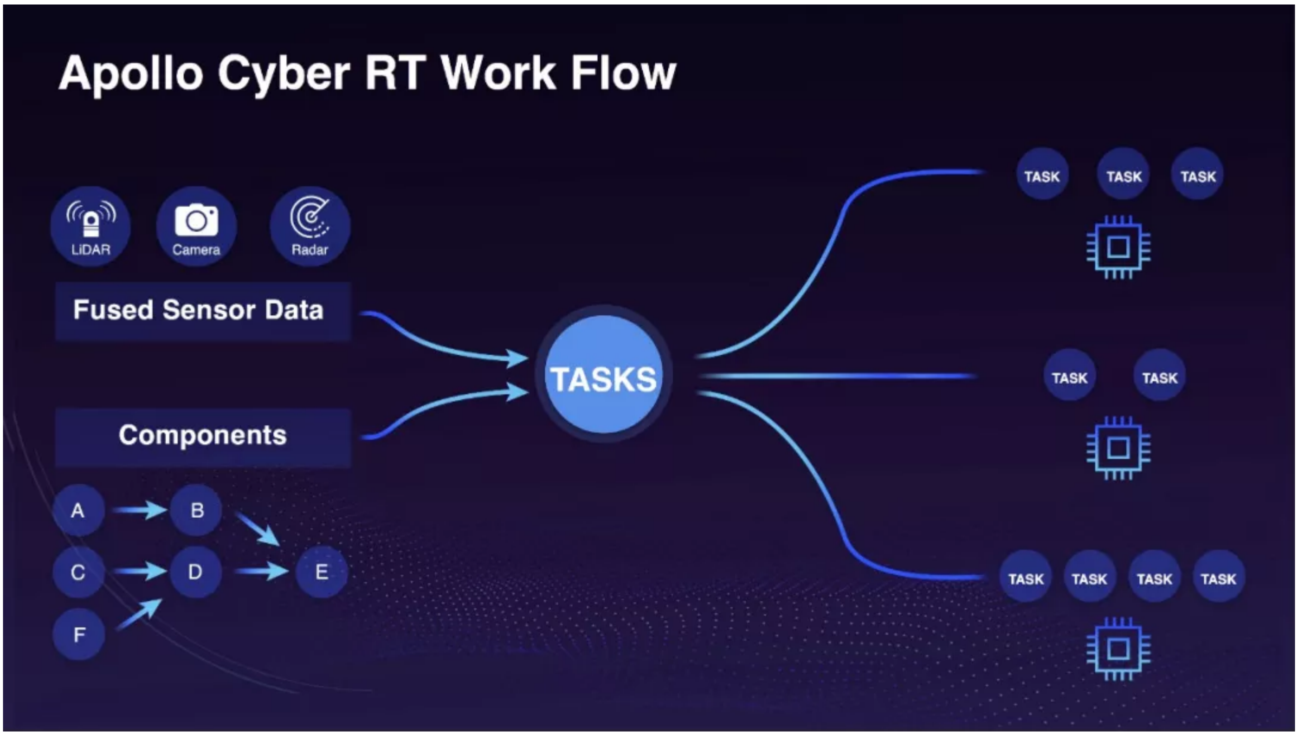
开发者说丨Apollo 3.5各功能模块启动过程解析

Apollo Cyber RT框架便是为Apollo构建此类框架的第一步，也是第一个专为自动驾驶技术设计的开源框架。



Apollo Cyber RT框架

Apollo Cyber RT 框架核心理念是基于的组件，组件有预先设定的输入输出。

实际上，每个组件就代表一个专用得算法模块。框架可以根据所有预定义的组件生成有向无环图（DAG)。

在运行时刻，框架把融合好的传感器数据和预定义的组件打包在一起形成用户级轻量任务，之后，框架的调度器可以根据资源可用性和任务优先级来派发这些任务。

Apollo 3.5彻底摒弃ROS，改用自研的Cyber作为底层通讯与调度平台。各功能模块的启动过程与之前版本天壤之别。

感谢社区荣誉布道师—贺博士 对Apollo 3.5 各功能模块的启动过程进行解析（关闭过程可作类似分析，不再赘述），希望给感兴趣的同学带来帮助。

DREAMVIEW模块启动过程

apollo 开发者社区

先从启动脚本文件scripts/bootstrap.sh开始剖析。

服务启动命令bash scripts/bootstrap.sh start实际上执行了scripts/bootstrap.sh脚本中的start函数：

```
dfunction start() {
    ./scripts/monitor.sh start
    ./scripts/dreamview.sh start
    if [ $? -eq 0 ]; then
        http_status="$(curl -o -I -L -s -w '%{http_code}' ${DREAMVIEW_URL})"
        if [ $http_status -eq 200 ]; then
            echo "Dreamview is running at" $DREAMVIEW_URL
        else
            echo "Failed to start Dreamview. Please check /apollo/data/log or /apollo/da
        fi
    fi
}
```

start函数内部分别调用脚本文件scripts/monitor.sh与scripts/dreamview.sh内部的start函数启动monitor与dreamview模块。

monitor 模 块 的 启 动 过 程 暂 且 按 下 不 表 ， 下 面 专 门 研 究 dreamview 模 块 的 start 函数。scripts/dreamview.sh文件内容如下：

