cyber/mianboard 目录下的函数

主要函数结构

```
├── 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
```

部分函数功能解释

- `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

mainboard

- DAG 文件是指有向无环图文件 (Directed Acyclic Graph)

- `int mian(int argc, char** argv)` -> 是程序的主函数 其中`argc`是一个整数 表示命令行参数的数量 `argv`是一个字符指针数组 其中每一个元素都是指向命令行参数的指针 例如如果你在命令行中运行并传入两个参数 ./pro arg1 arg2 那么 `argc`的值为 3(包括程序名称) 而`argv[0]` `argv[1]` `arg2` 分别为 "./pro" "arg1" "arg2"

```
#include "cyber/common/global_data.h"
#include "cyber/common/log.h"
#include "cyber/init.h"
#include "cyber/mainboard/module_argument.h"
#include "cyber/mainboard/module_controller.h"
#include "cyber/state.h"

using apollo::cyber::mainboard::ModuleArgument;
using apollo::cyber::mainboard::ModuleController;

int main(int argc, char** argv) {
  // parse the argument
  // 解析参数
  ModuleArgument module_args;
  module_args.ParseArgument(argc, argv);

  // initialize cyber
  // 初始化模块
  apollo::cyber::Init(argv[0]);

  // start module
  // 运行模块
```

```
  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;
}
```

---

module_argument

- 这里是其中的部分函数
- 解析参数 主要是解析加载 `DAG` 文件时候带的参数

- `binary_name_ = std::string(basename(argv[0]));` -> `basename()`从给定路径名中删除所有路径 直到并包括最后一个路径分隔符 (如果有) 例如 有一个路径名 "/usr/local/r/Pro.R" 那么`basename`函数将返回文件名"Pro.R"
- `GlobalData::Instance()->SetProcessGroup(process_group_);` -> 这里调用了 `GlobalData::Instance()`静态方法 它返回一个指向`GlobalData`类的单例对象的指针 然后调用该对象的 `SetProcessGroup`方法 并将变量`process_group_`传递给它

```
void ModuleArgument::ParseArgument(const int argc, char* const argv[]) {
  // 二进制模块名
  binary_name_ = std::string(basename(argv[0]));
  // 解析参数
  GetOptions(argc, argv);

  // 如果没有 process_group_ 和 shced_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 则设置为对应参数
  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
```

```
  for (std::string& dag : dag_conf_list_) {
    AINFO << "dag_conf: " << dag;
  }
}
```

---

module_controller

- 这里是其中的部分函数 LoadModule 函数
- 实现`cyber`模块的加载

- `module_config.module_library().front()` -> 调用了`modual_config`对象的`modual_library`方法 它返回一个包含模块库信息的容器 然后它调用了该容器的`front`方法 它返回一个容器中第一个元素的引用 这段代码的目的是获取模块配置中指定的第一个模块库
- `common::GetAbsolutePath(work_root, module_config.module_library());` -> 这段代码调用了`common::GetAbsolutePath`函数获取指定路径的绝对路径 `work_root`是一个字符串 表示工作目录的路径 `module_config.module_library()`返回一个包含模块库信息的容器 该函数将这两个参数组合起来 生成一个绝对路径

```
  // 找到模块路径
  if (module_config.module_library().front() == '/') {
    load_path = module_config.module_library();
  } else {
    load_path =
        common::GetAbsolutePath(work_root, module_config.module_library());
  }
```

- `class_loader_manager_.LoadLibrary(load_path);` -> `class_loader_manager_`加载模块 加载好对应的类之后再创建对应的对象 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)
- `const std::string& class_name = component.class_name();` -> 调用 `component`对象的`class_name`方法 它返回是一个字符串 表示组件的类名称
- `std::shared_ptr<ComponentBase> base = class_loader_manager_.CreateClassObj<ComponentBase>(class_name);` -> 调用`class_loader_manager_`对象的`CreateClassObj`方法 它用创建指定类名称的对象`class_name`是一个字符串 表示要创建的类的名称 该方法返回一个指向新创建对象的智能指针 然后赋值给`base`的变量 它是一个类型为`std::shared_ptr<ComponentBase>`的智能指针
- `std::shared_ptr<ComponentBase>` -> 是一个模板类 它表示一个指向`ComponentBase`类型对象的智能指针 智能指针是一个自动管理其所指向的对象生命周期的指针 当最后一个`std::shared_ptr`对象不再指向该对象时 该对象将被自动删除 这样我们不需要手动管理内存 可以避免内存泄漏和悬挂指针等问题
- `std::move(base)` -> 是一个函数 用于指示对象可以被"移动" 即允许从一个对象有效地转移资源到另一个对象 它产生一个标识其参数`t`的`xvalue`表达式
  举一个例子

```
  #include <iostream>
  #include <utility>
```

```cpp
#include <vector>

int main()
{
    std::vector<std::string> v1 = {"a", "b", "c"};
    std::vector<std::string> v2 = {"x", "y", "z"};

    std::cout << "v1: ";
    for (const auto& s : v1) std::cout << s << ' ';
    std::cout << "\nv2: ";
    for (const auto& s : v2) std::cout << s << ' ';
    std::cout << '\n';

    v2 = std::move(v1); // 移动赋值

    std::cout << "\nv1: ";
    for (const auto& s : v1) std::cout << s << ' ';
    std::cout << "\nv2: ";
    for (const auto& s : v2) std::cout << s << ' ';
}
```

* 输出的结果为

```
v1: a b c
v2: x y z

v1:
v2: a b c
```

```cpp
// 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));
    }
```