

## ./01-base/01-atomic\_hash\_map.md

---

### 代码总览

atomic\_hash\_map.h和atomic\_hash\_map\_test.cc

### 功能/知识

1.

- 
- 

2.

- 
- 

## ./01-base/学习apollo之cyber-base1.md

---

### cyber

---

#### —: base

#### 10. atomic\_hash\_map.h

##### a) BUILD

```
cc_library(  
    name = "atomic_hash_map",  
    hdrs = ["atomic_hash_map.h"],  
)
```

##### b) 模板类AtomicHashMap

- 它主要是维护了一个链表吧？
- 它的模板花样好多，用std::enable\_if强制一个模板类型的demo如下：

```
#include <atomic>  
#include <cstdint>  
#include <type_traits>  
#include <utility>  
template <typename K, typename
```

```
std::enable_if<std::is_integral<K>::value,int>::type = 0>
class A_CLASS {
    public:
        int a_int;
};
int main(int argc, const char* argv[]) {
    A_CLASS<int> a_class; //可以通过编译
    // A_CLASS<double> a_class; //无法通过编译
    return 0;
}
```

- 另，这个编译可以通过，但我还是不知道它的意义何在：

```
#include <atomic>
#include <cstdint>
#include <type_traits>
#include <utility>
template <typename K,int = 0>
class A_CLASS {
    public:
        int a_int;
};
int main(int argc, const char* argv[]) {
    A_CLASS<int> a_class;
    return 0;
}
```

好像知道了。

- 另：

```
#include <atomic>
#include <cstdint>
#include <type_traits>
#include <utility>
// template <typename K,typename std::enable_if<128,int>::type = 0>
//A_CLASS<int> a_class; 可以通过编译
// template <typename K,int I,typename std::enable_if<128,int>::type =
0> //A_CLASS<int,0> a_class; 可以通过编译
// template <typename K,int I=8,typename std::enable_if<128,int>::type
= 0> //A_CLASS<int> a_class;A_CLASS<int,0> a_class;均可以通过编译
template <typename K,int I=8,typename std::enable_if<I,int>::type = 0>
//A_CLASS<int> a_class;A_CLASS<int,1> a_class;可以通过编译
译.A_CLASS<int,0> a_class;无法通过编译
class A_CLASS {
    public:
        int a_int;
};
int main(int argc, const char* argv[]) {
```

```

A_CLASS<int> a_class;
// A_CLASS<int,0> a_class;
// A_CLASS<int,1> a_class;
return 0;
}

```

### c) 构造函数 `AtomicHashMap::AtomicHashMap`

常规构造。

**d) = delete** `AtomicHashMap(const AtomicHashMap &other) = delete;`  
`&operator=(const AtomicHashMap &other) = delete;`

禁止拷贝初始化。

### e) 函数 `Has`

和 `mode_num_` 按位与之后判断 `key` 是否存在。

### f) 函数 `Get`

和 `mode_num_` 按位与之后找 `key`。

### g) 函数 `Get`

f) 的重载。

它这个可以识别常量和变量的输入，不知道是什么机制，如：

```

apollo::cyber::base::AtomicHashMap<int, int> map;
int j=5;
map.Set(i*256, j); // 和下边调用的函数不一样
map.Set(i*256, 5); // 和上边调用的函数不一样

```

操纵的对象是一个 `Hash` 表 (`table_`)，首先索引到当前 `index`：`index = key & mode_num_`；这个没什么好说的，一行代码的事，然后操纵 `table_[index].Set` 是插入操作。对于 `table_[index]`，它是有序的，按序查找是否有 `key`，如果有，逻辑我还不太懂；如果没有，在当前的 `key`，和下一个 `key` 之间插入，以保证有序性，另如果当前是最后一个，把下一个置成 `nullptr`，这样也可以保证有序性。

总之，它应该是一个标准的 `hash` 表，加上了一些线程保护的机制。

### h) 函数 `Set`

获取 `key`。

### i) 函数 `Set`

h)的重载。

#### j) 函数Set

h)的重载。

e)~h)的一些小知识：

- \*\* 指针的指针
- && 右值引用
- `std::forward<V>` 完美转发, <https://blog.csdn.net/coolwriter/article/details/80970718>

#### k) 结构体Entry

一个链表的子单元吧, 原子保护多线程。

#### l) 构造函数Entry::Entry

常规。

#### m) 构造函数Entry::Entry

l)的重载, 用到了`std::atomic`, 关于多线程资源竞争还是不是很理解。有机会再看看吧。

#### n) 构造函数Entry::Entry

l)的重载。为什么原子量要用`store`赋值呢? 直接等于不好吗?

#### o) 析构函数Entry::~~Entry

p) 成员`key`, 成员`std::atomic<V *> value_ptr`, `std::atomic<Entry *> next`

#### q) 结构体Bucket

#### r) 构造函数Bucket::Bucket

初始化`head_`, `head_`是`Entry`的实例。

#### s) 析构函数Bucket::~~Bucket

逐个子单元删除`head_`。

#### t) 类函数Bucket::Has

查`head_`中是否含有`key`, `head_`的自单元似乎是以`key`从小到达排列的。

#### u) 类函数Bucket::Find

- 查`head_`中是否含有`key`的`Entry`的实例, 为什么有两个返回? : `prev_ptr`和`target_ptr`。

- `key`似乎是有向的。`target_ptr`满足`target_ptr->key == key`,`prev_ptr`满足`prev_ptr->next->key == key`, 即`prev_ptr->next==target_ptr`。

#### v) 类函数 `Bucket::Insert`

`Insert`字面上是插入, 但看里边的逻辑好复杂。

#### w) 类函数 `Bucket::Insert`

v)的重载。

#### x) 类函数 `Bucket::Get`

`Get`函数为什么会返回布尔值?

#### y) 成员变量 `table_`, `capacity_`, `mode_num_`

```
Bucket table_[TableSize];
uint64_t capacity_;
uint64_t mode_num_;
```

`TableSize`在模板中声明, 默认是128, `capacity_`的值是`TableSize`, `mode_num_`的值是`TableSize-1`,

### 11. `atomic_hash_map_test.cc`

#### a) `BUILD`的控制:

```
cc_test(
  name = "atomic_hash_map_test",
  size = "small",
  srcs = ["atomic_hash_map_test.cc"],
  deps = [
    "//cyber/base:atomic_hash_map",
    "@com_google_googletest//:gtest_main",
  ],
)
```

test暂不关注。

### 12. `atomic_rw_lock.h`

原子读写锁。似乎是用原子模拟一个锁,但c++11自带的锁不能满足读写的某种特性,所以它只能自己写一个。

#### a) `BUILD`的控制:

```
cc_library(
    name = "atomic_rw_lock",
    hdrs = ["atomic_rw_lock.h"],
    deps = [
        "//cyber/base:rw_lock_guard",
    ],
)
```

## b) 类AtomicRWLock

- 友元类，我信任你，所以你可以访问我的私处。主要用到了关键字**friend**。一个demo：

```
class CCar
{
private:
    int price;
    friend class CDriver; //声明 CDriver 为友元类
};
class CDriver
{
public:
    CCar myCar;
    void ModifyCar() //改装汽车
    {
        myCar.price += 1000; //因CDriver是CCar的友元类，故此处可以访问其私有成员
    }
};
int main()
{
    return 0;
}
```

另:大括号可以用于初始化.

```
int a(5);
std::cout<<a<<std::endl;    // 5
int b{6};
std::cout<<b<<std::endl;    // 6
```

另:**std::this\_thread::get\_id()**指的是获取当前线程

## c) 两个友元类声明：

```
friend class ReadLockGuard<AtomicRWLock>;
friend class WriteLockGuard<AtomicRWLock>;
```

还有带模板的写法哈，

#### d) 3个常量

```
static const int32_t RW_LOCK_FREE = 0;
static const int32_t WRITE_EXCLUSIVE = -1;
static const uint32_t MAX_RETRY_TIMES = 5;
```

可以按字面意思理解它们。

#### e) 构造函数 `AtomicRWLock::AtomicRWLock`

关键字 `explicit`，构造函数中进制隐式类型转换。

#### f) 类函数 `AtomicRWLock::ReadLock`

- `std::this_thread::yield();` 说是 save cpu，但逻辑似乎还是不是很。。。
- 

#### g) 类函数 `AtomicRWLock::ReadUnlock`

- 

#### h) 类函数 `AtomicRWLock::WriteUnlock`

- 

## [./01-base/学习apollo之cyber-base2.md](#)

---

### 13. `bounded_queue.h`

#### a) `BUILD` 的控制：

```
cc_library(
  name = "bounded_queue",
  hdrs = ["bounded_queue.h"],
  deps = [
    "//cyber/base:macros",
    "//cyber/base:wait_strategy",
  ],
)
```

#### b) 类 `BoundedQueue`

- 你是干啥的?
- 总之,这个头文件还是不很理解.

#### c) 类成员 `BoundedQueue::head_/tail_/commit_`.

- `alignas` 是一个关键字,内存对齐,许多教程说的是跨平台,但我还是不是很理解.但是对于理解代码应该没有干扰.

#### d) 类成员 `BoundedQueue::pool_size_`

- 

#### e) 类成员 `BoundedQueue::pool_`

- 

#### f) 类成员 `BoundedQueue::wait_strategy_`

- 初始化 `pool_[pool_size_]`
- 智能指针,知道在哪里销毁它:

```
#include <iostream>
#include <memory>
struct Task {
    int mId;
    Task(int id ) :mId(id) {
        std::cout << "Task::Constructor" << std::endl;
    }
    ~Task() {
        std::cout << "Task::Destructor" << std::endl;
    }
};
int main()
{
    Task* taskPtr(new Task(23));
    int id = taskPtr->mId;
    std::cout << id << std::endl;
    return 0;
}
```

会输出:

```
Task::Constructor
23
```

```
#include <iostream>
#include <memory>
```



```
struct Task {
    int mId;
    Task(int id ) :mId(id) {
        std::cout << "Task::Constructor" << std::endl;
    }
    ~Task() {
        std::cout << "Task::Destructor" << std::endl;
    }
};
int main()
{
    Task* taskPtr(new Task(23));
    int id = taskPtr->mId;
    std::cout << id << std::endl;
    delete taskPtr;
    return 0;
}
```

会输出:

```
Task::Constructor
23
Task::Destructor
```

```
#include <iostream>
#include <memory>
struct Task {
    int mId;
    Task(int id ) :mId(id) {
        std::cout << "Task::Constructor" << std::endl;
    }
    ~Task() {
        std::cout << "Task::Destructor" << std::endl;
    }
};
int main()
{
    std::unique_ptr<Task> taskPtr(new Task(23));
    int id = taskPtr->mId;
    std::cout << id << std::endl;
    return 0;
}
```

会输出:

```
Task::Constructor
23
```

Task::Destructor

- `reset`,智能指针的赋值.

#### g) 类成员 `BoundedQueue::break_all_wait_`

- `volatile`是一个关键字,防止编译器优化导致的多线程处理变量出错.

#### h) 构造函数 `BoundedQueue::BoundedQueue()`

- 

#### d) 两个 `delete`

取消赋值和拷贝操作.

#### e) 析构函数 `BoundedQueue<T>::~~BoundedQueue()`

释放一些东西.

`reinterpret_cast`指的是类型转换,

用法: `TYPE b = reinterpret_cast ( a )`

TYPE必须是一个指针、引用、算术类型、函数指针.

详见: [https://blog.csdn.net/m0\\_45867846/article/details/107082464](https://blog.csdn.net/m0_45867846/article/details/107082464)

#### f) 类函数 `BoundedQueue<T>::Init()`

调用重载函数 `BoundedQueue<T>::Init()`

#### g) 类函数 `BoundedQueue<T>::Init()`

- f)的重载函数.
- `std::calloc`,分配内存,通常和`std::free`搭配使用.
- `new`的几个用法:
  - `int *p = new int(3);`
  - `int *q = new int[3];`
  - `new(p) A(3);`指定内存地址new,又称placement new,但必须显式包含头文件`new`或者`new.h`,如:

```
#include<stdio.h>
#include<new> //必须显式包含,否则编译不通过
class A
{
    int i;
public:
    A(int _i) :i(_i*_i) {}
```

```

    void Say() { printf("i=%d\n", i); }
};
int main()
{
    char s[sizeof(A)];
    A* p = (A*)s;
    new(p) A(3);
    p->Say(); // i=9
}

```

#### h) 类函数 `BoundedQueue<T>::Enqueue()`

- 字面意思是入队.
- `compare_exchange_weak`的底层逻辑不懂的话可以简单理解成`tail_=new_tail`,同时返回`true`.

```

tail_.compare_exchange_weak(old_tail, new_tail,
                             std::memory_order_acq_rel,
                             std::memory_order_relaxed)

```

#### i) 类函数 `BoundedQueue<T>::Enqueue()`

- h)的重载.
- `cyber_unlikely` 是在`macros.h`的宏定义,它是`#define cyber_likely(x) (__builtin_expect((x), 1))` 总之是为了编译时生成更高效的代码,虽然我也不知道为什么.参考:  
[https://blog.csdn.net/qq\\_22660775/article/details/89028258](https://blog.csdn.net/qq_22660775/article/details/89028258)
- `NotifyOne()` `wait_strategy.h`中定义.

#### j) 类函数 `BoundedQueue<T>::Dequeue()`

- 字面意思是出队.

#### k) 类函数 `BoundedQueue<T>::WaitEnqueue()`

- 等待入队,如果入队,返回;如果没有入队,等待.

#### k2) 类函数 `BoundedQueue<T>::WaitEnqueue()`

- k)的重载.

#### l) 类函数 `BoundedQueue<T>::Size()`

- 队列的长度?

#### m) 类函数 `BoundedQueue<T>::Empty()`

- 队列是否为空.

**s) 类函数**`BoundedQueue<T>::GetIndex()`

- 取余数,他还标注了比%要快,会吗?但我写了两个程序测试了下,感觉差不多哦.
- 

**n) 类函数**`BoundedQueue<T>::SetWaitStrategy()`

- 设置`wait_strategy_`.

**o) 类函数**`BoundedQueue<T>::BreakAllWait()`

- 中断所有的`WaitEnqueue`

**p) 类函数**`BoundedQueue<T>::Head()`

- 

**q) 类函数**`BoundedQueue<T>::Tail()`

- 

**r) 类函数**`BoundedQueue<T>::Commit()`

- 

## [./01-base/学习apollo之cyber-base3.md](#)

---

**14.** `bounded_queue_test.cc`

`xx_test.cc`先不关注.

**15.** `concurrent_object_pool.h`**a) BUILD的控制：**

```
cc_library(
  name = "concurrent_object_pool",
  hdrs = ["concurrent_object_pool.h"],
  deps = [
    "//cyber/base:for_each",
  ],
)
```

它是一个常规链接库.

**b) 类**`CCObjectPool`:

- `std::enable_shared_from_this`, c11新特性,两个智能指针指向同一个实体, <https://blog.csdn.net/caoshangpa/article/details/79392878> 写的蛮好的.
- 这个函数也不是很懂,先往下看吧.

### c) 两个using

- 重命名,常规操作.
- 

### d) 类构造函数 `ObjectPool::ObjectPool()`

- `explicit`用在类构造函数之中,取消参数类型隐式变换,用于规范化代码.
- `template <typename... Args>`变参模板,如:

```
#include <iostream>
#include <string>
template<typename T>
T adder(T v) { //一定要有一个基本单元.也可能不一定.
    return v;
}
template<typename T, typename... Args>
T adder(T first, Args... args) {
    return first + adder(args...);
}
int main()
{
    std::cout<<adder(1,2,3,4)<<std::endl; //10
    std::cout<<adder('a','b','c','d')<<std::endl; //乱码

    std::cout<<adder(std::string("a"),std::string("b"),std::string("c"),std::string("d"))<<std::endl; //abcd
}
```

### e) 类构造函数 `ObjectPool::~ObjectPool()`

- d)的重载.
- C11的右值引用. [https://blog.csdn.net/with\\_dream/article/details/85137039](https://blog.csdn.net/with_dream/article/details/85137039) 但感觉还是云里雾里的.
- `FOR_EACH`里边的东西还是云里雾里的.

### f) 类析构函数 `ObjectPool::~~ObjectPool()`

- 虚函数.
- `std::free()`

### g) 类函数 `ObjectPool::GetObject()`

虚函数.

**h) 类函数**`ObjectPool::ReleaseObject()`

虚函数.

**i) 类函数**`ObjectPool::GetObject()`

- `this->shared_from_this()`?指向this的当前智能指针.
- 哈?