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

cd "${DIR}/.."

source "${DIR}/apollo_base.sh"

# run function from apollo_base.sh
# run command_name module_name
run dreamview "$@"
```

里面压根没有`start`函数，但我们找到一个`apollo_base.sh`脚本文件，并且有一条调用语句：`run dreamview "$@"`（展开以后就是`run dreamview start`）。

我们有理由判断，`run`函数存在于`apollo_base.sh`脚本文件，现在到里面一探究竟，不出意外果然有一个`run`函数：

```
function run() {
  local module=$1
  shift
  run_customized_path $module $module "$@"
}
```

上述代码中，`module`的值为`dreamview`，`$@`的值为`start`，因此后面继续调用`run_customized_path dreamview dreamview start`。继续顺藤摸瓜查看`run_customized_path`函数：

```
function run_customized_path() {
  local module_path=$1
  local module=$2
  local cmd=$3
  shift 3
  case $cmd in
    start)
      start_customized_path $module_path $module "$@"
      ;;
   # ...
}
```

实际调用的是`start_customized_path dreamview dreamview`。

再来查看 `start_customized_path`函数：

```
function start_customized_path() {
  MODULE_PATH=$1
  MODULE=$2
  shift 2

  is_stopped_customized_path "${MODULE_PATH}" "${MODULE}"
  if [ $? -eq 1 ]; then
    eval "nohup cyber_launch start /apollo/modules/${MODULE_PATH}/launch/${MODULE}.launc
    sleep 0.5
    is_stopped_customized_path "${MODULE_PATH}" "${MODULE}"
    if [ $? -eq 0 ]; then
      echo "Launched module ${MODULE}."
```

```
        return 0
      else
        echo "Could not launch module ${MODULE}. Is it already built?"
        return 1
      fi
    else
      echo "Module ${MODULE} is already running - skipping."
      return 2
    fi
  }
```

在start_customized_path函数内部，首先调用is_stopped_customized_path函数来判断（在内部实际通过指令$(pgrep -c -f "modules/dreamview/launch/dreamview.launch")来判断）dreamview模块是否已启动。

若该模块未启动，则使用指令nohup cyber_launch start/apollo/modules/dreamview/launch/dreamview.launch &以非挂断方式启动后台进程模块dreamview。

cyber_launch是Cyber平台提供的一个python工具程序，其完整路径为：${APOLLO_HOME}/cyber/tools/cyber_launch/cyber_launch。可通过sudo find / -name cyber_launch查找，${APOLLO_HOME}表示Apollo项目的根目录。

以我的机器为例，Docker外部为/home/davidhopper/code/apollo，Docker内部自不必说，全部为/apollo。为描述简单起见，下文全部以Docker内部的路径/apollo为准。

下面继续研究cyber_launch中的main函数：

```
def main():
    """ main function """
    cyber_path = os.getenv('CYBER_PATH')
    if cyber_path == None:
        logger.error('Error: environment variable CYBER_PATH not found, set environment
        sys.exit(1)
    os.chdir(cyber_path)
    parser = argparse.ArgumentParser(description='cyber launcher')
    subparsers = parser.add_subparsers(help='sub-command help')

    start_parser = subparsers.add_parser('start', help='launch/benchmark.launch')
    start_parser.add_argument('file', nargs='?', action='store', help='launch file, defa

```

```
    stop_parser = subparsers.add_parser('stop', help='stop all the module in launch file
    stop_parser.add_argument('file', nargs='?', action='store', help='launch file, defau

