast<int>
(ry_score_.size()));
    var_yaw = common::ISqrt(common::IRec(yaw_score_sdv +
params_.eps_mapper));

float z = center[2];
float rz = z * params_.rz_ratio;
float nr_bins_z = static_cast<float>(params_.nr_bins_z);
std::vector<float> buffer(static_cast<size_t>(2 *
nr_bins_z), 0);
float *score_z = buffer.data();
float dz = 2 * rz / nr_bins_z;
float z_start = std::max(z - rz, params_.depth_min);

```

```

float z_end = z + rz;
int count_z_test = 0;
for (float z_test = z_start; z_test <= z_end; z_test += dz)
{
    float center_test[3] = {center[0], center[1], center[2]};
    float sf = z_test * common::IRec(center_test[2]);
    common::IScale3(center_test, sf);
    float score_test = GetProjectionScore(*ry, bbox, hwl,
center_test);
    score_z[count_z_test++] = score_test;
}
float z_score_mean = common::IMean(score_z, count_z_test);
float z_score_sdv = common::ISdv(score_z, z_score_mean,
count_z_test);
var_z = common::ISqr(common::IRec(z_score_sdv +
params_.eps_mapper));

// fill the position_uncertainty_ and orientation_variance_
orientation_variance_(0) = var_yaw;
float bbox_cx = (bbox[0] + bbox[2]) / 2;
float focal = (k_mat_[0] + k_mat_[4]) / 2;
float sf_z_to_x = fabsf(bbox_cx - k_mat_[2]) *
common::IRec(focal);
float var_x = var_z * common::ISqr(sf_z_to_x);
float var_xz = sf_z_to_x * var_z;
position_uncertainty_(0, 0) = var_x;
position_uncertainty_(2, 2) = var_z;
position_uncertainty_(0, 2) = position_uncertainty_(2, 0) =
var_xz;
return success;
}

```

lane车道线

postprocessor 后处理

Process2D

```

bool DarkSCNNLanePostprocessor::Process2D(
    const LanePostprocessorOptions& options, CameraFrame*
frame) {

    frame->lane_objects.clear();
    auto start = std::chrono::high_resolution_clock::now();

    // 1. 拷贝检测到车道线数据到 lane_map
    cv::Mat lane_map(lane_map_height_, lane_map_width_,
CV_32FC1);
    memcpy(lane_map.data, frame->lane_detected_blob-
>cpu_data(),
           lane_map_width_ * lane_map_height_ * sizeof(float));

    // 2. 从lane_map中采样点，并且投影他们到世界坐标
    // TODO(techoe): Should be fixed
    int y = static_cast<int>(lane_map.rows * 0.9 - 1);
    // TODO(techoe): Should be fixed
    int step_y = (y - 40) * (y - 40) / 6400 + 1;

    xy_points.clear();
    xy_points.resize(lane_type_num_);
    uv_points.clear();
    uv_points.resize(lane_type_num_);

    // 3.1 如果y大于0，每次移动step
    while (y > 0) {
        // 3.2 每次移动一列
        for (int x = 1; x < lane_map.cols - 1; ++x) {
            // 3.3 获取最大行，最小列的点的值
            int value = static_cast<int>(round(lane_map.at<float>
(y, x)));
            // 3.4 如果lane在车道左边，value的值在spatialLUTind中
            if ((value > 0 && value < 5) || value == 11) {
                // right edge (inner) of the lane
                if (value != static_cast<int>
(round(lane_map.at<float>(y, x + 1)))) {
                    Eigen::Matrix<float, 3, 1> img_point(
                        static_cast<float>(x * roi_width_ /
lane_map.cols),
                        static_cast<float>(y * roi_height_ /
lane_map.rows + roi_start_));

```



```

Eigen::Matrix<float, 2, 1> xy_point;
Eigen::Matrix<float, 2, 1> uv_point;
if (std::fabs(xy_p(2)) < 1e-6) continue;
xy_point << xy_p(0) / xy_p(2), xy_p(1) / xy_p(2);
// Filter out lane line points
if (xy_point(0) < 0.0 || // This condition is only
for front camera
    xy_point(0) > max_longitudinal_distance_ ||
    std::abs(xy_point(1)) > 30.0) {
    continue;
}
uv_point << static_cast<float>(x * roi_width_ /
lane_map.cols),
           static_cast<float>(y * roi_height_ /
lane_map.rows + roi_start_);
if (xy_points[value].size() < minNumPoints_ ||
xy_point(0) < 50.0f ||
    std::fabs(xy_point(1) - xy_points[value].back()
(1)) < 1.0f) {
    xy_points[value].push_back(xy_point);
    uv_points[value].push_back(uv_point);
}
} else if (value >= lane_type_num_) {
    AWARN << "Lane line value shouldn't be equal or
more than: "
        << lane_type_num_;
}
}
step_y = (y - 45) * (y - 45) / 6400 + 1;
y -= step_y;
}

auto elapsed_1 = std::chrono::high_resolution_clock::now()
- start;
int64_t microseconds_1 =
    std::chrono::duration_cast<std::chrono::microseconds>
(elapsed_1).count();
time_1 += microseconds_1;

// 4. 移除异常值并进行ransac拟合
std::vector<Eigen::Matrix<float, 4, 1>> coeffs;
std::vector<Eigen::Matrix<float, 4, 1>> img_coeffs;

```

```

    std::vector<Eigen::Matrix<float, 2, 1>> selected_xy_points;
    coeffs.resize(lane_type_num_);
    img_coeffs.resize(lane_type_num_);
    for (int i = 1; i < lane_type_num_; ++i) {
        coeffs[i] << 0, 0, 0, 0;
        // 4.1 xy_points[i]即是每种类型lane中抽样的点的数组
        if (xy_points[i].size() < minNumPoints_) continue;
        Eigen::Matrix<float, 4, 1> coeff;
        // Solve linear system to estimate polynomial
        coefficients
        if (RansacFitting<float>(xy_points[i],
        &selected_xy_points, &coeff, 200,
                    static_cast<int>(minNumPoints_),
        0.1f)) {
            coeffs[i] = coeff;

            xy_points[i].clear();
            xy_points[i] = selected_xy_points;
        } else {
            xy_points[i].clear();
        }
    }

    auto elapsed_2 = std::chrono::high_resolution_clock::now()
- start;
    int64_t microseconds_2 =
        std::chrono::duration_cast<std::chrono::microseconds>
(elapsed_2).count();
    time_2 += microseconds_2 - microseconds_1;

    // 3. Write values into lane_objects
    std::vector<float> c0s(lane_type_num_, 0);
    for (int i = 1; i < lane_type_num_; ++i) {
        if (xy_points[i].size() < minNumPoints_) continue;
        c0s[i] = GetPolyValue(
            static_cast<float>(coeffs[i](3)), static_cast<float>
(coeffs[i](2)),
            static_cast<float>(coeffs[i](1)), static_cast<float>
(coeffs[i](0)),
            static_cast<float>(3.0));
    }
    // 在特殊情况下确定车道空间标签
    // 1.检查左边车道线是否小于最小采样点 2.检查右边车道线是否小于最小采样

```

点

```
if (xy_points[4].size() < minNumPoints_ &&
    xy_points[5].size() >= minNumPoints_) {
    std::swap(xy_points[4], xy_points[5]);
    std::swap(uv_points[4], uv_points[5]);
    std::swap(coeffs[4], coeffs[5]);
    std::swap(c0s[4], c0s[5]);
} else if (xy_points[6].size() < minNumPoints_ &&
           xy_points[5].size() >= minNumPoints_) {
    std::swap(xy_points[6], xy_points[5]);
    std::swap(uv_points[6], uv_points[5]);
    std::swap(coeffs[6], coeffs[5]);
    std::swap(c0s[6], c0s[5]);
}

if (xy_points[4].size() < minNumPoints_ &&
    xy_points[11].size() >= minNumPoints_) {
    // Use left lane boundary as the right most missing left
    lane,
    bool use_boundary = true;
    for (int k = 3; k >= 1; --k) {
        if (xy_points[k].size() >= minNumPoints_) {
            use_boundary = false;
            break;
        }
    }
    if (use_boundary) {
        std::swap(xy_points[4], xy_points[11]);
        std::swap(uv_points[4], uv_points[11]);
        std::swap(coeffs[4], coeffs[11]);
        std::swap(c0s[4], c0s[11]);
    }
}

if (xy_points[6].size() < minNumPoints_ &&
    xy_points[12].size() >= minNumPoints_) {
    // Use right lane boundary as the left most missing right
    lane,
    bool use_boundary = true;
    for (int k = 7; k <= 9; ++k) {
        if (xy_points[k].size() >= minNumPoints_) {
            use_boundary = false;
            break;
        }
    }
}
```

```

        }
    }

    if (use_boundary) {
        std::swap(xy_points[6], xy_points[12]);
        std::swap(uv_points[6], uv_points[12]);
        std::swap(coeffs[6], coeffs[12]);
        std::swap(c0s[6], c0s[12]);
    }
}

for (int i = 1; i < lane_type_num_; ++i) {
    base::LaneLine cur_object;
    if (xy_points[i].size() < minNumPoints_) {
        continue;
    }

    // [2] Set spatial label
    cur_object.pos_type = spatialLUT[i];

    // [3] Determine which lines are valid according to the y
    value at x = 3
    if ((i < 5 && c0s[i] < c0s[i + 1]) ||
        (i > 5 && i < 10 && c0s[i] > c0s[i - 1])) {
        continue;
    }
    if (i == 11 || i == 12) {
        std::sort(c0s.begin(), c0s.begin() + 10);
        if ((c0s[i] > c0s[0] && i == 12) || (c0s[i] < c0s[9] &&
i == 11)) {
            continue;
        }
    }
    // [4] 结果写入cur_object
    cur_object.curve_car_coord.x_start =
        static_cast<float>(xy_points[i].front()(0));
    cur_object.curve_car_coord.x_end =
        static_cast<float>(xy_points[i].back()(0));
    cur_object.curve_car_coord.a = static_cast<float>
(coeffs[i](3));
    cur_object.curve_car_coord.b = static_cast<float>
(coeffs[i](2));
    cur_object.curve_car_coord.c = static_cast<float>
(coeffs[i](1));
}

```

```

    cur_object.curve_car_coord.d = static_cast<float>
(coeffs[i](0));
    // if (cur_object.curve_car_coord.x_end -
    //     cur_object.curve_car_coord.x_start < 5) continue;
    // cur_object.order = 2;
    cur_object.curve_car_coord_point_set.clear();
    for (size_t j = 0; j < xy_points[i].size(); ++j) {
        base::Point2DF p_j;
        p_j.x = static_cast<float>(xy_points[i][j](0));
        p_j.y = static_cast<float>(xy_points[i][j](1));
        cur_object.curve_car_coord_point_set.push_back(p_j);
    }

    cur_object.curve_image_point_set.clear();
    for (size_t j = 0; j < uv_points[i].size(); ++j) {
        base::Point2DF p_j;
        p_j.x = static_cast<float>(uv_points[i][j](0));
        p_j.y = static_cast<float>(uv_points[i][j](1));
        cur_object.curve_image_point_set.push_back(p_j);
    }

    // cur_object.confidence.push_back(1);
    cur_object.confidence = 1.0f;
    frame->lane_objects.push_back(cur_object);
}

// Special case riding on a lane:
// 0: no center lane, 1: center lane as left, 2: center
lane as right
int has_center_ = 0;
for (auto lane_ : frame->lane_objects) {
    if (lane_.pos_type ==
base::LaneLinePositionType::EGO_CENTER) {
        if (lane_.curve_car_coord.d >= 0) {
            has_center_ = 1;
        } else if (lane_.curve_car_coord.d < 0) {
            has_center_ = 2;
        }
        break;
    }
}
// Change labels for all lanes from one side
if (has_center_ == 1) {

```

```

        for (auto& lane_ : frame->lane_objects) {
            int spatial_id = spatialLUTind[lane_.pos_type];
            if (spatial_id >= 1 && spatial_id <= 5) {
                lane_.pos_type = spatialLUT[spatial_id - 1];
            }
        }
    } else if (has_center_ == 2) {
        for (auto& lane_ : frame->lane_objects) {
            int spatial_id = spatialLUTind[lane_.pos_type];
            if (spatial_id >= 5 && spatial_id <= 9) {
                lane_.pos_type = spatialLUT[spatial_id + 1];
            }
        }
    }
}

auto elapsed = std::chrono::high_resolution_clock::now() - start;
int64_t microseconds =
    std::chrono::duration_cast<std::chrono::microseconds>
(elapsed).count();
// AINFO << "Time for writing: " << microseconds -
microseconds_2 << " us";
time_3 += microseconds - microseconds_2;
++time_num;

ADEBUG << "frame->lane_objects.size(): " << frame-
>lane_objects.size();

ADEBUG << "Avg sampling time: " << time_1 / time_num
    << " Avg fitting time: " << time_2 / time_num
    << " Avg writing time: " << time_3 / time_num;
ADEBUG << "darkSCNN lane_postprocess done!";
return true;
}

```

DarkSCNN

```

// Produce laneline output in camera coordinates (optional)
bool DarkSCNNLanePostprocessor::Process3D(
    const LanePostprocessorOptions& options, CameraFrame*
frame) {
    ConvertImagePoint2Camera(frame);

```

```

PolyFitCameraLaneline(frame);
return true;
}

```

图片坐标转换到相机

```

void
DarkSCNNLanePostprocessor::ConvertImagePoint2Camera(CameraFrame* frame) {
    float pitch_angle = frame->calibration_service-
>QueryPitchAngle();
    float camera_ground_height =
        frame->calibration_service-
>QueryCameraToGroundHeight();
    const Eigen::Matrix3f& intrinsic_params = frame-
>camera_k_matrix;
    const Eigen::Matrix3f& intrinsic_params_inverse =
intrinsic_params.inverse();
    std::vector<base::LaneLine>& lane_objects = frame-
>lane_objects;
    int laneline_num = static_cast<int>(lane_objects.size());
    for (int line_index = 0; line_index < laneline_num;
++line_index) {
        std::vector<base::Point2DF>& image_point_set =
            lane_objects[line_index].curve_image_point_set;
        std::vector<base::Point3DF>& camera_point_set =
            lane_objects[line_index].curve_camera_point_set;
        for (int i = 0; i < static_cast<int>
(image_point_set.size()); i++) {
            base::Point3DF camera_point;
            Eigen::Vector3d camera_point3d;
            const base::Point2DF& image_point = image_point_set[i];
            ImagePoint2Camera(image_point, pitch_angle,
camera_ground_height,
                            intrinsic_params_inverse,
&camera_point3d);
            camera_point.x = static_cast<float>(camera_point3d(0));
            camera_point.y = static_cast<float>(camera_point3d(1));
            camera_point.z = static_cast<float>(camera_point3d(2));
            camera_point_set.push_back(camera_point);
        }
    }
}

```

获取车道线点

```
void
DarkSCNNLanePostprocessor::PolyFitCameraLaneline(CameraFrame*
frame) {
    std::vector<base::LaneLine>& lane_objects = frame-
>lane_objects;
    int laneline_num = static_cast<int>(lane_objects.size());
    for (int line_index = 0; line_index < laneline_num;
++line_index) {
        const std::vector<base::Point3DF>& camera_point_set =
            lane_objects[line_index].curve_camera_point_set;
        // z: longitudinal direction
        // x: latitudinal direction
        float x_start = camera_point_set[0].z;
        float x_end = 0.0f;
        Eigen::Matrix<float, max_poly_order + 1, 1> camera_coeff;
        std::vector<Eigen::Matrix<float, 2, 1>> camera_pos_vec;
        for (int i = 0; i < static_cast<int>
(camera_point_set.size()); ++i) {
            x_end = std::max(camera_point_set[i].z, x_end);
            x_start = std::min(camera_point_set[i].z, x_start);
            Eigen::Matrix<float, 2, 1> camera_pos;
            camera_pos << camera_point_set[i].z,
camera_point_set[i].x;
            camera_pos_vec.push_back(camera_pos);
        }

        bool is_x_axis = true;
        bool fit_flag =
            PolyFit(camera_pos_vec, max_poly_order,
&camera_coeff, is_x_axis);
        if (!fit_flag) {
            continue;
        }
        lane_objects[line_index].curve_camera_coord.a =
camera_coeff(3, 0);
        lane_objects[line_index].curve_camera_coord.b =
camera_coeff(2, 0);
        lane_objects[line_index].curve_camera_coord.c =
camera_coeff(1, 0);
        lane_objects[line_index].curve_camera_coord.d =
camera_coeff(0, 0);
    }
}
```

```

        lane_objects[line_index].curve_camera_coord.x_start =
x_start;
        lane_objects[line_index].curve_camera_coord.x_end =
x_end;
        lane_objects[line_index].use_type =
base::LaneLineUseType::REAL;
    }
}

```

数据集

[CULane Dataset](https://xingangpan.github.io/projects/CULane.html) [<https://xingangpan.github.io/projects/CULane.html>]

[bdd](https://bdd-data.berkeley.edu/) [<https://bdd-data.berkeley.edu/>]

[tusimple](https://github.com/TuSimple/tusimple-benchmark) [<https://github.com/TuSimple/tusimple-benchmark>]

参考

[awesome-lane-detection](https://github.com/amusi/awesome-lane-detection) [<https://github.com/amusi/awesome-lane-detection>]

calibration_service & calibrator

首先看OnlineCalibrationService类的实现。

OnlineCalibrationService

初始化Init

```

bool OnlineCalibrationService::Init(
    const CalibrationServiceInitOptions &options) {
    master_sensor_name_ =
options.calibrator_working_sensor_name;
    sensor_name_ = options.calibrator_working_sensor_name;
    // 1. 初始化K矩阵
    auto &name_intrinsic_map = options.name_intrinsic_map;
    ACHECK(name_intrinsic_map.find(master_sensor_name_) !=
          name_intrinsic_map.end());
    CameraStatus camera_status;

```

```

    name_camera_status_map_.clear();
    for (auto iter = name_intrinsic_map.begin(); iter != name_intrinsic_map.end();
         ++iter) {
        camera_status.k_matrix[0] = static_cast<double>(iter->second(0, 0));
        camera_status.k_matrix[4] = static_cast<double>(iter->second(1, 1));
        camera_status.k_matrix[2] = static_cast<double>(iter->second(0, 2));
        camera_status.k_matrix[5] = static_cast<double>(iter->second(1, 2));
        camera_status.k_matrix[8] = 1.0;
        name_camera_status_map_.insert(
            std::pair<std::string, CameraStatus>(iter->first,
camera_status));
    }
    // 初始化校准参数
    CalibratorInitOptions calibrator_init_options;
    calibrator_init_options.image_width = options.image_width;
    calibrator_init_options.image_height =
options.image_height;
    calibrator_init_options.focal_x = static_cast<float>(
name_camera_status_map_[master_sensor_name_].k_matrix[0]);
    calibrator_init_options.focal_y = static_cast<float>(
name_camera_status_map_[master_sensor_name_].k_matrix[4]);
    calibrator_init_options.cx = static_cast<float>(
name_camera_status_map_[master_sensor_name_].k_matrix[2]);
    calibrator_init_options.cy = static_cast<float>(
name_camera_status_map_[master_sensor_name_].k_matrix[5]);
    calibrator_.reset();

BaseCalibratorRegisterer::GetInstanceByName(options.calibrator_method));
    ACHECK(calibrator_ != nullptr);
    ACHECK(calibrator_->Init(calibrator_init_options))
        << "Failed to init " << options.calibrator_method;
    return true;
}

```

更新帧信息

```
void OnlineCalibrationService::Update(CameraFrame *frame) {

    sensor_name_ = frame->data_provider->sensor_name();
    if (sensor_name_ == master_sensor_name_) {
        CalibratorOptions calibrator_options;
        calibrator_options.lane_objects =
            std::make_shared<std::vector<base::LaneLine>>(frame-
>lane_objects);
        calibrator_options.camera2world_pose =
            std::make_shared<Eigen::Affine3d>(frame-
>camera2world_pose);
        calibrator_options.timestamp = &(frame->timestamp);
        float pitch_angle = 0.f;
        // 1. 校准传感器参数
        bool updated = calibrator_->Calibrate(calibrator_options,
&pitch_angle);
        // rebuild the service when updated
        if (updated) {

name_camera_status_map_[master_sensor_name_].pitch_angle =
pitch_angle;
            for (auto iter = name_camera_status_map_.begin();
                iter != name_camera_status_map_.end(); iter++) {
                // 2. 更新俯仰角
                iter->second.pitch_angle =
                    iter->second.pitch_angle_diff +
name_camera_status_map_[master_sensor_name_].pitch_angle;
                // 3. 更新地平面
                iter->second.ground_plane[1] = cos(iter-
>second.pitch_angle);
                iter->second.ground_plane[2] = -sin(iter-
>second.pitch_angle);
            }
        }
        auto iter = name_camera_status_map_.find(sensor_name_);
        AINFO << "camera_ground_height: " << iter-
>second.camera_ground_height
            << " meter.";
        AINFO << "pitch_angle: " << iter->second.pitch_angle *
    }
}
```

```

180.0 / M_PI
    << " degree.";
is_service_ready_ = true;
}

```

calibrator

LaneLineCalibrator依赖LaneBasedCalibrator。

```

bool LaneLineCalibrator::Calibrate(const CalibratorOptions
&options,
                                    float *pitch_angle) {
    // 1. 加载当前车道线
    EgoLane ego_lane;
    if (!LoadEgoLaneline(*options.lane_objects, &ego_lane)) {
        AINFO << "Failed to get the ego lane.";
        return false;
    }

    double cam_ori[4] = {0};
    cam_ori[3] = 1.0;

    // 2. 相机坐标到世界坐标
    Eigen::Affine3d c2w = *options.camera2world_pose;

    double p2w[12] = {
        c2w(0, 0), c2w(0, 1), c2w(0, 2), c2w(0, 3), c2w(1, 0),
        c2w(1, 1),
        c2w(1, 2), c2w(1, 3), c2w(2, 0), c2w(2, 1), c2w(2, 2),
        c2w(2, 3),
    };

    ADEBUG << "c2w transform this frame:\n"
           << p2w[0] << ", " << p2w[1] << ", " << p2w[2] << ", "
           << p2w[3] << "\n"
           << p2w[4] << ", " << p2w[5] << ", " << p2w[6] << ", "
           << p2w[7] << "\n"
           << p2w[8] << ", " << p2w[9] << ", " << p2w[10] << ", "
           << p2w[11];

    common::IMultAx3x4(p2w, cam_ori, cam_coord_cur_);
}

```

```

time_diff_ = kTimeDiffDefault;
yaw_rate_ = kYawRateDefault;
velocity_ = kVelocityDefault;

timestamp_cur_ = *options.timestamp;
if (!is_first_frame_) {
    time_diff_ = fabsf(static_cast<float>(timestamp_cur_ -
timestamp_pre_));
    ADEBUG << timestamp_cur_ << " " << timestamp_pre_ <<
std::endl;
    camera::GetYawVelocityInfo(time_diff_, cam_coord_cur_,
cam_coord_pre_,
                                cam_coord_pre_pre_,
&yaw_rate_, &velocity_);
    std::string timediff_yawrate_velocity_text =
        absl::StrCat("time_diff_: ",
std::to_string(time_diff_).substr(0, 4),
                      " | yaw_rate_: ",
std::to_string(yaw_rate_).substr(0, 4),
                      " | velocity_: ",
std::to_string(velocity_).substr(0, 4));
    ADEBUG << timediff_yawrate_velocity_text << std::endl;
}

bool updated =
    calibrator_.Process(ego_lane, velocity_, yaw_rate_,
time_diff_);
if (updated) {
    *pitch_angle = calibrator_.get_pitch_estimation();
    float vanishing_row = calibrator_.get_vanishing_row();
    AINFO << "#updated pitch angle: " << *pitch_angle;
    AINFO << "#vanishing row: " << vanishing_row;
}

if (!is_first_frame_) {
    memcpy(cam_coord_pre_pre_, cam_coord_pre_, sizeof(double)
* 3);
}
is_first_frame_ = false;
memcpy(cam_coord_pre_, cam_coord_cur_, sizeof(double) * 3);
timestamp_pre_ = timestamp_cur_;
return updated;
}

```

LaneBasedCalibrator的校准过程。

```
bool LaneBasedCalibrator::Process(const EgoLane &lane, const
float &velocity,
                                    const float &yaw_rate,
                                    const float &time_diff) {
    float distance_traveled_in_meter = velocity * time_diff;
    float vehicle_yaw_changed = yaw_rate * time_diff;

    // 1. 检查是否直行
    if (!IsTravelingStraight(vehicle_yaw_changed)) {
        AINFO << "Do not calibrate if not moving straight: "
             << "yaw angle changed " << vehicle_yaw_changed;
        vp_buffer_.clear();
        return false;
    }

    VanishingPoint vp_cur;
    VanishingPoint vp_work;

    // 2. 从车道获取当前消失点的估计值
    if (!GetVanishingPoint(lane, &vp_cur)) {
        AINFO << "Lane is not valid for calibration.";
        return false;
    }
    vp_cur.distance_traveled = distance_traveled_in_meter;

    // Push vanishing point into buffer
    PushVanishingPoint(vp_cur);
    if (!PopVanishingPoint(&vp_work)) {
        AINFO << "Driving distance is not long enough";
        return false;
    }

    // 获取当前的俯仰估计
    pitch_cur_ = 0.0f;
    if (!GetPitchFromVanishingPoint(vp_work, &pitch_cur_)) {
        AINFO << "Failed to estimate pitch from vanishing
point.";
        return false;
    }
    vanishing_row_ = vp_work.pixel_pos[1];
```

```

// Get the filtered output using histogram
if (!AddPitchToHistogram(pitch_cur_)) {
    AINFO << "Calculated pitch is out-of-range.";
    return false;
}

accumulated_straight_driving_in_meter_ +=
distance_traveled_in_meter;

if (accumulated_straight_driving_in_meter_ >
    params_.min_distance_to_update_calibration_in_meter
&&
    pitch_histogram_.Process()) {
    pitch_estimation_ =
pitch_histogram_.get_val_estimation();
    const float cy = k_mat_[5];
    const float fy = k_mat_[4];
    vanishing_row_ = tanf(pitch_estimation_) * fy + cy;
    accumulated_straight_driving_in_meter_ = 0.0f;
    return true;
}
return false;
}

```

feature_extractor

TrackingFeatureExtractor追踪特征提取器。