**16. object\_pool\_test.cc**

暂不关注.

**17. reentrant\_rw\_lock.h**

- reentrant 可再入的.
- 它的结构和`atomic_rw_lock`好像啊.

**a) BUILD的控制：**

```
cc_library(
    name = "reentrant_rw_lock",
    hdrs = ["reentrant_rw_lock.h"],
)
```

**b) 友元类声明：**

```
friend class ReadLockGuard<ReentrantRWLock>;
friend class WriteLockGuard<ReentrantRWLock>;
```

**c) static const量**

```
static const int32_t RW_LOCK_FREE = 0;
static const int32_t WRITE_EXCLUSIVE = -1;
static const uint32_t MAX_RETRY_TIMES = 5;
static const std::thread::id null_thread;
```

**d) 类构造函数**`ReentrantRWLock::ReentrantRWLock()`**e) 类构造函数**`ReentrantRWLock::ReentrantRWLock()`

d)的重载

**f) 类函数**`ReentrantRWLock::ReadLock()`

d)的重载

## 18. `rw_lock_guard.h`

a) **BUILD**的控制：

```
cc_library(  
    name = "rw_lock_guard",  
    hdrs = ["rw_lock_guard.h"],  
)
```

常规链接库.

b) 类**ReadLockGuard**

- 它是构造时加锁,析构时解锁.一行代码实现两行的功能;
- 取消拷贝初始化.

c) 类**WriteLockGuard**

- 和b)极其类似.

## [./01-base/学习apollo之cyber-base4.md](#)

---

## 19. `signal.h`

`signal.h`之中维护了三个类:**Slot/Connection/Signal**,这三个类耦合十分紧密.

a) **BUILD**的控制：

```
cc_library(  
    name = "signal",  
    hdrs = ["signal.h"],  
)
```

常规链接库.

b) 类**Signal**

- 四个**using**,常规重命名.
- `std::list`是c++标准容器.
- 把类**Slot/Connection**声明在了**Signal**前面,为了能在**Signal**中使用**Slot/Connection**.

b1) 类成员**Signal::slots\_**

slot智能指针的std::list.

## b2) 类成员Signal::mutex\_

线程锁.

## c) 类构造函数Signal::Signal()

常规.

## d) 类析构函数Signal::~Signal()

调用DisconnectAllSlots()线程安全地清空所有slots\_(slots\_是slot智能指针的std::list(C++标准容器)),slot是Slot的实例,slot维护了一个std::function和一个connected(布尔值).

## e) 类运算符Signal::operator()

把slots\_的slot压如local,如果local不为空,执行每个slot中的function,线程安全地清除slots中所有connected==false的slot(ClearDisconnectedSlots()).因为要这样动态地添加删除slots,所以用std::list而不是std::vector?

## f) 类函数Signal::Connect()

线程安全地输入一个std::function,用std::function初始化一个slot,slot维护了一个std::function和一个connected(布尔值).slot压入slots\_,返回一个Connection,Connection维护了一个Slot和Signal指针.

## g) 类函数Signal::Disconnect()

输入一个connection,线程安全地:如果this(Signal)的slots\_有connection中的slot,该slot的connected置为false.清除slots\_中所有connected置为false的slot.

## h) 类函数Signal::DisconnectAllSlots()

线程安全地把slots\_中的所有slot的connected置为false,清空slot\_.

## i) delete

```
Signal(const Signal&) = delete;  
Signal& operator=(const Signal&) = delete;
```

取消拷贝赋值.

## j) 类函数Signal::ClearDisconnectedSlots()

线程安全地清除slots\_中所有connected==false的slot,

## k) 类Connection



- 维护了一个`Slot`智能指针和一个`Signal`智能指针.
- 两个`using`,常规重命名

**k1) 类成员**`Connection::slot_`

**k2) 类成员**`Connection::signal_`

**l) 类构造函数**`Connection::Connection()`

常规.

**m) 类构造函数**`Connection::Connection()`

l)的重载.

**n) 类析构函数**`Connection::~~Connection()`

常规.

**o) 类运算符**`Connection::=`

判断相等.

**p) 类函数**`Connection::HasSlot()`

输入一个`slot`,判断我自有的`slot_`和输入的`slot`是否相等.

**q) 类函数**`Connection::IsConnected()`

**r) 类函数**`Connection::Disconnect()`

**s) 类**`Slot`

- 维护了一个`std::function`和一个`connected`(布尔值),可以看做一个加强版的`std::function`.
- 一个`using`,常规重命名

**s1) 类成员**`Slot::cb_`

一个`std::function`

**s2) 类成员**`Slot::connected_`

一个布尔值

**t) 类构造函数**`Slot::Slot()`

常规.

**u) 类构造函数**`Slot::Slot()`

t)的重载.

#### v) 类析构函数 `Slot::~~Slot()`

常规.

#### w) 类析构函数 `Slot::~~Disconnect()`

常规.

#### x) 类析构函数 `Slot::~~connected()`

常规.

### 20. signal\_test.cc`

它是一个google test,阅读test十分有助于理解被测试代码.

#### a) BUILD的控制：

```
cc_test(
  name = "signal_test",
  size = "small",
  srcs = ["signal_test.cc"],
  deps = [
    "//cyber/base:signal",
    "@com_google_googletest//:gtest_main",
  ],
)
```

#### b) 由test分析signal.h

- `Signal`看起来和`std::function`极其类似,
- 和`std::function`不同的是`Signal`可以有多个`std::function`,如`sig(lhs, rhs)`;可以执行当前所有的`std::function`

## ./01-base/学习apollo之cyber-base5.md

---

### 19. thread\_pool.h

线程池?

#### a) BUILD的控制：

```
cc_library(
  name = "thread_pool",
```

```
    hdrs = ["thread_pool.h"],
)
```

常规链接库.

## b) 类ThreadPool

该头文件只有一个类.

### b1) 类成员ThreadPool::workers\_

线程vector。

### b2) 类构造函数ThreadPool::task\_queue\_`

涉及到了BoundedQueue，但BoundedQueue还是不理解。

### b3) 类构造函数ThreadPool::stop\_

原子。

## c) 类构造函数ThreadPool::ThreadPool()

- `std::vector`的`reserve().resize()`：改变的是`size()`与`capacity()`的大小,所以用完`resize()`之后一般用`at().reserve()`：改变的只是`capacity()`的大小，`size()`和内容不会改变,所以用完`reserve()`之后一般用`push_back()`或者`emplace_back()`.
- lambda表达式,的捕获`this`的规则. 所以`this`到底要不要显式地指明?我看`thread_pool.h`并没有显式地指明.

```
#include<iostream>
using namespace std;

struct Foo1 {
    int x = 1;

    void f(int x) {
        auto f = [x, this] { cout << x << endl; };
        f();
    }
};

struct Foo2 {
    int x = 1;

    void f(int x) {
        auto f = [this] { cout << this->x << endl; };
        // auto f = [this] { cout << x << endl; }; //编译报错
        f();
    }
};
```

```

struct Foo3 {
    int x = 1;

    void f(int x) {
        auto f = [x] { cout << x << endl; };
        f();
    }
};

int main(){
    Foo1 foo1;
    foo1.f(2); //2
    Foo2 foo2;
    foo2.f(2); //1
    Foo3 foo3;
    foo3.f(2); //2
}

```

- `std::thread+std::vector+push_back()+emplace_back()`的一些神奇现象:,,,,感觉 `emplace_back()`比`push_back()`更支持隐式变换.

```

#include <iostream>
#include <utility>
#include <thread>
#include <chrono>
#include <functional>
#include <atomic>
#include <vector>
void f1()
{
    std::cout << "I'm f1."<<std::endl;
}
int main()
{
    std::thread a_t(f1);
    a_t.join();
    std::vector<std::thread> t_s;
    // t_s.push_back(f1); //编译不通过
    t_s.push_back(std::thread(f1)); //编译通过
    t_s.at(0).join();
    // t_s.push_back([]{std::cout<<"666"<<std::endl;}); //编译不通过
    t_s.push_back(std::thread([]{std::cout<<"I'm lambda."<<std::endl;}));
    //编译通过
    t_s.at(1).join();
    t_s.emplace_back(f1); //编译通过
    t_s.at(2).join();
    t_s.emplace_back(std::thread(f1)); //编译通过
    t_s.at(3).join();
    t_s.emplace_back(std::thread([]{std::cout<<"I'm lambda."
<<std::endl;})); //编译通过
    t_s.at(4).join();
}

```

```

    t_s.emplace_back([]{std::cout<<"I'm lambda."<<std::endl;}); //编译通过
    t_s.at(5).join();
}

```

#### d) 类函数 `ThreadPool::Enqueue()`

- 这个函数还是不懂
- 尾置类型
- `std::result_of` 推断出函数的返回值类型, cpp reference 里的例子挺好的:

```

// result_of example
#include <iostream>
#include <type_traits>
int fn(int) {return int();} // function
typedef int(&fn_ref)(int); // function
reference
typedef int(*fn_ptr)(int); // function pointer
struct fn_class { int operator()(int i){return i;} }; // function-like
class
int main() {
    typedef std::result_of<decltype(fn)&(int)>::type A; // int
    typedef std::result_of<fn_ref(int)>::type B; // int
    typedef std::result_of<fn_ptr(int)>::type C; // int
    typedef std::result_of<fn_class(int)>::type D; // int
    std::cout << std::boolalpha;
    std::cout << "typedefs of int:" << std::endl;
    std::cout << "A: " << std::is_same<int,A>::value << std::endl; // true
    std::cout << "B: " << std::is_same<int,B>::value << std::endl; // true
    std::cout << "C: " << std::is_same<int,C>::value << std::endl; // true
    std::cout << "D: " << std::is_same<int,D>::value << std::endl; // true
    return 0;
}

```

- `std::make_shared` 智能指针的创建方法。cpp reference 的例子：

```

#include <iostream>
#include <memory>
int main () {
    std::shared_ptr<int> foo = std::make_shared<int> (10);
    // same as:
    std::shared_ptr<int> foo2 (new int(10));
    auto bar = std::make_shared<int> (20);
    auto baz = std::make_shared<std::pair<int,int>> (30,40);
    std::cout << "**foo: " << *foo << '\n';
    std::cout << "**bar: " << *bar << '\n';
    std::cout << "**baz: " << baz->first << ' ' << baz->second << '\n';
    return 0;
}

```

- `std::packaged_task`
  - `std::promise/std::future/std::packaged_task` 这篇文章写得很漂亮：  
<https://zhuanlan.zhihu.com/p/61311187> 我们按照它的实现以下：

```
// std::promise和std::future搭配使用，用于线程之间的数据传递
#include <chrono>
#include <future> // 引入future所在的头文件
#include <iostream>
#include <string>
#include <thread>
using namespace std;
class Food {
public:
    Food() {} // 默认构造函数
    // 通过菜名构建Food对象
    Food(string strName) : m_strName(strName) {}
    // 获取菜名
    string GetName() const { return m_strName; }
private:
    string m_strName; // 菜名
};
// 线程函数
// 根据菜名创建Food对象，并通过promise对象返回结果数据
void Cook(const string strName, promise<Food>& prom) {
    //为了突出效果，可以使线程休眠5s
    std::this_thread::sleep_for(std::chrono::seconds(5));
    // 做菜...
    Food food(strName);
    // 将创建完成的food对象放到promise传递出去
    prom.set_value(food);
}
int main() {
    // 用于存放结果数据的promise对象
    promise<Food> prom;
    // 获得promise所关联的future对象
    future<Food> fu = prom.get_future();
    // 创建分支线程执行Cook()函数
    // 同时将菜名和用于存放结果数据的promise对象传递给Cook()函数
    // ref()函数用于获取promise对象的引用
    thread t(Cook, "回锅肉", ref(prom));
    // 等待分支线程完成Food对象的创建，一旦完成，立即获取完成的Food对象
    //可以选择使用wait_for限制主线程等待时间
    Food food = fu.get(); //等待 prom.set_value(food);执行完成
    // 上菜
    cout << "客官，你点的" << food.GetName() << "来了，请慢用！" <<
endl;
    t.join(); // 等待分支线程最终完成
    return 0;
}
```

```
//为了减少码量, C++提供了packaged_task // std::promise和std::future搭配使用, 用于线程之间的数据传递
#include <future> // 引入future所在的头文件
#include <iostream>
#include <string>
using namespace std;
class Food { public: Food() {} // 默认构造函数 // 通过菜名构建Food对象 Food(string strName) : m_strName(strName) {} // 获取菜名 string GetName() const { return m_strName; } private: string m_strName; // 菜名 };
// 线程函数 // 根据菜名创建Food对象, 并通过promise对象返回结果数据 Food Cook(const string strName) { //为了突出效果, 可以使线程休眠5s std::this_thread::sleep_for(std::chrono::seconds(5)); // 做菜... Food food(strName); return food; }
int main() { // 使用线程函数的返回值和参数类型特化packaged_task类模板 // 利用其构造函数, 将线程函数打包成一个 packaged_task对象 packaged_task<Food(string)> cooker(Cook); //传入Cook函数: Food(string) // 从 packaged_task对象获得与之关联的future对象 future fu = cooker.get_future(); // 创建线程执行 packaged_task对象, 实际上执行的是Cook()函数 // 这里也不再需要传递promise对象 thread t(move(cooker), "回锅肉"); // 同样地获得结果数据 Food food = fu.get(); cout << "客官, 你点的" << food.GetName() << "来了, 请慢用!" << endl; t.join(); return 0; }
```

```
```c++
// 为了进一步减少码量, C++提供了`async`
// std::promise和std::future搭配使用, 用于线程之间的数据传递
#include <chrono>
#include <future> // 引入future所在的头文件
#include <iostream>
#include <string>
#include <thread>
using namespace std;
class Food {
public:
    Food() {} // 默认构造函数
    // 通过菜名构建Food对象
    Food(string strName) : m_strName(strName) {}
    // 获取菜名
    string GetName() const { return m_strName; }
private:
    string m_strName; // 菜名
};
// 线程函数
// 根据菜名创建Food对象, 并通过promise对象返回结果数据
Food Cook(const string strName) {
    //为了突出效果, 可以使线程休眠5s
    std::this_thread::sleep_for(std::chrono::seconds(5));
    // 做菜...
    Food food(strName);
    return food;
}
int main() {
    // 将Cook()函数异步 (async) 执行
    future<Food> fu = async(bind(Cook, "回锅肉"));
    cout<<"客官, 你点的"<<fu.get().GetName()<<"来了, 请慢用!"<<endl;
    return 0;
}
```

```
}
...
```

`std::promise`/`std::future`/`std::packaged\_task` 都是线程之间数据传递用的。

- `*auto* task = std::make_shared<std::packaged_task<return_type()>>(std::bind(std::forward<F>(*f*), std::forward<Args>(*args*)...));`这句话还是不甚理解。

### c) 类析构函数 `ThreadPool::~ThreadPool()`

- 原子的 `exchange`，赋值吗
- 为什么析构函数里边有线程的 `join()`?

## ./01-base/学习apollo之cyber-base6.md

---

### 20. thread\_safe\_queue.h`

线程池?

#### a) **BUILD**的控制：

```
cc_library(
    name = "thread_safe_queue",
    hdrs = ["thread_safe_queue.h"],
)
```

它是常规链接库。

#### b) 类 `ThreadSafeQueue`

该头文件只有一个类。

##### b1) 类成员 `ThreadSafeQueue::break_all_wait_`

- 是不是多线程临界资源都加上 `volatile` 比较好?

##### b2) 类成员 `ThreadSafeQueue::mutex_`

- 线程锁

##### b3) 类成员 `ThreadSafeQueue::queue_`

- 队列

##### b4) 类成员 `ThreadSafeQueue::cv_`



- `std::condition_variable`

### c) delete

取消拷贝赋值。

### d) ThreadSafeQueue::Enqueue()

- 线程安全地：向queue\_对尾加入一个元素
- 随机唤醒一个线程？(cv\_.notify\_one())
- `std::condition_variable`:线程同步用的
  - `std::condition_variable`的`notify_all()`和`wait()`