    #restart_parser = subparsers.add_parser('restart', help='restart the module')
    #restart_parser.add_argument('file', nargs='?', action='store', help='launch file, d

    params = parser.parse_args(sys.argv[1:])

    command = sys.argv[1]
    if command == 'start':
        start(params.file)
    elif command == 'stop':
        stop_launch(params.file)
    #elif command == 'restart':
    #    restart(params.file)
    else:
        logger.error('Invalid command %s' % command)
        sys.exit(1)

```

该函数无非进行一些命令行参数解析，

然后调用`start(/apollo/modules/dreamview/launch/dreamview.launch)`函数启动`dreamview`模块。

继续查看`start`函数，该函数内容很长，不再详细解释，

其主要功能是解析XML文件`/apollo/modules/dreamview/launch/dreamview.launch`中的各项元素：`name`、`dag_conf`、`type`、`process_name`、`exception_handler`，

其值分别为：

`dreamview`、`null`、`binary`、`/apollo/bazel-bin/modules/dreamview/dreamview --flagfile =/apollo/modules/common/data/global_flagfile.txt`、`respawn`，然后调用`ProcessWrapper(process_name.split()[0], 0, [""], process_name, process_type, exception_handler)`创建一个`ProcessWrapper`对象`pw`，

然后调用`pw.start()`函数启动`dreamview`模块：

```
def start(launch_file = ''):
    # ...

    process_list = []
    root = tree.getroot()
    for module in root.findall('module'):
        module_name = module.find('name').text
        dag_conf = module.find('dag_conf').text
```

```
            process_name = module.find('process_name').text
            sched_name = module.find('sched_name')
            process_type = module.find('type')
            exception_handler = module.find('exception_handler')
            # ...
            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, proc
                # default is library
                else:
                    pw = ProcessWrapper(g_binary_name, 0, dag_dict[str(process_name)], proce
                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)

        # no module in xml
        if not process_list:
            logger.error("No module was found in xml config.")
            return
        all_died = pmon.run()
        if not all_died:
            logger.info("Stop all processes...")
            stop()
        logger.info("Cyber exit.")
```

下面查看`ProcessWrapper`类里的`start`函数：

```
        def start(self):
            """
            start a manager in process name
            """
            if self.process_type == 'binary':
                args_list = self.name.split()
            else:
                args_list = [self.binary_path, '-d'] + self.dag_list
                if len(self.name) != 0:
                    args_list.append('-p')
                    args_list.append(self.name)
                if len(self.sched_name) != 0:
                    args_list.append('-s')
                    args_list.append(self.sched_name)

            self.args = args_list

            try:
                self.popen = subprocess.Popen(args_list, stdout=subprocess.PIPE, stderr=subp
```

```python
        except Exception, e:
            logger.error('Subprocess Popen exception: ' + str(e))
            return 2
        else:
            if self.popen.pid == 0 or self.popen.returncode is not None:
                logger.error('Start process [%s] failed.' % self.name)
                return 2

        th = threading.Thread(target=module_monitor, args=(self, ))
        th.setDaemon(True)
        th.start()
        self.started = True
        self.pid = self.popen.pid
        logger.info('Start process [%s] successfully. pid: %d' % (self.name, self.popen.
        logger.info('-' * 120)
        return 0
```

在该函数内部调用`/apollo/bazel-bin/modules/dreamview/dreamview --flagfile=/apollo/modules/common/data/global_flagfile.txt`最终启动了`dreamview`进程。

`dreamview`进程的`main`函数位于`/apollo/modules/dreamview/backend/main.cc`中，内容如下所示：

```cpp
int main(int argc, char *argv[]) {
  google::ParseCommandLineFlags(&argc, &argv, true);
  // add by caros for dv performance improve
  apollo::cyber::GlobalData::Instance()->SetProcessGroup("dreamview_sched");
  apollo::cyber::Init(argv[0]);

  apollo::dreamview::Dreamview dreamview;
  const bool init_success = dreamview.Init().ok() && dreamview.Start().ok();
  if (!init_success) {
    AERROR << "Failed to initialize dreamview server";
    return -1;
  }
  apollo::cyber::WaitForShutdown();
  dreamview.Stop();
  apollo::cyber::Clear();
  return 0;
}
```

该函数初始化`Cyber`环境，并调用`Dreamview::Init()`和`Dreamview::Start()`函数，启动`Dreamview`后台监护进程。

然后进入消息处理循环，直到等待`cyber::WaitForShutdown()`返回，清理资源并退出`main`函数。

Apollo 3.5使用`Cyber`启动`Localization`、`Perception`、`Prediction`、`Planning`、`Control`等功能模块。若只看各模块的BUILD文件，保证你无法找到该模块的启动入口`main`函数（Apollo 3.5之前的版本均是如此处理）。

下面以`Planning`模块为例具体阐述。

`Planning`模块`BUILD`文件中生成`binary`文件的配置项如下：

```
cc_binary(
  name = "libplanning_component.so",
  linkshared = True,
  linkstatic = False,
  deps = [":planning_component_lib"],
)
```

该配置项中没有`source`文件，仅包含一个依赖项:`planning_component_lib`。又注意到后者的定义如下：

```
cc_library(
    name = "planning_component_lib",
    srcs = [
        "planning_component.cc",
```

