cyber/data

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 协程进行绑定的回调函数) 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

部分函数功能解释

data_visitor_base

- `DataVisitor`继承`DataVisitorBase`类
- `DataVisitor`创建了一个`Notifier`类 并提供注册回调的接口 同时还引用了`DataNotifier::Instance()`单例

```
class DataVisitorBase {
 public:
  // 初始化的时候创建一个 Notifier
  DataVisitorBase() : notifier_(new Notifier()) {}

  // 设置注册回调
  void RegisterNotifyCallback(std::function<void()>&& callback) {
    notifier_->callback = callback;
  }

 protected:
  DataVisitorBase(const DataVisitorBase&) = delete;
  DataVisitorBase& operator=(const DataVisitorBase&) = delete;

  // 下一次消息的下标
  uint64_t next_msg_index_ = 0;
  // DataNotifier 单例
  DataNotifier* data_notifier_ = DataNotifier::Instance();
  std::shared_ptr<Notifier> notifier_;
};
```

data_visitor

- `DataDispatcher`中添加订阅的`channelBuffer`
- 在`DataNotifier`增加对应通道的`Notifier`
- 通过`DataVisitor`获取数据并进行融合

```
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)) {

    // 在 DataDispatcher 中增加 ChannelBuffer
    DataDispatcher<M0>::Instance()->AddBuffer(buffer_m0_);
    DataDispatcher<M1>::Instance()->AddBuffer(buffer_m1_);
    DataDispatcher<M2>::Instance()->AddBuffer(buffer_m2_);
    // 在 DataNotifier::Instance() 中增加创建好的 Notifier
    data_notifier_->AddNotifier(buffer_m0_.channel_id(), notifier_);
    // 对接受到的消息进行数据融合
    data_fusion_ =
        new fusion::AllLatest<M0, M1, M2>(buffer_m0_, buffer_m1_, buffer_m2_);
  }

  ~DataVisitor() {
    if (data_fusion_) {
      delete data_fusion_;
      data_fusion_ = nullptr;
    }
  }

  bool TryFetch(std::shared_ptr<M0>& m0, std::shared_ptr<M1>& m1,  // NOLINT
                std::shared_ptr<M2>& m2) {                         // NOLINT
    // 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_;
};
```

- 如果`DataVisitor`只访问一个消息 则不会对消息进行融合

- 如果`DataVisitor`访问两个以上的数据 那么需要进行融合 并且注册融合回调 之后`CacheBuffer`会调用融合回调进行数据处理 而不会把数据放入`CacheBuffer`中

```
// 当有两个消息的时候 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;
}


// 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;
}
```

- 如果有多个消息的时候 会以第一个消息为基准 然后把其他消息的最新消息一起放入融合的`buffer_fusion_`(我在`data/fusion/all_latest.h`这个文件看见了不同的`AllLatest` 而且处理的也有多个消息的)

```
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))) {
  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);
      });
}
```

---

data_fusion

- `DataFusion`类是一个虚类 定义了数据融合的接口`Fusion()` `Apollo`里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 取其他最新消息 当然也可以在这里实现其他的数据融合方式

data_dispatcher

- 添加`ChannelBuffer`到`buffers_map_` `key`为通道`id`(topic) `value`为订阅通道消息的`CacheBuffer`数组
- 分发通道中的消息 根据`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`有三个`CacheBuffer`订阅 那么每次都会往这三个`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

```
template <typename T>
class DataDispatcher {
 public:
  using BufferVector =
      std::vector<std::weak_ptr<CacheBuffer<std::shared_ptr<T>>>>;
  ~DataDispatcher() {}
  // 添加 ChannelBuffer 到 buffers_map_
  void AddBuffer(const ChannelBuffer<T>& channel_buffer);

  // 分发通道中的消息
  bool Dispatch(const uint64_t channel_id, const std::shared_ptr<T>& msg);

 private:
  // DataNotifier 单例
  DataNotifier* notifier_ = DataNotifier::Instance();
  std::mutex buffers_map_mutex_;
  // 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_;
  // 单例
  DECLARE_SINGLETON(DataDispatcher)
};
```

---

data_notifier

- `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`

```
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_;

  // 哈希表 key 为通道 id value 为 Notify 数组
  AtomicHashMap<uint64_t, NotifyVector> notifies_map_;

  DECLARE_SINGLETON(DataNotifier)
};
```

---

cache_buffer

- `CachrBuffer`实现了一个缓存队列

```
  void Fill(const T& value) {
    if (fusion_callback_) {
      // 融合回调
      fusion_callback_(value);
    } else {
      // 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_;
      }
    }
  }
```

---

channel_buffer

- `ChanelBuffer`是`CacheBuffer`的封装

```
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();
  // 判断最新的加一 why?
  } 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;
}
```

---

data 目录总结

- 数据的访问都是通过`DataVisitor`来实现的 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
- 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