```
// 用std::condition_variable唤醒所有线程的一个例子，参考
https://murphypei.github.io/blog/2019/04/cpp-concurrent-3.html
#include <condition_variable> // std::condition_variable
#include <iostream>           // std::cout
#include <mutex>               // std::mutex, std::unique_lock
#include <string>              // std::string
#include <thread>              // std::thread
std::mutex mtx;               // 全局互斥锁.
std::condition_variable cv;   // 全局条件变量.
void do_print_id(int id) {
    std::unique_lock<std::mutex> lck(mtx);
    cv.wait(lck); // 当前线程被阻塞.
    // 线程被唤醒，继续往下执行.
    std::cout << "thread " << id << '\n';
}
void go() {
    std::unique_lock<std::mutex> lck(mtx);
    cv.notify_all(); // 唤醒所有线程.
}
int main() {
    int thread_num = 100;
    std::thread threads[thread_num];
    // 开辟10个线程
    for (int i = 0; i < thread_num; ++i) threads[i] =
std::thread(do_print_id, i);
    //输入go, 会看到10个线程执行.
    std::cout << "please input go to wake up all threads" << std::endl;
    std::string input_str;
    while (1) {
        std::cin >> input_str;
        if (input_str != "go") {
            std::cout << "error input. please do again!" << std::endl;
        } else {
            break;
        }
    }
    go();
    for (auto& th : threads) th.join();
}
```

```
    return 0;
}
```

但是上例之中若程序执行过快可能会有问题（可能），如果注释掉cin：

```
// 该例之中并不是所有线程都会执行，这可能是线程执行过快所致？
#include <condition_variable> // std::condition_variable
#include <iostream>           // std::cout
#include <mutex>               // std::mutex, std::unique_lock
#include <string>              // std::string
#include <thread>              // std::thread
std::mutex mtx;               // 全局互斥锁。
std::condition_variable cv;   // 全局条件变量。
void do_print_id(int id) {
    std::unique_lock<std::mutex> lck(mtx);
    cv.wait(lck); // 当前线程被阻塞。
    // 线程被唤醒，继续往下执行。
    std::cout << "thread " << id << '\n';
}
void go() {
    std::unique_lock<std::mutex> lck(mtx);
    cv.notify_all(); // 唤醒所有线程。
}
int main() {
    int thread_num = 100;
    std::thread threads[thread_num];
    // 开辟10个线程
    for (int i = 0; i < thread_num; ++i) threads[i] =
std::thread(do_print_id, i);
    // //输入go, 会看到10个线程执行。
    // std::cout << "please input go to wake up all threads" << std::endl;
    // std::string input_str;
    // while (1) {
    //     std::cin >> input_str;
    //     if (input_str != "go") {
    //         std::cout << "error input. please do again!" << std::endl;
    //     } else {
    //         break;
    //     }
    // }
    go();
    for (auto& th : threads) th.join();
    return 0;
}
```

```
// 但加上一个全局标志位之后又不会又该问题了，好奇怪。
#include <condition_variable> // std::condition_variable
#include <iostream>           // std::cout
#include <mutex>               // std::mutex, std::unique_lock
```

```

#include <string>           // std::string
#include <thread>           // std::thread
std::mutex mtx;            // 全局互斥锁.
std::condition_variable cv; // 全局条件变量.
bool ready = false;       // 全局标志位
void do_print_id(int id) {
    std::unique_lock<std::mutex> lck(mtx);
    // 如果标志位不为true, 则等待
    while(!ready)
    {
        cv.wait(lck); // 当前线程被阻塞.
    }
    // 线程被唤醒, 继续往下执行.
    std::cout << "thread " << id << '\n';
}
void go() {
    std::unique_lock<std::mutex> lck(mtx);
    ready = true;
    cv.notify_all(); // 唤醒所有线程.
}
int main() {
    int thread_num = 100;
    std::thread threads[thread_num];
    // 开辟10个线程
    for (int i = 0; i < thread_num; ++i) threads[i] =
std::thread(do_print_id, i);
    //输入go, 会看到10个线程执行.
    // std::cout << "please input go to wake up all threads" << std::endl;
    // std::string input_str;
    // while (1) {
    //     std::cin >> input_str;
    //     if (input_str != "go") {
    //         std::cout << "error input. please do again!" << std::endl;
    //     } else {
    //         break;
    //     }
    // }
    // }
    go();
    for (auto& th : threads) th.join();
    return 0;
}

```

- `std::condition_variable`的`notify_one()`说的是随机唤起一个线程, 但是下例的测试显示随机唤起一个到多个线程。

```

// 测试 `notify_one()`
#include <condition_variable> // std::condition_variable
#include <iostream>           // std::cout
#include <mutex>               // std::mutex, std::unique_lock
#include <string>              // std::string
#include <thread>              // std::thread

```

```

std::mutex mtx;           // 全局互斥锁.
std::condition_variable cv; // 全局条件变量.
bool ready = false;      // 全局标志位
void do_print_id(int id) {
    std::unique_lock<std::mutex> lck(mtx);
    // 如果标志位不为true, 则等待
    while(!ready)
    {
        cv.wait(lck); // 当前线程被阻塞.
    }
    // 线程被唤醒, 继续往下执行.
    std::cout << "thread " << id << '\n';
}
void go() {
    std::unique_lock<std::mutex> lck(mtx);
    ready = true;
    cv.notify_one(); // 随机唤醒一个等待线程.
}
int main() {
    int thread_num = 100;
    std::thread threads[thread_num];
    // 开辟10个线程
    for (int i = 0; i < thread_num; ++i) threads[i] =
std::thread(do_print_id, i);
    //输入go, 会看到10个线程执行.
    // std::cout << "please input go to wake up all threads" << std::endl;
    // std::string input_str;
    // while (1) {
    //     std::cin >> input_str;
    //     if (input_str != "go") {
    //         std::cout << "error input. please do again!" << std::endl;
    //     } else {
    //         break;
    //     }
    // }
    go();
    for (auto& th : threads) th.join();
    return 0;
}

```

#### e) ThreadSafeQueue::Dequeue()

- 线程安全地从队列中取出第一个元素？并在队列中删除该元素？
- `std::move()`: 左值变为右值, 等同于`static_cast<T&&>(lvalue)`; 但为什么会出现在这里？

#### f) ThreadSafeQueue::WaitDequeue()

- 若`break_all_wait_==true`且`queue_`不为空, 执行`Dequeue()`, 否则线程等待。
- `wait()`有两种用法, 一种是: `void wait(unique_lock& lck)`; 上边已经介绍; 另一种是`template <class Predicate> void wait(unique_lock<mutex>& lck, Predicate pred)`; 类似

于：`while (!pred()) wait(lck);`，相当于加了条件判断语句。  
<https://blog.csdn.net/zzhongcy/article/details/85248597>

#### g) `ThreadSafeQueue::Empty()`

- 判断`queue_`是否为空。

#### g) `ThreadSafeQueue::BreakAllWait()`

- 唤醒所有线程。

## ./01-base/学习apollo之cyber-base7.md

---

### 21. `unbounded_queue.h`

无界限队列。

#### a) **BUILD**的控制：

```
cc_library(
  name = "unbounded_queue",
  hdrs = ["unbounded_queue.h"],
)
```

常规链接库。

#### b) 类`UnboundedQueue`

- 该头文件只有一个类。

#### b1) 类成员`UnboundedQueue::head_`

- 

#### b2) 类成员`UnboundedQueue::tail_`

- 

#### b3) 类成员`UnboundedQueue::tail_`

- 

#### c) 类构造函数`UnboundedQueue::UnboundedQueue()`

- 调用`Reset()`

#### d) `delete`

- 删除拷贝赋值。

#### e) 类析构函数 `UnboundedQueue::~UnboundedQueue()`

- 调用 `Destroy()`

#### f) 类函数 `UnboundedQueue::Clear()`

- 调用 `Destroy()` 和 `Reset()`

#### g) 类函数 `UnboundedQueue::Enqueue()`

- 
- 如果忽略多线程的话：

```
while (true) {
    if (tail_.compare_exchange_strong(old_tail, node)) {
        old_tail->next = node;
        old_tail->release();
        size_.fetch_add(1);
        break;
    }
}
可以写成：
tail_=node;
old_tail->next = node;
old_tail->release();
size_.fetch_add(1);
```

所以代码的含义是向队列末尾增加一个元素，也就是tail\_后边的一位。同时队列长度加1。

- 原子类型的 `fetch_add()`: 原子类型的自增？

#### h) 类函数 `UnboundedQueue::Dequeue()`

- 线程安全地从队列中取出首个元素

#### i) 类函数 `UnboundedQueue::Size()`

- 返回队列长度.

#### i1) 类函数 `UnboundedQueue::Empty()`

- 判断队列是否为空.

#### k) 类内类声明 `UnboundedQueue::Node`

- 链表的最小单元 `Node`
- 为什么构造时 `ref_count` 是2，析构时 `ref_count` 是 `ref_count.fetch_sub(1)?`

**l) 类函数 `UnboundedQueue::Reset()`**

- 构造出一个有一个Node的链表

**m) 类函数 `UnboundedQueue::Destroy()`**

- 删除链表的所有节点

**22. `unbounded_queue_test.cc`**

- `unbounded_queue.h`的test.

**a) 类函数 `UnboundedQueue::Reset()`**

- 构造出一个有一个Node的链表

**b) 类函数 `UnboundedQueue::Destroy()`**

- 删除链表的所有节点

**23. `wait_strategy.h`**

- 该头文件有1个父类和5个子类.

**a) `BUILD`的控制：**

```
cc_library(
    name = "wait_strategy",
    hdrs = ["wait_strategy.h"],
)
```

常规链接库.

**b) 类 `WaitStrategy`**

- 有4个虚函数: `NotifyOne()` `BreakAllWait()` `EmptyWait()` `~WaitStrategy`
- 关于虚函数这篇博客[https://blog.csdn.net/weixin\\_43329614/article/details/89103574](https://blog.csdn.net/weixin_43329614/article/details/89103574)强调了虚函数与派生类的函数形参和函数修饰符必须完全一样，否则虚函数的继承不起作用。另外，我按照这篇博客改了改，发现只有指针和引用可以实现父类用子类资源（函数和成员）的情况：

```
#include<iostream>
#include<string>
using namespace std;
class Base
{
public:
    string str="Base";
    virtual void func()
```

```

    {
        cout <<str<< ":Base!" << endl;
    }
};
class Derived :public Base
{
public:
    string str2="Derived";
    void func()
    {
        cout <<str2<< ":Derived!" << endl;
    }
};
Base base;
Derived derived;
Base base2=static_cast<Base>(derived);
Base* base3=&derived;
Base base4=derived;
Base& base5=derived;
Base base6=derived;
Base* base7=static_cast<Base*>(&derived);
int main()
{
    base.func();
    derived.func();
    base2.func();
    base3->func();
    base4.func();
    base5.func();
    base6.func();
    base7->func();
    std::cout<<sizeof(base)<<std::endl;
    std::cout<<sizeof(derived)<<std::endl;
    std::cout<<sizeof(base2)<<std::endl;
    std::cout<<sizeof(base5)<<std::endl;
    return 0;
}

```

### c)类BlockWaitStrategy

- 继承自类WaitStrategy
- 该类打包了std::mutex mutex\_和condition\_variable cv\_，应该是线程同步功能的进一步封装。

### d)类构造函数BlockWaitStrategy::BlockWaitStrategy()

•

### d1)类成员BlockWaitStrategy::mutex\_

- 线程锁



**d2)类成员`BlockWaitStrategy::cv\_**

- 线程同步:std::condition\_variable

**e)类函数 BlockWaitStrategy::NotifyOne()**

- **override**的作用是给编译器提供报错信息，在成员函数声明或定义中，**override** 确保该函数为虚函数并覆盖来自基类的虚函数。
- 随机唤醒一个线程。

**f)类函数 BlockWaitStrategy::EmptyWait()**

- 线程等待区
- 

**g)类函数 BlockWaitStrategy::BreakAllWait()**

- 唤醒所有线程

**h)类 SleepWaitStrategy**

- 继承自类WaitStrategy

**h1)类成员 SleepWaitStrategy::sleep\_time\_us\_**

- uint64\_t

**i)类构造函数 SleepWaitStrategy::SleepWaitStrategy()****j)类构造函数 SleepWaitStrategy::SleepWaitStrategy()**

- i)的重载。
- 给sleep\_time\_us\_赋值

**k)类函数 SleepWaitStrategy::EmptyWait()**

- 等待sleep\_time\_us\_时间

**l)类函数 SleepWaitStrategy::SetSleepTimeMicroSeconds()**

- 设置sleep\_time\_us\_

**m)类 YieldWaitStrategy**

- 继承自类WaitStrategy

**n)类构造函数 YieldWaitStrategy::YieldWaitStrategy()**

-

**o)类函数 `YieldWaitStrategy::EmptyWait()`**

- `std::this_thread::yield()` 让出时间片，减小CPU占用？

**p)类 `BusySpinWaitStrategy`**

- 继承自类 `WaitStrategy`

**q)类构造函数 `BusySpinWaitStrategy::BusySpinWaitStrategy()`**

- 

**r)类函数 `BusySpinWaitStrategy::EmptyWait()`**

- 什么也不做，返回 `true`
- 

**s)类 `TimeoutBlockWaitStrategy`**

- 继承自类 `WaitStrategy`

**s1)类成员 `TimeoutBlockWaitStrategy::mutex_`****s2)类成员 `TimeoutBlockWaitStrategy::cv_`****s3)类成员 `TimeoutBlockWaitStrategy::time_out_`****q)类构造函数 `TimeoutBlockWaitStrategy::TimeoutBlockWaitStrategy()`**

- 

**s)类构造函数 `TimeoutBlockWaitStrategy::TimeoutBlockWaitStrategy()`**

- q)的重载。
- `std::chrono::milliseconds` C11的时间对象，常用于定时之中。  
<https://blog.csdn.net/oncealong/article/details/28599655>

**t)类函数 `TimeoutBlockWaitStrategy::NotifyOne()`**

- 

**u)类函数 `TimeoutBlockWaitStrategy::EmptyWait()`**

- 

**v)类函数 `TimeoutBlockWaitStrategy::BreakAllWait()`**

- 

**w)类函数 `TimeoutBlockWaitStrategy::SetTimeout()`**

- 

## ./02-common/01-util.md

---

### 代码总览

提供了两个方法, 一个是字符串转哈希, 另一个是返回枚举变量的值(int).

### 功能/知识

#### 1. 方法Hash()

- 用到了`std::hash`, 哈希运算应该是生成一个数字签名.
- `std::size_t`是`unsigned long`.
- 为什么会有这种表达方法? `hash<string>{}(key)`; 注意里边的大括号, 删除又会报错.

#### 2. 方法ToInt()

- 这个方法用到了C++11的新特性: 尾置类型. 它相当于:

```
template <typename Enum>
typename std::underlying_type<Enum>::type ToInt(Enum const value) {
    return static_cast<typename std::underlying_type<Enum>::type>
(value);
}
```

- `std::underlying_type`用于获取枚举变量的基础类型, 它的标准用法是  
`underlying_type<a_enum>::value`
- 所以这个方法的功能是返回当前枚举变量的值, 如:

```
enum Week {Sun, Mon, Tue, Wed, Thu, Fri, Sat}; // 定义枚举类型week
Week today=Week::Fri;
std::cout << "today: " << apollo::cyber::common::ToInt(today) <<
std::endl; //会输出: today: 5
```

- 为什么它要用尾置类型呢? 为什么它要把简单问题复杂化呢?

## ./02-common/02-types.md

---

### 代码总览

提供了一些枚举变量和静态常量. 用于一些标识符?

### 功能/知识

## 1. 枚举变量Relation

- enum可以指定底层类型, 如下. 注意这个`std::uint8_t`, 是强制底层类型.