```
    ],
    hdrs = [
        "planning_component.h",
    ],
    copts = [
        "-DMODULE_NAME=\\\"planning\\\"",
    ],
    deps = [
        ":planning_lib",
        "//cyber",
        "//modules/common/adapters:adapter_gflags",
        "//modules/common/util:message_util",
        "//modules/localization/proto:localization_proto",
        "//modules/map/relative_map/proto:navigation_proto",
        "//modules/perception/proto:perception_proto",
        "//modules/planning/proto:planning_proto",
        "//modules/prediction/proto:prediction_proto",
    ],
)
```

在srcs文件planning_component.cc以及deps文件中均找不到main函数。

那么main函数被隐藏在哪里？如果没有main函数，binary文件libplanning_component.so又是如何启动的？

答案很简单，planning模块的binary文件libplanning_component.so作为cyber的一个组件启动，不需要main函数。

下面详细阐述在DreamView界面中启动Planning模块的过程。

DreamView前端界面操作此处不表，后端的消息响应函数HMI::RegisterMessageHandlers()位于/apollo/modules/dreamview/backend/hmi/hmi.cc文件中：

```
void HMI::RegisterMessageHandlers() {

  // ...
  websocket_->RegisterMessageHandler(
      "HMIAction",
      [this](const Json& json, WebSocketHandler::Connection* conn) {
        // Run HMIWorker::Trigger(action) if json is {action: ""}
        // Run HMIWorker::Trigger(action, value) if "value" field is provided.
        std::string action;
        if (!JsonUtil::GetStringFromJson(json, "action", &action)) {
          AERROR << "Truncated HMIAction request.";
          return;
```

```
        }
        HMIAction hmi_action;
        if (!HMIAction_Parse(action, &hmi_action)) {
          AERROR << "Invalid HMIAction string: " << action;
        }
        std::string value;
        if (JsonUtil::GetStringFromJson(json, "value", &value)) {
          hmi_worker_->Trigger(hmi_action, value);
        } else {
          hmi_worker_->Trigger(hmi_action);
        }

        // Extra works for current Dreamview.
        if (hmi_action == HMIAction::CHANGE_MAP) {
          // Reload simulation map after changing map.
          CHECK(map_service_->ReloadMap(true))
              << "Failed to load new simulation map: " << value;
        } else if (hmi_action == HMIAction::CHANGE_VEHICLE) {
          // Reload lidar params for point cloud service.
          PointCloudUpdater::LoadLidarHeight(FLAGS_lidar_height_yaml);
          SendVehicleParam();
        }
      });

  // ...
}
```

其中，`HMIAction_Parse(action, &hmi_action)`用于解析动作参数，`hmi_worker_->Trigger(hmi_action, value)`用于执行相关动作。

对于`Planning`模块的启动而言，`hmi_action`的值为`HMIAction::START_MODULE`，`value`的值为`Planning`。

实际上，`DreamView`将操作模式分为多种`hmi mode`，这些模式位于目录`/apollo/modules/dreamview/conf/hmi_modes`，每一个配置文件均对应一种`hmi mode`。

但不管处于哪种`hmi mode`，对于`Planning`模块的启动而言，`hmi_action`的值均为`HMIAction::START_MODULE`，`value`的值均为`Planning`。

当然，`Standard Mode`与`Navigation Mode`对应的`dag_files`不一样，`Standard Mode`的`dag_files`为`/apollo/modules/planning/dag/planning.dag`，`Navigation Mode`的`dag_files`为`/apollo/modules/planning/dag/planning_navi.dag`。

HMIWorker::Trigger(const HMIAction action, const std::string& value) 函数位于文件/apollo/modules/dreamview/backend/hmi/hmi_worker.cc中，其内容如下：

```
bool HMIWorker::Trigger(const HMIAction action, const std::string& value) {
  AINFO << "HMIAction " << HMIAction_Name(action) << "(" << value
        << ") was triggered!";
  switch (action) {
    // ...
    case HMIAction::START_MODULE:
      StartModule(value);
      break;
    // ...
  }
  return true;
}
```

继续研究HMIWorker::StartModule(const std::string& module)函数：

```
void HMIWorker::StartModule(const std::string& module) const {
  const Module* module_conf = FindOrNull(current_mode_.modules(), module);
  if (module_conf != nullptr) {
    System(module_conf->start_command());
  } else {
    AERROR << "Cannot find module " << module;
  }
}
```

上述函数中成员变量current_mode_保存着当前hmi mode对应配置文件包含的所有配置项。

例如modules/dreamview/conf/hmi_modes/mkz_standard_debug.pb.txt

里面就包含了MKZ标准调试模式下所有的功能模块，

该配置文件通过HMIWorker::LoadMode(const std::string& mode_config_path)函数读入到成员变量current_mode_中。

如果基于字符串module查找到了对应的模块名以及对应的启动配置文件dag_files，则调用System函数（内部实际调用std::system函数)基于命令module_conf->start_command()启动一个进程。

这个start_command从何而来？

需进一步分析`HMIWorker::LoadMode(const std::string& mode_config_path)`函数：