```

bool TrackingFeatureExtractor::Init(
    const FeatureExtractorInitOptions &init_options) {
    // setup bottom and top
    int feat_height = init_options.feat_blob->shape(2);
    int feat_width = init_options.feat_blob->shape(3);
    input_height_ =
        init_options.input_height == 0 ? feat_height :
    init_options.input_height;
    input_width_ =
        init_options.input_width == 0 ? feat_width :
    init_options.input_width;
    tracking_feature::FeatureParam feat_param;

```

```

    std::string config_path = cyber::common::GetAbsolutePath(
        init_options.root_dir, init_options.conf_file);
    // 1. 获取配置
    if (!cyber::common::GetProtoFromFile(config_path,
    &feat_param)) {
        return false;
    }
    if (feat_param.extractor_size() != 1) {
        return false;
    }
    CHECK_EQ(input_height_ / feat_height, input_width_ /
feat_width)
        << "Invalid aspect ratio: " << feat_height << "x" <<
feat_width
        << " from " << input_height_ << "x" << input_width_;
}

// 2. 获取ROI区域pooling
feat_blob_ = init_options.feat_blob;
for (int i = 0; i < feat_param.extractor_size(); i++) {
    switch (feat_param.extractor(i).feat_type()) {
        case
tracking_feature::ExtractorParam_FeatureType_ROIPooling:
            init_roiPooling(init_options,
feat_param.extractor(i).roi_pooling_param());
            break;
    }
}
if (roi_poolings_.empty()) {
    AERROR << "no proper extractor";
    return false;
}

return true;
}

```

初始化感兴趣区域？

```

void TrackingFeatureExtractor::init_roiPooling(
    const FeatureExtractorInitOptions &options,
    const tracking_feature::ROIPoolingParam &param) {
int feat_channel = options.feat_blob->shape(1);
feat_height_ = options.feat_blob->shape(2);

```

```

feat_width_ = options.feat_blob->shape(3);

std::shared_ptr<FeatureExtractorLayer>
feature_extractor_layer_ptr;
feature_extractor_layer_ptr.reset(new
FeatureExtractorLayer());
std::vector<int> shape{1, 5};
feature_extractor_layer_ptr->rois_blob.reset(new
base::Blob<float>(shape));
int pooled_w = param.pooled_w();
int pooled_h = param.pooled_h();
bool use_floor = param.use_floor();
feature_extractor_layer_ptr->pooling_layer.reset(
    new inference::ROIPoolingLayer<float>(pooled_h,
pooled_w, use_floor, 1,
                                         feat_channel));
feature_extractor_layer_ptr->top_blob.reset(
    new base::Blob<float>(1, feat_blob_->channels(),
pooled_h, pooled_w));
roi_poolings_.push_back(feature_extractor_layer_ptr);
}

```

解压tracking特征

```

bool TrackingFeatureExtractor::Extract(const
FeatureExtractorOptions &options,
                                         CameraFrame *frame) {

if (frame->detected_objects.empty()) {
    return true;
}
if (!options.normalized) {
    encode_bbox(&(frame->detected_objects));
}
for (auto feature_extractor_layer_ptr : roi_poolings_) {
    feature_extractor_layer_ptr->rois_blob->Reshape(
        {static_cast<int>(frame->detected_objects.size()),
5});
    float *rois_data =
        feature_extractor_layer_ptr->rois_blob-
    >mutable_cpu_data();
    for (const auto &obj : frame->detected_objects) {
        rois_data[0] = 0;
    }
}

```

```

    rois_data[1] =
        obj->camera_supplement.box.xmin *
static_cast<float>(feat_width_);
    rois_data[2] =
        obj->camera_supplement.box.ymin *
static_cast<float>(feat_height_);
    rois_data[3] =
        obj->camera_supplement.box.xmax *
static_cast<float>(feat_width_);
    rois_data[4] =
        obj->camera_supplement.box.ymax *
static_cast<float>(feat_height_);
    ADEBUG << rois_data[0] << " " << rois_data[1] << " " <<
rois_data[2]
        << " " << rois_data[3] << " " << rois_data[4];
    rois_data += feature_extractor_layer_ptr->rois_blob-
>offset(1);
}
feature_extractor_layer_ptr->pooling_layer->ForwardGPU(
    {feat_blob_, feature_extractor_layer_ptr->rois_blob},
    {frame->track_feature_blob});

if (!options.normalized) {
    decode_bbox(&(frame->detected_objects));
}
norm_.L2Norm(frame->track_feature_blob.get());
return true;
}

```

ProjectFeature投影特征

```

bool ProjectFeature::Init(const FeatureExtractorInitOptions
&options) {
    std::string efx_config = GetAbsolutePath(options.root_dir,
options.conf_file);
    ACHECK(cyber::common::GetProtoFromFile(efx_config,
&param_))
        << "Read config failed: " << efx_config;
    AINFO << "Load config Success: " <<
param_.ShortDebugString();
    std::string proto_file =
        GetAbsolutePath(options.root_dir, param_.proto_file());

```

```

    std::string weight_file =
        GetAbsolutePath(options.root_dir,
param_.weight_file());
    std::vector<std::string> input_names;
    std::vector<std::string> output_names;
    input_names.push_back(param_.input_blob());
    output_names.push_back(param_.feat_blob());
    const auto &model_type = param_.model_type();
    AINFO << "model_type=" << model_type;
    inference_.reset(inference::CreateInferenceByName(
        model_type, proto_file, weight_file, output_names,
input_names,
        options.root_dir));
    ACHECK(nullptr != inference_) << "Failed to init
CNNAdapter";
    gpu_id_ = GlobalConfig::Instance()->track_feature_gpu_id;
    inference_->set_gpu_id(gpu_id_);
    inference_->set_max_batch_size(100);
    std::vector<int> shape = {5, 64, 3, 3};
    std::map<std::string, std::vector<int>> shape_map{
        {param_.input_blob(), shape}};
    ACHECK(inference_->Init(shape_map));
    inference_->Infer();
    return true;
}

```

解压特征

```

bool ProjectFeature::Extract(const FeatureExtractorOptions
&options,
                                CameraFrame *frame) {
    auto input_blob = inference_-
>get_blob(param_.input_blob());
    auto output_blob = inference_-
>get_blob(param_.feat_blob());
    if (frame->detected_objects.empty()) {
        return true;
    }
    input_blob->Reshape(frame->track_feature_blob->shape());
    cudaMemcpy(
        input_blob->mutable_gpu_data(), frame-
>track_feature_blob->gpu_data(),

```

```

        frame->track_feature_blob->count() * sizeof(float),
cudaMemcpyDefault);

cudaDeviceSynchronize();
inference_->Infer();
cudaDeviceSynchronize();
frame->track_feature_blob->Reshape(
    {static_cast<int>(frame->detected_objects.size()),
output_blob->shape(1),
    output_blob->shape(2), output_blob->shape(3)});

cudaMemcpy(
    frame->track_feature_blob->mutable_gpu_data(),
output_blob->gpu_data(),
    frame->track_feature_blob->count() * sizeof(float),
cudaMemcpyDefault);

norm_.L2Norm(frame->track_feature_blob.get());
return true;
}

```

ExternalFeatureExtractor扩展特征初始化

```

bool ExternalFeatureExtractor::Init(
    const FeatureExtractorInitOptions &options) {
    std::string efx_config = GetAbsolutePath(options.root_dir,
options.conf_file);
    ACHECK(cyber::common::GetProtoFromFile(efx_config,
&param_))
        << "Read config failed: " << efx_config;
    AINFO << "Load config Success: " <<
param_.ShortDebugString();
    std::string proto_file =
        GetAbsolutePath(options.root_dir, param_.proto_file());
    std::string weight_file =
        GetAbsolutePath(options.root_dir,
param_.weight_file());
    std::vector<std::string> input_names;
    std::vector<std::string> output_names;
    input_names.push_back(param_.input_blob());
    output_names.push_back(param_.feat_blob());
    height_ = param_.resize_height();
    width_ = param_.resize_width();
}

```

```

const auto &model_type = param_.model_type();
AINFO << "model_type=" << model_type;
inference_.reset(inference_::CreateInferenceByName(
    model_type, proto_file, weight_file, output_names,
input_names,
    options.root_dir));
ACHECK(nullptr != inference_) << "Failed to init
CNNAdapter";
gpu_id_ = GlobalConfig::Instance()->track_feature_gpu_id;
inference_->set_gpu_id(gpu_id_);
std::vector<int> shape = {1, height_, width_, 3};
std::map<std::string, std::vector<int>> shape_map{
    {param_.input_blob(), shape}};

ACHECK(inference_->Init(shape_map));
inference_->Infer();
InitFeatureExtractor(options.root_dir);
image_.reset(new base::Image8U(height_, width_,
base::Color::BGR));
return true;
}

```

扩展特征解压

```

bool ExternalFeatureExtractor::Extract(const
FeatureExtractorOptions &options,
                                         CameraFrame *frame) {
    int raw_height = frame->data_provider->src_height();
    int raw_width = frame->data_provider->src_width();
    auto input_blob = inference_-
>get_blob(param_.input_blob());
    DataProvider::ImageOptions image_options;
    image_options.target_color = base::Color::BGR;
    auto offset_y_ = static_cast<int>(
        param_.offset_ratio() * static_cast<float>(raw_height)
+ 0.5f);
    image_options.crop_roi =
        base::RectI(0, offset_y_, raw_width, raw_height -
offset_y_);
    image_options.do_crop = true;
    // Timer timer;
    frame->data_provider->GetImage(image_options,
image_.get());
}

```

```
inference::ResizeGPU(*image_, input_blob, raw_width, 0);
inference_->Infer();
FeatureExtractorOptions feat_options;
feat_options.normalized = false;
feature_extractor_->set_roi(
    image_options.crop_roi.x, image_options.crop_roi.y,
    image_options.crop_roi.width,
image_options.crop_roi.height);
feature_extractor_->Extract(feat_options, frame);
AINFO << "Extract Done";
return true;
}
```

Table of Contents

- 什么是CNN?
- CNN的原理
 - 卷积层(Convolutional Layer)
 - 池化层(Max Pooling Layer)
 - 全连接层(Fully Connected Layer)
- 如何构建CNN
- 基本概念
- 论文汇总
- 引用

什么是CNN?

首先什么是CNN呢？我们在这里模仿儿童的学习方式，当小孩子学习一个陌生东西的时候，往往是从问题开始，这里我们拿CNN做对比，来介绍什么是CNN。



妈妈，这是什么？

这是小狗。

小狗是什么？

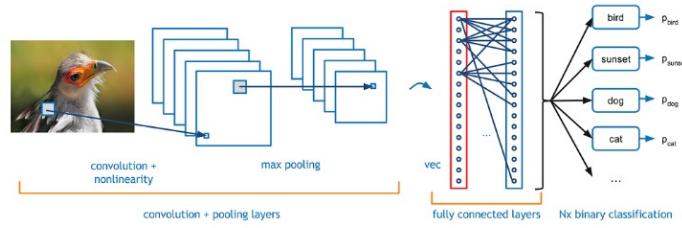
小狗是一种动物。

动物又是什么？

动物就是一种会动的生物，比如猫，狗啊都是动物。

那小狗有什么用呢？

小狗可以帮你看家，是人类的朋友。



这是什么？

这就是CNN(卷积神经网络)

CNN是什么？

CNN是一种神经网络。

神经网络又是什么？

神经网络是一种模仿生物神经网络(大脑)的结构和功能的数学模型或计算模型

CNN有什么用呢？

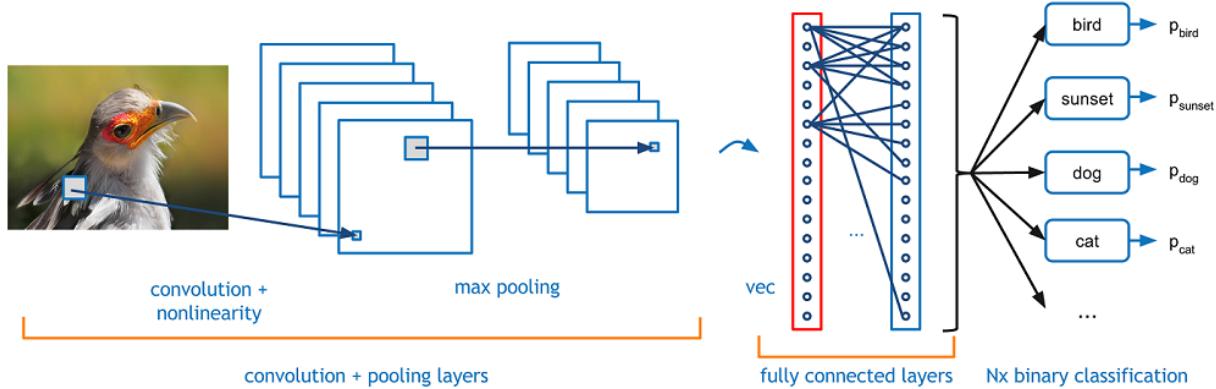
由于神经网络可以模拟人的判断能力，卷积神经网络在图像和语音识别方面能够给出更好的结果

miro

从上面的对话，我们知道CNN的全称是”Convolutional Neural Network”(卷积神经网络)。而神经网络是一种模仿生物神经网络（动物的中枢神经系统，特别是大脑）结构和功能的数学模型或计算模型。神经网络由大量的人工神经元组成，按不同的连接方式构建不同的网络。CNN是其中的一种，还有GAN(生成对抗网络)，RNN（递归神经网络）等，神经网络能够类似人一样具有简单的决定能力和简单的判断能力，在图像和语音识别方面能够给出更好的结果。

CNN的原理

CNN被广泛应用在图像识别领域，那么CNN是如何实现图像识别的呢？我们根据图中的例子来解释CNN的原理。



CNN是一种人工神经网络，CNN的结构可以分为3层：

1. 卷积层(**Convolutional Layer**) - 主要作用是提取特征。
2. 池化层(**Max Pooling Layer**) - 主要作用是下采样(downsampling)，却不会损坏识别结果。
3. 全连接层(**Fully Connected Layer**) - 主要作用是分类。

我们可以拿人类来做类比，比如你现在看到上图中的小鸟，人类如何识别它就是鸟的呢？首先你判断鸟的嘴是尖的，全身有羽毛和翅膀，有尾巴。然后通过这些联系起来判断这是一只鸟。而CNN的原理也类似，通过卷积层来查找特征，然后通过全连接层来做分类判断这是一只鸟，而池化层则是为了让训练的参数更少，在保持采样不变的情况下，忽略掉一些信息。

卷积层(**Convolutional Layer**)

那么卷基层是如何提取特征的呢？我们都知道卷积就是2个函数的叠加，应用在图像上，则可以理解为拿一个滤镜放在图像上，找出图像中的某些特征，而我们需要找到很多特征才能区分某一物体，所以我们会有很多滤镜，通过这些滤镜的组合，我们可以得出很多的特征。

首先一张图片在计算机中保存的格式为一个个的像素，比如一张长度为1080，宽度为1024的图片，总共包含了 $1080 * 1024$ 的像素，如果为RGB图片，因为RGB图片由3种颜色叠加而成，包含3个通道，因此我们需要用 $1080 * 1024 * 3$ 的数组来表示RGB图片。



我们看到的

29 10 19 20 00 00 12 22 13 14 90 89
78 27 39 00 00 00 12 90 87 89 19 19
22 32 65 56 89 11 22 33 11 15 67 28
00 29 38 48 21 22 27 81 92 33 13 13
11 77 45 45 37 89 91 23 12 38 74 63
46 41 31 37 13 12 46 46 24 32 44 57
84 38 38 74 63 46 41 31 37 13 12 46
46 24 32 44 57 84 38 15 67 28 00 29
38 48 21 22 27 81 92 33 13 13 11 77

电脑看到的

miro

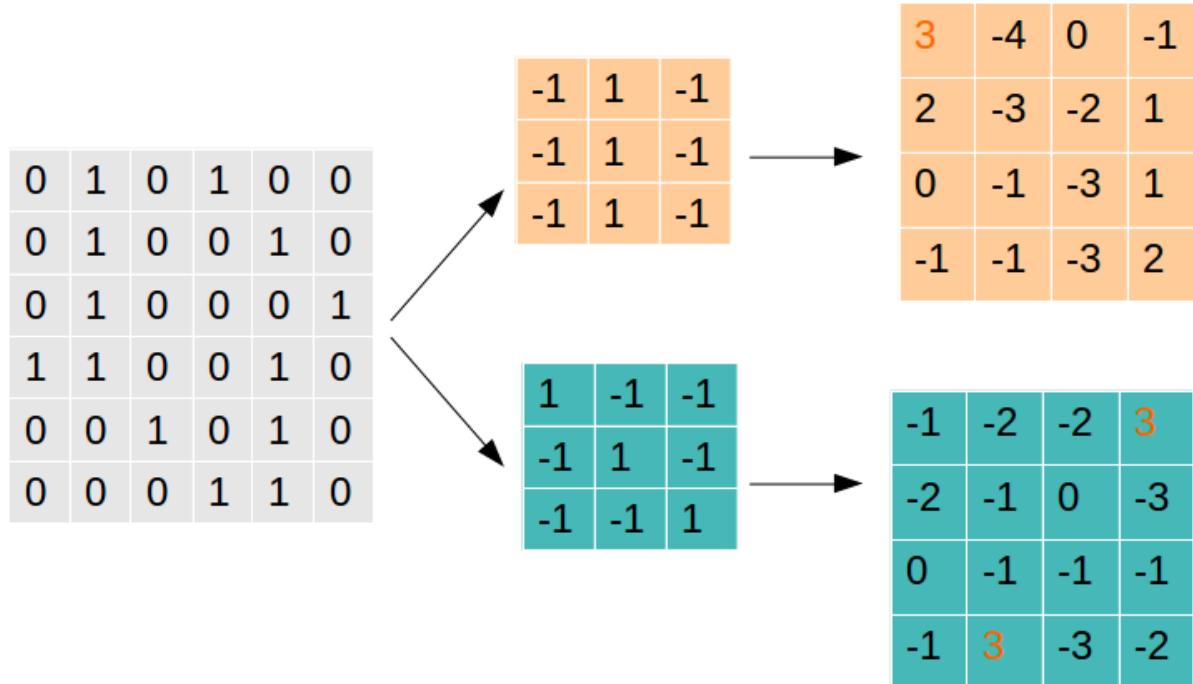
我们先从简单的情况开始考虑，假设我们有一组灰度图片，这样图片就可以表示为一个矩阵，假设我们的图片大小为 $5 * 5$ ，那么我们就可以得到一个 $5 * 5$ 的矩阵，接下来，我们用一组过滤器(Filter)来对图片过滤，过滤的过程就是求卷积的过程。假设我们的Filter的大小为 $3 * 3$ ，我们从图片的左上角开始移动Filter，并且把每次矩阵相乘的结果记录下来。可以通过下面的过程来演示。

1x1	1x0	1x1	0	0
0x0	1x1	1x0	1	0
0x1	0x0	1x1	1	1
0	0	1	1	0
0	1	1	0	0

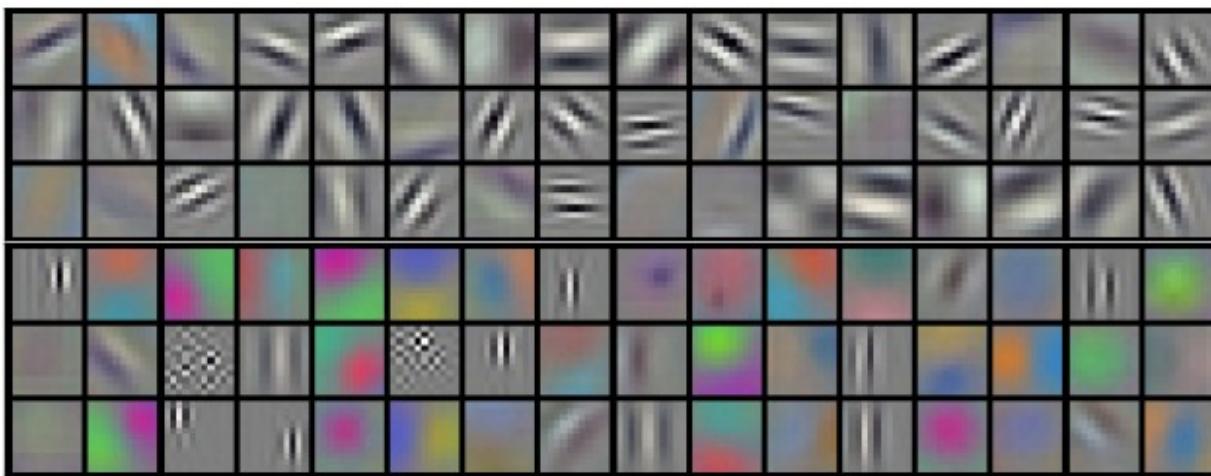
4		

每次Filter从矩阵的左上角开始移动，每次移动的步长是1，从左到右，从上到下，依次移动到矩阵末尾之后结束，每次都把Filter和矩阵对应的区域做乘法，得出一个新的矩阵。这其实就是做卷积的过程。而Filter的选择非常关键，Filter决定了过滤方式，通过不同的Filter会得

到不同的特征。举一个例子就是：

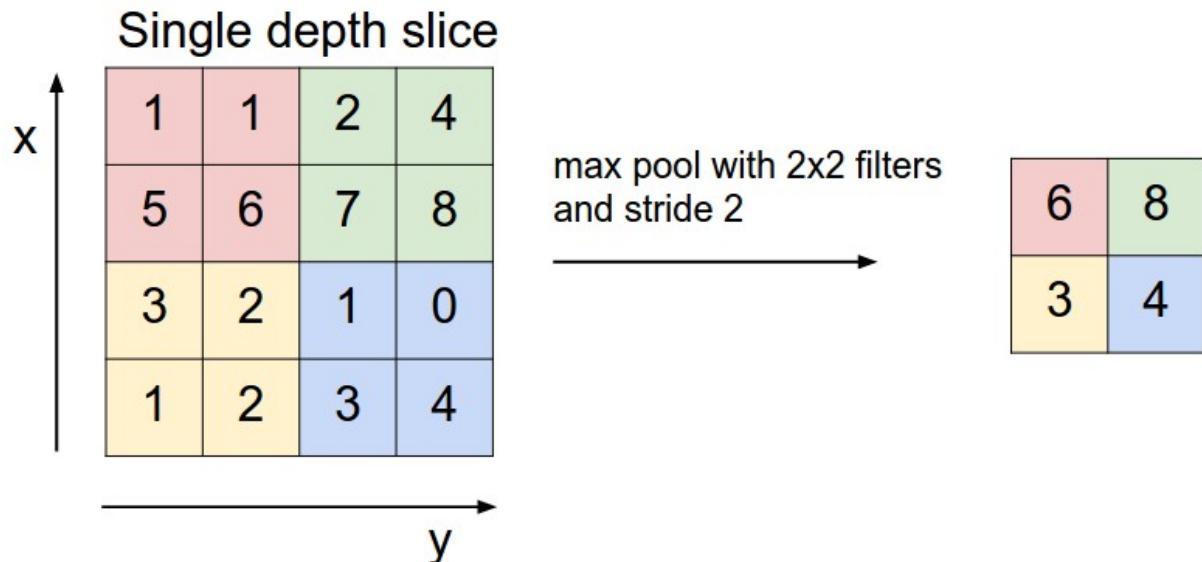


我们选择了2种Filter分别对图中的矩阵做卷积，可以看到值越大的就表示找到的特征越匹配，值越小的就表示找到的特征越偏离。Filter1主要是找到为”|”形状的特征，可以看到找到1处，转换后相乘值为3的网格就表示原始的图案中有”，而Filter2则表示找到””形状的特征，我们可以看到在图中可以找到2处。拿真实的图像举例子，我们经过卷积层的处理之后，得到如下的一些特征结果：



池化层(Max Pooling Layer)

经过卷积层处理的特征是否就可以直接用来分类了呢，答案是不能。我们假设一张图片的大小为 $500 * 500$ ，经过50个Filter的卷积层之后，得到的结果为 $500 * 500 * 50$ ，维度非常大，我们需要减少数据大小，而不会对识别的结果产生影响，即对卷积层的输出做下采样(downsampling)，这时候就引入了池化层。池化层的原理很简单，先看一个例子：



我们先从右边看起，可以看到把一个 $4 * 4$ 的矩阵按照 $2 * 2$ 做切分，每个 $2 * 2$ 的矩阵里，我们取最大的值保存下来，红色的矩阵里面最大值为6，所以输出为6，绿色的矩阵最大值为8，输出为8，黄色的为3，蓝色的为4。这样我们就把原来 $4 * 4$ 的矩阵变为了一个 $2 * 2$ 的矩阵。在看左边，我们发现原来 $224 * 224$ 的矩阵，缩小为 $112 * 112$ 了，减少了一半大小。

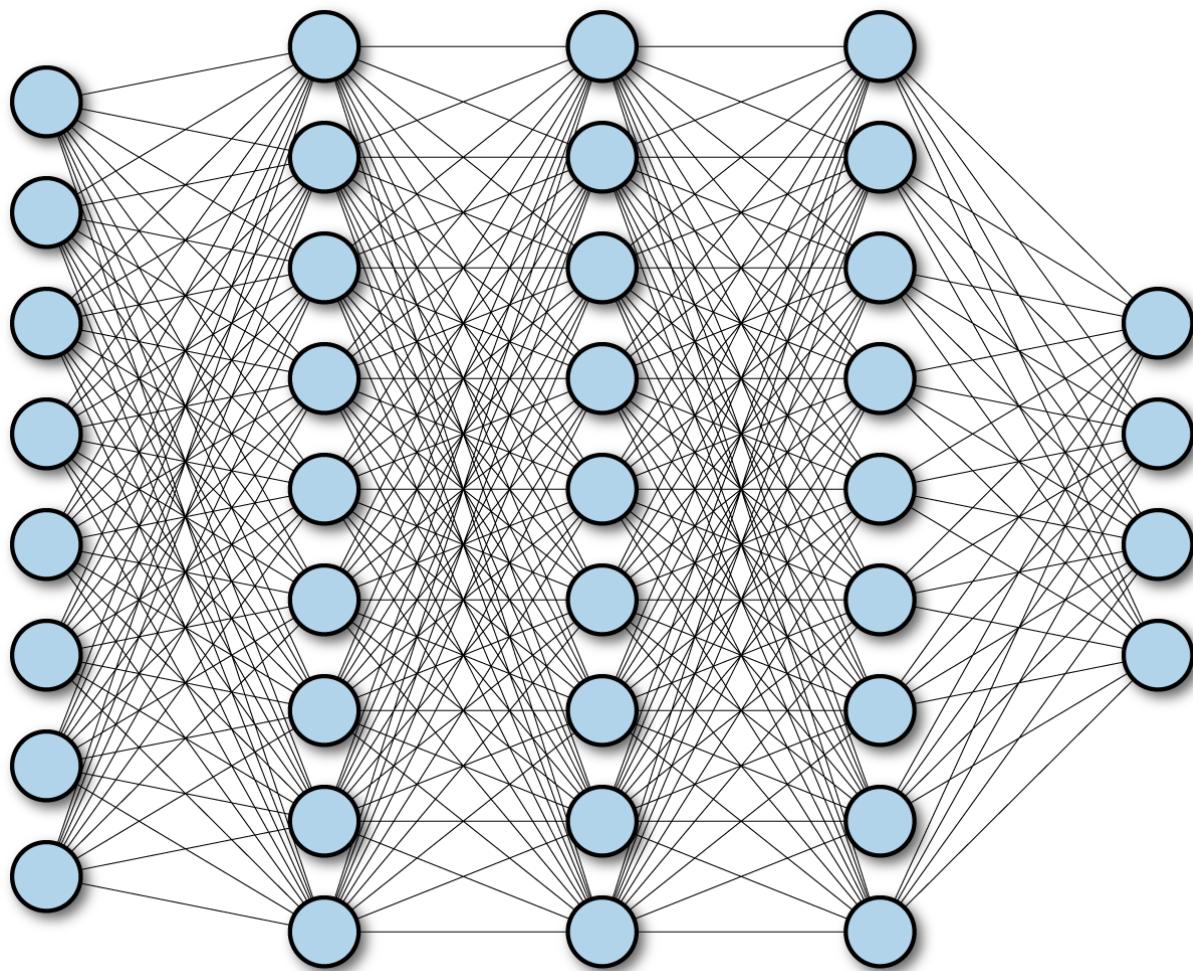
那么为什么这样做可行呢？丢失的一部分数据会不会对结果有影响，实际上，池化层不会对数据丢失产生影响，因为我们每次保留的输出都是局部最显著的一个输出，而池化之后，最显著的特征并没有丢失。我们只保留了认为最显著的特征，而把其他无用的信息丢掉，来减少运算。池化层的引入还保证了平移不变性，即同样的图像经过翻转变形之后，通过池化层，可以得到相似的结果。

既然是降采样，那么是否有其他方法实现降采样，也能达到同样的效果呢？当然有，通过其它的降采样方式，我们同样可以得到和池化层相同的结果，因此就可以拿这种方式替换掉池化层，可以起到相同的效果。

通常卷积层和池化层会重复多次形成具有多个隐藏层的网络，俗称深度神经网络。

全连接层(Fully Connected Layer)

全连接层的作用主要是进行分类。前面通过卷积和池化层得出的特征，在全连接层对这些总结好的特征做分类。全连接层就是一个完全连接的神经网络，根据权重每个神经元反馈的比重不一样，最后通过调整权重和网络得到分类的结果。



因为全连接层占用了神经网络80%的参数，因此对全连接层的优化就显得至关重要，现在也有用平均值来做最后的分类的。

如何构建CNN

现在我们已经清楚了CNN的原理，那么现在你想不想动手做一个CNN呢？下面我们通过tensorflow来实现一个CNN神经网络的例子：

1. 首先我们需要载入tensorflow环境， python代码如下：

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

# Imports
import numpy as np
import tensorflow as tf

tf.logging.set_verbosity(tf.logging.INFO)

# Our application logic will be added here

if __name__ == "__main__":
    tf.app.run()
```