```
enum Relation : std::uint8_t {  
    NO_RELATION = 0,  
    DIFF_HOST,   // different host  
    DIFF_PROC,   // same host, but different process  
    SAME_PROC,   // same process  
};
```

- 又如:

```
#include <iostream>  
#include <string>  
#include <typeinfo>  
enum class Relation1 : std::uint8_t {  
    NO_RELATION = 0,  
    DIFF_HOST,  
    DIFF_PROC,  
    SAME_PROC,  
};  
enum class Relation2 {  
    NO_RELATION = 0,  
    DIFF_HOST,  
    DIFF_PROC,  
    SAME_PROC,  
};  
int main(int argc, char* argv[]) {  
    std::cout << typeid(std::underlying_type<Relation1>::type).name() <<  
std::endl;  
    std::cout << typeid(std::underlying_type<Relation2>::type).name() <<  
std::endl;  
}
```

它会输出:

```
h  
i
```

`h`应该是`unsigned int`, `i`应该是`int`. 可能是编译器简写了.

- 这个博客讲枚举变量的值得作用域的, 讲得不错. 注意`enum class`和`enum struct`.  
<https://blog.csdn.net/datase/article/details/82773937>

[./02-common/03-time\\_conversion.md](#)

## 代码总览

这个是讲时间转换的.

## 功能/知识

### 1. 常量 `std::vector<std::pair<int32_t, int32_t>> LEAP_SECONDS`

- 描述UNIX时间和GPS时间的闰秒映射关系.
- 

## ./02-common/04-macros.md

---

## 代码总览

macro.h & macros\_test.cc

它提供了一些宏. 它依赖base下的macros.h.

## 功能/知识

### 1. 宏定义 `DEFINE_TYPE_TRAIT(HasShutdown, Shutdown)`

- 它在base/macros.h中, 它展开应该是这样子的:

```
template <typename T>
struct HasShutdown {
    template <typename Class>
    static constexpr bool Test(decltype(&Class::Shutdown)*) {
        return true;
    }
    template <typename>
    static constexpr bool Test(...) {
        return false;
    }

    static constexpr bool value = Test<T>(nullptr);
};

template <typename T>
constexpr bool HasShutdown<T>::value;
```

- `constexpr` 常量修饰符, 作用有二: 1是告诉编译器做优化; 2是初始化后限制修改.
- C++11中`constexpr`的限制, 如下: 此限制已在C++14中接解除. 参考 <https://www.qedev.com/dev/120195.html>.

```
constexpr int func (int x)
{
    if (x<0)
        x = -x;
    return x; // 编译不通过
}
```

```
constexpr int func (int x) { return x < 0 ? -x : x; } //编译通过
```

- 类的静态成员的使用要注意初始化, 否则编译不通过. 如下

```
#include <iostream>
#include <string>
#include <typeinfo>
class ClassWithShutdown {
public:
    static int foo_;
    static void set_foo(int val) {
        foo_ = val;
    }
};
//编译不通过
// int ClassWithShutdown::foo_ = 0;
int main(int argc, char* argv[]) {
    ClassWithShutdown::set_foo(1);
}
```

```
#include <iostream>
#include <string>
#include <typeinfo>
class ClassWithShutdown {
public:
    static int foo_;
    static void set_foo(int val) {
        foo_ = val;
    }
};
//编译通过
int ClassWithShutdown::foo_ = 0;
int main(int argc, char* argv[]) {
    ClassWithShutdown::set_foo(1);
}
```

- 讲decltype的,很有意思, <https://blog.csdn.net/helloworld19970916/article/details/82935374>. 一个用decltype判断类型的用法如下

```

#include <iostream>
#include <string>
#include <typeinfo>
class AClass {
public:
    int a_public_int;
    double a_public_double;
    std::string a_public_string;
    void a_public_function() { std::cout<<"I'm a public function."
<<std::endl; }
    static int a_static_public_int;
    static double a_static_public_double;
    static std::string a_static_public_string;
    static void a_static_public_function() { std::cout<<"I'm a static
public function."<<std::endl; }
private:
    int a_private_int;
    double a_private_double;
    std::string a_private_string;
    void a_private_function() { std::cout><<"I'm a private function."
<<std::endl; }
    static int a_static_private_int;
    static double a_static_private_double;
    static std::string a_static_private_string;
    static void a_static_private_function() { std::cout><<"I'm a static
private function."<<std::endl; }
};
int a_normal_function(){
    return 1;
}
int a_normal_function2(double a_d,double a_d2){
    return 1;
}
// 静态成员必须在定义类的文件中对静态成员变量进行初始化, 否则会编译出错。
int AClass::a_static_public_int = 0;
double AClass::a_static_public_double = 0.0;
std::string AClass::a_static_public_string = "0";
int AClass::a_static_private_int = 0;
double AClass::a_static_private_double = 0.0;
std::string AClass::a_static_private_string = "0";
int main(int argc, char* argv[]) {
    std::cout><<typeid(int).name()<<std::endl; //i
    std::cout><<typeid(int *).name()<<std::endl; //Pi
    std::cout><<typeid(double).name()<<std::endl; //d
    std::cout><<typeid(double *).name()<<std::endl; //Pd
    std::cout><<typeid(std::string).name()<<std::endl;
//Nst7__cxx112basic_stringIcSt11char_traitsIcESaIcEEE
    std::cout><<typeid(std::string *).name()<<std::endl;
//PNst7__cxx112basic_stringIcSt11char_traitsIcESaIcEEE
    std::cout><<typeid(AClass).name()<<std::endl; //6AClas
    std::cout><<typeid(decltype(&AClass::a_public_int) ).name()
<<std::endl; //M6AClassi

```

```

std::cout<<typeid(decltype(&AClass::a_public_int)*).name()
<<std::endl; //PM6AClassi
std::cout<<typeid(decltype(&AClass::a_public_double) ).name()
<<std::endl; //M6AClassd
std::cout<<typeid(decltype(&AClass::a_public_double)*).name()
<<std::endl; //PM6AClassd
std::cout<<typeid(decltype(&AClass::a_public_string) ).name()
<<std::endl;
//M6AClassNSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEEE
std::cout<<typeid(decltype(&AClass::a_public_string)*).name()
<<std::endl;
//PM6AClassNSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEEE
std::cout<<typeid(decltype(&AClass::a_public_function) ).name()
<<std::endl; //M6AClassFvvE
std::cout<<typeid(decltype(&AClass::a_public_function)*).name()
<<std::endl; //PM6AClassFvvE
std::cout<<typeid(decltype(&AClass::a_static_public_int) ).name()
<<std::endl; //Pi
std::cout<<typeid(decltype(&AClass::a_static_public_int)*).name()
<<std::endl; //PPi
std::cout<<typeid(decltype(&AClass::a_static_public_double)
).name()<<std::endl; //Pd
std::cout>
<<typeid(decltype(&AClass::a_static_public_double)*).name()
<<std::endl; //PPd
std::cout<<typeid(decltype(&AClass::a_static_public_string)
).name()<<std::endl;
//PNSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEEE
std::cout>
<<typeid(decltype(&AClass::a_static_public_string)*).name()
<<std::endl; //PPNSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEEE
std::cout<<typeid(decltype(&AClass::a_static_public_function)
).name()<<std::endl; //PFvvE
std::cout>
<<typeid(decltype(&AClass::a_static_public_function)*).name()
<<std::endl; //PPFvvE
std::cout>
<<typeid(decltype(AClass::a_static_public_function)*).name()
<<std::endl; //PFvvE
// std::cout<<typeid(decltype(&AClass::a_private_int) ).name()
<<std::endl; //编译不通过
// std::cout<<typeid(decltype(&AClass::a_private_int)*).name()
<<std::endl; //编译不通过
// std::cout<<typeid(decltype(&AClass::a_private_double) ).name()
<<std::endl; //编译不通过
// std::cout<<typeid(decltype(&AClass::a_private_double)*).name()
<<std::endl; //编译不通过
// std::cout<<typeid(decltype(&AClass::a_private_string) ).name()
<<std::endl; //编译不通过
// std::cout<<typeid(decltype(&AClass::a_private_string)*).name()
<<std::endl; //编译不通过
// std::cout<<typeid(decltype(&AClass::a_private_function) ).name()
<<std::endl; //编译不通过
// std::cout<<typeid(decltype(&AClass::a_private_function)*).name()

```



```

<<std::endl; //编译不通过
    // std::cout<<typeid(decltype(&AClass::a_static_private_int)
).name()<<std::endl; //编译不通过
    // std::cout>
<<typeid(decltype(&AClass::a_static_private_int)*).name()<<std::endl;
//编译不通过
    // std::cout><<typeid(decltype(&AClass::a_static_private_double)
).name()<<std::endl; //编译不通过
    // std::cout>
<<typeid(decltype(&AClass::a_static_private_double)*).name()
<<std::endl; //编译不通过
    // std::cout><<typeid(decltype(&AClass::a_static_private_string)
).name()<<std::endl; //编译不通过
    // std::cout>
<<typeid(decltype(&AClass::a_static_private_string)*).name()
<<std::endl; //编译不通过
    // std::cout><<typeid(decltype(&AClass::a_static_private_function)
).name()<<std::endl; //编译不通过
    // std::cout>
<<typeid(decltype(&AClass::a_static_private_function)*).name()
<<std::endl; //编译不通过
    std::cout<<typeid(decltype(a_normal_function)).name()<<std::endl;
//FivE
    std::cout<<typeid(decltype(a_normal_function2)).name()<<std::endl;
//FidDE
}

```

- 该宏的用处应该是判断类中是否有对应的方法Shutdown(),但是底层逻辑还是搞不懂。一个可以参考的测试demo如下：

```

#include <iostream>
#include <string>
#include <typeinfo>
template <typename T>
struct HasShutdown {
    template <typename Class>
    static constexpr bool Test(decltype(&Class::Shutdown)* ) {
        return true;
    }
    template <typename>
    static constexpr bool Test(...) {
        return false;
    }
    static constexpr bool value = Test<T>(nullptr);
};
template <typename T>
constexpr bool HasShutdown<T>::value;
class ClassWithShutdown {
public:
    void Shutdown() { set_foo(1);}
    static int foo() { return foo_; }
}

```

```

    static void set_foo(int val) { foo_ = val; }
private:
    static int foo_;
};
class ClassWithoutShutdown {
public:
    void Shutdown_invalid() { set_foo(1);}
    static int foo() { return foo_; }
    static void set_foo(int val) { foo_ = val; }

private:
    static int foo_;
};
int main(int argc, char* argv[]) {
    std::cout<<HasShutdown<ClassWithShutdown>::Test<ClassWithShutdown>
(nullptr )<<std::endl;
    std::cout<<HasShutdown<ClassWithShutdown>::value<<std::endl;

    std::cout<<HasShutdown<ClassWithoutShutdown>::Test<ClassWithoutShutdow
n>(nullptr )<<std::endl;
    std::cout<<HasShutdown<ClassWithoutShutdown>::value<<std::endl;
}

```

会输出:

```

1
1
0
0

```

## 2. 方法CallShutdown()

- 它有一个重载.
- `typename`的用法:
  1. 在模板定义语法中关键字`class`与`typename`的作用完全一样.
  2. `typename T::const_iterator it(proto.begin());`告诉编译器`T::const_iterator`是类型而不是变量. 参考 <https://blog.csdn.net/lyn631579741/article/details/110730145>.
- 这个方法的用法应该是如果类中有`Shutdown()`, 调用它. 如果没有, 编译报错. 一个demo:

```

#include <iostream>
#include <string>
#include <typeinfo>
template <typename T>
struct HasShutdown {
    template <typename Class>
    static constexpr bool Test(decltype(&Class::Shutdown)*) {
        return true;
    }
};

```

```

    }
    template <typename>
    static constexpr bool Test(...) {
        return false;
    }
    static constexpr bool value = Test<T>(nullptr);
};
template <typename T>
constexpr bool HasShutdown<T>::value;
class ClassWithShutdown {
public:
    void Shutdown() { std::cout<<"Shutdown"<<std::endl; }
    static int foo() { return foo_; }
    static void set_foo(int val) { foo_ = val; }
private:
    static int foo_;
};
class ClassWithoutShutdown {
public:
    void Shutdown_invalid() { std::cout><<"Shutdown invalid"<<std::endl; }
}
    static int foo() { return foo_; }
    static void set_foo(int val) { foo_ = val; }
private:
    static int foo_;
};
template <typename ClassWithShutdown>
typename std::enable_if<HasShutdown<ClassWithShutdown>::value>::type
CallShutdown(ClassWithShutdown *instance) {
    instance->Shutdown();
}
int main(int argc, char* argv[]) {
    ClassWithShutdown class_with_shutdown;
    CallShutdown(&(class_with_shutdown)); //可以通过编译, 运行会打印
Shutdown
    ClassWithoutShutdown class_without_shutdown;
    // CallShutdown(&(class_without_shutdown)); //不会通过编译
}

```

另外: `typename std::enable_if<bool, T>::type` 的类型是 `T`, `typename std::enable_if<bool, >::type` 的类型是 `void`, 所以 `typename std::enable_if<HasShutdown<ClassWithShutdown>::value>::type` 的类型是 `void`.

- 把一个方法转成 `void` 的含义是什么呢?

### 3. 宏定义 `#define UNUSED(param)`

- 转为 `void` 类型到底意味着什么?

### 4. 宏定义 `#define DISALLOW_COPY_AND_ASSIGN(classname)`

- 这个宏定义的含义是禁止通过拷贝初始化一个对象, 编译期检查. 禁止的是两种拷贝方式: `a(b)` 和 `a=b`.
- 用到了 `=delete` 一个例子如下:

```

class classname1 {
public:
    int a_num;
    classname1() {}
};
class classname2 {
public:
    int a_num;
    classname2() {}
    classname2(const classname2 &) = delete;
};
class classname3 {
public:
    int a_num;
    classname3() {}
    classname3(const classname3 &) = delete;
    classname3 &operator=(const classname3 &) = delete;
};
int main(int argc, char *argv[]) {
    classname1 ca1;
    classname2 ca2;
    classname3 ca3;
    classname1 ca1_copy(ca1);
    // classname2 ca2_copy(ca2); //无法通过编译
    // classname3 ca3_copy(ca3); //无法通过编译
    classname1 ca1_copy2;
    ca1_copy2 = ca1;
    classname2 ca2_copy2;
    ca2_copy2 = ca2;
    classname3 ca3_copy2;
    // ca3_copy2 = ca3; //无法通过编译
}

```

## 5. 宏定义`#define DECLARE_SINGLETON(classname)`

- 单例模式的定义.

## [./02-common/05-log.md](#)

---

### 代码总览

log.h和log\_test.cc

一些定义打印语句的宏 xxxxxx

### 功能/知识

#### 1. 宏定义`MODULE_NAME`

- 调用cyber/binary.h的GetName(). cyber/binary.h中维护了一个name(string), 通过GetName()得到它.

## 2. 宏定义ADEBUG\_MODULE

- VLOG(4)在glog中定义的. 字面阅读的话应该是定义了一个警告的等级.

## 3. 宏定义RETURN\_IF\_NULL

- #if !defined等价于#ifndef吗?
- 宏中#的用法, 阻止宏展开, 如:

```
#include <iostream>
#include <thread>
#include <mutex>
#define COUT_A(a) std::cout<<#a<<": "<<a<<std::endl;
int main()
{
    std::string Im_string="Im a.";
    COUT_A(Im_string)
}
```

### d) 宏定义ADEBUG\_MODULE

<<连接的字符串. 估计是用到一些文本的打印过程中。

VLOG(4)在glog中定义的. 字面阅读的话应该是定义了一个详细的等级。

## ./02-common/06-environment.md

### 代码总览

environment.h和environment\_test.cc

读取一些环境变量.

### 功能/知识

#### 1. 方法GetEnv()

- GetEnv()会查找环境变量\${var\_name}. 若\${var\_name}非空, 返回\${var\_name}; 若\${var\_name}为空, 返回default\_value.default\_value可以作为输入传入, 默认为"".
- std::getenv()返回操作系统环境变量. 一个demo:

```
#include <iostream>
#include <cstdlib>
int main()
{
```

```
if(const char* env_p = std::getenv("PATH"))
    std::cout << "Your PATH is: " << env_p << '\n';
}
```

我的系统返回:

```
Your PATH is:
/opt/ros/kinetic/bin:/home/shiqiang/bin:/home/shiqiang/.local/bin:/home/shiqiang/bin:/home/shiqiang/.local/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin:/home/shiqiang/bin:/home/shiqiang/bin
```

## 2. 方法WorkRoot()

- 获取环境变量\${CYBER\_PATH}. 若\${CYBER\_PATH}非空, 返回\${CYBER\_PATH}; 若\${CYBER\_PATH}为空, 返回"/apollo/cyber".
- 

## ./02-common/07-file.md

### 代码总览

file.h/file.cc/file\_test.cc.

proto与文件之间的读写转, 和对文件(夹)的一些操作.

### 功能/知识

#### 1. 方法GetProtoFromFile()

- 打开文件读取protobuf消息. 若是二进制文件(由文件名后缀判定), 优先调用二进制文件读取方法; 若是文本文件, 优先调用文本文件读取方法.
- `std::equal`, 判断两个vector string或者[]是否相等. 注意短链可以匹配的上长链. 一个demo:

```
// equal algorithm example
#include <algorithm> // std::equal
#include <iostream>  // std::cout
#include <string>    //std::string
int main() {
    std::string str1 = "abcdefg";
    std::string str2 = "abcdefg";
    std::string str3 = "abcd";
    std::string str4 = "efg";
    // yes
    if (std::equal(str1.begin(), str1.end(), str2.begin())) {
        std::cout << "yes" << std::endl;
    } else {
```