```
HMIMode HMIWorker::LoadMode(const std::string& mode_config_path) {
  HMIMode mode;
  CHECK(common::util::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.
    CHECK(!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());

    // Construct start_command:
    //     nohup mainboard -p  -d  ... &
    module.set_start_command("nohup mainboard");
    const auto& process_group = cyber_module.process_group();
    if (!process_group.empty()) {
      StrAppend(module.mutable_start_command(), " -p ", process_group);
    }
    for (const std::string& dag : cyber_module.dag_files()) {
      StrAppend(module.mutable_start_command(), " -d ", dag);
    }
    StrAppend(module.mutable_start_command(), " &");

    // Construct stop_command: pkill -f ''
    const std::string& first_dag = cyber_module.dag_files(0);
    module.set_stop_command(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();
  AINFO << "Loaded HMI mode: " << mode.DebugString();
  return mode;
}
```

通过该函数可以看到，构建出的`start_command`格式为`nohup mainboard -p-d... &`，

其中，`process_group`与`dag`均来自于当前hmi mode对应的配置文件。

以`modules/dreamview/conf/hmi_modes/mkz_close_loop.pb.txt`为例，

它包含两个`cyber_modules`配置项，对于`Computer`模块而言，它包含了11个`dag_files`文件（对应11个子功能模块），这些子功能模块全部属于名为`compute_sched`的`process_group`。

dag自不必言，每个子功能模块对应一个dag_files，

Planning子功能模块对应的dag_files为/apollo/modules/planning/dag/planning.dag。

```
cyber_modules {
  key: "Computer"
  value: {
    dag_files: "/apollo/modules/drivers/camera/dag/camera_no_compress.dag"
    dag_files: "/apollo/modules/drivers/gnss/dag/gnss.dag"
    dag_files: "/apollo/modules/drivers/radar/conti_radar/dag/conti_radar.dag"
    dag_files: "/apollo/modules/drivers/velodyne/dag/velodyne.dag"
    dag_files: "/apollo/modules/localization/dag/dag_streaming_msf_localization.dag"
    dag_files: "/apollo/modules/perception/production/dag/dag_streaming_perception.dag"
    dag_files: "/apollo/modules/perception/production/dag/dag_streaming_perception_traff
    dag_files: "/apollo/modules/planning/dag/planning.dag"
    dag_files: "/apollo/modules/prediction/dag/prediction.dag"
    dag_files: "/apollo/modules/routing/dag/routing.dag"
    dag_files: "/apollo/modules/transform/dag/static_transform.dag"
    process_group: "compute_sched"
  }
}
cyber_modules {
  key: "Controller"
  value: {
    dag_files: "/apollo/modules/canbus/dag/canbus.dag"
    dag_files: "/apollo/modules/control/dag/control.dag"
    dag_files: "/apollo/modules/guardian/dag/guardian.dag"
    process_group: "control_sched"
  }
}
# ...
```

至此，我们终于找到了Planning功能模块的启动命令为：

```
nohup mainboard -p compute_sched -d /apollo/modules/planning/dag/planning.dag &
```

nohup表示非挂断方式启动，mainboard无疑是启动的主程序，入口main函数必定包含于其中。

process_group的意义不是那么大，无非对功能模块分组而已；dag_files才是我们启动相关功能模块的真正配置文件。

查看cyber模块的构建文件/apollo/cyber/BUILD，可发现如下内容：

```
cc_binary(
    name = "mainboard",
    srcs = [
        "mainboard/mainboard.cc",
        "mainboard/module_argument.cc",
        "mainboard/module_argument.h",
        "mainboard/module_controller.cc",
        "mainboard/module_controller.h",
    ],
    copts = [
        "-pthread",
    ],
    linkstatic = False,
    deps = [
        ":cyber_core",
        "//cyber/proto:dag_conf_cc_proto",
    ],
)
```

至此，可执行文件`mainboard`的踪迹水落石出。

果不其然，入口函数`main`位于文件`cyber/mainboard/mainboard.cc`中：

```
int main(int argc, char** argv) {
  google::SetUsageMessage("we use this program to load dag and run user apps.");

  // parse the argument
  ModuleArgument module_args;
  module_args.ParseArgument(argc, argv);

  // initialize cyber
  apollo::cyber::Init(argv[0]);