其中tf.layers模块包含用于创建上述3种层的方法：

- conv2d()。构建一个二维卷积层。接受的参数为过滤器数量，过滤器核大小，填充和激活函数。
- max_pooling2d()。构建一个使用最大池化算法的二维池化层。接受的参数为池化过滤器大小和步长。
- dense()。构建全连接层，接受的参数为神经元数量和激活函数。
上述这些方法都接受张量作为输入，并返回转换后的张量作为输出。这样可轻松地将一个层连接到另一个层：只需从一个层创建方法中获取输出，并将其作为输入提供给另一个层即可。

输入层

对输入进行转换，输入的张量的形状应该为[batch_size, image_height, image_width, channels]。

- batch_size。在训练期间执行梯度下降法时使用的样本子集的大小。

- `image_height`。样本图像的高度。
- `image_width`。样本图像的宽度。
- `channels`。样本图像中颜色通道的数量。彩色图像有 3 个通道（红色、绿色、蓝色）。单色图像只有 1 个通道（黑色）。

```
input_layer = tf.reshape(features["x"], [-1, 28, 28, 1])
```

卷积层

我们的第一个卷积层创建32个 $5 * 5$ 的过滤器。

```
conv1 = tf.layers.conv2d(  
    inputs=input_layer,  
    filters=32,  
    kernel_size=[5, 5],  
    padding="same",  
    activation=tf.nn.relu)
```

池化层

接下来，我们将第一个池化层连接到刚刚创建的卷积层。

```
pool1 = tf.layers.max_pooling2d(inputs=conv1, pool_size=[2, 2], strides=2)
```

卷积层2和池化层2

对于卷积层 2，我们配置 64 个 5×5 过滤器，并应用 ReLU 激活函数；对于池化层 2，我们使用与池化层 1 相同的规格（一个 2×2 最大池化过滤器，步长为 2）：

```
conv2 = tf.layers.conv2d(  
    inputs=pool1,  
    filters=64,  
    kernel_size=[5, 5],  
    padding="same",  
    activation=tf.nn.relu)
```

```
pool2 = tf.layers.max_pooling2d(inputs=conv2, pool_size=[2, 2], strides=2)
```

全连接层

接下来，我们需要向CNN添加全连接层（具有1024个神经元和ReLU激活函数），以对卷积/池化层提取的特征执行分类。不过，在我们连接该层之前，我们需要先扁平化特征图(pool2)，以将其变形为[batch_size, features]，使张量只有两个维度：

```
pool2_flat = tf.reshape(pool2, [-1, 7 * 7 * 64])
```

在上面的 reshape() 操作中，-1 表示 batch_size 维度将根据输入数据中的样本数量动态计算。每个样本都具有 7 (pool2 高度) * 7 (pool2 宽度) * 64 (pool2 通道) 个特征。

```
dense = tf.layers.dense(inputs=pool2_flat, units=1024, activation=tf.nn.relu)
```

为了改善模型的结果，我们还会使用 layers 中的 dropout 方法，向密集层应用丢弃正则化：

```
dropout = tf.layers.dropout(
    inputs=dense, rate=0.4, training=mode ==
tf.estimator.ModeKeys.TRAIN)
```

对数层

我们的神经网络中的最后一层是对数层，该层返回预测的原始值。我们创建一个具有 10 个神经元（介于 0 到 9 之间的每个目标类别对应一个神经元）的密集层，并应用线性激活函数（默认函数）：

```
logits = tf.layers.dense(inputs=dropout, units=10)
```

生成预测

```
tf.argmax(input=logits, axis=1)
```

我们可以使用 `tf.nn.softmax` 应用 softmax 激活函数，以从对数层中得出概率：

```
tf.nn.softmax(logits, name="softmax_tensor")
```

我们将预测编译为字典，并返回：

```
predictions = {
    "classes": tf.argmax(input=logits, axis=1),
    "probabilities": tf.nn.softmax(logits,
name="softmax_tensor")
}
if mode == tf.estimator.ModeKeys.PREDICT:
    return tf.estimator.EstimatorSpec(mode=mode,
predictions=predictions)
```

基本概念

- 卷积核 - 卷积核就是图像处理时，给定输入图像，在输出图像中每一个像素是输入图像中一个小区域中像素的加权平均，其中权值由一个函数定义，这个函数称为卷积核。[kernel](#)
[[https://en.wikipedia.org/wiki/Kernel_\(image_processing\)](https://en.wikipedia.org/wiki/Kernel_(image_processing))]
- 卷积 卷积可以对应到2个函数叠加，因此用一个filter和图片叠加就可以求出整个图片的情况，可以用在图像的边缘检测，图片锐化，模糊等方面。[Convolution](#) [<https://en.wikipedia.org/wiki/Convolution>]

论文汇总

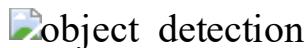
以下内容引用自[A Beginner's Guide to Convolutional Neural Networks \(CNNs\)](#) [<https://skymind.ai/wiki/convolutional-network>]，主要是为了整理和学习相关内容，从新整理了一遍。

ImageNet分类



- Microsoft (Deep Residual Learning) [Paper](#) [<https://arxiv.org/pdf/1512.03385v1.pdf>] [Slide](#) [http://image-net.org/challenges/talks/ilsvrc2015_deep_residual_learning_kaiminghe.pdf]
 - Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun, Deep Residual Learning for Image Recognition, arXiv:1512.03385.
- Microsoft (PReLU/Weight initialization) [Paper](#) [<https://arxiv.org/pdf/1502.01852.pdf>]
 - Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun, Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification, arXiv:1502.01852.
- Batch Normalization [Paper](#) [<https://arxiv.org/pdf/1502.03167.pdf>]
 - Sergey Ioffe, Christian Szegedy, Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, arXiv:1502.03167.
- GoogLeNet [Paper](#) [<http://arxiv.org/pdf/1409.4842>]
 - Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, Andrew Rabinovich, CVPR, 2015.
- VGG-Net [Web](#) [http://www.robots.ox.ac.uk/~vgg/research/very_deep/] [Paper](#) [<https://arxiv.org/pdf/1409.1556.pdf>]
 - Karen Simonyan and Andrew Zisserman, Very Deep Convolutional Networks for Large-Scale Visual Recognition, ICLR, 2015.
- AlexNet [Paper](#) [<http://papers.nips.cc/book/advances-in-neural-information-processing-systems-25-2012>]
 - Alex Krizhevsky, Ilya Sutskever, Geoffrey E. Hinton, ImageNet Classification with Deep Convolutional Neural Networks, NIPS, 2012.

物体检测(Object Detection)



- PVANET [paper](#) [<https://arxiv.org/pdf/1608.08021.pdf>] [Code](#) [<https://github.com/sanghoon/pva-faster-rcnn>]
 - Kye-Hyeon Kim, Sanghoon Hong, Byungseok Roh, Yeongjae Cheon, Minje Park, PVANET: Deep but Lightweight Neural Networks for Real-time Object Detection, arXiv:1608.08021

- OverFeat, NYU [Paper](#) [<https://arxiv.org/pdf/1312.6229.pdf>]
 - OverFeat: Integrated Recognition, Localization and Detection using Convolutional Networks, ICLR, 2014.
- R-CNN, UC Berkeley [Paper-CVPR14](#) [https://www.cv-foundation.org/openaccess/content_cvpr_2014/papers/Girshick_Rich_Feature_Hierarchies_2014_CVPR_paper.pdf] [Paper-arXiv14](#) [<https://arxiv.org/pdf/1311.2524.pdf>]
 - Ross Girshick, Jeff Donahue, Trevor Darrell, Jitendra Malik, Rich feature hierarchies for accurate object detection and semantic segmentation, CVPR, 2014.
- SPP, Microsoft Research [Paper](#) [<https://arxiv.org/pdf/1406.4729.pdf>]
 - Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun, Spatial Pyramid Pooling in Deep Convolutional Networks for Visual Recognition, ECCV, 2014.
- Fast R-CNN, Microsoft Research [Paper](#) [<https://arxiv.org/pdf/1504.08083.pdf>]
 - Ross Girshick, Fast R-CNN, arXiv:1504.08083.
- Faster R-CNN, Microsoft Research [Paper](#) [<https://arxiv.org/pdf/1506.01497.pdf>]
 - Shaoqing Ren, Kaiming He, Ross Girshick, Jian Sun, Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks, arXiv:1506.01497.
- R-CNN minus R, Oxford [Paper](#) [<https://arxiv.org/pdf/1506.06981.pdf>]
 - Karel Lenc, Andrea Vedaldi, R-CNN minus R, arXiv:1506.06981.
- End-to-end people detection in crowded scenes [Paper](#) [<https://arxiv.org/abs/1506.04878>]
 - Russell Stewart, Mykhaylo Andriluka, End-to-end people detection in crowded scenes, arXiv:1506.04878.
- You Only Look Once: Unified, Real-Time object Detection [Paper](#) [<https://arxiv.org/abs/1506.02640>], [Paper Version 2](#) [<https://arxiv.org/abs/1612.08242>], [C Code](#) [<https://github.com/pjreddie/darknet>], [Tensorflow Code](#) [<https://github.com/thtrieu/darkflow>]
 - Joseph Redmon, Santosh Divvala, Ross Girshick, Ali Farhadi, You Only Look Once: Unified, Real-Time Object Detection, arXiv:1506.02640
 - Joseph Redmon, Ali Farhadi (Version 2)
- Inside-Outside Net [Paper](#) [<https://arxiv.org/abs/1512.04143>]
 - Sean Bell, C. Lawrence Zitnick, Kavita Bala, Ross Girshick, Inside-Outside Net: Detecting Objects in Context with Skip Pooling and Recurrent Neural Networks

- Deep Residual Network (Current State-of-the-Art) [Paper](#) [<https://arxiv.org/abs/1512.03385>]
 - Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun, Deep Residual Learning for Image Recognition
- Weakly Supervised Object Localization with Multi-fold Multiple instance Learning [Paper](#) [<https://arxiv.org/pdf/1503.00949.pdf>]
- R-FCN [Paper](#) [<https://arxiv.org/abs/1605.06409>] [Code](#) [<https://github.com/dajifeng001/R-FCN>]
 - Jifeng Dai, Yi Li, Kaiming He, Jian Sun, R-FCN: Object Detection via Region-based Fully Convolutional Networks
- SSD [Paper](#) [<https://arxiv.org/pdf/1512.02325v2.pdf>] [Code](#) [<https://github.com/weiliu89/caffe/tree/ssd>]
 - Wei Liu1, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, Alexander C. Berg, SSD: Single Shot MultiBox Detector, arXiv:1512.02325
- Speed/accuracy trade-offs for modern convolutional object detectors [Paper](#) [<https://arxiv.org/pdf/1611.10012v1.pdf>]
 - Jonathan Huang, Vivek Rathod, Chen Sun, Menglong Zhu, Anoop Korattikara, Alireza Fathi, Ian Fischer, Zbigniew Wojna, Yang Song, Sergio Guadarrama, Kevin Murphy, Google Research, arXiv:1611.10012

视频分类(Video Classification)

- Nicolas Ballas, Li Yao, Pal Chris, Aaron Courville, “Delving Deeper into Convolutional Networks for Learning Video Representations”, ICLR 2016. [Paper](#) [<https://arxiv.org/pdf/1511.06432v4.pdf>]
- Michael Mathieu, camille couprie, Yann Lecun, “Deep Multi Scale Video Prediction Beyond Mean Square Error”, ICLR 2016. [Paper](#) [<https://arxiv.org/pdf/1511.05440v6.pdf>]

对象跟踪(Object Tracking)

- Seunghoon Hong, Tackgeun You, Suha Kwak, Bohyung Han, Online Tracking by Learning Discriminative Saliency Map with Convolutional Neural Network, arXiv:1502.06796. [Paper](#) [<https://arxiv.org/pdf/1502.06796.pdf>]

- Hanxi Li, Yi Li and Fatih Porikli, DeepTrack: Learning Discriminative Feature Representations by Convolutional Neural Networks for Visual Tracking, BMVC, 2014. [Paper](#) [<http://www.bmva.org/bmvc/2014/files/paper028.pdf>]
- N Wang, DY Yeung, Learning a Deep Compact Image Representation for Visual Tracking, NIPS, 2013. [Paper](#) [<http://winsty.net/papers/dlt.pdf>]
- N Wang, DY Yeung, Learning a Deep Compact Image Representation for Visual Tracking, NIPS, 2013. [Paper](#) [<http://winsty.net/papers/dlt.pdf>]
- Chao Ma, Jia-Bin Huang, Xiaokang Yang and Ming-Hsuan Yang, Hierarchical Convolutional Features for Visual Tracking, ICCV 2015 [Paper](#) [https://www.cv-foundation.org/openaccess/content_iccv_2015/papers/Ma_Hierarchical_Convolutional_Features_ICCV_2015_paper.pdf] [Code](#) [<https://github.com/jbhuang0604/CF2>]
- Lijun Wang, Wanli Ouyang, Xiaogang Wang, and Huchuan Lu, Visual Tracking with fully Convolutional Networks, ICCV 2015 [Paper](#) [http://202.118.75.4/lu/Paper/ICCV2015/iccv15_lijun.pdf] [Code](#) [<https://github.com/scott89/FCNT>]
- Hyeonseob Nam and Bohyung Han, Learning Multi-Domain Convolutional Neural Networks for Visual Tracking, [Paper](#) [<https://arxiv.org/pdf/1510.07945.pdf>] [Code](#) [<https://github.com/HyeonseobNam/MDNet>] [Project Page](#) [<http://cvlab.postech.ac.kr/research/mdnet/>]

底层视觉(Low-Level Vision)

超分辨率(Super-Resolution)

- 迭代图像重建(Iterative Image Reconstruction)
 - Sven Behnke: Learning Iterative Image Reconstruction. IJCAI, 2001. [Paper](#) [<http://www.ais.uni-bonn.de/behnke/papers/ijcai01.pdf>]
 - Sven Behnke: Learning Iterative Image Reconstruction in the Neural Abstraction Pyramid. International Journal of Computational Intelligence and Applications, vol. 1, no. 4, pp. 427-438, 2001. [Paper](#) [<http://www.ais.uni-bonn.de/behnke/papers/ijcia01.pdf>]
- Super-Resolution (SRCNN) [Web](#) [<http://mmlab.ie.cuhk.edu.hk/projects/SRCNN.html>] [Paper-ECCV14](#) [http://personal.ie.cuhk.edu.hk/~ccloy/files/eccv_2014_deepresolution.pdf] [Paper-arXiv15](#) [<https://arxiv.org/pdf/1501.00092.pdf>]

- Chao Dong, Chen Change Loy, Kaiming He, Xiaoou Tang, Learning a Deep Convolutional Network for Image Super-Resolution, ECCV, 2014.
 - Chao Dong, Chen Change Loy, Kaiming He, Xiaoou Tang, Image Super-Resolution Using Deep Convolutional Networks, arXiv:1501.00092.
- Very Deep Super-Resolution
 - Jiwon Kim, Jung Kwon Lee, Kyoung Mu Lee, Accurate Image Super-Resolution Using Very Deep Convolutional Networks, arXiv:1511.04587, 2015. [Paper](#) [<https://arxiv.org/abs/1511.04587>]
- Deeply-Recursive Convolutional Network
 - Jiwon Kim, Jung Kwon Lee, Kyoung Mu Lee, Deeply-Recursive Convolutional Network for Image Super-Resolution, arXiv:1511.04491, 2015. [Paper](#) [<https://arxiv.org/abs/1511.04491>]
- Casade-Sparse-Coding-Network
 - Zhaowen Wang, Ding Liu, Wei Han, Jianchao Yang and Thomas S. Huang, Deep Networks for Image Super-Resolution with Sparse Prior. ICCV, 2015. [Paper](#) [<http://www.ifp.illinois.edu/~dingliu2/iccv15/iccv15.pdf>] [Code](#) [<http://www.ifp.illinois.edu/~dingliu2/iccv15/>]
- Perceptual Losses for Super-Resolution
 - Justin Johnson, Alexandre Alahi, Li Fei-Fei, Perceptual Losses for Real-Time Style Transfer and Super-Resolution, arXiv:1603.08155, 2016. [Paper](#) [<https://arxiv.org/abs/1603.08155>] [Supplementary](#) [<https://cs.stanford.edu/people/jcjohns/papers/fast-style/fast-style-supp.pdf>]
- SRGAN
 - Christian Ledig, Lucas Theis, Ferenc Huszar, Jose Caballero, Andrew Cunningham, Alejandro Acosta, Andrew Aitken, Alykhan Tejani, Johannes Totz, Zehan Wang, Wenzhe Shi, Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network, arXiv:1609.04802v3, 2016. [Paper](#) [<https://arxiv.org/pdf/1609.04802v3.pdf>]
- Others
 - Osendorfer, Christian, Hubert Soyer, and Patrick van der Smagt, Image Super-Resolution with Fast Approximate Convolutional Sparse Coding, ICONIP, 2014. [Paper ICONIP-2014](#) [http://brml.org/uploads/tx_sibibtex/281.pdf]

其他应用(Other Applications)

- Optical Flow (FlowNet) [Paper](#) [<http://arxiv.org/pdf/1504.06852>]
 - Philipp Fischer, Alexey Dosovitskiy, Eddy Ilg, Philip Häusser, Caner Hazırbaş, Vladimir Golkov, Patrick van der Smagt, Daniel Cremers, Thomas Brox, FlowNet: Learning Optical Flow with Convolutional Networks, arXiv:1504.06852.
- Compression Artifacts Reduction [Paper-arXiv15](#) [<http://arxiv.org/pdf/1504.06993>]
 - Chao Dong, Yubin Deng, Chen Change Loy, Xiaoou Tang, Compression Artifacts Reduction by a Deep Convolutional Network, arXiv:1504.06993.
- Blur Removal
 - Christian J. Schuler, Michael Hirsch, Stefan Harmeling, Bernhard Schölkopf, Learning to Deblur, arXiv:1406.7444 [Paper](#) [<https://arxiv.org/pdf/1406.7444.pdf>]
 - Jian Sun, Wenfei Cao, Zongben Xu, Jean Ponce, Learning a Convolutional Neural Network for Non-uniform Motion Blur Removal, CVPR, 2015 [Paper](#) [<https://arxiv.org/pdf/1503.00593.pdf>]
- Image Deconvolution [Web](#) [<http://lxu.me/projects/dcnn/>] [Paper](#) [http://lxu.me/mypapers/dcnn_nips14.pdf]
 - Li Xu, Jimmy SJ. Ren, Ce Liu, Jiaya Jia, Deep Convolutional Neural Network for Image Deconvolution, NIPS, 2014.
- Deep Edge-Aware Filter [Paper](#) [<http://proceedings.mlr.press/v37/xub15.pdf>]
 - Li Xu, Jimmy SJ. Ren, Qiong Yan, Renjie Liao, Jiaya Jia, Deep Edge-Aware Filters, ICML, 2015.
- Computing the Stereo Matching Cost with a Convolutional Neural Network [Paper](#) [https://www.cv-foundation.org/openaccess/content_cvpr_2015/papers/Zbontar_Computing_the_Stereo_2015_CVPR_paper.pdf]
 - Jure Žbontar, Yann LeCun, Computing the Stereo Matching Cost with a Convolutional Neural Network, CVPR, 2015.
- Colorful Image Colorization Richard Zhang, Phillip Isola, Alexei A. Efros, ECCV, 2016 [Paper](#) [<http://arxiv.org/pdf/1603.08511.pdf>], [Code](#) [<https://github.com/richzhang/colorization>]
- Ryan Dahl, [Blog](#) [<https://tinyclouds.org/colorize/>]

- Feature Learning by Inpainting [Paper](https://arxiv.org/pdf/1604.07379v1.pdf) [<https://arxiv.org/pdf/1604.07379v1.pdf>]
[Code](https://github.com/pathak22/context-encoder) [<https://github.com/pathak22/context-encoder>]
 - Deepak Pathak, Philipp Krahenbuhl, Jeff Donahue, Trevor Darrell, Alexei A. Efros, Context Encoders: Feature Learning by Inpainting, CVPR, 2016

边缘检测(Edge Detection)



- Holistically-Nested Edge Detection [Paper](https://arxiv.org/pdf/1504.06375.pdf) [<https://arxiv.org/pdf/1504.06375.pdf>]
[Code](https://github.com/s9xie/hed) [<https://github.com/s9xie/hed>]
 - Saining Xie, Zhuowen Tu, Holistically-Nested Edge Detection, arXiv:1504.06375.
- DeepEdge [Paper](http://arxiv.org/pdf/1412.1123) [<http://arxiv.org/pdf/1412.1123>]
 - Gedas Bertasius, Jianbo Shi, Lorenzo Torresani, DeepEdge: A Multi-Scale Bifurcated Deep Network for Top-Down Contour Detection, CVPR, 2015.
- DeepContour [Paper](http://mc.eistar.net/UpLoadFiles/Papers/DeepContour_cvpr15.pdf) [http://mc.eistar.net/UpLoadFiles/Papers/DeepContour_cvpr15.pdf]
 - Wei Shen, Xinggang Wang, Yan Wang, Xiang Bai, Zhijiang Zhang, DeepContour: A Deep Convolutional Feature Learned by Positive-Sharing Loss for Contour Detection, CVPR, 2015.