```

        std::cout << "false" << std::endl;
    }
    // yes
    if (std::equal(str3.begin(), str3.end(), str1.begin())) {
        std::cout << "yes" << std::endl;
    } else {
        std::cout << "false" << std::endl;
    }
    // false
    if (std::equal(str1.begin(), str1.end(), str3.begin())) {
        std::cout << "yes" << std::endl;
    } else {
        std::cout << "false" << std::endl;
    }
    // yes
    if (std::equal(str4.rbegin(), str4.rend(), str1.rbegin())) {
        std::cout << "yes" << std::endl;
    } else {
        std::cout << "false" << std::endl;
    }
}
}

```

## 2. 方法PathExists()

- 判断目录或文件是否存在.
- `stat`可以用来判断文件是否存在, 如:

```

#include <sys/stat.h>
#include <sys/types.h>
#include <iostream>
using namespace std;
int main()
{
    struct stat info;
    cout<<stat("test15.cc",&info)<<endl; //存在 0, 不存在 -1.
    //不管存在与否, info都会有很多字段 (有值的那种) 可以参考
    https://blog.csdn.net/natpan/article/details/81453209
    cout<<"st_dev is:"<<info.st_dev<<endl;
    cout<<"st_ino is:"<<info.st_ino<<endl;
    cout<<"st_mode is:"<<info.st_mode<<endl;
    cout<<"st_nlink is:"<<info.st_nlink<<endl;
    cout<<"st_uid si:"<<info.st_uid<<endl;
    cout<<"st_gid is:"<<info.st_gid<<endl;
    cout<<"st_rdev is:"<<info.st_rdev<<endl;
    cout<<"st_size is:"<<info.st_size<<endl;
    cout<<"st_blksize is:"<<info.st_blksize<<endl;
    cout<<"st_atime is:"<<info.st_atime<<endl;
    cout<<"st_mtime is:"<<info.st_mtime<<endl;
    cout<<"st_ctime is :"<<info.st_ctime<<endl;
}

```

### 3. 函数DirectoryExists()

- 判断目录是否存在. 注意目录和文件的字段st\_mode是不一样的. 一个demo:

```
#include <sys/stat.h>
#include <sys/types.h>
#include <iostream>
using namespace std;
int main()
{
    struct stat info;
    cout<<stat("test15.cc",&info)<<endl;
    cout<<"st_mode is:"<<info.st_mode<<endl; // st_mode is:33204
    cout<<stat("/home/shiqiang/",&info)<<endl;
    cout<<"st_mode is:"<<info.st_mode<<endl; // st_mode is:16877
}
```

### 4. 方法Glob()

- 用正则的方法查找文件. demo:

```
#include <iostream>
#include <vector>
#include <string>
#include <glob.h>
std::vector<std::string> Glob(const std::string &pattern) {
    glob_t globs = {};
    std::vector<std::string> results;
    if (glob(pattern.c_str(), GLOB_TILDE, nullptr, &globs) == 0) {
        for (size_t i = 0; i < globs.gl_pathc; ++i) {
            results.emplace_back(globs.gl_pathv[i]);
            std::cout<<results.at(results.size()-1)<<std::endl;
        }
    }
    globfree(&globs);
    return results;
}

int main()
{
    // 输入的是正则规则
    Glob("test*.cc");
}
```

### 5. 方法EnsureDirectory()

- 确保文件夹存在, 如果文件夹不存在就创建它.
- 一个使用mkdir的demo如下:



```

#include <iostream>
#include <string>
#include <vector>
#include <sys/stat.h>
int main() {
    std::string directory_path =
"/home/shiqiang/temp_ws/shishisfile/";
    for (size_t i = 1; i < directory_path.size(); ++i) {
        if (directory_path[i] == '/') {
            directory_path[i] = 0;
            std::cout<<"directory_path:"<<directory_path.c_str()
<<std::endl;
            if (mkdir(directory_path.c_str(), S_IRWXU) != 0) { //如果没有成
创建
                if (errno != EEXIST) { //如果没有成功创建的原因不是因为文件夹已经存
在
                    return 0;
                }
            }
            directory_path[i] = '/';
        }
    }
}

```

## 6. 方法CopyDir()

- 作用是拷贝目录.
- 用struct dirent和readdir()的一个demo：

```

#include <sys/stat.h>
#include <iostream>
#include <string>
#include <vector>
#include <dirent.h>

int main() {
    std::string from="./";
    DIR *directory = opendir(from.c_str());
    struct dirent *entry;
    while ((entry = readdir(directory)) != nullptr) {
        std::cout<<"d_name: "<<entry->d_name<<", d_type: "<<(int)entry-
>d_type<<std::endl; //文件和目录的d_type是不一样的
    }
    closedir(directory);
    return 0;
}

```

## 7. 方法GetCurrentPath()

- 获取当前路径. 一个demo:

```
#include <unistd.h>
#include <iostream>
#include <string>
int main() {
    char tmp[4096];
    getcwd(tmp, sizeof(tmp));
    std::cout<<tmp<<std::endl;
    return 0;
}
```

## ./02-common/08-global\_data.md

---

### 代码总览

global\_data.h和global\_data.cc.

维护了一个类: `GlobalData`.

也是获取一些环境变量.

### 功能/知识

#### 1. 类方法`GlobalData::InitHostInfo()`

- 初始化一些东西吧.
- 赋值`host_ip_`, 如果环境变量`CYBER_IP`有值, 把`CYBER_IP`赋给`host_ip_`; 如果果环境变量`CYBER_IP`无值, 当前设备IP赋给`host_ip_`. 只支持IPv4?
- 测试`gethostname`的一个demo:

```
#include <unistd.h>
#include <iostream>
#include <string>
int main() {
    char host_name[1024];
    gethostname(host_name, sizeof(host_name)); //返回hostname, 我的是
carbon
    std::cout<<host_name<<std::endl;
    return 0;
}
```

- `unistd.h`为Linux/Unix系统中内置头文件, 包含了许多系统服务的函数原型, 例如`read`函数 `write`函数和`getpid`函数等. 其作用相当于windows操作系统的`windows.h`, 是操作系统为用户提供的统一API接口, 方便调用系统提供的一些服务.

- 
- 

## 2. 方法 `program_path()`

- 返回当前程序的绝对路径.
- `/proc/self/exe` 是一个符号链接, 代表当前程序的绝对路径. 用 `readlink` 读取 `/proc/self/exe` 可以获取当前程序的绝对路径. 一个 demo:

```
#include <unistd.h>
#include <iostream>
int main(int argc, const char* argv[]) {
    char path[1025];
    auto len = readlink("/proc/self/exe", path, sizeof(path) - 1);
    if (len == -1) {
        return 0;
    }
    path[len] = '\0';
    std::cout << path << std::endl; //
/home/shiqiang/apollo/with_apollo/temp/a.out
    return 0;
}
```

会打印:

```
shiqiang@carbon:~/apollo/with_apollo/temp$ ./a.out
/home/shiqiang/apollo/with_apollo/temp/a.out
```

- 

## [./03-apollo/学习apollo之cyber-blocker1.md](#)

---

### 29. `blocker.h`

- 这个代码写得很漂亮, 很干净。

#### a) build 的红控制

```
cc_library(
    name = "blocker",
    hdrs = ["blocker.h"],
)
```

#### b) 类 `BlockerBase`

- 一个基类函数，专门用于继承，里边的函数全都是虚函数

#### c) 结构体BlockerAttr

- 维护了两个成员变量：`capacity(size_t)`和`channel_name(string)`,然后就是各种构造方法

#### d) 类Blocker

- 继承自BlockerBase
- 声明友元类BlockerManager，这边只声明可以吗？
- 模板

#### e) using 关系图：

- `std::unordered_map`, 一般会把`unordered_map`和`map`比较，`map`底层是红黑树，`unordered_map`底层是哈希表。
- 哈希表在查找上更有优势吗？

#### f) 类构造函数Blocker::Blocker()

- 赋值`attr_`（BlockerAttr），c)介绍的结构体

#### g) 类析构函数Blocker::~Blocker()

- 清除：`observed_msg_queue_`，`published_msg_queue_`，`published_callbacks_`。都是成员变量

#### h) 类函数Blocker::Publish()

- 调用i)

#### i) 类函数Blocker::Publish()

- 调用`Enqueue()`和`Notify()`。
- 所以作用是？

#### j) 类函数Blocker::ClearObserved()

- 线程安全地清除`observed_msg_queue_`

#### k) 类函数Blocker::ClearPublished()

- 线程安全地清除`published_msg_queue_`

#### l) 类函数Blocker::Observe()

- 线程安全地把`observed_msg_queue_`赋值为`published_msg_queue_`

**m) 类函数Blocker::IsObservedEmpty()**

- 线程安全地判断`observed_msg_queue_`是否为空。
- 如果仅是读取的话需要加线程锁吗？难道读取的过程内部还执行了写？

**n) 类函数Blocker::IsPublishedEmpty()**

- 线程安全地判断`published_msg_queue_`是否为空。
- `IsPublishedEmpty()`和`IsObservedEmpty()`共用一把锁。线程锁的复用问题。一把锁可以锁不同的资源吗？既然是不同的资源，它们之间有干涉吗？需要锁吗？

**o) 类函数Blocker::Subscribe()**

- `published_callbacks_`中加入一对键值。

**p) 类函数Blocker::Unsubscribe()**

- `published_callbacks_`中由键删除值。

**q) 类函数Blocker::GetLatestObserved()**

- 声明是正常写法，定义确实尾置类型，这样可以吗？
- 返回队列`observed_msg_queue_`的首项。

**s) 类函数Blocker::GetLatestObservedPtr()**

- 和q)基本上一样，不过返回的是指针。

**t) 类函数Blocker::GetOldestObservedPtr()**

- 返回队列`observed_msg_queue_`的尾项，指针形式。

**u) 类函数Blocker::GetLatestPublishedPtr()**

- 返回队列`observed_msg_queue_`的首项，指针形式。

**v) 类函数Blocker::ObservedBegin()**

- 返回指向队列`observed_msg_queue_`首项的迭代器。

**w) 类函数Blocker::ObservedEnd()**

- 返回指向队列`observed_msg_queue_`尾项的迭代器。

**x) 类函数Blocker::capacity()**

- 返回`attr_.capacity`，它是某个值容量上限？

**y) 类函数Blocker::set\_capacity()**

- 线程安全地，设置attr\_.capacity
- attr\_.capacity是published\_msg\_queue\_的容量。
- published\_msg\_queue长度超出attr\_.capacity时，优先删除队尾的量，知道满足attr\_.capacity

#### z) 类函数Blocker::Reset()

- 私有
- 线程安全地清除：observed\_msg\_queue\_，published\_msg\_queue\_，published\_callbacks\_。都是成员变量

#### z1) 类函数Blocker::Enqueue()

- 私有
- published\_msg\_queue\_的首部添加值，检查size并尝试清除队尾。

#### z2) 类函数Blocker::Notify()

- 私有
- 对于published\_callbacks\_中的每个元素std::pair<std::string, Callback>，它的second：Callback，也就是std::function<void(const MessagePtr&)>，把msg传给它并执行该function。

### 30. blocker\_manager.h&blocker\_manager.cc

- 从命名的角度看这个代码是blocker的管理者。
- 维护了一个类：BlockerManager

#### a) build的红控制

```
cc_library(
    name = "blocker_manager",
    srcs = ["blocker_manager.cc"],
    hdrs = ["blocker_manager.h"],
    deps = [
        ":blocker",
    ],
)
```

#### b) 类BlockerManager

#### c) using

- 

#### d)类析构函数：BlockerManager::~~BlockerManager()

- 清除blockers\_
- blockers\_是成员变量，

**e) 类函数BlockerManager::Instance()**

- 静态函数
- 创建一个BlockerManager智能指针，这个智能指针本身也是静态的，所以在一段代码中多次调用Instance()只会创建一个智能指针而非多次创建？

**f) 类函数BlockerManager::GetOrCreateBlocker()**

- 根据attr.channel\_name查找blockers\_(unordered\_map<string, shared\_ptr<BlockerBase>>),不存在则创建。
- std::dynamic\_pointer\_cast:智能指针的向下转换  
<https://blog.csdn.net/u010846653/article/details/74535519>

**f1) 类函数BlockerManager::GetBlocker()**

- 根据attr.channel\_name查找blockers\_(unordered\_map<string, shared\_ptr<BlockerBase>>),不存在则返回空指针。

**g) 类函数BlockerManager::Publish()**

- 模板
- 调用了f) 创建或调用了一个blocker(Blocker<T>)
- 调用blocker的成员函数Publish()。
- typename

**h) 类函数BlockerManager::Publish()**

- g)的重载。

**i) 类函数BlockerManager::Subscribe()**

- 模板
- 调用了f) 创建或调用了一个blocker(Blocker<T>)
- 调用blocker的成员函数Subscribe()。

**j) 类函数BlockerManager::Unsubscribe()**

- 模板
- 调用了f1) 根据channel\_name查询blocker
- 调用blocker的成员函数Unsubscribe()。

**k) 类函数BlockerManager::Observe()**

- blockers\_中的每一个second(BlockerBase)执行它的Observe()

**l) 类函数BlockerManager::Reset()**

- blockers\_中的每一个second(BlockerBase)执行它的Reset()

### m) 部分私有函数

```
BlockerManager();
BlockerManager(const BlockerManager&) = delete;
BlockerManager& operator=(const BlockerManager&) = delete;
```

这些构造函数以及delete是想只留有Instance()这个函数来创建对象吗？

### n) 成员变量blockers\_

- BlockerMap

### o) 成员变量blocker\_mutex\_

- 线程锁

## ./03-apollo/学习apollo之cyber-data2.md

### data2. data\_notifier.h

- 详细方法参考: 学习apollo之cyber-data2.html
- 维护了一个类: DataNotifier,

### ①.宏定义DECLARE\_SINGLETON(DataNotifier)

- 它的展开是:

```
public:
    static DataNotifier *Instance(bool create_if_needed = true) {
        static ataNotifierD *instance = nullptr;
        if (!instance && create_if_needed) {
            static std::once_flag flag;
            std::call_once(flag, [&] { instance = new (std::nothrow)
DataNotifier(); });
        }
        return instance;
    }
    static void CleanUp() {
        auto instance = Instance(false);
        if (instance != nullptr) {
            CallShutdown(instance);
        }
    }
private:
    DataNotifier();
    DataNotifier(const DataNotifier &) = delete;
    DataNotifier &operator=(const DataNotifier &) = delete;
```



- 这是一个单例模式的写法, 它的作用是在一个系统中一个类只能被实例一次.
- 写单例模式的范式是:构造函数私有; 公有静态函数获取指向该实例的指针; 析构函数公有.

## ./03-apollo/apollo的启动脚本1-2020年12月17日.md

### apollo的启动脚本1

研究apollo是怎么运行的

#### 分析 bootstrap.sh 脚本

命令是：

```
bash scripts/bootstrap.sh
```

bootstrap.sh 的内容是：

```
#!/usr/bin/env bash
# 删除自带注释
DIR="$(cd "$(dirname "${BASH_SOURCE[0]}")" && pwd)" #所执行脚本所在的父目录，也
就是docker中的/apollo/scripts
DREAMVIEW_URL="http://localhost:8888" #

cd "${DIR}/.." # docker中的/apollo

# Make sure supervisord has correct coredump file limit.
ulimit -c unlimited # 似乎是权限设置？ 参考https://www.runoob.com/linux/linux-
comm-ulimit.html

source "${DIR}/apollo_base.sh" # 执行apollo_base.sh

function start() {
    ./scripts/monitor.sh start
    ./scripts/dreamview.sh start
    if [ $? -eq 0 ]; then #source "${DIR}/apollo_base.sh"没有出现错误，$*的用法
参考https://blog.csdn.net/helloxiaozhe/article/details/80940066 挺有意思的
        sleep 2 # wait for some time before starting to check
        http_status="$(curl -o /dev/null -x '' -I -L -s -w '%{http_code}'
"${DREAMVIEW_URL}")" #似乎是在检查服务器是否启动，更多curl的用法，参考
https://www.ruanyifeng.com/blog/2019/09/curl-reference.html
        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/data/core for more information"
        fi
    }
```

```

    fi
}

function stop() {
    ./scripts/dreamview.sh stop
    ./scripts/monitor.sh stop
}

case $1 in
    start)
        start
        ;;
    stop)
        stop
        ;;
    restart)
        stop
        start
        ;;
    *) # 默认是start
        start
        ;;
esac

```

## 分析 apollo\_bash.sh 脚本（由bootstrap.sh脚本启动）