  // start module
  ModuleController controller(module_args);
  if (!controller.Init()) {
    controller.Clear();
    AERROR << "module start error.";
    return -1;
  }

  apollo::cyber::WaitForShutdown();
  controller.Clear();
  AINFO << "exit mainboard.";

  return 0;
}
```

`main`函数十分简单，首先是解析参数，初始化`cyber`环境，

接下来创建一个`ModuleController`类对象`controller`，

之后调用`controller.Init()`启动相关功能模块。

最后，一直等待`cyber::WaitForShutdown()`返回，清理资源并退出`main`函数。

`ModuleController::Init()`函数十分简单，内部调用了`ModuleController::LoadAll()`函数：

```
bool ModuleController::LoadAll() {
  const std::string work_root = common::WorkRoot();
  const std::string current_path = common::GetCurrentPath();
  const std::string dag_root_path = common::GetAbsolutePath(work_root, "dag");

  for (auto& dag_conf : args_.GetDAGConfList()) {
    std::string module_path = "";
    if (dag_conf == common::GetFileName(dag_conf)) {
      // case dag conf argument var is a filename
      module_path = common::GetAbsolutePath(dag_root_path, dag_conf);
    } else if (dag_conf[0] == '/') {
      // case dag conf argument var is an absolute path
      module_path = dag_conf;
    } else {
      // case dag conf argument var is a relative path
      module_path = common::GetAbsolutePath(current_path, dag_conf);
      if (!common::PathExists(module_path)) {
        module_path = common::GetAbsolutePath(work_root, dag_conf);
      }
    }
    AINFO << "Start initialize dag: " << module_path;
    if (!LoadModule(module_path)) {
      AERROR << "Failed to load module: " << module_path;
      return false;
    }
  }
  return true;
}
```

上述函数处理一个`dag_conf`配置文件循环，

读取配置文件中的所有`dag_conf`，

并逐一调用

`bool ModuleController::LoadModule(const std::string& path)`函数加载功能模块：

```
bool ModuleController::LoadModule(const std::string& path) {
  DagConfig dag_config;
  if (!common::GetProtoFromFile(path, &dag_config)) {
    AERROR << "Get proto failed, file: " << path;
    return false;
  }
```

```
  return LoadModule(dag_config);
}
```

上述函数从磁盘配置文件读取配置信息，

并调用`bool ModuleController::LoadModule(const DagConfig& dag_config)`

函数加载功能模块：

```
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 not exist: " << load_path;
      return false;
    }

    class_loader_manager_.LoadLibrary(load_path);

    for (auto& component : module_config.components()) {
      const std::string& class_name = component.class_name();
      std::shared_ptr base =
          class_loader_manager_.CreateClassObj(class_name);
      if (base == nullptr) {
        return false;
      }

      if (!base->Initialize(component.config())) {
        return false;
      }
      component_list_.emplace_back(std::move(base));
    }

    for (auto& component : module_config.timer_components()) {
      const std::string& class_name = component.class_name();
      std::shared_ptr base =
          class_loader_manager_.CreateClassObj(class_name);
      if (base == nullptr) {
        return false;
      }

      if (!base->Initialize(component.config())) {
        return false;
      }
```

```
            component_list_.emplace_back(std::move(base));
        }
    }
    return true;
}
```

上述函数看似很长，

核心思想无非是调用`class_loader_manager_.LoadLibrary(load_path);`

加载功能模块，创建并初始化功能模块类对象，并将该功能模块加入到`cyber`的组件列表中统一调度管理。

整个`Planning`模块的启动过程已阐述完毕，但仍有一个问题需要解决：

`Planning`模块是如何作为`Cyber`的一个组件注册并动态创建的？

3.1 组件注册过程

首先看组件注册过程。

注意到modules/planning/planning_component.h

的组件类PlanningComponent继承自

cyber::Component</prediction::predictionobstacles,>,

里面管理着PlanningBase类对象指针（Apollo 3.5基于场景概念进行规划，目前从PlanningBase类派生出三个规划类：StdPlanning（高精地图模式）、NaviPlanning（实时相对地图模式）、OpenSpacePlanning（自由空间模式），可通过目录modules/planning/dag下的配置文件指定选用何种场景）。

同时，使用宏CYBER_REGISTER_COMPONENT(PlanningComponent)将规划组件PlanningComponent注册到Cyber的组件类管理器。查看源代码可知：

```
#define CYBER_REGISTER_COMPONENT(name) \
  CLASS_LOADER_REGISTER_CLASS(name, apollo::cyber::ComponentBase)
```

而后者的定义为：

```
#define CLASS_LOADER_REGISTER_CLASS(Derived, Base) \
  CLASS_LOADER_REGISTER_CLASS_INTERNAL_1(Derived, Base, __COUNTER__)
```

继续展开得到：