语义分割(Semantic Segmentation)



- PASCAL VOC2012 Challenge Leaderboard (01 Sep. 2016)

		mean	submission	date
▶	SegModel [?]	81.8	23-Aug-2016	▼
▶	RRR-ResNet152-COCO [?]	81.7	25-Aug-2016	▼
▶	CentraleSupelec Deep G-CRF [?]	80.2	12-Aug-2016	
▶	CMT-FCN-ResNet-CRF [?]	80.0	02-Aug-2016	
▶	DeepLabv2-CRF [?]	79.7	06-Jun-2016	
▶	CASIA_SegResNet_CRF_COCO [?]	79.3	03-Jun-2016	
▶	LRR_4x_ResNet_COCO [?]	79.3	18-Jul-2016	
▶	Adelaide_VeryDeep_FCN_VOC [?]	79.1	13-May-2016	
▶	LRR_4x_COCO [?]	78.7	16-Jun-2016	
▶	CASIA_IVA_OASeg [?]	78.3	21-May-2016	
▶	Oxford_TVGV_HO_CRF [?]	77.9	16-Mar-2016	
▶	Adelaide_Context_CNN_CRF_COCO [?]	77.8	06-Nov-2015	
▶	CUHK_DPN_COCO [?]	77.5	22-Sep-2015	
▶	Adelaide_Context_CNN_CRF_COCO [?]	77.2	13-Aug-2015	
▶	DeepLab-CRF-Attention-DT [?]	76.3	03-Feb-2016	
▶	CentraleSuperBoundaries++ [?]	76.0	13-Jan-2016	
▶	LRR_4x_de_pyramid_VOC [?]	75.9	07-Jun-2016	
▶	DeepLab-CRF-Attention [?]	75.7	03-Feb-2016	
▶	Adelaide_Context_CNN_CRF_VOC [?]	75.3	30-Aug-2015	
▶	MSRA_BoxSup [?]	75.2	18-May-2015	
▶	MERL/umd_Deep_GCRF_COCO [?]	74.8	15-Jan-2016	
▶	POSTECH_DeconvNet_CRF_VOC [?]	74.8	18-Aug-2015	
▶	Oxford_TVGV_CRF_RNN_COCO [?]	74.7	22-Apr-2015	

from PASCAL VOC2012 leaderboards

[<http://host.robots.ox.ac.uk:8080/leaderboard/displaylb.php?challengeid=11&compid=6>]

- SEC: Seed, Expand and Constrain
 - Alexander Kolesnikov, Christoph Lampert, Seed, Expand and Constrain: Three Principles for Weakly-Supervised Image

Segmentation, ECCV, 2016. [Paper](#)

[<http://pub.ist.ac.at/~akolesnikov/files/ECCV2016/main.pdf>] [Code](#)

[<https://github.com/kolesman/SEC>]

- Adelaide
 - Guosheng Lin, Chunhua Shen, Ian Reid, Anton van den Hengel, Efficient piecewise training of deep structured models for semantic segmentation, arXiv:1504.01013. [Paper](#)
[<https://arxiv.org/pdf/1504.01013.pdf>] (1st ranked in VOC2012)
 - Guosheng Lin, Chunhua Shen, Ian Reid, Anton van den Hengel, Deeply Learning the Messages in Message Passing Inference, arXiv:1508.02108. [Paper](#) [<https://arxiv.org/pdf/1508.02108.pdf>] (4th ranked in VOC2012)
- Deep Parsing Network (DPN)
 - Ziwei Liu, Xiaoxiao Li, Ping Luo, Chen Change Loy, Xiaoou Tang, Semantic Image Segmentation via Deep Parsing Network, arXiv:1509.02634 / ICCV 2015 [Paper](#) [<https://arxiv.org/pdf/1509.02634.pdf>] (2nd ranked in VOC 2012)
- CentraleSuperBoundaries, INRIA [Paper](#) [<https://arxiv.org/pdf/1511.07386.pdf>]
 - Iasonas Kokkinos, Surpassing Humans in Boundary Detection using Deep Learning, arXiv:1411.07386 (4th ranked in VOC 2012)
- BoxSup [Paper](#) [<http://arxiv.org/pdf/1503.01640.pdf>]
 - Jifeng Dai, Kaiming He, Jian Sun, BoxSup: Exploiting Bounding Boxes to Supervise Convolutional Networks for Semantic Segmentation, arXiv:1503.01640. (6th ranked in VOC2012)
- POSTECH
 - Hyeyoung Noh, Seunghoon Hong, Bohyung Han, Learning Deconvolution Network for Semantic Segmentation, arXiv:1505.04366. [Paper](#) [<https://arxiv.org/pdf/1505.04366.pdf>] (7th ranked in VOC2012)
 - Seunghoon Hong, Hyeyoung Noh, Bohyung Han, Decoupled Deep Neural Network for Semi-supervised Semantic Segmentation, arXiv:1506.04924. [Paper](#) [<https://arxiv.org/pdf/1506.04924.pdf>]
 - Seunghoon Hong, Junhyuk Oh, Bohyung Han, and Honglak Lee, Learning Transferrable Knowledge for Semantic Segmentation with Deep Convolutional Neural Network, arXiv:1512.07928 [Paper](#) [<https://arxiv.org/pdf/1512.07928.pdf>] [Project Page](#)
[<http://cvlab.postech.ac.kr/research/transfernet/>]

- Conditional Random Fields as Recurrent Neural Networks [Paper](#)
[\[https://arxiv.org/pdf/1502.03240.pdf\]](https://arxiv.org/pdf/1502.03240.pdf)
 - Shuai Zheng, Sadeep Jayasumana, Bernardino Romera-Paredes, Vibhav Vineet, Zhizhong Su, Dalong Du, Chang Huang, Philip H. S. Torr, Conditional Random Fields as Recurrent Neural Networks, arXiv:1502.03240. (8th ranked in VOC2012)
- DeepLab
 - Liang-Chieh Chen, George Papandreou, Kevin Murphy, Alan L. Yuille, Weakly-and semi-supervised learning of a DCNN for semantic image segmentation, arXiv:1502.02734. [Paper](#)
[\[https://arxiv.org/pdf/1502.02734.pdf\]](https://arxiv.org/pdf/1502.02734.pdf) (9th ranked in VOC2012)
- Zoom-out [Paper](#) [https://www.cv-foundation.org/openaccess/content_cvpr_2015/papers/Mostajabi_Feedforward_Semantic_Segmentation_2015_CVPR_paper.pdf]
 - Mohammadreza Mostajabi, Payman Yadollahpour, Gregory Shakhnarovich, Feedforward Semantic Segmentation With Zoom-Out Features, CVPR, 2015
- Joint Calibration [Paper](#) [<https://arxiv.org/pdf/1507.01581.pdf>]
 - Holger Caesar, Jasper Uijlings, Vittorio Ferrari, Joint Calibration for Semantic Segmentation, arXiv:1507.01581.
- Fully Convolutional Networks for Semantic Segmentation [Paper-arXiv15](#) [<https://arxiv.org/pdf/1411.4038.pdf>]
 - Jonathan Long, Evan Shelhamer, Trevor Darrell, Fully Convolutional Networks for Semantic Segmentation, CVPR, 2015.
- Hypercolumn [Paper](#) [https://www.cv-foundation.org/openaccess/content_cvpr_2015/papers/Hariharan_Hypercolumns_for_Object_2015_CVPR_paper.pdf]
 - Bharath Hariharan, Pablo Arbelaez, Ross Girshick, Jitendra Malik, Hypercolumns for Object Segmentation and Fine-Grained Localization, CVPR, 2015.
- Deep Hierarchical Parsing [Paper](#) [https://www.cv-foundation.org/openaccess/content_cvpr_2015/papers/Sharma_Deep_Hierarchical_Parsing_2015_CVPR_paper.pdf]
 - Abhishek Sharma, Oncel Tuzel, David W. Jacobs, Deep Hierarchical Parsing for Semantic Segmentation, CVPR, 2015.
- Learning Hierarchical Features for Scene Labeling [Paper-ICML12](#) [<http://yann.lecun.com/exdb/publis/pdf/farabet-icml-12.pdf>] [Paper-PAMI13](#)

[<http://yann.lecun.com/exdb/publis/pdf/farabet-pami-13.pdf>]

- Clement Farabet, Camille Couprie, Laurent Najman, Yann LeCun, Scene Parsing with Multiscale Feature Learning, Purity Trees, and Optimal Covers, ICML, 2012.
- Clement Farabet, Camille Couprie, Laurent Najman, Yann LeCun, Learning Hierarchical Features for Scene Labeling, PAMI, 2013.
- University of Cambridge [Web](#) [<http://mi.eng.cam.ac.uk/projects/segnets/>]
 - Vijay Badrinarayanan, Alex Kendall and Roberto Cipolla “SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation.” arXiv preprint arXiv:1511.00561, 2015.
[Paper](#) [<https://arxiv.org/abs/1511.00561>]
- Alex Kendall, Vijay Badrinarayanan and Roberto Cipolla “Bayesian SegNet: Model Uncertainty in Deep Convolutional Encoder-Decoder Architectures for Scene Understanding.” arXiv preprint arXiv:1511.02680, 2015. [Paper](#) [<https://arxiv.org/abs/1511.00561>]
- Princeton
 - Fisher Yu, Vladlen Koltun, “Multi-Scale Context Aggregation by Dilated Convolutions”, ICLR 2016, [Paper](#) [<https://arxiv.org/pdf/1511.07122v2.pdf>]
- Univ. of Washington, Allen AI
 - Hamid Izadinia, Fereshteh Sadeghi, Santosh Kumar Divvala, Yejin Choi, Ali Farhadi, “Segment-Phrase Table for Semantic Segmentation, Visual Entailment and Paraphrasing”, ICCV, 2015, [Paper](#) [https://www.cv-foundation.org/openaccess/content_iccv_2015/papers/Izadinia_Segment-Phrase_Table_for_ICCV_2015_paper.pdf]
- INRIA
 - Iasonas Kokkinos, “Pusing the Boundaries of Boundary Detection Using deep Learning”, ICLR 2016, [Paper](#) [<https://arxiv.org/pdf/1511.07386v2.pdf>]
- UCSB
 - Niloufar Pourian, S. Karthikeyan, and B.S. Manjunath, “Weakly supervised graph based semantic segmentation by learning communities of image-parts”, ICCV, 2015, [Paper](#) [https://www.cv-foundation.org/openaccess/content_iccv_2015/papers/Pourian_Weakly_Supervised_Graph_ICCV_2015_paper.pdf]

视觉注意力和显著性(Visual Attention and Saliency)



- Mr-CNN [Paper](#) [https://www.cv-foundation.org/openaccess/content_cvpr_2015/papers/Liu_Predicting_Eye_Fixations_2015_CVPR_paper.pdf]
 - Nian Liu, Junwei Han, Dingwen Zhang, Shifeng Wen, Tianming Liu, Predicting Eye Fixations using Convolutional Neural Networks, CVPR, 2015.
-

Saurabh Singh, Derek Hoiem, David Forsyth, Learning a Sequential Search for Landmarks, CVPR, 2015.

- Multiple Object Recognition with Visual Attention [Paper](#) [<https://arxiv.org/pdf/1412.7755.pdf>]
 - Jimmy Lei Ba, Volodymyr Mnih, Koray Kavukcuoglu, Multiple Object Recognition with Visual Attention, ICLR, 2015.
- Recurrent Models of Visual Attention [Paper](#) [<http://papers.nips.cc/paper/5542-recurrent-models-of-visual-attention.pdf>]
 - Volodymyr Mnih, Nicolas Heess, Alex Graves, Koray Kavukcuoglu, Recurrent Models of Visual Attention, NIPS, 2014.

Learning a Sequential Search for Landmarks [Paper](#) [https://www.cv-foundation.org/openaccess/content_cvpr_2015/papers/Singh_Learning_a_Sequential_Search_for_Landmarks_2015_CVPR_paper.pdf]

[foundation.org/openaccess/content_cvpr_2015/papers/Singh_Learning_a_Sequential_2015_CVPR_paper.pdf](https://www.cv-foundation.org/openaccess/content_cvpr_2015/papers/Singh_Learning_a_Sequential_2015_CVPR_paper.pdf)

物体识别(Object Recognition)

- Weakly-supervised learning with convolutional neural networks [Paper](https://www.cv-foundation.org/openaccess/content_cvpr_2015/papers/Oquab_Is_Object_Localization_2015_CVPR_paper.pdf) [https://www.cv-foundation.org/openaccess/content_cvpr_2015/papers/Oquab_Is_Object_Localization_2015_CVPR_paper.pdf]
 - Maxime Oquab, Leon Bottou, Ivan Laptev, Josef Sivic, Is object localization for free? – Weakly-supervised learning with convolutional neural networks, CVPR, 2015.
- FV-CNN [Paper](https://www.cv-foundation.org/openaccess/content_cvpr_2015/papers/Cimpoi_Deep_Filter_Banks_2015_CVPR_paper.pdf) [https://www.cv-foundation.org/openaccess/content_cvpr_2015/papers/Cimpoi_Deep_Filter_Banks_2015_CVPR_paper.pdf]
 - Mircea Cimpoi, Subhransu Maji, Andrea Vedaldi, Deep Filter Banks for Texture Recognition and Segmentation, CVPR, 2015.

人体姿势估计(Human Pose Estimation)

- Zhe Cao, Tomas Simon, Shih-En Wei, and Yaser Sheikh, Realtime Multi-Person 2D Pose Estimation using Part Affinity Fields, CVPR, 2017.
- Leonid Pishchulin, Eldar Insafutdinov, Siyu Tang, Bjoern Andres, Mykhaylo Andriluka, Peter Gehler, and Bernt Schiele, Deepcut: Joint subset partition and labeling for multi person pose estimation, CVPR, 2016.
- Shih-En Wei, Varun Ramakrishna, Takeo Kanade, and Yaser Sheikh, Convolutional pose machines, CVPR, 2016.
- Alejandro Newell, Kaiyu Yang, and Jia Deng, Stacked hourglass networks for human pose estimation, ECCV, 2016.

- Tomas Pfister, James Charles, and Andrew Zisserman, Flowing convnets for human pose estimation in videos, ICCV, 2015.
- Jonathan J. Tompson, Arjun Jain, Yann LeCun, Christoph Bregler, Joint training of a convolutional network and a graphical model for human pose estimation, NIPS, 2014.

Understanding CNN



- Karel Lenc, Andrea Vedaldi, Understanding image representations by measuring their equivariance and equivalence, CVPR, 2015. [Paper](#) [https://www.cv-foundation.org/openaccess/content_cvpr_2015/papers/Lenc_Understanding_Image_Representations_2015_CVPR_paper.pdf]
- Anh Nguyen, Jason Yosinski, Jeff Clune, Deep Neural Networks are Easily Fooled: High Confidence Predictions for Unrecognizable Images, CVPR, 2015. [Paper](#) [https://www.cv-foundation.org/openaccess/content_cvpr_2015/papers/Nguyen_Deep_Neural_Networks_2015_CVPR_paper.pdf]
- Aravindh Mahendran, Andrea Vedaldi, Understanding Deep Image Representations by Inverting Them, CVPR, 2015. [Paper](#) [https://www.cv-foundation.org/openaccess/content_cvpr_2015/papers/Mahendran_Understanding_Deep_Image_2015_CVPR_paper.pdf]
- Bolei Zhou, Aditya Khosla, Agata Lapedriza, Aude Oliva, Antonio Torralba, Object Detectors Emerge in Deep Scene CNNs, ICLR, 2015. [arXiv Paper](#) [<https://arxiv.org/abs/1412.6856>]
- Alexey Dosovitskiy, Thomas Brox, Inverting Visual Representations with Convolutional Networks, arXiv, 2015. [Paper](#) [<https://arxiv.org/abs/1506.02753>]
- Matthew Zeiler, Rob Fergus, Visualizing and Understanding Convolutional Networks, ECCV, 2014. [Paper](#) [<https://cs.nyu.edu/~fergus/papers/zeilerECCV2014.pdf>]

Image and Language

Image Captioning



- Pay Less Attention with Lightweight and Dynamic Convolutions [Paper](#) [<https://arxiv.org/abs/1901.10430>]
- UCLA / Baidu [Paper](#) [<https://arxiv.org/pdf/1410.1090.pdf>]
 - Junhua Mao, Wei Xu, Yi Yang, Jiang Wang, Alan L. Yuille, Explain Images with Multimodal Recurrent Neural Networks, arXiv:1410.1090.
- Toronto [Paper](#) [<https://arxiv.org/pdf/1411.2539.pdf>]
 - Ryan Kiros, Ruslan Salakhutdinov, Richard S. Zemel, Unifying Visual-Semantic Embeddings with Multimodal Neural Language Models, arXiv:1411.2539.
- Berkeley [Paper](#) [<https://arxiv.org/pdf/1411.4389.pdf>]
 - Jeff Donahue, Lisa Anne Hendricks, Sergio Guadarrama, Marcus Rohrbach, Subhashini Venugopalan, Kate Saenko, Trevor Darrell, Long-term Recurrent Convolutional Networks for Visual Recognition and Description, arXiv:1411.4389.
- Google [Paper](#) [<https://arxiv.org/pdf/1411.4555.pdf>]
 - Oriol Vinyals, Alexander Toshev, Samy Bengio, Dumitru Erhan, Show and Tell: A Neural Image Caption Generator, arXiv:1411.4555.
- Stanford [Web](#) [<https://cs.stanford.edu/people/karpathy/deepimagesent/>] [Paper](#) [<https://cs.stanford.edu/people/karpathy/cvpr2015.pdf>]
 - Andrej Karpathy, Li Fei-Fei, Deep Visual-Semantic Alignments for Generating Image Description, CVPR, 2015.
- UML / UT [Paper](#) [<https://arxiv.org/pdf/1412.4729.pdf>]
 - Subhashini Venugopalan, Huijuan Xu, Jeff Donahue, Marcus Rohrbach, Raymond Mooney, Kate Saenko, Translating Videos to Natural Language Using Deep Recurrent Neural Networks, NAACL-HLT, 2015.
- CMU / Microsoft [Paper-arXiv](#) [<https://arxiv.org/pdf/1411.5654.pdf>]
 - Xinlei Chen, C. Lawrence Zitnick, Learning a Recurrent Visual Representation for Image Caption Generation, arXiv:1411.5654.
 - Xinlei Chen, C. Lawrence Zitnick, Mind's Eye: A Recurrent Visual Representation for Image Caption Generation, CVPR 2015

- Microsoft [Paper](#) [<https://arxiv.org/pdf/1411.4952.pdf>]
 - Hao Fang, Saurabh Gupta, Forrest Iandola, Rupesh Srivastava, Li Deng, Piotr Dollár, Jianfeng Gao, Xiaodong He, Margaret Mitchell, John C. Platt, C. Lawrence Zitnick, Geoffrey Zweig, From Captions to Visual Concepts and Back, CVPR, 2015.
- Univ. Montreal / Univ. Toronto [Web](#) [<http://kelvinxu.github.io/projects/capgen.html>]
 [Paper](#) [<http://www.cs.toronto.edu/~zemei/documents/captionAttn.pdf>]
 - Kelvin Xu, Jimmy Lei Ba, Ryan Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhutdinov, Richard S. Zemel, Yoshua Bengio, Show, Attend, and Tell: Neural Image Caption Generation with Visual Attention, arXiv:1502.03044 / ICML 2015
- Idiap / EPFL / Facebook [Paper](#) [<https://arxiv.org/pdf/1502.03671.pdf>]
 - Remi Lebret, Pedro O. Pinheiro, Ronan Collobert, Phrase-based Image Captioning, arXiv:1502.03671 / ICML 2015
- UCLA / Baidu [Paper](#) [<https://arxiv.org/pdf/1504.06692.pdf>]
 - Junhua Mao, Wei Xu, Yi Yang, Jiang Wang, Zhiheng Huang, Alan L. Yuille, Learning like a Child: Fast Novel Visual Concept Learning from Sentence Descriptions of Images, arXiv:1504.06692
- MS + Berkeley
 - Jacob Devlin, Saurabh Gupta, Ross Girshick, Margaret Mitchell, C. Lawrence Zitnick, Exploring Nearest Neighbor Approaches for Image Captioning, arXiv:1505.04467 [Paper](#) [<https://arxiv.org/pdf/1505.04467.pdf>]
 - Jacob Devlin, Hao Cheng, Hao Fang, Saurabh Gupta, Li Deng, Xiaodong He, Geoffrey Zweig, Margaret Mitchell, Language Models for Image Captioning: The Quirks and What Works, arXiv:1505.01809 [Paper]
- Adelaide [Paper](#) [<https://arxiv.org/pdf/1505.01809.pdf>]
 - Qi Wu, Chunhua Shen, Anton van den Hengel, Lingqiao Liu, Anthony Dick, Image Captioning with an Intermediate Attributes Layer, arXiv:1506.01144
- Tilburg [Paper](#) [<https://arxiv.org/pdf/1506.03694.pdf>]
 - Grzegorz Chrupala, Akos Kadar, Afra Alishahi, Learning language through pictures, arXiv:1506.03694
- Univ. Montreal [Paper](#) [<https://arxiv.org/pdf/1507.01053.pdf>]
 - Kyunghyun Cho, Aaron Courville, Yoshua Bengio, Describing Multimedia Content using Attention-based Encoder-Decoder

Networks, arXiv:1507.01053

- Cornell [Paper](#) [<https://arxiv.org/pdf/1508.02091.pdf>]
 - Jack Hessel, Nicolas Savva, Michael J. Wilber, Image Representations and New Domains in Neural Image Captioning, arXiv:1508.02091
- MS + City Univ. of HongKong [Paper](#) [https://www.cv-foundation.org/openaccess/content_iccv_2015/papers/Yao_Learning_Query_and_ICCV_2015_paper.pdf]
 - Ting Yao, Tao Mei, and Chong-Wah Ngo, “Learning Query and Image Similarities with Ranking Canonical Correlation Analysis”, ICCV, 2015

Video Captioning

- Berkeley [Web](#) [<http://jeffdonahue.com/lrcn/>] [Paper](#) [<https://arxiv.org/pdf/1411.4389.pdf>]
 - Jeff Donahue, Lisa Anne Hendricks, Sergio Guadarrama, Marcus Rohrbach, Subhashini Venugopalan, Kate Saenko, Trevor Darrell, Long-term Recurrent Convolutional Networks for Visual Recognition and Description, CVPR, 2015.
- UT / UML / Berkeley [Paper](#) [<https://arxiv.org/pdf/1412.4729.pdf>]
 - Subhashini Venugopalan, Huijuan Xu, Jeff Donahue, Marcus Rohrbach, Raymond Mooney, Kate Saenko, Translating Videos to Natural Language Using Deep Recurrent Neural Networks, arXiv:1412.4729.
- Microsoft [Paper](#) [<https://arxiv.org/pdf/1505.01861.pdf>]
 - Yingwei Pan, Tao Mei, Ting Yao, Houqiang Li, Yong Rui, Joint Modeling Embedding and Translation to Bridge Video and Language, arXiv:1505.01861.
- UT / Berkeley / UML [Paper](#) [<https://arxiv.org/pdf/1505.00487.pdf>]
 - Subhashini Venugopalan, Marcus Rohrbach, Jeff Donahue, Raymond Mooney, Trevor Darrell, Kate Saenko, Sequence to Sequence—Video to Text, arXiv:1505.00487.
- Univ. Montreal / Univ. Sherbrooke [Paper](#) [<https://arxiv.org/pdf/1502.08029.pdf>]
 - Li Yao, Atousa Torabi, Kyunghyun Cho, Nicolas Ballas, Christopher Pal, Hugo Larochelle, Aaron Courville, Describing Videos by Exploiting Temporal Structure, arXiv:1502.08029
- MPI / Berkeley [Paper](#) [<https://arxiv.org/pdf/1506.01698.pdf>]

- Anna Rohrbach, Marcus Rohrbach, Bernt Schiele, The Long-Short Story of Movie Description, arXiv:1506.01698
- Univ. Toronto / MIT [Paper](#) [<https://arxiv.org/pdf/1506.06724.pdf>]
 - Yukun Zhu, Ryan Kiros, Richard Zemel, Ruslan Salakhutdinov, Raquel Urtasun, Antonio Torralba, Sanja Fidler, Aligning Books and Movies: Towards Story-like Visual Explanations by Watching Movies and Reading Books, arXiv:1506.06724
- Univ. Montreal [Paper](#) [<https://arxiv.org/pdf/1507.01053.pdf>]
 - Kyunghyun Cho, Aaron Courville, Yoshua Bengio, Describing Multimedia Content using Attention-based Encoder-Decoder Networks, arXiv:1507.01053
- TAU / USC [paper](#) [<https://arxiv.org/pdf/1612.06950.pdf>]
 - Dotan Kaufman, Gil Levi, Tal Hassner, Lior Wolf, Temporal Tessellation for Video Annotation and Summarization, arXiv:1612.06950.

Question Answering



- Virginia Tech / MSR [Web](#) [<https://visualqa.org/>] [Paper](#) [<https://arxiv.org/pdf/1505.00468.pdf>]
 - Stanislaw Antol, Aishwarya Agrawal, Jiasen Lu, Margaret Mitchell, Dhruv Batra, C. Lawrence Zitnick, Devi Parikh, VQA: Visual Question Answering, CVPR, 2015 SUNw:Scene Understanding workshop.
- MPI / Berkeley [Web](#) [<https://www.mpi-inf.mpg.de/departments/computer-vision-and-multimodal-computing/research/vision-and-language/visual-turing-challenge/>] [Paper](#) [<https://arxiv.org/pdf/1505.01121.pdf>]
 - Mateusz Malinowski, Marcus Rohrbach, Mario Fritz, Ask Your Neurons: A Neural-based Approach to Answering Questions about Images, arXiv:1505.01121.
- Toronto [Paper](#) [<https://arxiv.org/pdf/1505.02074.pdf>] [Dataset](#) [<http://www.cs.toronto.edu/~mren/imageqa/data/cocoqa/>]
 - Mengye Ren, Ryan Kiros, Richard Zemel, Image Question Answering: A Visual Semantic Embedding Model and a New Dataset, arXiv:1505.02074 / ICML 2015 deep learning workshop.

- Baidu / UCLA [Paper](#) [<https://arxiv.org/pdf/1505.05612.pdf>] [Dataset] //todo
 - Hauyuan Gao, Junhua Mao, Jie Zhou, Zhiheng Huang, Lei Wang, Wei Xu, Are You Talking to a Machine? Dataset and Methods for Multilingual Image Question Answering, arXiv:1505.05612.
- POSTECH [Paper](#) [<https://arxiv.org/pdf/1511.05756.pdf>] [Project Page](#) [<http://cvlab.postech.ac.kr/research/dppnet/>]
 - Hyeonwoo Noh, Paul Hongsuck Seo, and Bohyung Han, Image Question Answering using Convolutional Neural Network with Dynamic Parameter Prediction, arXiv:1511.05765
- CMU / Microsoft Research [Paper](#) [<https://arxiv.org/pdf/1511.02274v2.pdf>]
 - Yang, Z., He, X., Gao, J., Deng, L., & Smola, A. (2015). Stacked Attention Networks for Image Question Answering. arXiv:1511.02274.
- MetaMind [Paper](#) [<https://arxiv.org/pdf/1603.01417v1.pdf>]
 - Xiong, Caiming, Stephen Merity, and Richard Socher. “Dynamic Memory Networks for Visual and Textual Question Answering.” arXiv:1603.01417 (2016).
- SNU + NAVER [Paper](#) [<https://arxiv.org/abs/1606.01455>]
 - Jin-Hwa Kim, Sang-Woo Lee, Dong-Hyun Kwak, Min-Oh Heo, Jeonghee Kim, Jung-Woo Ha, Byoung-Tak Zhang, Multimodal Residual Learning for Visual QA, arXiv:1606:01455
- UC Berkeley + Sony [Paper](#) [<https://arxiv.org/pdf/1606.01847.pdf>]
 - Akira Fukui, Dong Huk Park, Daylen Yang, Anna Rohrbach, Trevor Darrell, and Marcus Rohrbach, Multimodal Compact Bilinear Pooling for Visual Question Answering and Visual Grounding, arXiv:1606.01847
- Postech [Paper](#) [<https://arxiv.org/pdf/1606.03647.pdf>]
 - Hyeonwoo Noh and Bohyung Han, Training Recurrent Answering Units with Joint Loss Minimization for VQA, arXiv:1606.03647
- SNU + NAVER [Paper](#) [<https://arxiv.org/abs/1610.04325>]
 - Jin-Hwa Kim, Kyoung Woon On, Jeonghee Kim, Jung-Woo Ha, Byoung-Tak Zhang, Hadamard Product for Low-rank Bilinear Pooling, arXiv:1610.04325.