```

#!/usr/bin/env bash
# 删除自带注释
TOP_DIR="$(cd "$(dirname "${BASH_SOURCE[0]}")/.." && pwd -P)" #pwd -p 参考
https://www.cnblogs.com/sjxbg/p/5513840.html
source ${TOP_DIR}/scripts/apollo.bashrc #运行apollo.bahsrc脚本

HOST_ARCH="$(uname -m)" #会返回x86_64

function set_lib_path() {
    local CYBER_SETUP="${APOLLO_ROOT_DIR}/cyber/setup.bash"
    [ -e "${CYBER_SETUP}" ] && . "${CYBER_SETUP}" #如果该脚本存在的话，执行该脚本

    # TODO(storypku):
    # /usr/local/apollo/local_integ/lib

    # FIXME(all): remove PYTHONPATH settings
    export PYTHONPATH="${APOLLO_ROOT_DIR}/modules/tools:${PYTHONPATH}" #设值，
环境变量
    # Set teleop paths
    export
PYTHONPATH="${APOLLO_ROOT_DIR}/modules/teleop/common:${PYTHONPATH}" #设值，
环境变量
    add_to_path "/apollo/modules/teleop/common/scripts" #apollo.bashrc中的自定义命令
}

```

```

function create_data_dir() { #创建一些文件夹
    local DATA_DIR="${APOLLO_ROOT_DIR}/data"
    mkdir -p "${DATA_DIR}/log"
    mkdir -p "${DATA_DIR}/bag"
    mkdir -p "${DATA_DIR}/core"
}

function determine_bin_prefix() { # ?
    APOLLO_BIN_PREFIX=$APOLLO_ROOT_DIR
    if [ -e "${APOLLO_ROOT_DIR}/bazel-bin" ]; then
        APOLLO_BIN_PREFIX="${APOLLO_ROOT_DIR}/bazel-bin"
    fi
    export APOLLO_BIN_PREFIX
}

function setup_device_for_aarch64() { #检查can网路
    local can_dev="/dev/can0"
    if [ ! -e "${can_dev}" ]; then
        warning "No CAN device named ${can_dev}. "
        return
    fi

    sudo ip link set can0 type can bitrate 500000 #初始化can?
    sudo ip link set can0 up #初始化can?
}

function setup_device_for_amd64() { # 另一种can?
    # setup CAN device
    for INDEX in $(seq 0 3); do # 0 1 2 3
        # soft link if sensorbox exist
        if [ -e /dev/zyng_can${INDEX} ] && [ ! -e /dev/can${INDEX} ]; then
            sudo ln -s /dev/zyng_can${INDEX} /dev/can${INDEX}
        fi
        if [ ! -e /dev/can${INDEX} ]; then
            sudo mknod --mode=a+rw /dev/can${INDEX} c 52 $INDEX
        fi
    done

    # setup nvidia device
    sudo /sbin/modprobe nvidia
    sudo /sbin/modprobe nvidia-uvm
    if [ ! -e /dev/nvidia0 ]; then
        info "mknod /dev/nvidia0"
        sudo mknod -m 666 /dev/nvidia0 c 195 0
    fi
    if [ ! -e /dev/nvidiactl ]; then
        info "mknod /dev/nvidiactl"
        sudo mknod -m 666 /dev/nvidiactl c 195 255
    fi
    if [ ! -e /dev/nvidia-uvm ]; then
        info "mknod /dev/nvidia-uvm"
        sudo mknod -m 666 /dev/nvidia-uvm c 243 0
    fi
}

```

```

if [ ! -e /dev/nvidia-uvm-tools ]; then
    info "mknod /dev/nvidia-uvm-tools"
    sudo mknod -m 666 /dev/nvidia-uvm-tools c 243 1
fi
}

function setup_device() {
    if [ "$(uname -s)" != "Linux" ]; then
        info "Not on Linux, skip mapping devices."
        return
    fi
    if [[ "${HOST_ARCH}" == "x86_64" ]]; then
        setup_device_for_amd64
    else
        setup_device_for_aarch64
    fi
}

function decide_task_dir() {
    # Try to find largest NVMe drive.
    DISK="$(df | grep "^/dev/nvme" | sort -nr -k 4 \
        | awk '{print substr($0, index($0, $6))}')"

    # Try to find largest external drive.
    if [ -z "${DISK}" ]; then
        DISK="$(df | grep "/media/${DOCKER_USER}" | sort -nr -k 4 \
            | awk '{print substr($0, index($0, $6))}')"
    fi

    if [ -z "${DISK}" ]; then
        echo "Cannot find portable disk. Fallback to apollo data dir."
        DISK="/apollo"
    fi

    # Create task dir.
    BAG_PATH="${DISK}/data/bag"
    TASK_ID=$(date +%Y-%m-%d-%H-%M-%S)
    TASK_DIR="${BAG_PATH}/${TASK_ID}"
    mkdir -p "${TASK_DIR}"

    echo "Record bag to ${TASK_DIR}..."
    export TASK_ID="${TASK_ID}"
    export TASK_DIR="${TASK_DIR}"
}

function is_stopped_customized_path() {
    MODULE_PATH=$1
    MODULE=$2
    NUM_PROCESSES="$(pgrep -f
"modules/${MODULE_PATH}/launch/${MODULE}.launch" | grep -cv '^1$')"
    if [ "${NUM_PROCESSES}" -eq 0 ]; then
        return 1
    else
        return 0
    fi
}

```

```

    fi
}

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
${APOLLO_ROOT_DIR}/modules/${MODULE_PATH}/launch/${MODULE}.launch &
        sleep 0.5
        is_stopped_customized_path "${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
}

function start() {
    MODULE=$1
    shift

    start_customized_path $MODULE $MODULE "$@"
}

function start_prof_customized_path() {
    MODULE_PATH=$1
    MODULE=$2
    shift 2

    echo "Make sure you have built with 'bash apollo.sh build_prof'"
    LOG="${APOLLO_ROOT_DIR}/data/log/${MODULE}.out"
    is_stopped_customized_path "${MODULE_PATH}" "${MODULE}"
    if [ $? -eq 1 ]; then
        PROF_FILE="/tmp/${MODULE}.prof"
        rm -rf $PROF_FILE
        BINARY="${APOLLO_BIN_PREFIX}/modules/${MODULE_PATH}/${MODULE}"
        eval "CPUPROFILE=$PROF_FILE $BINARY \
            --flagfile=modules/${MODULE_PATH}/conf/${MODULE}.conf \
            --log_dir=${APOLLO_ROOT_DIR}/data/log $@ </dev/null >${LOG} 2>&1 &"
        sleep 0.5
        is_stopped_customized_path "${MODULE_PATH}" "${MODULE}"
        if [ $? -eq 0 ]; then
            echo -e "Launched module ${MODULE} in prof mode. \nExport profile by
command:"

```

```

        echo -e "${YELLOW}google-pprof --pdf $BINARY $PROF_FILE >
${MODULE}_prof.pdf${NO_COLOR}"
        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
}

function start_prof() {
    MODULE=$1
    shift

    start_prof_customized_path $MODULE $MODULE "$@"
}

function start_fe_customized_path() {
    MODULE_PATH=$1
    MODULE=$2
    shift 2

    is_stopped_customized_path "${MODULE_PATH}" "${MODULE}"
    if [ $? -eq 1 ]; then
        eval "cyber_launch start
${APOLLO_ROOT_DIR}/modules/${MODULE_PATH}/launch/${MODULE}.launch"
    else
        echo "Module ${MODULE} is already running - skipping."
        return 2
    fi
}

function start_fe() {
    MODULE=$1
    shift

    start_fe_customized_path $MODULE $MODULE "$@"
}

function start_gdb_customized_path() {
    MODULE_PATH=$1
    MODULE=$2
    shift 2

    eval "gdb --args ${APOLLO_BIN_PREFIX}/modules/${MODULE_PATH}/${MODULE} \
        --flagfile=modules/${MODULE_PATH}/conf/${MODULE}.conf \
        --log_dir=${APOLLO_ROOT_DIR}/data/log $@"
}

function start_gdb() {
    MODULE=$1

```

```

shift

start_gdb_customized_path $MODULE $MODULE "$@"
}

function stop_customized_path() {
    MODULE_PATH=$1
    MODULE=$2

    is_stopped_customized_path "${MODULE_PATH}" "${MODULE}"
    if [ $? -eq 1 ]; then
        echo "${MODULE} process is not running!"
        return
    fi

    cyber_launch stop
    "${APOLLO_ROOT_DIR}/modules/${MODULE_PATH}/launch/${MODULE}.launch"
    if [ $? -eq 0 ]; then
        echo "Successfully stopped module ${MODULE}."
    else
        echo "Module ${MODULE} is not running - skipping."
    fi
}

function stop() {
    MODULE=$1
    stop_customized_path $MODULE $MODULE
}

# Note: This 'help' function here will overwrite the bash builtin command
# 'help'.
# TODO: add a command to query known modules.
function help() {
    cat << EOF
Invoke ". scripts/apollo_base.sh" within docker to add the following
commands to the environment:
Usage: COMMAND [<module_name>]

COMMANDS:
    help:      show this help message
    start:     start the module in background
    start_fe:  start the module without putting in background
    start_gdb: start the module with gdb
    stop:      stop the module
EOF
}

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 $module "$@"
        ;;
    start_fe)
        start_fe_customized_path $module_path $module "$@"
        ;;
    start_gdb)
        start_gdb_customized_path $module_path $module "$@"
        ;;
    start_prof)
        start_prof_customized_path $module_path $module "$@"
        ;;
    stop)
        stop_customized_path $module_path $module
        ;;
    help)
        help
        ;;
    *)
        start_customized_path $module_path $module $cmd "$@"
        ;;
esac
}

# Write log to a file about the env when record a bag.
function record_bag_env_log() {
    if [ -z "${TASK_ID}" ]; then
        TASK_ID=$(date +%Y-%m-%d-%H-%M)
    fi

    git status > /dev/null 2>&1
    if [ $? -ne 0 ]; then
        echo "Not in Git repo, maybe because you are in release container."
        echo "Skip log environment."
        return
    fi

    commit=$(git log -1)
    echo -e "Date:$(date)\n" >> Bag_Env_$TASK_ID.log
    git branch | awk '/\*/ { print "current branch: " $2; }' >>
Bag_Env_$TASK_ID.log
    echo -e "\nNewest commit:\n$commit" >> Bag_Env_$TASK_ID.log
    echo -e "\ngit diff:" >> Bag_Env_$TASK_ID.log
    git diff >> Bag_Env_$TASK_ID.log
    echo -e "\n\n\n\n" >> Bag_Env_$TASK_ID.log
    echo -e "git diff --staged:" >> Bag_Env_$TASK_ID.log
    git diff --staged >> Bag_Env_$TASK_ID.log
}

# run command_name module_name
function run_module() {
    local module=$1
    shift
    run_customized_path $module $module "$@"
}

```



```
unset OMP_NUM_THREADS #删除变量，参考：https://www.runoob.com/linux/linux-comm-unset.html

if [ $APOLLO_IN_DOCKER = "true" ]; then #apollo.bashrc中
    create_data_dir
    set_lib_path $1
    if [ -z $APOLLO_BASE_SOURCED ]; then
        determine_bin_prefix
        export APOLLO_BASE_SOURCED=1
    fi
fi
```

## dreamview.sh

核心是这两行

```
# run_module command_name module_name
run_module dreamview "$@"
```

## monitor.sh

核心是这两行

```
# run_module command_name module_name
run_module monitor "$@"
```

## 小结

整个启动脚本看下来核心就是在这：

```
eval "nohup cyber_launch start # apollo_base.sh 154行
${APOLLO_ROOT_DIR}/modules/${MODULE_PATH}/launch/${MODULE}.launch &"
```

如：

```
cyber_launch start apollo/modules/deamview/launch/dreamview.launch &
```

[./03-apollo/apollo的启动脚本2-2020年12月17日.md](#)

# apollo的启动脚本2

---

## 研究下 cyber\_launch 命令

```
cyber_launch start apollo/modules/deamview/launch/dreamview.launch &
```

该命令在模块cyber下。

另一个开源项目 <https://github.com/daohu527/Dig-into-Apollo> 有介绍， 这里我记录下学习该开源项目分析 cyber的部分的阅读笔记

## design/readme.md

## [./03-apollo/学习apollo之cyber-blocker2.md](#)

---

### 31. `intra_reader.h`

- 这个代码写得很漂亮，很干净。

#### a) build的红控制

```
cc_library(  
    name = "intra_reader",  
    hdrs = ["intra_reader.h"],  
    deps = [  
        ":blocker_manager",  
    ],  
)
```

虽然build中没有写reader.h，但是在头文件里确实包含了：`#include "cyber/node/reader.h"`，神奇

#### b) 类IntraReader

- 继承了`apollo::cyber::Reader<MessageT`
- To be filled

#### b) 类IntraWriter

- 继承了`apollo::cyber::Reader<MessageT`
- To be filled

## [./03-apollo/学习apollo之cyber-class\\_loader1.md](#)

---

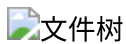
### 24. `utility/class_factory.h` & `utility/class_factory.cc`

无界限队列。

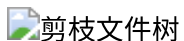
### a) **BUILD**的控制：

```
cc_library(
    name = "class_loader",
    srcs = [
        "class_loader.cc",
        "utility/class_factory.cc",
        "utility/class_loader_utility.cc",
    ],
    hdrs = [
        "class_loader.h",
        "class_loader_register_macro.h",
        "utility/class_factory.h",
        "utility/class_loader_utility.h",
    ],
    deps = [
        "//cyber:init",
        "//cyber/common:log",
        "@poco//:PocoFoundation",
    ],
)
```

这个build控制多个文件，文件展开树：



删除重复文件之后的展开树：



这下知道该先看哪个文件了（`class_factory.h`&`class_factory.cc`->`class_loader_utility.h`&`class_loader_utility.cc`->`class_loader_register_macro.h`->`class_loader.h`&`class_loader.cc`）。

这几个文件涉及到头文件相互包含，看了一篇博客<https://blog.csdn.net/hazir/article/details/38600419>，写的比较好，现复现一下他的工作：(好像看错了，它并没有头文件相互包含，只是有一些头文件重复包含了，不过这里学习下也可以的)

- C++是允许头文件相互包含的先看一个小例子：
  - 代码:

```
// header1.h
#ifndef _HEADER_1_H__
#define _HEADER_1_H__
#include "header2.h"
#endif
```

```
// header2.h
#ifndef _HEADER_2_H__
#define _HEADER_2_H__
#include "header1.h"
#endif
```

```
// source1.cc
#include "header1.h"
int main(int argc, char **argv){
    return 0;
}
```

- 编译：

```
g++ source1.cc
```

编译会成功的。

但若没有`#ifndef`，编译会报错：

- 代码:

```
// header1.h
#include "header2.h"
```

```
// header2.h
#include "header1.h"
```

```
// source1.cc
#include "header1.h"
int main(int argc, char **argv){
    return 0;
}
```

- 编译：

```
g++ source1.cc
```

会报错：

```
In file included from header2.h:2:0,
                  from header1.h:2,
                  from header2.h:2,
                  from header1.h:2,
                  from header2.h:2,
                  from header1.h:2,
                  .....
                  from header2.h:2,
                  from header1.h:2,
                  from header2.h:2,
                  from header1.h:2,
                  from source1.cc:2:
header1.h:2:20: error: #include nested too deeply
```

所以还是要加上`#ifndef`防止递归无穷包含。

再看一个例子：

```
// header1.h
#ifndef _HEADER_1_H__
#define _HEADER_1_H__
#include "header2.h"
#define CONST_FOR_HEADER_2 1 //a macro used in header2.h
bool func(Header2Class* CA){ //defined in header2.h
    return true;
}
#endif
```

```
// header2.h
#ifndef _HEADER_2_H__
#define _HEADER_2_H__
#include "header1.h"
class Header2Class{ // used in header1.h
    int mVar;
    void setMem(){ mVar = CONST_FOR_HEADER_2; }; //macro CONST_FOR_HEADER_2
    is defined in header1.h
};
#endif
```

```
// source1.cc
#include "header1.h"
int main(int argc, char **argv){
    return 0;
}
```

```
// source2.cc
#include"header2.h"
int main(int argc, char **argv){
    return 0;
}
```

编译：