```
#define CLASS_LOADER_REGISTER_CLASS_INTERNAL_1(Derived, Base, UniqueI
  CLASS_LOADER_REGISTER_CLASS_INTERNAL(Derived, Base, UniqueID)
```

仍然需要进一步展开：

```
#define CLASS_LOADER_REGISTER_CLASS_INTERNAL(Derived, Base, UniqueID
  namespace {
  struct ProxyType##UniqueID {
    ProxyType##UniqueID() {
      apollo::cyber::class_loader::utility::RegisterClass(      \
          #Derived, #Base);
    }
  };
  static ProxyType##UniqueID g_register_class_##UniqueID;
  }
```

将PlanningComponent代入上述宏，最终得到：

```
namespace {
struct ProxyType__COUNTER__ {
  ProxyType__COUNTER__() {
    apollo::cyber::class_loader::utility::RegisterClass(
        "PlanningComponent", "apollo::cyber::ComponentBase");
  }
};
static ProxyType__COUNTER__ g_register_class___COUNTER__;
}
```

注意两点：

第一，上述定义位于`namespace apollo::planning`内；

第二，`___COUNTER__`是C语言的一个计数器宏，这里仅代表一个占位符，实际展开时可能就是78之类的数字，亦即`ProxyType__COUNTER__`实际上应为`ProxyType78`之类的命名。

上述代码简洁明了，首先定义一个结构体`ProxyType__COUNTER__`，

该结构体仅包含一个构造函数，

在内部调用`apollo::cyber::class_loader::utility::RegisterClass`

`<planningcomponent,< span="">< span="">apollo::cyber::ComponentBase>`注册a

`ComponentBase`类的派生类`PlanningComponent`。

并定义一个静态全局结构体

`ProxyType__COUNTER__`变量：`g_register_class___COUNTER__`。

继续观察

`apollo::cyber::class_loader::utility::RegisterClass`函数：

```
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* new_class_factrory_obj =
      new utility::ClassFactory(class_name, base_class_name);
  new_class_factrory_obj->AddOwnedClassLoader(GetCurActiveClassLoade
  new_class_factrory_obj->SetRelativeLibraryPath(GetCurLoadingLibrar

  GetClassFactoryMapMapMutex().lock();
  ClassClassFactoryMap& factory_map =
      GetClassFactoryMapByBaseClass(typeid(Base).name());
```

```
    factory_map[class_name] = new_class_factrory_obj;
    GetClassFactoryMapMapMutex().unlock();
  }
```

该函数创建一个模板类

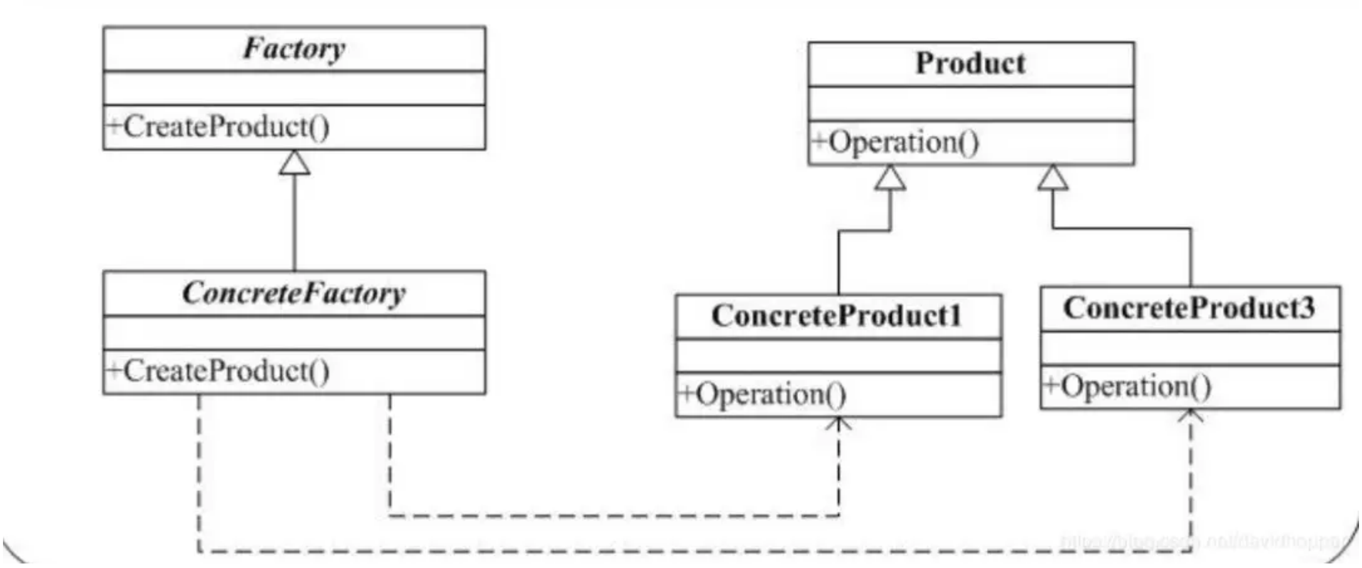
`utility::ClassFactory`对象`new_class_factrory_obj`，

为其添加类加载器，设置加载库的路径，

最后将工厂类对象加入到`ClassClassFactoryMap`对象`factory_map`统一管理。

通过该函数，我们可以清楚地看到，

`Cyber`使用工厂方法模式完成产品类（例如`PlanningComponent`）对象的创建：



3.2 动态创建过程

根据第二节内容，

功能模块类`PlanningComponent`对象在

`bool ModuleController::LoadModule(const DagConfig& dag_config)`函数内部创建：