Image Generation

- Convolutional / Recurrent Networks

- Aäron van den Oord, Nal Kalchbrenner, Oriol Vinyals, Lasse Espeholt, Alex Graves, Koray Kavukcuoglu. “Conditional Image Generation with PixelCNN Decoders” [Paper](#)
[\[https://arxiv.org/pdf/1606.05328v2.pdf\]](https://arxiv.org/pdf/1606.05328v2.pdf) [Code](#)
[\[https://github.com/kundan2510/pixelCNN\]](https://github.com/kundan2510/pixelCNN)
 - Alexey Dosovitskiy, Jost Tobias Springenberg, Thomas Brox, “Learning to Generate Chairs with Convolutional Neural Networks”, CVPR, 2015. [Paper](#) [https://www.cv-foundation.org/openaccess/content_cvpr_2015/papers/Dosovitskiy_Learning_to_Generate_2015_CVPR_paper.pdf]
 - Karol Gregor, Ivo Danihelka, Alex Graves, Danilo Jimenez Rezende, Daan Wierstra, “DRAW: A Recurrent Neural Network For Image Generation”, ICML, 2015. [Paper](#)
[\[https://arxiv.org/pdf/1502.04623v2.pdf\]](https://arxiv.org/pdf/1502.04623v2.pdf)
- Adversarial Networks
 - Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, Yoshua Bengio, Generative Adversarial Networks, NIPS, 2014. [Paper](#)
[\[https://arxiv.org/abs/1406.2661\]](https://arxiv.org/abs/1406.2661)
 - Emily Denton, Soumith Chintala, Arthur Szlam, Rob Fergus, Deep Generative Image Models using a Laplacian Pyramid of Adversarial Networks, NIPS, 2015. [Paper](#)
[\[https://arxiv.org/abs/1506.05751\]](https://arxiv.org/abs/1506.05751)
 - Lucas Theis, Aäron van den Oord, Matthias Bethge, “A note on the evaluation of generative models”, ICLR 2016. [Paper](#)
[\[https://arxiv.org/abs/1511.01844\]](https://arxiv.org/abs/1511.01844)
 - Zhenwen Dai, Andreas Damianou, Javier Gonzalez, Neil Lawrence, “Variationally Auto-Encoded Deep Gaussian Processes”, ICLR 2016. [Paper](#) [<https://arxiv.org/pdf/1511.06455v2.pdf>]
 - Elman Mansimov, Emilio Parisotto, Jimmy Ba, Ruslan Salakhutdinov, “Generating Images from Captions with Attention”, ICLR 2016, [Paper](#) [<https://arxiv.org/pdf/1511.02793v2.pdf>]
 - Jost Tobias Springenberg, “Unsupervised and Semi-supervised Learning with Categorical Generative Adversarial Networks”, ICLR 2016, [Paper](#) [<https://arxiv.org/pdf/1511.06390v1.pdf>]
 - Harrison Edwards, Amos Storkey, “Censoring Representations with an Adversary”, ICLR 2016, [Paper](#)

- [<https://arxiv.org/pdf/1511.05897v3.pdf>]
- Takeru Miyato, Shin-ichi Maeda, Masanori Koyama, Ken Nakae, Shin Ishii, “Distributional Smoothing with Virtual Adversarial Training”, ICLR 2016, [Paper](#) [<https://arxiv.org/pdf/1507.00677v8.pdf>]
- Jun-Yan Zhu, Philipp Krahenbuhl, Eli Shechtman, and Alexei A. Efros, “Generative Visual Manipulation on the Natural Image Manifold”, ECCV 2016. [Paper](#) [<https://arxiv.org/pdf/1609.03552v2.pdf>] [Code](#) [<https://github.com/junyanz/iGAN>] [Video](#) [<https://youtu.be/9c4z6YsBGQ0>]
- Mixing Convolutional and Adversarial Networks
 - Alec Radford, Luke Metz, Soumith Chintala, “Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks”, ICLR 2016. [Paper](#) [<https://arxiv.org/pdf/1511.06434.pdf>]

Other Topics

- Visual Analogy [Paper](#) [<https://web.eecs.umich.edu/~honglak/nips2015-analogy.pdf>]
 - Scott Reed, Yi Zhang, Yuting Zhang, Honglak Lee, Deep Visual Analogy Making, NIPS, 2015
- Surface Normal Estimation [Paper](#) [https://www.cv-foundation.org/openaccess/content_cvpr_2015/papers/Wang_Designing_Deep_Networks_2015_CVPR_paper.pdf]
 - Xiaolong Wang, David F. Fouhey, Abhinav Gupta, Designing Deep Networks for Surface Normal Estimation, CVPR, 2015.
- Action Detection [Paper](#) [https://www.cv-foundation.org/openaccess/content_cvpr_2015/papers/Gkioxari_Finding_Action_Tubes_2015_CVPR_paper.pdf]
 - Georgia Gkioxari, Jitendra Malik, Finding Action Tubes, CVPR, 2015.
- Crowd Counting [Paper](#) [https://www.cv-foundation.org/openaccess/content_cvpr_2015/papers/Zhang_Cross-Scene_Crowd_Counting_2015_CVPR_paper.pdf]
 - Cong Zhang, Hongsheng Li, Xiaogang Wang, Xiaokang Yang, Cross-scene Crowd Counting via Deep Convolutional Neural Networks, CVPR, 2015.
- 3D Shape Retrieval [Paper](#) [https://www.cv-foundation.org/openaccess/content_cvpr_2015/papers/Wang_Sketch-

Based_3D_Shape_2015_CVPR_paper.pdf]

- Fang Wang, Le Kang, Yi Li, Sketch-based 3D Shape Retrieval using Convolutional Neural Networks, CVPR, 2015.
- Weakly-supervised Classification
 - Samaneh Azadi, Jiashi Feng, Stefanie Jegelka, Trevor Darrell, “Auxiliary Image Regularization for Deep CNNs with Noisy Labels”, ICLR 2016, [Paper](#) [<https://arxiv.org/pdf/1511.07069v2.pdf>]
- Artistic Style [Paper](#) [<https://arxiv.org/abs/1508.06576>] [Code](#) [<https://github.com/jcjohnson/neural-style>]
 - Leon A. Gatys, Alexander S. Ecker, Matthias Bethge, A Neural Algorithm of Artistic Style.
- Human Gaze Estimation
 - Xucong Zhang, Yusuke Sugano, Mario Fritz, Andreas Bulling, Appearance-Based Gaze Estimation in the Wild, CVPR, 2015. [Paper](#) [https://www.cv-foundation.org/openaccess/content_cvpr_2015/papers/Zhang_Appearance-Based_Gaze_Estimation_2015_CVPR_paper.pdf] [Website](#) [<https://www.mpi-inf.mpg.de/departments/computer-vision-and-multimodal-computing/research/gaze-based-human-computer-interaction/appearance-based-gaze-estimation-in-the-wild-mpiiigaze/>]
- Face Recognition
 - Yaniv Taigman, Ming Yang, Marc’Aurelio Ranzato, Lior Wolf, DeepFace: Closing the Gap to Human-Level Performance in Face Verification, CVPR, 2014. [Paper](#) [https://www.cs.toronto.edu/~ranzato/publications/taigman_cvpr14.pdf]
 - Yi Sun, Ding Liang, Xiaogang Wang, Xiaoou Tang, DeepID3: Face Recognition with Very Deep Neural Networks, 2015. [Paper](#) [<https://arxiv.org/abs/1502.00873>]
 - Florian Schroff, Dmitry Kalenichenko, James Philbin, FaceNet: A Unified Embedding for Face Recognition and Clustering, CVPR, 2015. [Paper](#) [<https://arxiv.org/abs/1503.03832>]
- Facial Landmark Detection
 - Yue Wu, Tal Hassner, KangGeon Kim, Gerard Medioni, Prem Natarajan, Facial Landmark Detection with Tweaked Convolutional Neural Networks, 2015. [Paper](#) [<https://arxiv.org/abs/1511.04031>] [Project](#) [https://talhassner.github.io/home/publication/2017_TPAMI_2]

Courses

- Deep Vision
 - [Stanford] [CS231n: Convolutional Neural Networks for Visual Recognition](http://cs231n.stanford.edu/) [<http://cs231n.stanford.edu/>]
 - [CUHK] [ELEG 5040: Advanced Topics in Signal Processing\(Introduction to Deep Learning\)](https://piazza.com/cuhk.edu.hk/spring2015/eleg5040/home).
[<https://piazza.com/cuhk.edu.hk/spring2015/eleg5040/home>]
- More Deep Learning
 - [Stanford] [CS224d: Deep Learning for Natural Language Processing](http://cs224d.stanford.edu/) [<http://cs224d.stanford.edu/>]
 - [Oxford] [Deep Learning by Prof. Nando de Freitas](https://www.cs.ox.ac.uk/people/nando.defreitas/machinelearning/)
[<https://www.cs.ox.ac.uk/people/nando.defreitas/machinelearning/>]
 - [NYU] [Deep Learning by Prof. Yann LeCun](https://cilvr.cs.nyu.edu/doku.php?id=courses:deeplearning2014:start)
[<https://cilvr.cs.nyu.edu/doku.php?id=courses:deeplearning2014:start>]

Books

- Free Online Books
 - [Deep Learning by Ian Goodfellow, Yoshua Bengio, and Aaron Courville](https://www.deeplearningbook.org/) [<https://www.deeplearningbook.org/>]
 - [Neural Networks and Deep Learning by Michael Nielsen](http://neurallnetworksanddeeplearning.com/)
[<http://neurallnetworksanddeeplearning.com/>]
 - [Deep Learning Tutorial by LISA lab, University of Montreal](http://deeplearning.net/tutorial/deeplearning.pdf)
[<http://deeplearning.net/tutorial/deeplearning.pdf>]

Videos

- Talks
 - Deep Learning, Self-Taught Learning and Unsupervised Feature Learning By Andrew Ng [video](https://www.youtube.com/watch?v=n1ViNeWhC24) [<https://www.youtube.com/watch?v=n1ViNeWhC24>]
 - Recent Developments in Deep Learning By Geoff Hinton [video](https://www.youtube.com/watch?v=vShMxxqtDDs)
[<https://www.youtube.com/watch?v=vShMxxqtDDs>]
 - The Unreasonable Effectiveness of Deep Learning by Yann LeCun [video](https://www.youtube.com/watch?v=sc-KbuZqGkI) [<https://www.youtube.com/watch?v=sc-KbuZqGkI>]

- Deep Learning of Representations by Yoshua Bengio [video](#)
[\[https://www.youtube.com/watch?v=4xsVFLnHC_0\]](https://www.youtube.com/watch?v=4xsVFLnHC_0)

Applications

- Adversarial Training
 - Code and hyperparameters for the paper “Generative Adversarial Networks” [Web](#) [<https://github.com/goodfeli/adversarial>]
- Understanding and Visualizing
 - Source code for “Understanding Deep Image Representations by Inverting Them,” CVPR, 2015. [Web](#) [<https://github.com/aravindhdm/deep-goggle>]
- Semantic Segmentation
 - Source code for the paper “Rich feature hierarchies for accurate object detection and semantic segmentation,” CVPR, 2014. [Web](#) [<https://github.com/rbgirshick/rcnn>]
 - Source code for the paper “Fully Convolutional Networks for Semantic Segmentation,” CVPR, 2015. [Web](#) [<https://github.com/longjon/caffe/tree/future>]
- Super-Resolution
 - Image Super-Resolution for Anime-Style-Art [Web](#) [<https://github.com/nagadomi/waifu2x>]
- Edge Detection
 - Source code for the paper “DeepContour: A Deep Convolutional Feature Learned by Positive-Sharing Loss for Contour Detection,” CVPR, 2015. [Web](#) [<https://github.com/shenwei1231/DeepContour>]
 - Source code for the paper “Holistically-Nested Edge Detection”, ICCV 2015. [Web](#) [<https://github.com/s9xie/hed>]

Blogs

- Deep down the rabbit hole: CVPR 2015 and beyond @Tombone’s Computer Vision Blog [Web](#) [<http://www.computervisionblog.com/2015/06/deep-down-rabbit-hole-cvpr-2015-and.html>]
- CVPR recap and where we’re going @ Zoya Bylinskii (MIT PhD Student)’s Blog [Web](#) [<http://zoyathinks.blogspot.kr/2015/06/cvpr-recap-and-where-were->

going.html]

- Facebook's AI Painting@Wired [Web](https://www.wired.com/2015/06/facebook-googles-fake-brains-spawn-new-visual-reality/) [https://www.wired.com/2015/06/facebook-googles-fake-brains-spawn-new-visual-reality/]
- Inceptionism: Going Deeper into Neural Networks@Google Research [Web](http://googleresearch.blogspot.kr/2015/06/inceptionism-going-deeper-into-neural.html) [http://googleresearch.blogspot.kr/2015/06/inceptionism-going-deeper-into-neural.html]
- Implementing Neural networks [Web](https://peterroelants.github.io/) [https://peterroelants.github.io/]

引用

[A Beginner's Guide to Convolutional Neural Networks \(CNNs\)](https://skymind.ai/wiki/convolutional-network).

[https://skymind.ai/wiki/convolutional-network]

[A Beginner's Guide To Understanding Convolutional Neural Networks](https://adeshpande3.github.io/A-Beginner%27s-Guide-To-Understanding-Convolutional-Neural-Networks/)

[https://adeshpande3.github.io/A-Beginner%27s-Guide-To-Understanding-Convolutional-Neural-Networks/]

[Convolutional Neural Networks \(CNNs / ConvNets\)](https://cs231n.github.io/convolutional-networks/).

[http://cs231n.github.io/convolutional-networks/]

[An intuitive guide to Convolutional Neural Networks](https://www.freecodecamp.org/news/an-intuitive-guide-to-convolutional-neural-networks-260c2de0a050/)

[https://www.freecodecamp.org/news/an-intuitive-guide-to-convolutional-neural-networks-260c2de0a050/]

[Fully Connected Deep Networks](https://www.oreilly.com/library/view/tensorflow-for-deep/9781491980446/ch04.html) [https://www.oreilly.com/library/view/tensorflow-for-deep/9781491980446/ch04.html]

[使用 Estimator 构建卷积神经网络](https://www.tensorflow.org/tutorials/estimators/cnn?hl=zh-cn) [https://www.tensorflow.org/tutorials/estimators/cnn?hl=zh-cn]

fusion

fusion传感器数据融合，整体的融合有硬件融合和数据融合，主要的工作是时间同步和数据融合。

app

lib

inference推理

深度学习模型包括训练和推理2个过程，训练模型的过程是通过设计神经网络，通过数据训练出模型的参数。而推理则是部署深度学习的过程，实际上目前训练深度学习主要用的是python语言，而部署的时候大部分会采用c++，并且通过gpu进行加速，而inference模块则实现了上述功能。

inference主要实现了tensorflow, caffe, paddlepaddle3种框架的实现，并且通过cuda进行计算加速。

caffe

paddlepaddle

tensorrt

目录介绍

```
|-- BUILD
|-- caffe // caffe框架
|-- inference.cc // 定义了推理接口
|-- inference_factory.cc // 推理工厂，用来创建推理器
|-- inference_factory.h
|-- inference_factory_test.cc
|-- inference.h
|-- inference_test.cc
|-- inference_test_data // 推理测试数据
|-- layer.cc // 定义了layer接口
|-- layer.h
|-- layer_test.cc
|-- operators // 算子? ?
|-- paddlepaddle // paddlepaddle框架
|-- tensorrt // tensorrt框架
|-- test
```

```
└── tools // yolo, lane等的例子  
└── utils // cuda加速工具
```

CreateInferenceByName

CreateInferenceByName可以创建3种形式的推理器caffe, paddlepaddle, tensorrt。这里的tensorrt就是英伟达的加速库吗？也就是说如果是其它模型则采用tensorrt部署， caffe和paddlepaddle则采用这2种高级别的api部署？

```
Inference *CreateInferenceByName(const std::string &name,  
                                  const std::string  
&proto_file,  
                                  const std::string  
&weight_file,  
                                  const  
std::vector<std::string> &outputs,  
                                  const  
std::vector<std::string> &inputs,  
                                  const std::string  
&model_root) {  
    if (name == "CaffeNet") {  
        return new CaffeNet(proto_file, weight_file, outputs,  
                           inputs);  
    } else if (name == "RTNet") {  
        return new RTNet(proto_file, weight_file, outputs,  
                          inputs);  
    } else if (name == "RTNetInt8") {  
        return new RTNet(proto_file, weight_file, outputs,  
                          inputs, model_root);  
    } else if (name == "PaddleNet") {  
        return new PaddleNet(proto_file, weight_file, outputs,  
                           inputs);  
    }  
    return nullptr;  
}
```

3种推理模型分别在caffe, paddlepaddle, tensorrt目录中，其中有用到cuda进行加速，其中”.cu”是cuda对C++的扩展。

lidar

app

app目录主要实现3个功能lidar_obstacle_detection, lidar_obstacle_segmentation, lidar_obstacle_tracking三个功能。

lib 目录

整个激光雷达的处理流程是什么？？？先分割，找地面，然后找障碍物？？？

classifier

ground_detector

map_manager

object_builder

object_filter_bank

pointcloud_preprocessor

roi_filter

感兴趣区域过滤

scene_manager

场景管理？？？

segmentation

分割

tracker

追踪

tools

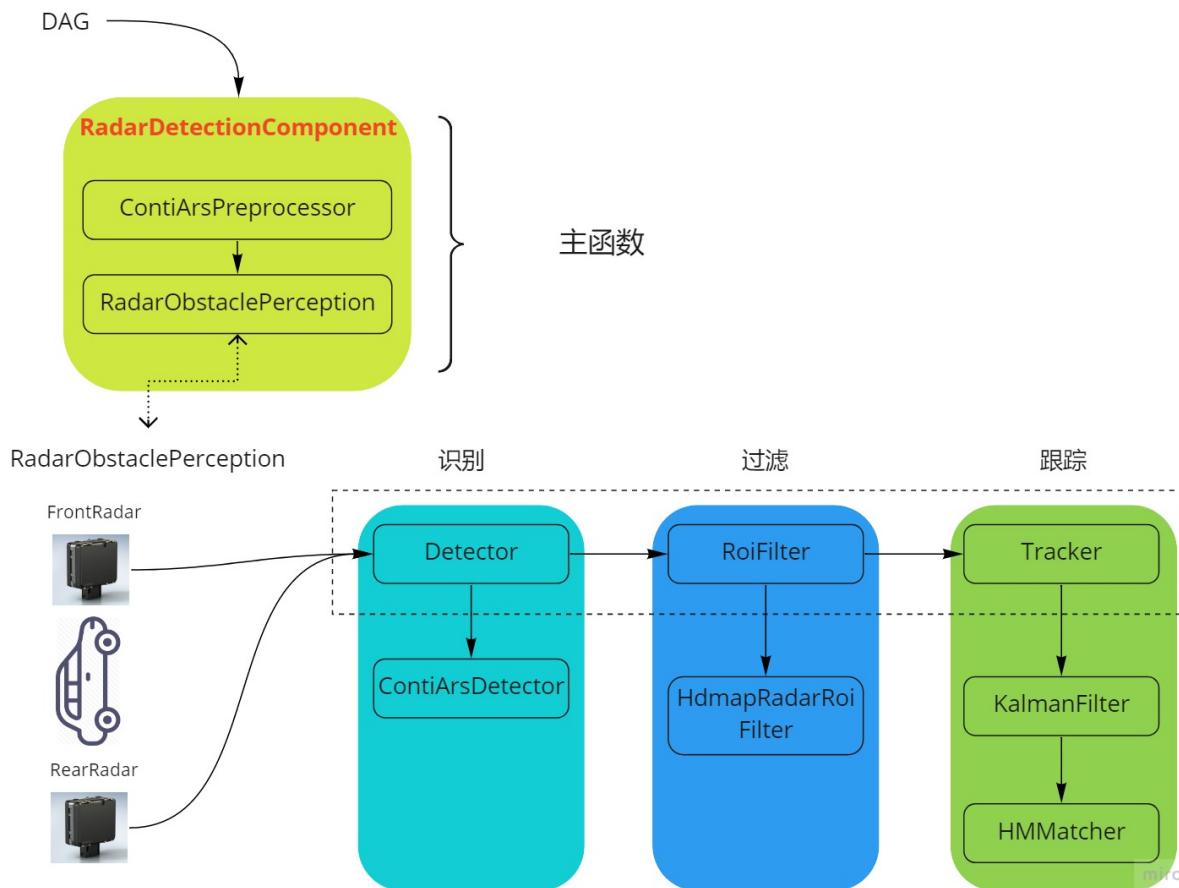
tools目录主要有2个工具，一个是OfflineLidarObstaclePerception，另一个是msg_exporter_main，下面分别介绍这2个工具的作用。

radar

毫米波雷达的处理流程相对比较简单，毫米波雷达的入口在”RadarDetectionComponent”中，在”RadarDetectionComponent”中首先对毫米波雷达数据做预处理(ContiArsPreprocessor)，实际上就是对毫米波雷达每一帧的时间做校正，之后在对毫米波雷达障碍物做识别”RadarObstaclePerception”，由于毫米波雷达可以直接输出障碍物的信息，识别的工作实际上已经简化了，识别出障碍物之后再根据ROI区域对检测结果做过滤，最后再采用卡尔曼滤波和匈牙利匹配对障碍物做追踪和匹配，上述这种追踪算法实际上是SORT算法，论文是2016年发表的”Simple Online and Realtime Tracking”。

下图是毫米波雷达模块的整个调用流程，可以看到在”RadarDetectionComponent”模块中实现了预处理(ContiArsPreprocessor)和障碍物检测(RadarObstaclePerception)。而障碍物检测”RadarObstaclePerception”在radar目录中实现，主要分为检测、

感兴趣区域过滤、追踪3个步骤。



疑问

不知道毫米波雷达对静态障碍物的检测和识别效果怎么样？

下面我们将主要介绍radar目录的具体实现，实际上radar,camera,lidar的目录结构都大体类似，在app中申明功能，在lib中实现。

```
.  
├── app           // 功能定义  
└── common  
└── lib  
    ├── detector   // 物体检测  
    ├── dummy  
    ├── interface  
    ├── preprocessor // 预处理  
    ├── roi_filter  // 感兴趣区域过滤  
    └── tracker      // 目标追踪  
        └── common
```

```
└── conti_ars_tracker
    └── filter          // 卡尔曼滤波
        └── matcher      // 匈牙利算法
```

app

毫米波雷达的实现在”radar_obstacle_perception.cc”中实现，实现的类为”RadarObstaclePerception”。主要调用了”lib”目录中的方法，来实现雷达检测目标的输出。

Init初始化

Init初始化中会指定预处理器，识别器，追踪器等几个组件，接下来毫米波雷达会采用上述组件进行障碍物的识别和追踪。

```
bool RadarObstaclePerception::Init(const std::string&
pipeline_name) {
    std::string model_name = pipeline_name;
    // 1. 读取配置
    const ModelConfig* model_config = nullptr;
    ACHECK(ConfigManager::Instance()-
>GetModelConfig(model_name, &model_config))
        << "not found model: " << model_name;

    std::string detector_name;
    ACHECK(model_config->get_value("Detector", &detector_name))
        << "Detector not found";

    std::string roi_filter_name;
    ACHECK(model_config->get_value("RoiFilter",
&roi_filter_name))
        << "RoiFilter not found";

    std::string tracker_name;
    ACHECK(model_config->get_value("Tracker", &tracker_name))
        << "Tracker not found";

    // 2. 给指定的识别器，过滤器，追踪器赋值
    BaseDetector* detector =
```

```

BaseDetectorRegisterer::GetInstanceByName(detector_name);
CHECK_NOTNULL(detector);
detector_.reset(detector);

BaseRoiFilter* roi_filter =
BaseRoiFilterRegisterer::GetInstanceByName(roi_filter_name);
CHECK_NOTNULL(roi_filter);
roi_filter_.reset(roi_filter);

BaseTracker* tracker =
BaseTrackerRegisterer::GetInstanceByName(tracker_name);
CHECK_NOTNULL(tracker);
tracker_.reset(tracker);

// 3. 识别器, 过滤器, 追踪器初始化
ACHECK(detector_->Init()) << "radar detector init error";
ACHECK(roi_filter_->Init()) << "radar roi filter init
error";
ACHECK(tracker_->Init()) << "radar tracker init error";

return true;
}

```

Perceive识别

接下来是整个识别流程，输入为障碍物，配置选项，输出为障碍物对象。其中处理器，识别器，追踪器具体的实现在”lib”目录中。

```

bool RadarObstaclePerception::Perceive(
    const drivers::ContiRadar& corrected_obstacles,
    const RadarPerceptionOptions& options,
    std::vector<base::ObjectPtr>* objects) {

    const std::string& sensor_name = options.sensor_name;
    base::FramePtr detect_frame_ptr(new base::Frame());
    // 1. 识别障碍物
    if (!detector_->Detect(corrected_obstacles,
        options.detector_options,
        detect_frame_ptr)) {

```

```

        AERROR << "radar detect error";
        return false;
    }

    // 2. 感兴趣区域过滤
    if (!roi_filter_->RoiFilter(options.roi_filter_options,
detect_frame_ptr)) {
        ADEBUG << "All radar objects were filtered out";
    }

    // 3. 追踪
    base::FramePtr tracker_frame_ptr =
std::make_shared<base::Frame>();
    if (!tracker_->Track(*detect_frame_ptr,
options.track_options,
                    tracker_frame_ptr)) {
        AERROR << "radar track error";
        return false;
    }

    // 4. 输出结果
*objects = tracker_frame_ptr->objects;

    return true;
}

```

lib 目录

preprocessor 预处理

预处理主要是对时间戳做预处理。

detector 识别

这里不是获取的雷达的raw_data而是直接获取的radar输出的感知结果。实现的类为”ContiArsDetector”主要实现了”Detect”方法。

```

bool ContiArsDetector::Detect(const drivers::ContiRadar&
corrected_obstacles,
                               const DetectorOptions& options,
                               base::FramePtr radar_frame) {
    // 1. 通过校准好的数据填充雷达帧
    RawObs2Frame(corrected_obstacles, options, radar_frame);
    radar_frame->timestamp =
    corrected_obstacles.header().timestamp_sec();
    radar_frame->sensor2world_pose = *
    (options.radar2world_pose);
    return true;
}