```
g++ source1.cc -o app1 & g++ source2.cc -o app2
```

报错：

```
[1] 9207
In file included from header2.h:3:0,
               from source2.cc:2:
header1.h:6:11: error: 'Header2Class' was not declared in this scope
  bool func(Header2Class* CA){ //defined in header2.h
             ^
header1.h:6:25: error: 'CA' was not declared in this scope
  bool func(Header2Class* CA){ //defined in header2.h
             ^
header1.h:6:28: error: expected ',', or ';' before '{' token
  bool func(Header2Class* CA){ //defined in header2.h
             ^
In file included from header1.h:3:0,
               from source1.cc:1:
header2.h: In member function 'void Header2Class::setMem()':
header2.h:7:25: error: 'CONST_FOR_HEADER_2' was not declared in this scope
  void setMem(){ mVar = CONST_FOR_HEADER_2 };    //macro CONST_FOR_HEADER_2
is defined in head1er.h
             ^
In file included from source2.cc:2:0:
header2.h: In member function 'void Header2Class::setMem()':
header2.h:7:44: error: expected ';' before '}' token
  void setMem(){ mVar = CONST_FOR_HEADER_2 };    //macro CONST_FOR_HEADER_2
is defined in head1er.h
                               ^
[1]+  Exit 1                  g++ source1.cc -o app1
```

先分析第一个问题：**error: 'Header2Class' was not declared in this scope**报错是 **source2.cc**发出的，cc(cpp,c)文件找头文件的话是直接讲头文件包含进来：

```
// source2.cc 找头文件后会拼接成一个新的文件
// 拼接后展开来的新文件
```

```

// header1.h
#ifndef _HEADER_1_H__
#define _HEADER_1_H__
// #include"header2.h"
#define CONST_FOR_HEADER_2 1 //a macro used in header2.h
bool func(Header2Class* CA){ //defined in header2.h
    return true;
}
#endif

// header2.h
#ifndef _HEADER_2_H__
#define _HEADER_2_H__
// #include"header1.h"
class Header2Class{ // used in header1.h
    int mVar;
    void setMem(){ mVar = CONST_FOR_HEADER_2; }; //macro CONST_FOR_HEADER_2
    is defined in head1er.h
};
#endif

// source2.cc
#include"header2.h"
int main(int argc, char **argv){
    return 0;
}

```

<---这里错误看得很清楚，CA在使用前并没有定义，而是在之后定义

所以要在header1.h之中声明类Header2Class。修改之后的文件为：

```

// header1.h
#ifndef _HEADER_1_H__
#define _HEADER_1_H__
#include"header2.h"
#define CONST_FOR_HEADER_2 1 //a macro used in header2.h
class Header2Class; // <-----这里新增一行
bool func(Header2Class* CA){ //defined in header2.h
    return true;
}
#endif

```

```

// header2.h
#ifndef _HEADER_2_H__
#define _HEADER_2_H__
#include"header1.h"
class Header2Class{ // used in header1.h
    int mVar;
    void setMem(){ mVar = CONST_FOR_HEADER_2; }; //macro CONST_FOR_HEADER_2

```

```
is defined in head1er.h
};
#endif
```

```
// source1.cc
#include"header1.h"
int main(int argc, char **argv){
    return 0;
}
```

```
// source2.cc
#include"header2.h"
int main(int argc, char **argv){
    return 0;
}
```

编译:

```
g++ source1.cc -o app1 & g++ source2.cc -o app2
```

过程不报错了，但会另一个问题：

```
[1] 10617
In file included from header1.h:3:0,
        from source1.cc:1:
header2.h: In member function 'void Header2Class::setMem()':
header2.h:7:25: error: 'CONST_FOR_HEADER_2' was not declared in this scope
    void setMem(){ mVar = CONST_FOR_HEADER_2 };    //macro CONST_FOR_HEADER_2
is defined in head1er.h
        ^
In file included from source2.cc:2:0:
header2.h: In member function 'void Header2Class::setMem()':
header2.h:7:44: error: expected ';' before '}' token
    void setMem(){ mVar = CONST_FOR_HEADER_2 };    //macro CONST_FOR_HEADER_2
is defined in head1er.h
        ^
[1]+  Exit 1                  g++ source1.cc -o app1
```

分析第二个问题：error: 'CONST\_FOR\_HEADER\_2' was not declared in this scope报错是source1.cc发出的，编译过程中source1.cc会展开成：

```
// header2.h
#ifndef __HEADER_2_H__
```



```

#define __HEADER_2_H__
// #include"header1.h"

class Header2Class{ // used in header1.h
    int mVar;
    void setMem(){ mVar = CONST_FOR_HEADER_2 };    //macro CONST_FOR_HEADER_2
is defined in head1er.h <---错误也很清楚，CONST_FOR_HEADER_2在使用前并没有定义，
而是在之后定义
};
#endif

// header1.h
#ifndef __HEADER_1_H__
#define __HEADER_1_H__
// #include"header2.h"
#define CONST_FOR_HEADER_2 1 //a macro used in header2.h
class Header2Class;
bool func(Header2Class* CA){ //defined in header2.h
    return true;
}
#endif

// source1.cc
#include"header1.h"
int main(int argc, char **argv){
    return 0;
}

```

所以会报错，改动方法是更改宏定义`#define CONST_FOR_HEADER_2 1`的位置，改动之后代码为：

```

// header1.h
#ifndef __HEADER_1_H__
#define __HEADER_1_H__
#include"header2.h"
// #define CONST_FOR_HEADER_2 1 //a macro used in header2.h <-----以前在
这里
class Header2Class;
bool func(Header2Class* CA){ //defined in header2.h
    return true;
}
#endif

```

```

// header2.h
#ifndef __HEADER_2_H__
#define __HEADER_2_H__
#include"header1.h"

// #define CONST_FOR_HEADER_2 1 <-----移动到这儿
class Header2Class{ // used in header1.h

```

```
int mVar;
void setMem(){ mVar = CONST_FOR_HEADER_2; };    //macro CONST_FOR_HEADER_2
is defined in head1er.h
};
#endif
```

```
// source1.cc
#include"header1.h"
int main(int argc, char **argv){
    return 0;
}
```

```
// source2.cc
#include"header2.h"
int main(int argc, char **argv){
    return 0;
}
```

可以通过编译！

#### b) 类声明 **ClassLoader** :

- 仅仅声明了一个类。

#### c) 类 **AbstractClassFactoryBase**

- 很简单的一个类

##### c1) 类成员 **AbstractClassFactoryBase::relative\_class\_loaders\_**

- `std::vector<ClassLoader*>`

##### c2) 类成员 **AbstractClassFactoryBase::relative\_library\_path\_**

- `std::string`

##### c3) 类成员 **AbstractClassFactoryBase::base\_class\_name\_**

- `std::string`

##### c4) 类成员 **AbstractClassFactoryBase::class\_name\_**

- `std::string`

#### d) 类构造函数: **AbstractClassFactoryBase::AbstractClassFactoryBase()**

- 赋值`relative_library_path_base_class_name_class_name_relative_library_path_`赋为空值

**e) 类析构函数:**`AbstractClassFactoryBase::~~AbstractClassFactoryBase()`

- 虚函数

**f) 类函数:**`AbstractClassFactoryBase::SetRelativeLibraryPath()`

- 设置相关链接库路径，给`relative_library_path_`赋值

**g) 类函数:**`AbstractClassFactoryBase::AddOwnedClassLoader()`

- 向`relative_class_loaders_`中添加新值，`relative_class_loaders_`是一个类加载器指针(`ClassLoader*`)的`vector`，`relative_class_loaders_`是一个类成员。

**h) 类函数:**`AbstractClassFactoryBase::RemoveOwnedClassLoader()`

- 在`relative_class_loaders_`删除一个元素，但`relative_class_loaders_`是一个`vector`，它的删除元素操作是否效率低？或者该方法不会频繁使用？

**i) 类函数:**`AbstractClassFactoryBase::IsOwnedBy()`

- `relative_class_loaders_`是否含有`loader`。

**j) 类函数:**`AbstractClassFactoryBase::IsOwnedByAnybody()`

- 判断`relative_class_loaders_`是否为空。

**k) 类函数:**`AbstractClassFactoryBase::GetRelativeClassLoaders()`

- 返回`relative_class_loaders_`。

**l) 类函数:**`AbstractClassFactoryBase::GetRelativeLibraryPath()`

- 返回`relative_library_path_`。

**m) 类函数:**`AbstractClassFactoryBase::GetBaseClassName()`

- 返回`base_class_name_`。

**n) 类函数:**`AbstractClassFactoryBase::GetClassName()`

- 返回`class_name_`。

**o) 类:**`AbstractClassFactory`

- 继承自`AbstractClassFactoryBase`

**p) 类构造函数:**`AbstractClassFactory::AbstractClassFactory()`

- 集成父类的构造函数。

#### q) 类函数: `AbstractClassFactory::CreateObj()`

- 虚函数。
- `const = 0 const` 和 `=0`，要分开理解，`=0` 表示这个成员函数是纯虚函数，也就是它可以没有定义，只有接口，由它的继承类具体定义它的行为，当然，你也可以给它定义缺省的函数体  
<https://www.cnblogs.com/Stephen-Qin/p/10128922.html>
- 所以该类只能被继承，而不能创建它的对象。使能创建它的派生类。这里可以自己写一个例子感受下。

#### r) 类函数: `AbstractClassFactory::AbstractClassFactory()`

- 为什么不是 `public`？

#### s) 类函数: `AbstractClassFactory::AbstractClassFactory()`

- 为什么不是 `public`？
- 允许拷贝构造

#### t) 类函数: `AbstractClassFactory::AbstractClassFactory()`

- 为什么不是 `public`？
- 允许拷贝构造

#### u) 类: `ClassFactory`

- 继承自 `AbstractClassFactory`

#### v) 类构造函数: `ClassFactory::ClassFactory()`

#### w) 类函数: `ClassFactory::CreateObj()`

•

### 25. `class_loader_utility.h` & `class_loader_utility.cc`

- 它的 `build` 在 24) 中已经介绍。
- 用 C++ 写面向过程风格的代码就是这样写的吗？
- 这份代码不好读的原因是用了大量的 `static+return+&(引用)` 结构；`map` 里边套 `map`。

#### a) 类声明 `ClassLoader`：

- 仅仅声明了一个类。

#### b) `using`：

- 5 个 `using`，它们套来套去的，关系如下：



### c) 函数GetClassFactoryMapMap()

- 维护一个BaseToClassFactoryMapMap实例
- 函数中大量使用了static+return+&(引用)结构：

```
BaseToClassFactoryMapMap& GetClassFactoryMapMap() {
    static BaseToClassFactoryMapMap instance;
    return instance;
}
```

- 它是否相当于：?它是否相当于语法糖？

```
BaseToClassFactoryMapMap instance;
```

### d) 函数GetClassFactoryMapMapMutex()

- 维护一个std::recursive\_mutex实例
- std::recursive\_mutex递归锁，相对于std::mutex，std::recursive\_mutex在锁里边又加了一个计数器。许多文章上说这样可以防止死锁。

```
// 该例子使用std::mutex，可能无法在同一线程里重复加锁，会输出：
/*
1.
*/
#include <iostream>          // std::cout
#include <thread>            // std::thread
#include <mutex>             // std::mutex
int counter=0;
std::mutex mtx; // locks access to counter
void attempt_10k_increases() {
    mtx.lock();
    std::cout<<"1. "<<std::endl;
    mtx.lock();
    std::cout<<"2. "<<std::endl;
    for (int i=0; i<10000; ++i) {
        ++counter;
    }
    mtx.unlock();
    mtx.unlock();
    std::cout<<"3. "<<std::endl;
}

int main (int argc, const char* argv[]) {
    std::thread threads[1];
    for (int i=0; i<1; ++i)
        threads[i] = std::thread(attempt_10k_increases);
}
```

```

    for (auto& th : threads) th.join();
    return 0;
}

```

```

// 该例子使用std::recursive_mutex, 允许重复加锁, 会输出:
/*
1.
2.
3.
*/
#include <iostream>          // std::cout
#include <thread>             // std::thread
#include <mutex>              // std::mutex
int counter=0;
std::mutex mtx; // locks access to counter
void attempt_10k_increases() {
    mtx.lock();
    std::cout<<"1. ";<<std::endl;
    mtx.lock();
    std::cout<<"2. ";<<std::endl;
    for (int i=0; i<10000; ++i) {
        ++counter;
    }
    mtx.unlock();
    mtx.unlock();
    std::cout<<"3. ";<<std::endl;
}

int main (int argc, const char* argv[]) {
    std::thread threads[1];
    for (int i=0; i<1; ++i)
        threads[i] = std::thread(attempt_10k_increases);
    for (auto& th : threads) th.join();
    return 0;
}

```

#### e) 函数GetLibPathPocoShareLibVector()

- 维护一个LibpathPocolibVector实例。

#### f) 函数GetLibPathPocoShareLibMutex()

- 维护一个std::recursive\_mutex实例。

#### g) 函数GetClassFactoryMapByBaseClass()

- 由键查值：BaseToClassFactoryMapMap, 如果没有, 创建一对键值, 键它讲的是typeid\_base\_class\_name(std::string)

**g1) 函数GetCurLoadingLibraryNameReference()**

- 仅在.cc代码中出现。
- 维护一个`library_name(string)`实例

**h) 函数GetCurLoadingLibraryName()**

- 调用`GetCurLoadingLibraryNameReference`,返回值而非引用.

**j) 函数SetCurLoadingLibraryName()**

- 设置`library_name,g1)`中维护的.

**j1) 函数GetCurActiveClassLoaderReference()**

- 维护一个`ClassLoader*`

**k) 函数GetCurActiveClassLoader()**

- 调用`GetCurActiveClassLoaderReference`,返回值而非引用.

**l) 函数SetCurActiveClassLoader()**

- 调用`ClassLoader,j1)`中维护的.

**m) 函数IsLibraryLoaded()**

- `LibpathPocolibVector`和`ClassFactoryVector`的交汇。
- 如果`LibpathPocolibVector`有但`ClassFactoryVector`没有，返回true。
- 这个代码好复杂，虽然每行代码都看得懂，但放在一起不知道他要干什么。
- 

**n) 函数IsLibraryLoadedByAnybody()**

- 线程安全地:判断`LibPathPocoShareLibVector`中是否存在键  
`library_path,LibPathPocoShareLibVector`在e)中维护

**o) 函数LoadLibrary()**

- 如果放入到`LibpathPocolibVector`意味着类已经被加载了吗？
- `Poco::SharedLibrary a_shared_library(library_path)` `Poco::SharedLibrary`的用法吗？
- 创建一个`std::pair<std::string, PocoLibraryPtr>(*library_path*, new Poco::SharedLibrary(*library_path*))`，并加入到`LibpathPocolibVector`中去

**p) 函数UnloadLibrary()**

- 删除`LibpathPocolibVector`中的元素。

**r) 函数RegisterClass()**

- `utility::AbstractClassFactory<Base>* new_class_factory_obj =new utility::ClassFactory<Derived, Base>(*class_name*, *base_class_name*);` 用一个基类指针指向一个派生类对象. 可以调用派生类的资源.
- `new_class_factory_obj`的`relative_class_loaders_(std::vector<ClassLoader*>)`中添加一个值, 使用该函数前必须调用`GetCurActiveClassLoader()`
- `new_class_factory_obj`的`relative_library_path_`设置新值, 使用该函数前必须调用`GetCurLoadingLibraryName()`
- 线程安全地:
- `typeid` 类型名, 编译期确定。
- 在一个ClassFactoryMap中新建一对键值, 其中键是基类的名字(`typeid(Base).name()`), 值通过`factory_map[class_name] = new_class_factory_obj;`赋值, 也就是一个基类指针.
- 总体上: `RegisterClass<Derived, Base>()` 会给`GetClassFactoryMapMap()`中的静态变量`instance(BaseToClassFactoryMapMap)`: 插入一对键值对, 键是基类的类型名称`typeid(Base).name()`, 值的类型是`ClassClassFactoryMap()(std::map<std::string, utility::AbstractClassFactoryBase*>)`所以值也是键值对map, 并针对值新建一对键值对: 把派生类的名称-基类指针 (指向一个派生类对象) (`factory_map[class_name] = new_class_factory_obj;`)
- 注意它是两层键值对map: `std::map<std::string, std::map<std::string, utility::AbstractClassFactoryBase*>>`

#### s) 函数CreateClassObj()

- 线程安全地: 对map: `std::map<std::string, std::map<std::string, utility::AbstractClassFactoryBase*>>`查map[`typeid(Base).name()`][`*class_name*`], 若存在值, 它的类型是`utility::AbstractClassFactoryBase*`, 类型转换为`utility::AbstractClassFactory<Base>*`并赋给factory
  - 如果有值factory且factor的`relative_class_loaders_`中有loader创建一个新的派生类, 并返回该派生类。

#### t) 函数GetValidClassNames()

- 线程安全地: 查找一个拥有loader的factory
- 遍历地获取某个名字

#### u) 函数GetAllClassFactoryObjectsOfLibrary()

- 仅在.cc代码中出现。
- 对 w)GetAllClassFactoryObjects() 再过滤一遍, 满足(`utility::AbstractClassFactoryBase*`)->`GetRelativeLibraryPath() == library_path`的。

#### v) 函数GetAllClassFactoryObjects()

- 仅在.cc代码中出现。



- 格式转换，本来是：`std::map<std::string, utility::AbstractClassFactoryBase*>`,转成  
`std::vector<utility::AbstractClassFactoryBase*>`
- 没有用到全局变量

#### w) 函数GetAllClassFactoryObjects()

- 仅在.cc代码中出现。
- v)的重载
- 线程安全地：把一个BaseToClassFactoryMapMap中的所有utility::AbstractClassFactoryBase\*用一个vector维护。
- 注意BaseToClassFactoryMapMap是两层map，vector抽取了两层之后得到的utility::AbstractClassFactoryBase\*
- std::vector 的insert()函数：
- BaseToClassFactoryMapMap是GetClassFactoryMapMap()中的静态变量。

#### w1) 函数DestroyClassFactoryObjectsOfLibrary()

- 对于一个class\_factory\_map(ClassClassFactoryMap\*非全局变量) 的每个元素AbstractClassFactoryBase\*(map的second)，调用其类内方法判断是否满足某些条件(GetRelativeLibraryPath(), IsOwnedBy)，满足则删除元素。
- `std::map.erase():itr = a_map.erase(itr)`;删除当前元素，返回下一个元素的迭代器。

#### w2) 函数DestroyClassFactoryObjectsOfLibrary()

- w1)的重载。
- 线程安全地
- 操作全局变量: BaseToClassFactoryMapMap
- `std::map.erase():itr = a_map.erase(itr)`;删除当前元素，返回下一个元素的迭代器。

#### x) 函数FindLoadedLibrary()

- 仅在.cc代码中出现。
- 键值查找，返回迭代器。
- 为什么要用vector<pair<string, XXXX>>,而不是用map<string, XXXX>

## ./03-apollo/学习apollo之cyber-class\_loader2.md

---

### 26. class\_loader\_register\_macro.h

- 它的build在24)中已经介绍。

#### a) 宏定义CLASS\_LOADER\_REGISTER\_CLASS\_INTERNAL(Derived, Base, UniqueID)

- #运算符，字符串化；##运算符，粘合字符串。
- 所以宏展开之后应该是：

