cyber/component

Component 介绍

- component是cyber为了帮助我们特意实现的类
- component加载的时候会自动帮我们创建一个node 通过node来订阅和发布对应消息 每个component有且只能对应一个node
- component对用户提供两个接口Init()和Proc() 用户在Init()中进行初始化 在Proc()中接受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 也可以对应到在dreamviewer上动态的打开关系模块

Component 工作流程

- 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
- 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)
- 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)
- component帮助用户把初始化和数据收发的流程进行封装 减少用户的工作量 component封装了整个数据的收发流程 component本身并不是一个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入口在cyber/mainboard/http://mainboard.cc中
- 主函数会先进行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) {
  // 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
  // 运行模块 启动 cyber 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;
}
```

component 动态加载

- `cyber`函数在`ModuleController::Init()`进行加载 具体的加载过程在`ModuleController::LoadModule`中
- 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

```
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;
    }

    // 加载动态库
    class_loader_manager_.LoadLibrary(load_path);

    // 加载消息触发模块
    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));
    }
```

```
    // 加载定时触发模块
    for (auto& component : module_config.timer_components()) {
      // 创建对象
      const std::string& class_name = component.class_name();
      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));
    }
  }
  return true;
}
```

部分函数功能解释

component

- `component`一共有四个模板类 分别对应接受 0-3 个消息 这里主要分析 2 个消息的情况
- 创建`node`节点(一个`component`只能有一个`node`节点 之后用户可以用`node_`在`init`中创建`reader`或`writer`)
- 调用用户自定义的初始化函数`Init()`(子类的`Init`方法)
- 创建`reader` 订阅几个消息就创建几个`reader`
- 创建回调函数 实际上执行用户定义算法`Proc()`函数
- 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
- 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

```
template <typename M0, typename M1>
bool Component<M0, M1, NullType, NullType>::Initialize(
    const ComponentConfig& config) {

  // 创建 Node
  node_.reset(new Node(config.name()));
  LoadConfigFiles(config);

  if (config.readers_size() < 2) {
    AERROR << "Invalid config file: too few readers.";
    return false;
  }

  // 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();

  // 创建 reader1
  auto reader1 = node_->template CreateReader<M1>(reader_cfg);

  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;

  // 创建 reader0
  if (cyber_likely(is_reality_mode)) {
    reader0 = node_->template CreateReader<M0>(reader_cfg);
  } else {
    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;
  }
  readers_.push_back(std::move(reader0));
  readers_.push_back(std::move(reader1));

  if (cyber_unlikely(!is_reality_mode)) {
    return true;
  }

  auto sched = scheduler::Instance();

  // 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());
  }

  // 创建数据访问器
  auto dv = std::make_shared<data::DataVisitor<M0, M1>>(config_list);

  // 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());
}
```

创建协程

- 创建协程对应上诉代码中的

```
croutine::RoutineFactory factory = croutine::CreateRoutineFactory<M0, M1>(func, dv);
```

- 协程通过工厂模式创建 里面包含一个回调函数和一个`dv`(数据访问器)
- 该方法在`cyber/croutine/routine_factory.h`中
- 工厂中设置`DataVisitor`
- 工厂中创建设置协程执行函数 回调包括三个步骤 -> 从 `DataVisitor`中获取数据 -> 执行回调函数 -> 继续休眠

```
template <typename M0, typename M1, typename F>
RoutineFactory CreateRoutineFactory(
    F&& f, const std::shared_ptr<data::DataVisitor<M0, M1>>& dv) {
  RoutineFactory factory;

  // 在工厂中设置 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);
```

```
        // 从 DataVisitor 中获取数据
        if (dv->TryFetch(msg0, msg1)) {
          // 执行回调函数
          f(msg0, msg1);
          // 继续休眠
          CRoutine::Yield(RoutineState::READY);
        } else {
          CRoutine::Yield();
        }
      }
    };
  };
  return factory;
}
```

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());
```

- 如何在`Sheduler`中创建任务

```
bool Scheduler::CreateTask(std::function<void()>&& func,
                           const std::string& name,
                           std::shared_ptr<DataVisitorBase> visitor) {
  if (cyber_unlikely(stop_.load())) {
    ADEBUG << "scheduler is stoped, cannot create task!";
    return false;
  }

  // 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;

  // 分发协程任务
  if (!DispatchTask(cr)) {
    return false;
  }

  // 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 对象

- 实际上`Component`分为两类 -> 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`
- 执行流程大致为 -> 创建`Node` -> 调用用户自定义初始化函数 -> 创建定时器 定时调用`Proc()`函数

```
bool TimerComponent::Initialize(const TimerComponentConfig& config) {
  if (!config.has_name() || !config.has_interval()) {
    AERROR << "Missing required field in config file.";
    return false;
  }

  // 创建 node
  node_.reset(new Node(config.name()));
  LoadConfigFiles(config);

  // 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());

  // 创建定时器 定时调用 Proc() 函数
  auto func = [self]() { self->Process(); };
  timer_.reset(new Timer(config.interval(), func, false));
  timer_->Start();
  return true;
}
```