```

具体在雷达帧中填充了什么数据呢？下面我们来看下具体的实现。

```

void ContiArsDetector::RawObs2Frame(
    const drivers::ContiRadar& corrected_obstacles,
    const DetectorOptions& options, base::FramePtr
radar_frame) {
    // 1. 雷达到世界，雷达到IMU的位置关系，以及车的角速度
    const Eigen::Matrix4d& radar2world = *
    (options.radar2world_pose);
    const Eigen::Matrix4d& radar2novatel = *
    (options.radar2novatel_trans);
    const Eigen::Vector3f& angular_speed =
    options.car_angular_speed;
    Eigen::Matrix3d rotation_novatel;

    rotation_novatel << 0, -angular_speed(2), angular_speed(1),
    angular_speed(2),
    0, -angular_speed(0), -angular_speed(1),
    angular_speed(0), 0;
    Eigen::Matrix3d rotation_radar =
    radar2novatel.topLeftCorner(3, 3).inverse() *
    rotation_novatel *

    radar2novatel.topLeftCorner(3, 3);
    Eigen::Matrix3d radar2world_rotate = radar2world.block<3,
3>(0, 0);
    Eigen::Matrix3d radar2world_rotate_t =
    radar2world_rotate.transpose();
    // Eigen::Vector3d radar2world_translation =
    radar2world.block<3, 1>(0, 3);
}

```

```

ADEBUG << "radar2novatel: " << radar2novatel;
ADEBUG << "angular_speed: " << angular_speed;
ADEBUG << "rotation_radar: " << rotation_radar;
for (const auto radar_obs : corrected_obstacles.contiobs())
{
    base::ObjectPtr radar_object =
    std::make_shared<base::Object>();
    radar_object->id = radar_obs.obstacle_id();
    radar_object->track_id = radar_obs.obstacle_id();
    Eigen::Vector4d local_loc(radar_obs.longitude_dist(),
                             radar_obs.lateral_dist(), 0,
                             1);
    Eigen::Vector4d world_loc =
        static_cast<Eigen::Matrix<double, 4, 1, 0, 4, 1>>(
            radar2world *
local_loc);
    radar_object->center = world_loc.block<3, 1>(0, 0);
    radar_object->anchor_point = radar_object->center;

    Eigen::Vector3d local_vel(radar_obs.longitude_vel(),
                             radar_obs.lateral_vel(), 0);

    Eigen::Vector3d angular_trans_speed =
        rotation_radar * local_loc.topLeftCorner(3, 1);
    Eigen::Vector3d world_vel =
        static_cast<Eigen::Matrix<double, 3, 1, 0, 3, 1>>(
            radar2world_rotate * (local_vel +
angular_trans_speed));
    Eigen::Vector3d vel_temp =
        world_vel + options.car_linear_speed.cast<double>();
    radar_object->velocity = vel_temp.cast<float>();

    Eigen::Matrix3d dist_rms;
    dist_rms.setZero();
    Eigen::Matrix3d vel_rms;
    vel_rms.setZero();
    dist_rms(0, 0) = radar_obs.longitude_dist_rms();
    dist_rms(1, 1) = radar_obs.lateral_dist_rms();
    vel_rms(0, 0) = radar_obs.longitude_vel_rms();
    vel_rms(1, 1) = radar_obs.lateral_vel_rms();
    radar_object->center_uncertainty =
        (radar2world_rotate * dist_rms * dist_rms.transpose());
}

```

```

*
    radar2world_rotate_t)
        .cast<float>();

radar_object->velocity_uncertainty =
    (radar2world_rotate * vel_rms * vel_rms.transpose() *
     radar2world_rotate_t)
        .cast<float>();
    double local_obj_theta = radar_obs.oritation_angle() /
180.0 * PI;
    Eigen::Vector3f direction(static_cast<float>
(cos(local_obj_theta)),
                                static_cast<float>
(sin(local_obj_theta)), 0.0f);
    direction = radar2world_rotate.cast<float>() * direction;
    radar_object->direction = direction;
    radar_object->theta = std::atan2(direction(1),
direction(0));
    radar_object->theta_variance =
        static_cast<float>(radar_obs.oritation_angle_rms() /
180.0 * PI);
    radar_object->confidence = static_cast<float>
(radar_obs.probexist());

int motion_state = radar_obs.dynprop();
double prob_target = radar_obs.probexist();
if ((prob_target > MIN_PROBEXIST) &&
    (motion_state == CONTI_MOVING || motion_state ==
CONTI_ONCOMING ||
     motion_state == CONTI_CROSSING_MOVING)) {
    radar_object->motion_state = base::MotionState::MOVING;
} else if (motion_state == CONTI_DYNAMIC_UNKNOWN) {
    radar_object->motion_state =
base::MotionState::UNKNOWN;
} else {
    radar_object->motion_state =
base::MotionState::STATIONARY;
    radar_object->velocity.setZero();
}

int cls = radar_obs.obstacle_class();
if (cls == CONTI_CAR || cls == CONTI_TRUCK) {
    radar_object->type = base::ObjectType::VEHICLE;
}

```

```

    } else if (cls == CONTI_PEDESTRIAN) {
        radar_object->type = base::ObjectType::PEDESTRIAN;
    } else if (cls == CONTI_MOTOCYCLE || cls ==
CONTI_BICYCLE) {
        radar_object->type = base::ObjectType::BICYCLE;
    } else {
        radar_object->type = base::ObjectType::UNKNOWN;
    }

    radar_object->size(0) = static_cast<float>
(radar_obs.length());
    radar_object->size(1) = static_cast<float>
(radar_obs.width());
    radar_object->size(2) = 2.0f; // vehicle template (pnc
required)
    if (cls == CONTI_POINT) {
        radar_object->size(0) = 1.0f;
        radar_object->size(1) = 1.0f;
    }
    // extreme case protection
    if (radar_object->size(0) * radar_object->size(1) < 1.0e-
4) {
        if (cls == CONTI_CAR || cls == CONTI_TRUCK) {
            radar_object->size(0) = 4.0f;
            radar_object->size(1) = 1.6f; // vehicle template
        } else {
            radar_object->size(0) = 1.0f;
            radar_object->size(1) = 1.0f;
        }
    }
    MockRadarPolygon(radar_object);

    float local_range = static_cast<float>
(local_loc.head(2).norm());
    float local_angle =
        static_cast<float>(std::atan2(local_loc(1),
local_loc(0)));
    radar_object->radar_supplement.range = local_range;
    radar_object->radar_supplement.angle = local_angle;

    radar_frame->objects.push_back(radar_object);

ADEBUG << "obs_id: " << radar_obs.obstacle_id() << ", "

```

```

        << "long_dist: " << radar_obs.longitude_dist() <<
", "
        << "lateral_dist: " << radar_obs.lateral_dist() <<
", "
        << "long_vel: " << radar_obs.longitude_vel() << ", "
"
        << "lateral_vel: " << radar_obs.lateral_vel() << ", "
"
        << "rcs: " << radar_obs.rcs() << ", "
        << "meas_state: " << radar_obs.meas_state();
    }
}

```

roi_filter 过滤

根据地图过滤感兴趣的区域。主要的实现
在”HdmapRadarRoiFilter”中，主要是判断障碍物是否在感兴趣区域之
内。

```

bool HdmapRadarRoiFilter::RoiFilter(const RoiFilterOptions&
options,
                                      base::FramePtr
radar_frame) {
    std::vector<base::ObjectPtr> origin_objects = radar_frame-
>objects;
    // 1. 判断障碍物是否在感兴趣区域之内
    return common::ObjectInRoiCheck(options.roi,
origin_objects,
                                      &radar_frame->objects);
}

```

疑问

1. 感兴趣区域是如何组成的，为何定义的是点云的格式？？？

tracker 追踪

追踪的主要实现在”ContiArsTracker”中，用的算法是SORT算法，先对目标做检测，然后用卡尔曼滤波对物体的运动做估计，然后用匈牙利算法对多个目标做匹配，得到多个目标的跟踪结果。

ContiArsTracker

初始化

```
bool ContiArsTracker::Init() {
    std::string model_name = name_;
    const lib::ModelConfig *model_config = nullptr;
    bool state = true;
    if (!lib::ConfigManager::Instance()->GetModelConfig(model_name,
&model_config)) {
        AERROR << "not found model: " << model_name;
        state = false;
    }
    if (!model_config->get_value("tracking_time_window",
&s_tracking_time_win_)) {
        AERROR << "track_time_window is not found.";
        state = false;
    }
    if (!model_config->get_value("macher_name",
&matcher_name_)) {
        AERROR << "macher_name is not found.";
        state = false;
    }
    std::string chosen_filter;
    if (!model_config->get_value("chosen_filter",
&chosen_filter)) {
        AERROR << "chosen_filter is not found.";
        state = false;
    }
    RadarTrack::SetChosenFilter(chosen_filter);
    int tracked_times_threshold;
    if (!model_config->get_value("tracked_times_threshold",
&tracked_times_threshold)) {
        AERROR << "tracked_times_threshold is not found.";
        state = false;
    }
}
```

```

}

RadarTrack::SetTrackedTimesThreshold(tracked_times_threshold)
;
bool use_filter;
if (!model_config->get_value("use_filter", &use_filter)) {
    AERROR << "use_filter is not found.";
    state = false;
}
RadarTrack::SetUseFilter(use_filter);
// Or use register class instead.
if (matcher_name_ == "HMM matcher") {
    matcher_ = new HMM matcher();
    matcher_->Init(); // use proto later
} else {
    AERROR << "Not supported matcher : " << matcher_name_;
    state = false;
}

track_manager_ = new RadarTrackManager();
ACHECK(track_manager_ != nullptr)
    << "Failed to get RadarTrackManager instance.";
return state;
}

```

追踪

```

bool ContiArsTracker::Track(const base::Frame
&detected_frame,
                           const TrackerOptions &options,
                           base::FramePtr tracked_frame) {
    TrackObjects(detected_frame);
    CollectTrackedFrame(tracked_frame);
    return true;
}

```

AdaptiveKalmanFilter

自适应卡尔曼滤波器

HMM matcher

匈牙利匹配

1. 配置文件目录

配置文件

在”modules\perception\production\data\perception\radar\models\tracker\hm_matcher.conf”中。

```
max_match_distance : 2.5  
bound_match_distance : 10.0
```

Match

首先是找到没有追踪到的目标，和丢失追踪的目标？这里的”unassigned_tracks”和”unassigned_objects”如何定义呢？

```
bool HMM matcher::Match(const std::vector<RadarTrackPtr>  
&radar_tracks,  
                        const base::Frame &radar_frame,  
                        const TrackObjectMatcherOptions  
&options,  
                        std::vector<TrackObjectPair>  
*assignments,  
                        std::vector<size_t> *unassigned_tracks,  
                        std::vector<size_t>  
*unassigned_objects) {  
    // 1. traceid相等，并且距离小于max_match_distance  
    IDMatch(radar_tracks, radar_frame, assignments,  
    unassigned_tracks,  
            unassigned_objects);  
    TrackObjectPropertyMatch(radar_tracks, radar_frame,  
    assignments,  
                            unassigned_tracks,  
    unassigned_objects);  
    return true;  
}
```

HMM matcher

雷达障碍物的匹配通过HMMatcher来实现，本质上采用的是匈牙利算法，这里的实现稍微有点变化，下面我们就着重分析下整个匹配算法。

HMMatcher初始化Init()，Init获取”hm_matcher.conf”配置文件路径”modules\perception\production\data\perception\radar\models\tracker\hm_matcher.conf”，并读取”max_match_distance”和”bound_match_distance”的值。这里默认值为

```
max_match_distance : 2.5  
bound_match_distance : 10.0
```

那么接下来看HMMatcher如何来进行匹配的。

```
bool HMMatcher::Match(const std::vector<RadarTrackPtr>  
&radar_tracks,  
                      const base::Frame &radar_frame,  
                      const TrackObjectMatcherOptions  
&options,  
                      std::vector<TrackObjectPair>  
*assignments,  
                      std::vector<size_t> *unassigned_tracks,  
                      std::vector<size_t>  
*unassigned_objects) {  
    IDMatch(radar_tracks, radar_frame, assignments,  
    unassigned_tracks,  
            unassigned_objects);  
    TrackObjectPropertyMatch(radar_tracks, radar_frame,  
    assignments,  
                            unassigned_tracks,  
    unassigned_objects);  
    return true;  
}
```

先看下IDMatch中做了什么？

```
void BaseMatcher::IDMatch(const std::vector<RadarTrackPtr>  
&radar_tracks,  
                          const base::Frame &radar_frame,  
                          std::vector<TrackObjectPair>  
*assignments,
```

```

        std::vector<size_t>
*unassigned_tracks,
                           std::vector<size_t>
*unassigned_objects) {
    size_t num_track = radar_tracks.size();
    const auto &objects = radar_frame.objects;
    double object_timestamp = radar_frame.timestamp;
    size_t num_obj = objects.size();
    if (num_track == 0 || num_obj == 0) {
        unassigned_tracks->resize(num_track);
        unassigned_objects->resize(num_obj);
        std::iota(unassigned_tracks->begin(), unassigned_tracks-
>end(), 0);
        std::iota(unassigned_objects->begin(),
unassigned_objects->end(), 0);
        return;
    }
    std::vector<bool> track_used(num_track, false);
    std::vector<bool> object_used(num_obj, false);
    for (size_t i = 0; i < num_track; ++i) {
        const auto &track_object = radar_tracks[i]-
>GetObsRadar();
        double track_timestamp = radar_tracks[i]->GetTimestamp();
        if (track_object.get() == nullptr) {
            AERROR << "track_object is not available";
            continue;
        }
        int track_object_track_id = track_object->track_id;
        for (size_t j = 0; j < num_obj; ++j) {
            int object_track_id = objects[j]->track_id;
            if (track_object_track_id == object_track_id &&
                RefinedTrack(track_object, track_timestamp,
objects[j],
                           object_timestamp)) {
                assignments->push_back(std::pair<size_t, size_t>(i,
j));
                track_used[i] = true;
                object_used[j] = true;
            }
        }
    }
    for (size_t i = 0; i < track_used.size(); ++i) {
        if (!track_used[i]) {

```

```

        unassigned_tracks->push_back(i);
    }
}
for (size_t i = 0; i < object_used.size(); ++i) {
    if (!object_used[i]) {
        unassigned_objects->push_back(i);
    }
}
}

```

接着看TrackObjectPropertyMatch如何进行匹配。

```

void HMMatcher::TrackObjectPropertyMatch(
    const std::vector<RadarTrackPtr> &radar_tracks,
    const base::Frame &radar_frame,
    std::vector<TrackObjectPair> *assignments,
    std::vector<size_t> *unassigned_tracks,
    std::vector<size_t> *unassigned_objects) {
    // 1. 如果任意一个为空，则结束，那么物体和追踪是如何对应的？？
    if (unassigned_tracks->empty() || unassigned_objects-
>empty()) {
        return;
    }
    std::vector<std::vector<double> >
    association_mat(unassigned_tracks->size());
    for (size_t i = 0; i < association_mat.size(); ++i) {
        association_mat[i].resize(unassigned_objects->size(), 0);
    }
    // 计算追踪物体和当前帧物体的距离，并且保存在association_mat中。
    ComputeAssociationMat(radar_tracks, radar_frame,
    *unassigned_tracks,
                           *unassigned_objects,
    &association_mat);

    // 把association_mat赋值给global_costs
    common::SecureMat<double> *global_costs =
        hungarian_matcher_.mutable_global_costs();
    global_costs->Resize(unassigned_tracks->size(),
    unassigned_objects->size());
    for (size_t i = 0; i < unassigned_tracks->size(); ++i) {
        for (size_t j = 0; j < unassigned_objects->size(); ++j) {
            (*global_costs)(i, j) = association_mat[i][j];
        }
    }
}

```

```

    }

    // 通过直方图来计算匹配? ? ?

    std::vector<TrackObjectPair> property_assignments;
    std::vector<size_t> property_unassigned_tracks;
    std::vector<size_t> property_unassigned_objects;
    hungarian_matcher_.Match(
        BaseMatcher::GetMaxMatchDistance(),
        BaseMatcher::GetBoundMatchDistance(),

        common::GatedHungarianMatcher<double>::OptimizeFlag::OPTMIN,
        &property_assignments, &property_unassigned_tracks,
        &property_unassigned_objects);

    // 是否目标和追踪到的目标数量一定相等? ? ?
    // assignments 存放匹配好的对象, unassigned_tracks没有匹配的追
    踪, unassigned_objects没有匹配的目标
    for (size_t i = 0; i < property_assignments.size(); ++i) {
        size_t gt_idx = unassigned_tracks-
>at(property_assignments[i].first);
        size_t go_idx = unassigned_objects-
>at(property_assignments[i].second);
        assignments->push_back(std::pair<size_t, size_t>(gt_idx,
go_idx));
    }
    std::vector<size_t> temp_unassigned_tracks;
    std::vector<size_t> temp_unassigned_objects;
    for (size_t i = 0; i < property_unassigned_tracks.size();
++i) {
        size_t gt_idx = unassigned_tracks-
>at(property_unassigned_tracks[i]);
        temp_unassigned_tracks.push_back(gt_idx);
    }
    for (size_t i = 0; i < property_unassigned_objects.size();
++i) {
        size_t go_idx = unassigned_objects-
>at(property_unassigned_objects[i]);
        temp_unassigned_objects.push_back(go_idx);
    }
    *unassigned_tracks = temp_unassigned_tracks;
    *unassigned_objects = temp_unassigned_objects;
}

```

这里的匹配，采用的是”Kuhn-Munkres Algorithm”来进行多目标追踪的。

```
template <typename T>
void GatedHungarianMatcher<T>::Match(
    T cost_thresh, T bound_value, OptimizeFlag opt_flag,
    std::vector<std::pair<size_t, size_t>>* assignments,
    std::vector<size_t>* unassigned_rows,
    std::vector<size_t>* unassigned_cols) {

    /* initialize matcher */
    cost_thresh_ = cost_thresh;
    opt_flag_ = opt_flag;
    bound_value_ = bound_value;
    assignments_ptr_ = assignments;
    // 这里为何要检查cost_thresh < bound_value
    MatchInit();

    /* compute components */
    std::vector<std::vector<size_t>> row_components;
    std::vector<std::vector<size_t>> col_components;
    this->ComputeConnectedComponents(&row_components,
    &col_components);
    CHECK_EQ(row_components.size(), col_components.size());

    /* compute assignments */
    assignments_ptr_->clear();
    assignments_ptr_->reserve(std::max(rows_num_, cols_num_));
    for (size_t i = 0; i < row_components.size(); ++i) {
        this->OptimizeConnectedComponent(row_components[i],
        col_components[i]);
    }

    this->GenerateUnassignedData(unassigned_rows,
    unassigned_cols);
}
```

Frame

Frame代表了规划模块中的一帧，包括了多种信息。

FindDriveReferenceLineInfo

找到代价最小并且可以行驶的参考线，然后返回结果。

```
const ReferenceLineInfo *Frame::FindDriveReferenceLineInfo()
{
    double min_cost = std::numeric_limits<double>::infinity();
    drive_reference_line_info_ = nullptr;
    // 遍历找到最小代价，并且可以行驶的reference line
    for (const auto &reference_line_info :
reference_line_info_) {
        if (reference_line_info.IsDrivable() &&
            reference_line_info.Cost() < min_cost) {
            drive_reference_line_info_ = &reference_line_info;
            min_cost = reference_line_info.Cost();
        }
    }
    return drive_reference_line_info_;
}
```

FindTargetReferenceLineInfo

返回第一个变道类型的参考线，如果没有找到，则返回最后一个参考线。

```
const ReferenceLineInfo *Frame::FindTargetReferenceLineInfo()
{
    const ReferenceLineInfo *target_reference_line_info =
    nullptr;
    for (const auto &reference_line_info :
reference_line_info_) {
        if (reference_line_info.IsChangeLanePath()) {
            return &reference_line_info;
        }
    }
}
```

```

    }
    target_reference_line_info = &reference_line_info;
}
return target_reference_line_info;
}

```

FindFailedReferenceLineInfo

找到是变道类型并且不能行驶的参考线，如果没有找到则返回空。

```

const ReferenceLineInfo *Frame::FindFailedReferenceLineInfo()
{
    for (const auto &reference_line_info :
reference_line_info_) {
        // Find the unsuccessful lane-change path
        if (!reference_line_info.IsDrivable() &&
            reference_line_info.IsChangeLanePath()) {
            return &reference_line_info;
        }
    }
    return nullptr;
}

```

DriveReferenceLineInfo

返回FindDriveReferenceLineInfo中找到的参考线。即代价最小并且可以行驶的参考线。

```

const ReferenceLineInfo *Frame::DriveReferenceLineInfo()
const {
    return drive_reference_line_info_;
}

```

UpdateReferenceLinePriority

更新参考线的优先级，其中key为lane的id，value为优先级。(todo)这里的lanes是hdmap::RouteSegments类型，为什么id只有一个？

```

void Frame::UpdateReferenceLinePriority(
    const std::map<std::string, uint32_t> &id_to_priority) {
    for (const auto &pair : id_to_priority) {
        const auto id = pair.first;
        const auto priority = pair.second;
        auto ref_line_info_itr =
            std::find_if(reference_line_info_.begin(),
reference_line_info_.end(),
                         [&id] (const ReferenceLineInfo
&ref_line_info) {
                             return ref_line_info.Lanes().Id() ==
id;
                         });
        if (ref_line_info_itr != reference_line_info_.end()) {
            ref_line_info_itr->SetPriority(priority);
        }
    }
}

```

CreateReferenceLineInfo

根据reference_lines创建reference_line_info_（也是一个数组），并且添加障碍物到数组。

```

bool Frame::CreateReferenceLineInfo(
    const std::list<ReferenceLine> &reference_lines,
    const std::list<hdmap::RouteSegments> &segments) {
    reference_line_info_.clear();
    auto ref_line_iter = reference_lines.begin();
    auto segments_iter = segments.begin();
    // 1. reference_line_info_ 添加成员, (todo) 这里ref_line_iter和
    segments_iter是一一对应的吗?
    while (ref_line_iter != reference_lines.end()) {
        if (segments_iter->StopForDestination()) {
            is_near_destination_ = true;
        }
        reference_line_info_.emplace_back(vehicle_state_,
planning_start_point_,
                                         *ref_line_iter,
                                         *segments_iter);
        ++ref_line_iter;
    }
}

```

```

    ++segments_iter;
}

// 2. 如果reference_line_info_大小为2
if (reference_line_info_.size() == 2) {
    common::math::Vec2d xy_point(vehicle_state_.x(),
vehicle_state_.y());
    common::SLPoint first_sl;
    if
(!reference_line_info_.front().reference_line().XYToSL(xy_point,
&first_sl)) {
        return false;
    }
    common::SLPoint second_sl;
    if
(!reference_line_info_.back().reference_line().XYToSL(xy_point,
&second_sl)) {
        return false;
    }
    // 2.1 根据车当前的位置求出到起点的距离
    const double offset = first_sl.l() - second_sl.l();

    reference_line_info_.front().SetOffsetToOtherReferenceLine(offset);

    reference_line_info_.back().SetOffsetToOtherReferenceLine(-
offset);
}

bool has_valid_reference_line = false;
// 3. 初始化障碍物，如果有一个成功则表示有合理的参考线
for (auto &ref_info : reference_line_info_) {
    if (!ref_info.Init(obstacles())) {
        AERROR << "Failed to init reference line";
    } else {
        has_valid_reference_line = true;
    }
}

```

```
    return has_valid_reference_line;  
}
```

Dig into Apollo - Reference line

license MIT

尽吾志也而不能至者，可以无悔矣

Table of Contents

- 介绍
- 参考线
- 平滑器
- 参考线提供者

参考线介绍(Reference line)

参考线是根据routing规划的路线，生成一系列参考轨迹，提供给规划算法做为参考，从而生成最终的规划轨迹。为什么要提供参考呢？因为道路是结构化道路，在没有参考的情况下，需要通过搜索算法来查找路线，这种场景在机器人路径规划中比较普遍，机器人在一个开放空间只要没有障碍物它就可以行走，而车不一样，车是在道路上行驶的，在提供参考的情况下，节省了查找的时间和复杂度，降低了算法的难度，这也就是参考线的意义。

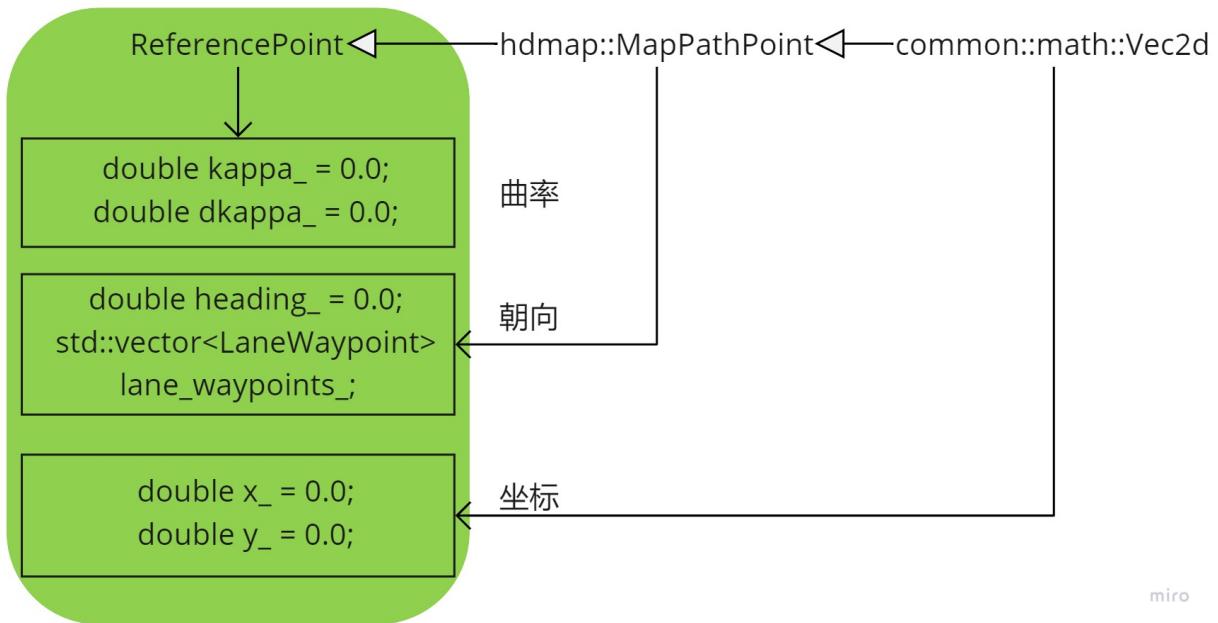
1. ReferenceLine和ReferenceLineInfo的关系 ReferenceLine提供的是轨迹信息，而ReferenceLineInfo在ReferenceLine的基础上新添加了决策信息。