```
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;
    // ...
    class_loader_manager_.LoadLibrary(load_path);
```

```
    for (auto& component : module_config.components()) {
      const std::string& class_name = component.class_name();
      std::shared_ptr base =
          class_loader_manager_.CreateClassObj(class_name);
      if (base == nullptr) {
        return false;
      }

      if (!base->Initialize(component.config())) {
        return false;
      }
      component_list_.emplace_back(std::move(base));
    }

    // ...
  }
  return true;
}
```

已经知道，PlanningComponent对象是通过class_loader_manager_.CreateClassObj(class_name)创建出来的，而class_loader_manager_是一个class_loader::ClassLoaderManager类对象。

现在的问题是：class_loader::ClassLoaderManager与3.1节中的工厂类utility::AbstractClassFactory如何联系起来的？

先看ClassLoaderManager::CreateClassObj函数（位于文件 cyber/class_loader/class_loader_manager.h 中）：

```
template <typename Base>
std::shared_ptr ClassLoaderManager::CreateClassObj(
    const std::string& class_name) {
  std::vector class_loaders = GetAllValidClassLoaders();
  for (auto class_loader : class_loaders) {
    if (class_loader->IsClassValid(class_name)) {
      return (class_loader->CreateClassObj(class_name));
    }
  }
  AERROR << "Invalid class name: " << class_name;
  return std::shared_ptr();
}
```

上述函数中，从所有class_loaders中找出一个正确的class_loader，并调用class_loader->CreateClassObj(class_name)

（位于文件cyber/class_loader/class_loader.h中）

创建功能模块组件类对象：

```
template <typename Base>
std::shared_ptr ClassLoader::CreateClassObj(
    const std::string& class_name) {
  if (!IsLibraryLoaded()) {
    LoadLibrary();
  }

  Base* class_object = utility::CreateClassObj(class_name, this);
  if (nullptr == class_object) {
    AWARN << "CreateClassObj failed, ensure class has been registere
          << "classname: " << class_name << ",lib: " << GetLibraryPa
    return std::shared_ptr();
  }

  std::lock_guard<std::mutex> lck(classobj_ref_count_mutex_);
  classobj_ref_count_ = classobj_ref_count_ + 1;
  std::shared_ptr classObjSharePtr(
      class_object, std::bind(&ClassLoader::OnClassObjDeleter, this,
                              std::placeholders::_1));
  return classObjSharePtr;
}
```

上述函数继续调用utility::CreateClassObj(class_name, this)

（位于文件cyber/class_loader/utility/class_loader_utility.h中）

创建功能模块组件类对象：

```
template <typename Base>
Base* CreateClassObj(const std::string& class_name, ClassLoader* loa
  GetClassFactoryMapMapMutex().lock();
  ClassClassFactoryMap& factoryMap =
      GetClassFactoryMapByBaseClass(typeid(Base).name());
  AbstractClassFactory* factory = nullptr;
  if (factoryMap.find(class_name) != factoryMap.end()) {
    factory = dynamic_cast(
        factoryMap[class_name]);
  }
  GetClassFactoryMapMapMutex().unlock();

  Base* classobj = nullptr;
  if (factory && factory->IsOwnedBy(loader)) {
    classobj = factory->CreateObj();
  }

  return classobj;
}
```

上述函数使用`factory = dynamic_cast( factoryMap[class_name]);`

获取对应的工厂对象指针，

至此终于将`class_loader::ClassLoaderManager`与3.1节中的工厂类`utility::AbstractClassFactory`联系起来了。

工厂对象指针找到后，使用`classobj = factory->CreateObj();`就顺理成章地将功能模块类对象创建出来了。

END