cyber/class_load

主要函数结构

```
├── 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
```

- 类加载器的作用就是动态的加载动态库 然后实例化对象 -> 首先`apollo`中的各个`module`都会编译一个动态库 拿`planing`模块来举例子 在`planning/dag/planning.dag`中会加载下面这个模块:

```
    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 之后在调用`Initialize`方法初始化

部分函数功能解释

class_loader

- 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都是调用`utilty`类中的实现

```
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;
  // 获取类的名称
```

```
  template <typename Base>
  std::vector<std::string> GetValidClassNames();
  // 实例化对象
  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 OnClassObjDeleter(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_;
};
```

- 实列化对象的实现
- 在创建类的时候 类引用计数加 1 并且绑定类的析构函数(`OnClassObjDeleter`) 删除对象让类引用计数减 1

```
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);

  if (class_object == nullptr) {
    AWARN << "CreateClassObj failed, ensure class has been registered. "
          << "classname: " << class_name << ",lib: " << GetLibraryPath();
    return std::shared_ptr<Base>();
  }

  // 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::OnClassObjDeleter<Base>, this,
                              std::placeholders::_1));
  return classObjSharePtr;
}
```

```
template <typename Base>
void ClassLoader::OnClassObjDeleter(Base* obj) {
  if (nullptr == obj) {
    return;
  }

  std::lock_guard<std::mutex> lck(classobj_ref_count_mutex_);
  delete obj;
  -- classobj_ref_count_;
}
```

---

class_loader_manager

- 类加载器管理 实际是管理不同的`classloader` 而不同的`libpath`对应不同的`classloader`
- `ClassLoaderManager`主要的数据结构如下：

```
    std::map<std::string, ClassLoader*> libpath_loader_map_;
```

- 其中`key`为`library_path` 而`value`为`ClassLoader`
- 也就是说`ClassLoaderManager`对`ClassLoader`进行保存和管理

```
    bool ClassLoaderManager::LoadLibrary(const std::string& library_path) {
    std::lock_guard<std::mutex> lck(libpath_loader_map_mutex_);
    if (!IsLibraryValid(library_path)) {
        libpath_loader_map_[library_path] =
            new class_loader::ClassLoader(library_path);
        }
        return IsLibraryValid(library_path);
    }
```