参考线(ReferenceLineInfo)

参考线信息，在参考线的基础添加了决策信息，ST图等。

参考线中的点(ReferencePoint)

参考线中的点继承至`hdmap::MapPathPoint`，而`hdmap::MapPathPoint`又继承至`common::math::Vec2d`，也就是说参考线中的点实际上包含了路径点的信息，它有原来路径点中的朝向和坐标信息，同时还新增加了曲率信息，通过下图我们可以清楚的看出上述关系。



miro

参考线(ReferenceLine)

理解了参考线中的点之后，我们再看参考线的数据结构。

```

std::vector<SpeedLimit> speed_limit_; // 速度限制
std::vector<ReferencePoint> reference_points_; // 参考线的点
hdmap::Path map_path_; // 路径
uint32_t priority_ = 0; // 优先级
  
```

其中速度限制主要标明参考线中哪些段有速度限制，因为是一个数组，因此一个参考线中可以有多段不同的限速。优先级则表示了当前参考线的优先级，用于后面有多个参考线的时候进行挑选。

参考线中的点也是一个数组，也就是说参考线是由参考点组成的，而`map_path_`则是最不好理解的，实际上`map_path_`就是地图中参考线，把参考线中的点转换到地图中，因此`map_path_`中的点和参考点数组的大小是一致的。

除此之外，参考线还提供了一些方法，通过这些方法我们可以拼接参考线，也可以判断参考线所在位置的路的宽度，以及是否在路上等信息。我们先分析这些方法的功能实现，然后再介绍哪些场景需要用到这些功能。

构造函数

我们可以看到有2种方式来生成ReferenceLine，可以通过一组参考点来生成，也可以通过地图路径来生成，这2者实际上是等价的，下面我们开始分析。

1. 通过一组参考点生成，`reference_points_`直接拷贝赋值了，然后再用`reference_points`生成`hdmap::MapPathPoint`，最后保存到`map_path_`。

```
ReferenceLine::ReferenceLine(
    const std::vector<ReferencePoint>& reference_points)
: reference_points_(reference_points),
  map_path_(std::move(std::vector<hdmap::MapPathPoint>(
      reference_points.begin(), reference_points.end())))
```

2. 通过地图路径生成参考线。遍历路径中的点，然后取`lane_waypoints`中的第一个点，保存到参考点的数组中。

```
ReferenceLine::ReferenceLine(const MapPath& hdmap_path)
: map_path_(hdmap_path) {
    for (const auto& point : hdmap_path.path_points()) {
        const auto& lane_waypoint = point.lane_waypoints()[0];
        reference_points_.emplace_back(
            hdmap::MapPathPoint(point, point.heading(),
lane_waypoint), 0.0, 0.0);
    }
}
```

缝合参考线(Stitch)

缝合参考线是把2段参考线连接起来，代码中也给出了下面2种情况。并且每次拼接的时候，会尽可能多的采用自身的参考线。

```
* Example 1
* this:    |-----A-----x-----B-----|
* other:          |-----C-----x-----D-----|
* Result: |-----A-----x-----B-----x-----D-----|
* In the above example, A-B is current reference line, and
C-D is the other
* reference line. If part B and part C matches, we update
current reference
* line to A-B-D.
*
* Example 2
* this:          |-----A-----x-----B-----|
* other: |-----C-----x-----D-----|
* Result: |-----C-----x-----A-----x-----B-----|
* In the above example, A-B is current reference line, and
C-D is the other
* reference line. If part A and part D matches, we update
current reference
* line to C-A-B.
```

接下来我们分析下代码。

```
bool ReferenceLine::Stitch(const ReferenceLine& other) {
    // 1. 找到起点的交点
    auto first_point = reference_points_.front();
    common::SLPoint first_sl;
    if (!other.XYToSL(first_point, &first_sl)) {
        AWARN << "Failed to project the first point to the other
reference line.";
        return false;
    }
    bool first_join = first_sl.s() > 0 && first_sl.s() <
other.Length();
    // 2. 找到终点的交点
    auto last_point = reference_points_.back();
    common::SLPoint last_sl;
    if (!other.XYToSL(last_point, &last_sl)) {
        AWARN << "Failed to project the last point to the other
reference line.;"
```

```

        return false;
    }
    bool last_join = last_sl.s() > 0 && last_sl.s() <
other.Length();
    // 3. 如果起点和终点都没有交点，则退出
    if (!first_join && !last_join) {
        AERROR << "These reference lines are not connected.";
        return false;
    }

    // 累积s值
    const auto& accumulated_s =
other.map_path().accumulated_s();
    // 参考点
    const auto& other_points = other.reference_points();
    auto lower = accumulated_s.begin();
    static constexpr double kStitchingError = 1e-1;

    if (first_join) {
        // 4. 如果横向偏移大于0.1m，则退出
        if (first_sl.l() > kStitchingError) {
            AERROR << "lateral stitching error on first join of
reference line too "
                "big, stitching fails";
            return false;
        }
        lower = std::lower_bound(accumulated_s.begin(),
accumulated_s.end(),
                           first_sl.s());
        // 4.1 因为this的起点在other之后，插入other的起点到this的起点
        size_t start_i = std::distance(accumulated_s.begin(),
lower);
        reference_points_.insert(reference_points_.begin(),
other_points.begin(),
                           other_points.begin() + start_i);
    }
    if (last_join) {
        // 5.1 如果横向偏移大于0.1m，则退出
        if (last_sl.l() > kStitchingError) {
            AERROR << "lateral stitching error on first join of
reference line too "
                "big, stitching fails";
            return false;
        }
    }
}

```

```

    }
    // 5.2 因为this的终点小于other的终点，把other终点拼接到参考线的终点
    auto upper = std::upper_bound(lower, accumulated_s.end(),
last_sl.s());
    auto end_i = std::distance(accumulated_s.begin(), upper);
reference_points_.insert(reference_points_.end(),
                           other_points.begin() + end_i,
other_points.end());
}
map_path_ =
MapPath(std::move(std::vector

```

分割参考线(Segment)

分割参考线的方法是根据起点s，向前和向后的查看距离把参考线进行分割。有2个方法，我们只看其中一个就可以了。

```

bool ReferenceLine::Segment(const double s, const double
look_backward,
                               const double look_forward) {
const auto& accumulated_s = map_path_.accumulated_s();

// 1. 查找向后的索引(look_backward)
auto start_index =
    std::distance(accumulated_s.begin(),
                  std::lower_bound(accumulated_s.begin(),
accumulated_s.end(),
                           s - look_backward));
// 2. 查找向前的索引(look_forward)
auto end_index =
    std::distance(accumulated_s.begin(),
                  std::upper_bound(accumulated_s.begin(),
accumulated_s.end(),
                           s + look_forward));
// 3. 如果只有一个点
if (end_index - start_index < 2) {
    AERROR << "Too few reference points after shrinking.";
}

```

```

        return false;
    }

    // 4. 更新当前的参考线，并且返回成功
    reference_points_ =
        std::vector<ReferencePoint>(reference_points_.begin() +
start_index,
                                         reference_points_.begin() +
end_index);
    map_path_ = MapPath(std::vector<hdmap::MapPathPoint>(
        reference_points_.begin(), reference_points_.end()));
    return true;
}

```

其它方法介绍

这里把其它一些方法的功能进行介绍，具体的代码就不展开分析了。

```

// 根据s值获取参考点（会根据s进行插值）
ReferencePoint GetReferencePoint(const double s) const;
// 根据x,y找到最近的点，并且进行插值
ReferencePoint GetReferencePoint(const double x, const
double y) const;

// PathPoint转换为FrenetFramePoint
common::FrenetFramePoint GetFrenetPoint(
    const common::PathPoint& path_point) const;

//
std::pair<std::array<double, 3>, std::array<double, 3>>
ToFrenetFrame(
    const common::TrajectoryPoint& traj_point) const;

// 查找起点和终点分别为start_s和end_s的参考点
std::vector<ReferencePoint> GetReferencePoints(double
start_s,
                                         double end_s)
const;
// 获取离s最近的索引
size_t GetNearestReferenceIndex(const double s) const;

// 离s最近的ReferencePoint

```

```

    ReferencePoint GetNearestReferencePoint(const
common::math::Vec2d& xy) const;
    ReferencePoint GetNearestReferencePoint(const double s)
const;

    // 根据起点s和终点s获取LaneSegment
    std::vector<hdmap::LaneSegment> GetLaneSegments(const
double start_s,
                                                    const
double end_s) const;
    // 获取box在参考线上的投影框
    bool GetApproximateSLBoundary(const common::math::Box2d&
box,
                                    const double start_s, const
double end_s,
                                    SLBoundary* const
sl_boundary) const;
    bool GetSLBoundary(const common::math::Box2d& box,
                       SLBoundary* const sl_boundary) const;
    bool GetSLBoundary(const hdmap::Polygon& polygon,
                       SLBoundary* const sl_boundary) const;
    // SL坐标到XY坐标相互转换
    bool SLToXY(const common::SLPoint& sl_point,
                common::math::Vec2d* const xy_point) const;
    bool XYToSL(const common::math::Vec2d& xy_point,
                common::SLPoint* const sl_point) const;
    // 获取s距离处路的宽度
    bool GetLaneWidth(const double s, double* const
lane_left_width,
                    double* const lane_right_width) const;
    // 获取s距离处的偏移
    bool GetOffsetToMap(const double s, double* l_offset)
const;
    // 获取s距离处路的宽度
    bool GetRoadWidth(const double s, double* const
road_left_width,
                    double* const road_right_width) const;
    // 获取s距离处路的类型
    hdmap::Road::Type GetRoadType(const double s) const;
    // 获取s距离处道路
    void GetLaneFromS(const double s,
                      std::vector<hdmap::LaneInfoConstPtr>*
lanes) const;

```

```

// 获取乘车宽度
double GetDrivingWidth(const SLBoundary& sl_boundary)
const;
// 是否在路上
bool IsOnLane(const common::SLPoint& sl_point) const;
bool IsOnLane(const common::math::Vec2d& vec2d_point)
const;
template <class XYPoint>
bool IsOnLane(const XYPoint& xy) const {
    return IsOnLane(common::math::Vec2d(xy.x(), xy.y()));
}
bool IsOnLane(const SLBoundary& sl_boundary) const;

bool IsOnRoad(const common::SLPoint& sl_point) const;
bool IsOnRoad(const common::math::Vec2d& vec2d_point)
const;
bool IsOnRoad(const SLBoundary& sl_boundary) const;
// 是否堵路了
bool IsBlockRoad(const common::math::Box2d& box2d, double
gap) const;
// 是否有重叠
bool HasOverlap(const common::math::Box2d& box) const;
// 查找s处的速度限制
double GetSpeedLimitFromS(const double s) const;
// 添加start_s到end_s处的速度限制，并且进行排序
void AddSpeedLimit(double start_s, double end_s, double
speed_limit);

```

平滑器(ReferenceLineSmoothen)

平滑器的主要作用是对参考线做平滑，一共有3种类型的平滑器。

- DiscretePointsReferenceLineSmoothen
- QpSplineReferenceLineSmoothen
- SpiralReferenceLineSmoothen

它们都继承至ReferenceLineSmoothen，需要提供设置锚点SetAnchorPoints和Smooth2个接口。

```

struct AnchorPoint {
    common::PathPoint path_point;

```

```
    double lateral_bound = 0.0;
    double longitudinal_bound = 0.0;
    // enforce smoother to strictly follow this reference point
    bool enforced = false;
};

class ReferenceLineSmoother {
public:
    explicit ReferenceLineSmoother(const
ReferenceLineSmootherConfig& config)
        : config_(config) {}

    // 虚函数，设置锚点
    virtual void SetAnchorPoints(
        const std::vector<AnchorPoint>& anchor_points) = 0;

    // 虚函数，平滑参考线
    virtual bool Smooth(const ReferenceLine&, ReferenceLine*&
const) = 0;

    virtual ~ReferenceLineSmoother() = default;
protected:
    ReferenceLineSmootherConfig config_;
};
```

DiscretePointsReferenceLineSmoother

todo

QpSplineReferenceLineSmoother

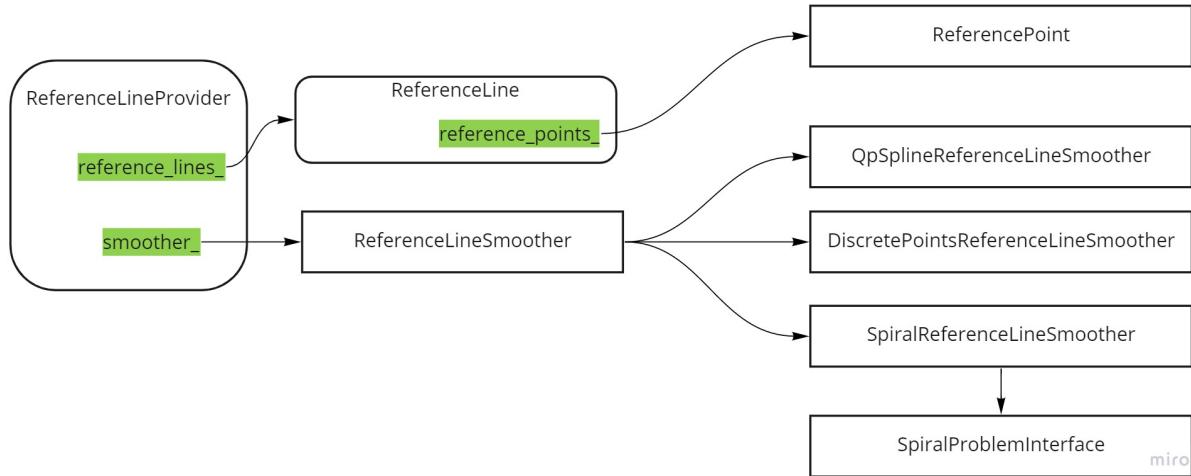
todo

SpiralReferenceLineSmoother

todo

参考线提供者(ReferenceLineProvider)

planning模块中分为2个任务，一个线程单独的执行ReferenceLineProvider，另外一个线程根据生成好的参考线进行路径规划。而参考线提供者根据车辆的位置，pnc地图来生成参考线，并且进行平滑之后输出给planning模块做后续的路径规划。



我们带着以下几个问题来阅读代码。

1. 整个流程的过程是怎样的？
2. 如何生成的参考线，输入是什么？输出是什么？
3. 参考线用图来形象的表示？

构造函数

从参考线提供者的构造函数中就可以看出，它的输入是车辆状态和地图，输出是参考线。

```

ReferenceLineProvider::ReferenceLineProvider(
    const common::VehicleStateProvider
    *vehicle_state_provider,
    const hdmap::HDMap *base_map,
    const std::shared_ptr<relative_map::MapMsg>
    &relative_map)
    // 1. 初始化车辆状态提供者
    : vehicle_state_provider_(vehicle_state_provider) {
    // 2. 如果是导航模式则启动相对地图，如果不是则启动pnc_map
    if (!FLAGS_use_navigation_mode) {
        pnc_map_ = std::make_unique<hdmap::PncMap>(base_map);
        relative_map_ = nullptr;
    }
}
  
```

```

    } else {
        pnc_map_ = nullptr;
        relative_map_ = relative_map;
    }
    // 3. 初始化平滑器

ACHECK(cyber::common::GetProtoFromFile(FLAGS_smoothen_config_
filename,
                                         &smoother_config_))
    << "Failed to load smoother config file "
    << FLAGS_smoothen_config_filename;
if (smoother_config_.has_qp_spline()) {
    smoother_.reset(new
QpSplineReferenceLineSmoothen(smoothere_config_));
} else if (smoother_config_.has_spiral()) {
    smoother_.reset(new
SpiralReferenceLineSmoothen(smoothere_config_));
} else if (smoother_config_.has_discrete_points()) {
    smoother_.reset(new
DiscretePointsReferenceLineSmoothen(smoothere_config_));
} else {
    ACHECK(false) << "unknown smoother config "
                << smoother_config_.DebugString();
}
is_initialized_ = true;
}

```

开始线程

ReferenceLineProvider通过start()方法开始启动新的线程并且执行，通过stop()方法停止。

```

bool ReferenceLineProvider::Start() {
    if (FLAGS_use_navigation_mode) {
        return true;
    }
    if (!is_initialized_) {
        AERROR << "ReferenceLineProvider has NOT been
initiated.";
        return false;
    }
    // 1. 启动异步任务运行GenerateThread

```

```

    if (FLAGS_enable_reference_line_provider_thread) {
        task_future_ =
cyber::Async(&ReferenceLineProvider::GenerateThread, this);
    }
    return true;
}

```

然后通过以下2个方法，刷新routing请求和车辆状态。也就是说参考线通过实时的routing请求和车辆状态来生成参考线。

```

    bool UpdateRoutingResponse(const routing::RoutingResponse&
routing);

    void UpdateVehicleState(const common::VehicleState&
vehicle_state);

```

通过GetReferenceLines来获取参考线。实际上并发模式和不是并发模式执行的函数都是一样，只不过并发模式下另外的线程已经计算好了，因此可以直接赋值。

```

bool ReferenceLineProvider::GetReferenceLines(
    std::list<ReferenceLine> *reference_lines,
    std::list<hdmap::RouteSegments> *segments) {
    ...
    // 1. 如果有单独的线程，则直接赋值
    if (FLAGS_enable_reference_line_provider_thread) {
        std::lock_guard<std::mutex> lock(reference_lines_mutex_);
        if (!reference_lines_.empty()) {
            reference_lines->assign(reference_lines_.begin(),
reference_lines_.end());
            segments->assign(route_segments_.begin(),
route_segments_.end());
            return true;
        }
    } else {
        double start_time = Clock::NowInSeconds();
        // 2. 否则，创建并且更新参考线
        if (CreateReferenceLine(reference_lines, segments)) {
            UpdateReferenceLine(*reference_lines, *segments);
            double end_time = Clock::NowInSeconds();
            last_calculation_time_ = end_time - start_time;
            return true;
        }
    }
}

```

```

        }
    }

AWARN << "Reference line is NOT ready.";
if (reference_line_history_.empty()) {
    AERROR << "Failed to use reference line latest history";
    return false;
}
// 3. 如果失败，则采用上一次的规划轨迹
reference_lines-
>assign(reference_line_history_.back().begin(),
reference_line_history_.back().end());
segments->assign(route_segments_history_.back().begin(),
route_segments_history_.back().end());
AWARN << "Use reference line from history!";
return true;
}

```

从上面代码可以得出，参考线的生成主要集中在2个函数中
CreateReferenceLine和UpdateReferenceLine。

CreateReferenceLine

创建参考线

```

bool ReferenceLineProvider::CreateReferenceLine(
    std::list<ReferenceLine> *reference_lines,
    std::list<hdmap::RouteSegments> *segments) {

    // 1. 获取车辆状态
    common::VehicleState vehicle_state;
    {
        std::lock_guard<std::mutex> lock(vehicle_state_mutex_);
        vehicle_state = vehicle_state_;
    }
    // 2. 获取routing
    routing::RoutingResponse routing;
    {
        std::lock_guard<std::mutex> lock(routing_mutex_);
        routing = routing_;
    }
}

```

```

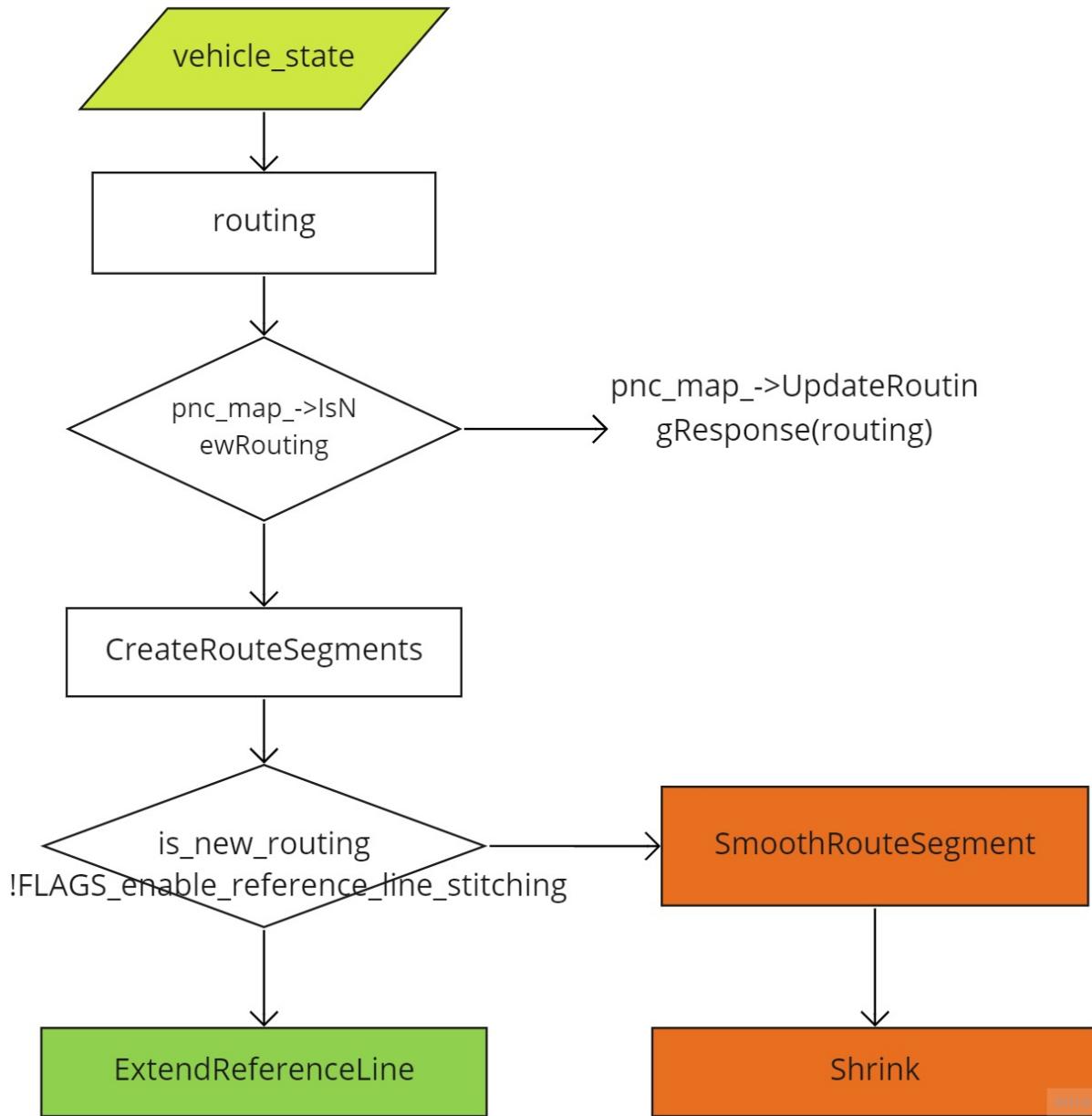
bool is_new_routing = false;
{
    // 2.1 如果是新routing, 那么更新routing
    std::lock_guard<std::mutex> lock(pnc_map_mutex_);
    if (pnc_map_->IsNewRouting(routing)) {
        is_new_routing = true;
        if (!pnc_map_->UpdateRoutingResponse(routing)) {
            AERROR << "Failed to update routing in pnc map";
            return false;
        }
    }
}

// 3. 创建routing segment
if (!CreateRouteSegments(vehicle_state, segments)) {
    AERROR << "Failed to create reference line from routing";
    return false;
}
if (is_new_routing ||
!FLAGS_enable_reference_line_stitching) {
    for (auto iter = segments->begin(); iter != segments-
>end();) {
        reference_lines->emplace_back();
        // 4.1.1 平滑routing segment
        if (!SmoothRouteSegment(*iter, &reference_lines-
>back())) {
            AERROR << "Failed to create reference line from route
segments";
            reference_lines->pop_back();
            iter = segments->erase(iter);
        } else {
            common::SLPoint sl;
            if (!reference_lines->back().XYToSL(vehicle_state,
&sl)) {
                AWARN << "Failed to project point: {" <<
vehicle_state.x() << ","
                << vehicle_state.y() << "} to stitched
reference line";
            }
        }
        // 4.1.2 收缩参考线
        Shrink(sl, &reference_lines->back(), &(*iter));
        ++iter;
    }
}

```

```
    }
    return true;
} else { // stitching reference line
// 4.2 根据routing segment扩展参考线
    for (auto iter = segments->begin(); iter != segments-
>end();) {
        reference_lines->emplace_back();
        if (!ExtendReferenceLine(vehicle_state, &(*iter),
                                 &reference_lines->back())))
            AERROR << "Failed to extend reference line";
        reference_lines->pop_back();
        iter = segments->erase(iter);
    } else {
        ++iter;
    }
}
return true;
}
```

创建参考线的过程如下，首先获取routing的消息，然后生成routingsegment，之后结合车辆状态，扩展和分割参考线。



接下来我们分别分析没有发送新routing和发送了新routing的生成参考线的流程。

旧routing生成参考线

创建routing路段

根据车辆当前状态，从Pnc地图中获取routing路段，最后判断是否要提高变道的优先级。

```
bool ReferenceLineProvider::CreateRouteSegments (
    const common::VehicleState &vehicle_state,
    std::list<hdmap::RouteSegments> *segments) {
{
    // 1. pnc_map获取RouteSegments
    std::lock_guard<std::mutex> lock(pnc_map_mutex_);
    if (!pnc_map_->GetRouteSegments(vehicle_state, segments))
    {
        AERROR << "Failed to extract segments from routing";
        return false;
    }
}

// 2. 是否提高变道的优先级
if (FLAGS_prioritize_change_lane) {
    PrioritzeChangeLane(segments);
}
return !segments->empty();
}
```

GetRouteSegments

pnc地图中获取RouteSegments的过程实际上是根据车辆当前的速度查看前面一小段和后面一小段的路径。

```
bool PncMap::GetRouteSegments(const VehicleState
&vehicle_state,
                                std::list<RouteSegments> *const
route_segments) {
    // 1. 前看距离
    double look_forward_distance =
        LookForwardDistance(vehicle_state.linear_velocity());
    // 2. 后看距离
    double look_backward_distance =
FLAGS_look_backward_distance;
    return GetRouteSegments(vehicle_state,
look_backward_distance,
                                look_forward_distance,
```

```
route_segments);  
}
```

接着我们继续看GetRouteSegments

```
bool PncMap::GetRouteSegments(const VehicleState  
&vehicle_state,  
                               const double backward_length,  
                               const double forward_length,  
                               std::list<RouteSegments> *const  
route_segments) {  
    // 1. 更新车辆状态  
    if (!UpdateVehicleState(vehicle_state)) {  
        AERROR << "Failed to update vehicle state in pnc_map.";  
        return false;  
    }  
    // 2. 车辆是否在路上  
    if (!adc_waypoint_.lane || adc_route_index_ < 0 ||  
        adc_route_index_ >= static_cast<int>  
(route_indices_.size())) {  
        AERROR << "Invalid vehicle state in pnc_map, update  
vehicle state first.";  
        return false;  
    }  
  
    const auto &route_index =  
route_indices_[adc_route_index_].index;  
    const int road_index = route_index[0];  
    const int passage_index = route_index[1];  
    const auto &road = routing_.road(road_index);  
    // 3. 找到相邻的所有passage id  
    auto drive_passages = GetNeighborPassages(road,  
passage_index);  
    for (const int index : drive_passages) {  
        const auto &passage = road.passage(index);  
        RouteSegments segments;  
        // 3.1 转换passage到segments  
        if (!PassageToSegments(passage, &segments)) {  
            ADEBUG << "Failed to convert passage to lane  
segments.";  
            continue;  
        }  
        const PointENU nearest_point =
```

```

        index == passage_index
            ? adc_waypoint_.lane-
>GetSmoothPoint(adc_waypoint_.s)
            : PointFactory::ToPointENU(adc_state_);
common::SLPoint sl;
LaneWaypoint segment_waypoint;
// 3.2 获取车辆到segments的投影
if (!segments.GetProjection(nearest_point, &sl,
&segment_waypoint)) {
    ADEBUG << "Failed to get projection from point: "
        << nearest_point.ShortDebugString();
    continue;
}
if (index != passage_index) {
    if (!segments.CanDriveFrom(adc_waypoint_)) {
        ADEBUG << "You cannot drive from current waypoint to
passage: "
            << index;
        continue;
    }
}
route_segments->emplace_back();
const auto last_waypoint = segments.LastWaypoint();
// 4. 根据车辆的当前位置前后一段距离, 填充segments
if (!ExtendSegments(segments, sl.s() - backward_length,
                    sl.s() + forward_length,
&route_segments->back())) {
    AERROR << "Failed to extend segments with s=" << sl.s()
        << ", backward: " << backward_length
        << ", forward: " << forward_length;
    return false;
}
// 5. 根据passage设置route_segments的属性
if (route_segments-
>back().IsWaypointOnSegment(last_waypoint)) {
    route_segments-
>back().SetRouteEndWaypoint(last_waypoint);
}
route_segments->back().SetCanExit(passage.can_exit());
route_segments-
>back().SetNextAction(passage.change_lane_type());
const std::string route_segment_id =
absl::StrCat(road_index, "_", index);

```

```

    route_segments->back().SetId(route_segment_id);
    route_segments-
>back().SetStopForDestination(stop_for_destination_);
    if (index == passage_index) {
        route_segments->back().SetIsOnSegment(true);
        route_segments-
>back().SetPreviousAction(routing::FORWARD);
    } else if (sl.l() > 0) {
        route_segments-
>back().SetPreviousAction(routing::RIGHT);
    } else {
        route_segments-
>back().SetPreviousAction(routing::LEFT);
    }
}
return !route_segments->empty();
}

```

扩展参考线(ExtendReferenceLine)

```

bool ReferenceLineProvider::ExtendReferenceLine(const
VehicleState &state,
                                                 RouteSegments
*segments,
                                                 ReferenceLine
*reference_line) {
    RouteSegments segment_properties;
    segment_properties.SetProperties(*segments);
    auto prev_segment = route_segments_.begin();
    auto prev_ref = reference_lines_.begin();
    // 1. 查找和segments连接的segment
    while (prev_segment != route_segments_.end()) {
        if (prev_segment->IsConnectedSegment(*segments)) {
            break;
        }
        ++prev_segment;
        ++prev_ref;
    }
    if (prev_segment == route_segments_.end()) {
        if (!route_segments_.empty() && segments->IsOnSegment())
{
            AWARN << "Current route segment is not connected with

```

```

previous route "
    "segment";
}
return SmoothRouteSegment(*segments, reference_line);
}
common::SLPoint sl_point;
Vec2d vec2d(state.x(), state.y());
LaneWaypoint waypoint;
// 2. 获取车辆到segment的投影
if (!prev_segment->GetProjection(vec2d, &sl_point,
&waypoint)) {
    AWARN << "Vehicle current point: " << vec2d.DebugString()
        << " not on previous reference line";
    return SmoothRouteSegment(*segments, reference_line);
}
const double prev_segment_length =
RouteSegments::Length(*prev_segment);
const double remain_s = prev_segment_length - sl_point.s();
const double look_forward_required_distance =
    PncMap::LookForwardDistance(state.linear_velocity());
// 3. 如果remain_s大于look_forward_required_distance，则返回当前
前segment
if (remain_s > look_forward_required_distance) {
    *segments = *prev_segment;
    segments->SetProperties(segment_properties);
    *reference_line = *prev_ref;
    ADEBUG << "Reference line remain " << remain_s
        << ", which is more than required " <<
    look_forward_required_distance
        << " and no need to extend";
    return true;
}
// 4. 如果小于，则需要补充下一个segment
double future_start_s =
    std::max(sl_point.s(), prev_segment_length -
FLAGS_reference_line_stitch_overlap_distance);
double future_end_s =
    prev_segment_length +
FLAGS_look_forward_extend_distance;
RouteSegments shifted_segments;
std::unique_lock<std::mutex> lock(pnc_map_mutex_);
if (!pnc_map_->ExtendSegments(*prev_segment,

```

```

future_start_s, future_end_s,
                           &shifted_segments)) {
    lock.unlock();
    AERROR << "Failed to shift route segments forward";
    return SmoothRouteSegment(*segments, reference_line);
}
lock.unlock();
if (prev_segment-
>IsWaypointOnSegment(shifted_segments.LastWaypoint())) {
    *segments = *prev_segment;
    segments->SetProperties(segment_properties);
    *reference_line = *prev_ref;
    ADEBUG << "Could not further extend reference line";
    return true;
}
hdmap::Path path(shifted_segments);
ReferenceLine new_ref(path);
if (!SmoothPrefixedReferenceLine(*prev_ref, new_ref,
reference_line)) {
    AWARN << "Failed to smooth forward shifted reference
line";
    return SmoothRouteSegment(*segments, reference_line);
}
if (!reference_line->Stitch(*prev_ref)) {
    AWARN << "Failed to stitch reference line";
    return SmoothRouteSegment(*segments, reference_line);
}
if (!shifted_segments.Stitch(*prev_segment)) {
    AWARN << "Failed to stitch route segments";
    return SmoothRouteSegment(*segments, reference_line);
}
*segments = shifted_segments;
segments->SetProperties(segment_properties);
common::SLPoint sl;
if (!reference_line->XYToSL(vec2d, &sl)) {
    AWARN << "Failed to project point: " <<
vec2d.DebugString()
                << " to stitched reference line";
}
return Shrink(sl, reference_line, segments);
}

```

Shrink

根据本车位置裁剪参考线和RouteSegments

```
bool ReferenceLineProvider::Shrink(const common::SLPoint &sl,
                                    ReferenceLine
*reference_line,
                                    RouteSegments *segments) {
    static constexpr double kMaxHeadingDiff = M_PI * 5.0 / 6.0;
    // 1. 确定前后查看的距离
    double new_backward_distance = sl.s();
    double new_forward_distance = reference_line->Length() -
        sl.s();
    bool need_shrink = false;
    if (sl.s() > FLAGS_look_backward_distance * 1.5) {
        ADEBUG << "reference line back side is " << sl.s()
            << ", shrink reference line: origin length: "
            << reference_line->Length();
        new_backward_distance = FLAGS_look_backward_distance;
        need_shrink = true;
    }
    // 2. 角度差小于kMaxHeadingDiff
    const auto index = reference_line-
>GetNearestReferenceIndex(sl.s());
    const auto &ref_points = reference_line-
>reference_points();
    const double cur_heading = ref_points[index].heading();
    auto last_index = index;
    while (last_index < ref_points.size() &&
           AngleDiff(cur_heading,
ref_points[last_index].heading()) <
           kMaxHeadingDiff) {
        ++last_index;
    }
    --last_index;
    if (last_index != ref_points.size() - 1) {
        need_shrink = true;
        common::SLPoint forward_sl;
        reference_line->XYToSL(ref_points[last_index],
&forward_sl);
        new_forward_distance = forward_sl.s() - sl.s();
    }
}
```

```

// 3. 分割并且收缩参考线
if (need_shrink) {
    if (!reference_line->Segment(sl.s(),
new_backward_distance,
                                new_forward_distance)) {
        AWARN << "Failed to shrink reference line";
    }
    if (!segments->Shrink(sl.s(), new_backward_distance,
                           new_forward_distance)) {
        AWARN << "Failed to shrink route segment";
    }
}
return true;
}

```

自此没有更新routing情况下的参考线生成流程已经分析完成了。

新routing生成参考线

todo

UpdatedReferenceLine

根据是否更新了routing重新给参考线进行赋值。

```

void ReferenceLineProvider::UpdateReferenceLine(
    const std::list<ReferenceLine> &reference_lines,
    const std::list<hdmap::RouteSegments> &route_segments) {
    // 1. 如果参考线大小和roue路线段不相等，则返回
    if (reference_lines.size() != route_segments.size()) {
        AERROR << "The calculated reference line size(" <<
reference_lines.size()
                << ") and route_segments size(" <<
route_segments.size()
                << ") are different";
        return;
    }
    std::lock_guard<std::mutex> lock(reference_lines_mutex_);
    // 2.1 如果新旧参考线大小不等，则拷贝
    if (reference_lines_.size() != reference_lines.size()) {
        reference_lines_ = reference_lines;
    }
}

```

```

        route_segments_ = route_segments;
    } else {
        // 2.2 如果相等，则依次拷贝
        auto segment_iter = route_segments.begin();
        auto internal_iter = reference_lines_.begin();
        auto internal_segment_iter = route_segments_.begin();
        for (auto iter = reference_lines.begin();
              iter != reference_lines.end() &&
              segment_iter != route_segments.end() &&
              internal_iter != reference_lines_.end() &&
              internal_segment_iter != route_segments_.end());
              ++iter, ++segment_iter, ++internal_iter,
              ++internal_segment_iter) {
            if (iter->reference_points().empty()) {
                *internal_iter = *iter;
                *internal_segment_iter = *segment_iter;
                continue;
            }
            if (common::util::SamePointXY(
                    iter->reference_points().front(),
                    internal_iter->reference_points().front()) &&
                common::util::SamePointXY(iter-
                                         >reference_points().back(),
                                         internal_iter-
                                         >reference_points().back()) &&
                std::fabs(iter->Length() - internal_iter->Length())
                <
                    common::math::kMathEpsilon) {
                continue;
            }
            *internal_iter = *iter;
            *internal_segment_iter = *segment_iter;
        }
    }
    // 3. 存储并且更新最近3次的参考线和routing历史信息
    reference_line_history_.push(reference_lines_);
    route_segments_history_.push(route_segments_);
    static constexpr int kMaxHistoryNum = 3;
    if (reference_line_history_.size() > kMaxHistoryNum) {
        reference_line_history_.pop();
        route_segments_history_.pop();
    }
}

```

如何获取场景？？？

如何分析并且获取场景？？？？

```
void ScenarioManager::Run() {  
    auto environment_features =  
        FeatureExtractor::ExtractEnvironmentFeatures();  
  
    auto ptr_scenario_features =  
        ScenarioAnalyzer::Analyze(environment_features);  
  
    current_scenario_ = ptr_scenario_features->scenario();  
  
    // TODO(all) other functionalities including lane, junction  
    filters  
}
```

如何生成LaneGraph

Obstacle::BuildLaneGraph()

1. 障碍物如何生成Lane图，作用是什么？？？
2. common目录的”RoadGraph”生成的图和上述的图有什么关系？？？

protobuf中mutable_的作用

可以看出，对于每个字段会生成一个has函数(has_number)、clear清除函数(clear_number)、set函数(set_number)、get函数(number和mutable_number)。这儿解释下get函数中的两个函数的区别，对于原型为const std::string &number() const的get函数而言，返回的是常量字段，不能对其值进行修改。但是在有一些情况下，对字段进行修改是必要的。

的，所以提供了一个mutable版的get函数，通过获取字段变量的指针，从而达到改变其值的目的。

计划

1. 分析下LaneGraph和RoadGraph是如何生成的？？？
2. 分析LTSM和MLP的实现？？？
3. 分析预测曲线的原理？？？
4. planning模块是如何利用这些预测的障碍物信息的？？？

1. 为什么会出现找不到节点的情况？？？ 节点没有连接，所以需要找到节点附近最近的路段。

2. Astar算法 <http://www-cs-students.stanford.edu/~amitp/gameprog.html#Paths>

<http://theory.stanford.edu/~amitp/GameProgramming/>

<https://www.geeksforgeeks.org/a-search-algorithm/>

<http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html>

3. “RoutingComponent”类继承至”cyber::Component”，并且申明为”public”方式，”cyber::Component”是一个模板类，它定义了”Initialize”和”Process”方法。

```
template <typename M0>
class Component<M0, NullType, NullType, NullType> : public
ComponentBase {
public:
    Component() {}
    ~Component() override {}
    bool Initialize(const ComponentConfig& config) override;
    bool Process(const std::shared_ptr<M0>& msg);

private:
    virtual bool Proc(const std::shared_ptr<M0>& msg) = 0;
};
```

// todo 模板方法中为虚函数，而继承类中为公有方法？为什么？

SearchRoute

```
bool Navigator::SearchRoute(const RoutingRequest& request,
                           RoutingResponse* const response)
{
    ...
    // 初始化规划点和起点
    std::vector<const TopoNode*> way_nodes;
```

```

    std::vector<double> way_s;
    if (!Init(request, graph_.get(), &way_nodes, &way_s)) {
        return false;
    }
    // 根据节点和起点，查找返回结果，注意这里返回的是一段范围
    std::vector<NodeWithRange> result_nodes;
    if (!SearchRouteByStrategy(graph_.get(), way_nodes, way_s,
    &result_nodes)) {
        return false;
    }
    if (result_nodes.empty()) {
        return false;
    }
    // 插入起点和终点
    result_nodes.front().SetStartS(request.waypoint().begin()-
    >s());
    result_nodes.back().SetEndS(request.waypoint().rbegin()-
    >s());
    // 生成通道区域
    if (!result_generator_->GeneratePassageRegion(
        graph_->MapVersion(), request, result_nodes,
    topo_range_manager_,
        response)) {
        return false;
    }
    ...
}

```

FillLaneInfoIfMissing

如果routing请求中没有包含lane信息，则会自动补全这一部分信息。

```

RoutingRequest Routing::FillLaneInfoIfMissing(
    const RoutingRequest& routing_request) {
    RoutingRequest fixed_request(routing_request);
    // 遍历routing请求的点
    for (int i = 0; i < routing_request.waypoint_size(); ++i) {
        const auto& lane_waypoint = routing_request.waypoint(i);
        // routing_request请求的点有lane_id，则表示在路上，不用查找
        if (lane_waypoint.has_id()) {

```

```

        continue;
    }
    auto point =
common::util::MakePointENU(lane_waypoint.pose().x(),
                           lane_waypoint.pose().y(),
                           lane_waypoint.pose().z());
    double s = 0.0;
    double l = 0.0;
    hdmap::LaneInfoConstPtr lane;
    // FIXME(all): select one reasonable lane candidate for
    point=>lane
    // is one to many relationship.
    // 找到当前点最近的lane信息
    if (hdmap_->GetNearestLane(point, &lane, &s, &l) != 0) {
        AERROR << "Failed to find nearest lane from map at
position: "
                << point.DebugString();
        return routing_request;
    }
    auto waypoint_info = fixed_request.mutable_waypoint(i);
    waypoint_info->set_id(lane->id().id());
    waypoint_info->set_s(s);
}
return fixed_request;
}

```

BlackListRangeGenerator生成黑名单路段 通过RoutingRequest中的black_lane和black_road来生成黑名单路段，这里会设置一整段路都为黑名单。

```

void BlackListRangeGenerator::GenerateBlackMapFromRequest (
    const RoutingRequest& request, const TopoGraph* graph,
    TopoRangeManager* const range_manager) const {
    AddBlackMapFromLane(request, graph, range_manager);
    AddBlackMapFromRoad(request, graph, range_manager);
    range_manager->SortAndMerge();
}

```

通过Terminal来设置黑名单，应用场景是设置routing的起点和终点。这里的起点和终点都是一个点，功能是把lane切分为2个subNode

```
void BlackListRangeGenerator::AddBlackMapFromTerminal(
    const TopoNode* src_node, const TopoNode* dest_node,
    double start_s,
    double end_s, TopoRangeManager* const range_manager)
const {
    double start_length = src_node->Length();
    double end_length = dest_node->Length();
    if (start_s < 0.0 || start_s > start_length) {
        AERROR << "Illegal start_s: " << start_s << ", length: "
        << start_length;
        return;
    }
    if (end_s < 0.0 || end_s > end_length) {
        AERROR << "Illegal end_s: " << end_s << ", length: " <<
        end_length;
        return;
    }

    double start_cut_s = MoveSBackward(start_s, 0.0);
    range_manager->Add(src_node, start_cut_s, start_cut_s);
    AddBlackMapFromOutParallel(src_node, start_cut_s /
    start_length,
                                range_manager);

    double end_cut_s = MoveSForward(end_s, end_length);
    range_manager->Add(dest_node, end_cut_s, end_cut_s);
    AddBlackMapFromInParallel(dest_node, end_cut_s /
    end_length, range_manager);
    range_manager->SortAndMerge();
}
```

TODO: 如果在dreamview里设置多个routing点的情况，那么会出现第一段的终点和第二段的起点有overlap的情况，排序之后，会出现2厘米的gap?目前看起来不会影响，因为会往前开，另外为什么要设置往后偏移1厘米，如果刚好停在边界点上，会如何处理？？？

TODO: 判断是否足够进行切换Lane,

```

bool TopoNode::IsOutOfRangeEnough(const
std::vector<NodeSRange>& range_vec,
                                  double start_s, double end_s)
{
    // 是否足够切换Lane
    if (!NodeSRange::IsEnoughForChangeLane(start_s, end_s)) {
        return false;
    }
    int start_index = BinarySearchForSLarger(range_vec,
start_s);
    int end_index = BinarySearchForSSmaller(range_vec, end_s);

    int index_diff = end_index - start_index;
    if (start_index < 0 || end_index < 0) {
        return false;
    }
    if (index_diff > 1) {
        return true;
    }

    double pre_s_s = std::max(start_s,
range_vec[start_index].StartS());
    double suc_e_s = std::min(end_s,
range_vec[end_index].EndS());

    if (index_diff == 1) {
        double dlt = range_vec[start_index].EndS() - pre_s_s;
        dlt += suc_e_s - range_vec[end_index].StartS();
        return NodeSRange::IsEnoughForChangeLane(dlt);
    }
    if (index_diff == 0) {
        return NodeSRange::IsEnoughForChangeLane(pre_s_s,
suc_e_s);
    }
    return false;
}

```

GeneratePassageRegion

生成passageRegion，这里需要注意每个Passage中的直行都是合并了的，也就是说passage中只有每次换向的时候才会从新启用新的passage。

SubTopoGraph

构建subtopograph的作用就是为了方便routing切割lane，然后把lane分割成几个子节点，子节点的网络是如何建立的？如何根据这些节点来进行查找和计算代价？？？

Navigator::MergeRoute

没有看出来从哪里mergeRoute

AStarStrategy::Search

具体的查找过程，每次还是会从子网络查找

Reference

城市道路分析：<https://geoffboeing.com/2016/11/osmnx-python-street-networks/> <https://automating-gis-processes.github.io/2018/notebooks/L6/network-analysis.html>
https://socialhub.technion.ac.il/wp-content/uploads/2017/08/revise_version-final.pdf <https://stackoverflow.com/questions/29639968/shortest-path-using-openstreetmap-datanodes-and-ways>

查找依赖库

由于c++中是允许重载，而函数名要唯一，所以需要名字修饰

```
ldd -r /media/data/k8s/apollo/bazel-bin/cyber/py_wrapper/../../_solib_k8/libcyber_Sclass_Uloader_Slibclass_Uloader.so  
c++filt _ZN4Poco13SharedLibraryC1ERKSS
```

也可以用”nm”命令查询：

```
nm -u /media/data/k8s/apollo/bazel-bin/cyber/py_wrapper/../../_solib_k8/libcyber_Sclass_Uloader_Slibclass_Uloader.so | grep _ZN4Poco13SharedLibraryC1ERKSS
```

[undefined symbol](https://blog.csdn.net/stpeace/article/details/76561814) [https://blog.csdn.net/stpeace/article/details/76561814]

[名字修饰](https://zh.wikipedia.org/wiki/%E5%90%8D%E5%AD%97%E4%BF%AE%E9%A5%B0)

[https://zh.wikipedia.org/wiki/%E5%90%8D%E5%AD%97%E4%BF%AE%E9%A5%B0]

CoreDump

<https://www.jianshu.com/p/3dc143c53ca2>

编译

gdb调试

<https://blog.csdn.net/davidhopper/article/details/82589722>
启动gdb调试

```
gdb -q bazel-bin/modules/map/relative_map/navigation_lane_test
```

进入GDB调试界面后，使用l命令查看源代码，使用b 138在源代码第138行（可根据需要修改为自己所需的代码位置）设置断点，使用r命令运行navigation_lane_test程序，进入断点暂停后，使用p navigation_lane_查看当前变量值（可根据需要修改为其他变量名），使用n单步调试一条语句，使用s单步调试进入函数内部，使用c继续执行后续程序。如果哪个部分测试通不过，调试信息会立刻告诉你具体原因，可使用bt查看当前调用堆栈。

bazel编译

编译清除

```
bazel clean --expunge
```

todo

tf_static的问题：

1. 为什么把obs_sensor2novatel_tf2_frame_id改为localization了？？？
2. 为什么发布的时候需要传入node，即多个node往一个topic发送，还是说多个线程访问同一个node（location和perception同时通过一个node发布topic）？
3. 为什么需要根据child_frame_id做过滤，即使child_frame_id不能一样，否则会后面会覆盖前面的？
（StaticTransformComponent::SendTransform）
4. 订阅的时候是每个模块都有listener吗，即遵循tf的设计原则。即代码可以复用，但是实例是每个模块自己new一个。
5. Buffer::SubscriptionCallbackImpl 中为什么时间戳比更新就清空整个List？

```
if (now.ToNanosecond() < last_update_.ToNanosecond()) {  
    AINFO << "Detected jump back in time. Clearing TF  
buffer.";  
    clear();  
    // cache static transform stamped again.  
    for (auto& msg : static_msgs_) {  
        setTransform(msg, authority, true);  
    }  
}
```

6. Buffer是一个单例，”apollo::transform::Buffer”提供了查询接口，即每个模块共享这个单例，和tf的设计原则还是不相符合？注意线程安全？

坐标系

UTM坐标系

UTM坐标系的坐标原点位于本初子午线与赤道交点，以正东方向为x轴正方向（UTM Easting），正北方向为y轴正方向（UTM Northing）

GPS坐标系

以地心为原点，连接南北两极并同纬线垂直相交的线叫做经线，垂直于经线的绕地球一圈就是纬线。

UTM坐标转GPS坐标

参考 [<https://www.ibm.com/developerworks/cn/java/j-coordconvert/index.html>]

<https://joancharmant.com/blog/sub-frame-accurate-synchronization-of-two-usb-cameras/>

<https://forums.ni.com/t5/Machine-Vision/Synchronized-capture-for-multiple-USB-cameras/td-p/1879647?profile.language=en>

<https://raspberrypi.stackexchange.com/questions/73588/usb-camera-synchronization-with-external-trigger>

<https://stackoverflow.com/questions/25161920/creating-synchronized-stereo-videos-using-webcams>

<https://discourse.vvvv.org/t/200-300-cameras-synchronized-triggering-and-access/13840>

<https://www.aliexpress.com/item/ELP-USB-Camera-Module-Micro-Mini-Industrial-USB-2-0-Synchronized-Dual-Lens-Stereo-Camera-Driverless/32866119916.html>

<https://raspberrypi.stackexchange.com/questions/58376/multiple-frame-synced-still-cameras>

lgsvl介绍

apollo桥接

lgsvl通过socket实现了apollo和lgsvl的桥接，主要的原理如下：

数据格式

需要重点关注的是数据格式的转换：

地图制作

lgsvl默认支持的地图为”San Francisco”，目前还不支持导入地图，如果需要自己制作地图可以按照下面的流程，在cityengine中制作城市的3维地图，然后导出模型到unity，用unity的地图编辑器编辑并且导出hd_map到apollo，之后在unity中创建场景，并且导入车的模型，整个地图的制作就完成了。

lgsvl场景介绍

我们以”SimpleMap”场景来举例子，介绍场景是如何组成的，SimpleMap由以下几个场景组成：

```
Directional light // 平行光
EventSystem // 事件系统
XE_Rigged-apollo // 车模型
ProceduralRoad // 地图中的房屋以及模型
MapSimple // 道路，交通灯模型
HDMapTool // apollo高精度地图制作工具
VectorMapTool // autoware地图制作工具
PointCloudTool // ?
PointCloudBounds // ?
```

```
MapOrigin // 地图起点，用来确定gps坐标? ?  
spawn_transform //  
spawn_transform(1)
```

EventSystem 事件系统检测游戏的事件，并且提供调用接口。例如键盘鼠标回调，射线检测。

Spawn GameObject 特殊的创建游戏对象事件。

[<https://docs.unity3d.com/Manual/UNetSpawning.html>]

sanfrancisco

Terrains 地形

碰撞采用的是[网格碰撞](#) [<http://docs.manew.com/Components/class-MeshCollider.html>]

Unity帮助文档

<http://docs.manew.com/Manual/71.html>

回放

<https://gamedev.stackexchange.com/questions/141149/action-replay-in-unity-3d>

https://www.gamasutra.com/blogs/AustonMontville/20141105/229437/Implementing_a_replay_system_in_Unity_and_how_Id_do_it_differently_next_time.php

<https://forum.unity.com/threads/best-approach-to-replay-system.624517/>