```
// #define CLASS_LOADER_REGISTER_CLASS_INTERNAL(Derived, Base, UniqueID)
namespace {
    struct ProxyTypeUniqueID {
        ProxyTypeUniqueID() {
            apollo::cyber::class_loader::utility::RegisterClass<Derived, Base>(
                "Derived", "Base");
        }
    };
    static ProxyTypeUniqueID g_register_class_UniqueID;
}
```

#### b) 宏定义 `CLASS_LOADER_REGISTER_CLASS_INTERNAL_1(Derived, Base, UniqueID)`

- 和a)一样

#### c) 宏定义 `#define CLASS_LOADER_REGISTER_CLASS(Derived, Base)`

- 相对于a), `UniqueID`被替换成了 `__COUNTER__`。

### 27. `class_loader.h` & `class_loader.cc`

- 它的build在24)中已经介绍。
- 代码中维护了一个类: `ClassLoader`。

#### a) 类 `ClassLoader`

- 它维护了一个类。

#### b) 类构造函数: `ClassLoader::ClassLoader()`

- 初始化一些成员: `library_path_, loadlib_ref_count_, classobj_ref_count_`

#### c) 类析构函数: `ClassLoader::~ClassLoader()`

- 调用 `UnloadLibrary()`;

#### d) 类函数: `ClassLoader::IsLibraryLoaded()`

- 类是否被加载？

#### e) 类函数: `ClassLoader::LoadLibrary()`

- 加载类，每加载一次， `loadlib_ref_count_`加1。

#### f) 类函数: `ClassLoader::UnloadLibrary()`

- 卸载类，每卸载一次， `loadlib_ref_count_`减1。

#### g) 类函数: `ClassLoader::GetLibraryPath()`

- 返回：`library_path_`

#### h) 类函数: `ClassLoader::GetValidClassNames()`

- 模板函数。
- 作用？

#### i) 类函数: `ClassLoader::CreateClassObj()`

- 模板函数。
- 作用？
- `std::shared_ptr`在构造时可以传入一个析构器  
[https://blog.csdn.net/baidu\\_31541363/article/details/95802210](https://blog.csdn.net/baidu_31541363/article/details/95802210)

```
std::shared_ptr<Base> classObjSharePtr(class_object,
std::bind(&ClassLoader::OnClassObjDeleter<Base>,
this, std::placeholders::_1));
```

#### j) 类函数: `ClassLoader::IsClassValid()`

- 模板函数。
- 根据名字找类？

#### k) 类函数: `ClassLoader::OnClassObjDeleter()`

- 模板函数。
- 删除一个`Base*`

### 28. `class_loader_manager.h`&`class_loader_manager.cc`

- 有一个类`ClassLoaderManager`，这个类应该是管理多个`class_loader`
- 这块看得云里雾里的原因大概有太多代码嵌套，类嵌套，this指针，另外一些类的目的（尤其是类名称的含义）不是很，，， 再看吧。

#### a) Build

```
cc_library(
    name = "class_loader_manager",
    srcs = ["class_loader_manager.cc"],
    hdrs = ["class_loader_manager.h"],
    deps = [
        ":class_loader",
    ],
)
```

链接库。

**b) 类构造函数**`ClassLoaderManager::ClassLoaderManager()`

**c) 类析构函数**`ClassLoaderManager::~~ClassLoaderManager()`

**d) 类函数**`ClassLoaderManager::LoadLibrary()`

- 向`libpath_loader_map_`更新一对键值，键:`library_path`; 值`new class_loader::ClassLoader(library_path)`

**e) 类函数**`ClassLoaderManager::UnloadAllLibrary()`

- 卸载所有的`class_loader`

**g) 类函数**`ClassLoaderManager::IsLibraryValid()`

- 是否有`library_name`

**h) 类函数**`ClassLoaderManager::CreateClassObj()`

- 模板函数
- `class_loader`里的方法写得太乱了。
- 似乎是创建一个类实例指针。
- 、

**i) 类函数**`ClassLoaderManager::CreateClassObj()`

- 模板函数
- h)的重载
- ？
- 

**j) 类函数**`ClassLoaderManager::IsClassValid()`

- 模板函数
- ？

**k) 类函数**`ClassLoaderManager::GetValidClassNames()`

- 模板函数
- ？

**l) 类函数**`ClassLoaderManager::GetClassLoaderByLibPath()`

- 键值查找`libpath_loader_map_`

**m) 类函数**`ClassLoaderManager::GetAllValidClassLoaders()`

- map转vector, map的second

**n) 类函数**`ClassLoaderManager::GetAllValidLibPath()`

- map转vector, map的first

- 

**o) 类函数**`ClassLoaderManager::UnloadLibrary()`

- 卸载class\_loader
- ?

## ./03-apollo/学习apollo之cyber-commo3.md

### cyber

#### 一: common

#### 5. log.h

#### a) BUILD

**BUILD**中的控制方法：

```
cc_library(
  name = "log",
  hdrs = ["log.h"],
  deps = [
    "//cyber:binary",
    "@com_github_google_glog//:glog",
  ],
)
```

依赖cyber下边的binary.h,和网络上的glog?，生成链接库。

#### b) 宏定义LEFT\_BRACKET和RIGHT\_BRACKET

常规宏常量，没什么好讲的。

#### c) 宏定义MODULE\_NAME

调用cyber/binary.h的GetName(), cyber/binary.h中维护了一个name(string)，通过GetName()得到它。

**d) 宏定义ADEBUG\_MODULE**

<<连接的字符串。估计是用到一些文本的打印过程中。

VLOG(4)在glog中定义的。字面阅读的话应该是定义了一个详细的等级。

**e) 宏定义ADEBUG**

打印信息的前缀。

**f) 宏定义ALOG\_MODULE\_STREAM**

ALOG\_MODULE\_STREAM貌似用到了宏递归？。

简单字符串拼接。

**g) 宏定义ALOG\_MODULE**

拼接log\_severity和module。

**h) AINFO AWARN AERROR AFATAL**

模块名和INFO等的拼接。

**i) ALOG\_MODULE\_STREAM\_INFO ALOG\_MODULE\_STREAM\_WARN ALOG\_MODULE\_STREAM\_ERROR  
ALOG\_MODULE\_STREAM\_FATAL**

调用了glog吧。

**i) AINFO\_IF AWARN\_IF AERROR\_IF AFATAL\_IF**

调用了ALOG\_IF。

**j) ALOG\_IF****k) ACHECK AINFO\_EVERY AWARN\_EVERY AERROR\_EVERY**

glog的一些打印语法以及[模块名]

**l) RETURN\_IF\_NULL**

#if !defined等价于#ifndef吗？

宏中#的用法，阻止宏展开，如：

```
#include <iostream>
#include <thread>
#include <mutex>
#define COUT_A(a) std::cout<<#a<<": "<<a<<std::endl;
int main()
```

```
{  
    std::string Im_string="Im a."  
    COUT_A(Im_string)  
}
```

#### m) **RETURN\_VAL\_IF\_NULL**

内容结构与l)较为类似。

#### n) **RETURN\_IF**

内容结构与l)较为类似。

#### o) **RETURN\_VAL\_IF**

内容结构与l)较为类似。

#### p) **\_RETURN\_VAL\_IF\_NULL2\_\_**

内容结构与l)较为类似。

#### q) **\_RETURN\_VAL\_IF2\_\_**

内容结构与l)较为类似。

#### r) **\_RETURN\_IF2\_\_**

内容结构与l)较为类似。

### 6. **log\_test.cc**

a) **log.h**的测试，暂不介绍了。

### 7. **macros\_test.cc**

a) **macros.h**的测试，暂不介绍了。

### 8. **environment.h**

a) **BUILD**中的控制方法：

```
cc_library(  
    name = "environment",  
    hdrs = ["environment.h"],  
    deps = [  
        "//cyber/common:log",  
    ],  
)
```

链接库，依赖 5.log.h。

## b) 内联函数GetEnv

std::getenv返回操作系统环境变量。一个demo：

```
#include <iostream>
#include <cstdlib>
int main()
{
    if(const char* env_p = std::getenv("PATH"))
        std::cout << "Your PATH is: " << env_p << '\n';
}
```

我的当前系统返回：

```
Your PATH is:
/opt/ros/kinetic/bin:/home/shiqiang/bin:/home/shiqiang/.local/bin:/home/shi
qiang/bin:/home/shiqiang/.local/bin:/usr/local/sbin:/usr/local/bin:/usr/sbi
n:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin:/home/shiqiang/
bin:/home/shiqiang/bin
```

GetEnv会查找环境变量\${var\_name}。若\${var\_name}非空，返回\${var\_name}；若\${var\_name}为空，返回default\_value，default\_value可以作为输入传入，默认为""。

## c) 内联函数WorkRoot

获取环境变量\${CYBER\_PATH}。若\${CYBER\_PATH}非空，返回\${CYBER\_PATH}；若\${CYBER\_PATH}为空，返回"/apollo/cyber"。

## 9. file.h&file.cc

### a) BUILD中的控制方法：

```
cc_library(
    name = "file",
    srcs = ["file.cc"],
    hdrs = ["file.h"],
    deps = [
        "//cyber/common:log",
        "@com_google_protobuf//:protobuf",
    ],
)
```

链接库，依赖 5.log.h。



**b) 枚举FileType**

有两个底层类型，`TYPE_FILE`文件类型？`TYPE_DIR`目录类型？

**c) 函数SetProtoToASCIIFile**

`protobuf`消息转文本文件。

**d) 函数SetProtoToASCIIFile**

`protobuf`消息转文本文件。c) 的重载。

**e) 函数GetProtoFromASCIIFile**

打开文本文件读取`protobuf`消息。

**f) 函数SetProtoToBinaryFile**

`protobuf`消息转二进制文件。

**g) 函数GetProtoFromBinaryFile**

打开二进制文件读取`protobuf`消息。

**h) 函数GetProtoFromFile**

打开文件读取`protobuf`消息。若是二进制文件（由文件名后缀判定），优先调用二进制文件读取方法；若是文本文件，优先调用文本文件读取方法。

`std::equal`，判断两个`vector`、`string`或者`[]`是否相等。注意短链可以匹配的上长链。一个demo：

```
// equal algorithm example
#include <algorithm> // std::equal
#include <iostream> // std::cout
#include <string> //std::string

int main() {
    std::string str1 = "abcdefg";
    std::string str2 = "abcdefg";
    std::string str3 = "abcd";
    std::string str4 = "efg";
    // yes
    if (std::equal(str1.begin(), str1.end(), str2.begin())) {
        std::cout << "yes" << std::endl;
    } else {
        std::cout << "false" << std::endl;
    }
    // yes
    if (std::equal(str3.begin(), str3.end(), str1.begin())) {
        std::cout << "yes" << std::endl;
    } else {
```

```

    std::cout << "false" << std::endl;
}
// false
if (std::equal(str1.begin(), str1.end(), str3.begin())) {
    std::cout << "yes" << std::endl;
} else {
    std::cout << "false" << std::endl;
}
// yes
if (std::equal(str4.rbegin(), str4.rend(), str1.rbegin())) {
    std::cout << "yes" << std::endl;
} else {
    std::cout << "false" << std::endl;
}
}

```

### i) 函数 **GetContent**

以字符串形式读取文件内容。

### j) 函数 **GetAbsolutePath**

返回绝对路径，字符串拼接。

### k) 函数 **PathExists**

判断目录或文件是否存在。

**stat**可以用来判断文件是否存在。如：

```

#include <sys/stat.h>
#include <sys/types.h>
#include <iostream>
using namespace std;
int main()
{
    struct stat info;
    cout<<stat("test15.cc",&info)<<endl; //存在 0，不存在 -1.
    //不管存在与否，info都会有很多字段（有值的那种）可以参考
https://blog.csdn.net/natpan/article/details/81453209
    cout<<"st_dev is:"<<info.st_dev<<endl;
    cout<<"st_ino is:"<<info.st_ino<<endl;
    cout<<"st_mode is:"<<info.st_mode<<endl;
    cout<<"st_nlink is:"<<info.st_nlink<<endl;
    cout<<"st_uid si:"<<info.st_uid<<endl;
    cout<<"st_gid is:"<<info.st_gid<<endl;
    cout<<"st_rdev is:"<<info.st_rdev<<endl;
    cout<<"st_size is:"<<info.st_size<<endl;
    cout<<"st_blksize is:"<<info.st_blksize<<endl;
    cout<<"st_atime is:"<<info.st_atime<<endl;
    cout<<"st_mtime is:"<<info.st_mtime<<endl;

```

```
    cout<<"st_ctime is :"<<info.st_ctime<<endl;
}
```

## I) 函数 **DirectoryExists**

判断目录是否存在。注意目录和文件的st\_mode是不一样的。一个demo：

```
#include <sys/stat.h>
#include <sys/types.h>
#include <iostream>
using namespace std;
int main()
{
    struct stat info;
    cout<<stat("test15.cc",&info)<<endl;
    cout<<"st_mode is:"<<info.st_mode<<endl; // st_mode is:33204
    cout<<stat("/home/shiqiang/",&info)<<endl;
    cout<<"st_mode is:"<<info.st_mode<<endl; // st_mode is:16877
}
```

## I) 函数 **Glob**

用正则的方法查找文件。demo：

```
#include <iostream>
#include <vector>
#include <string>
#include <glob.h>

std::vector<std::string> Glob(const std::string &pattern) {
    glob_t globs = {};
    std::vector<std::string> results;
    if (glob(pattern.c_str(), GLOB_TILDE, nullptr, &globs) == 0) {
        for (size_t i = 0; i < globs.gl_pathc; ++i) {
            results.emplace_back(globs.gl_pathv[i]);
            std::cout<<results.at(results.size()-1)<<std::endl;
        }
    }
    globfree(&globs);
    return results;
}

int main()
{
    // 输入的是正则规则
    Glob("test*.cc");
}
```

### m) 函数CopyFile

拷贝文件。

### n) 函数EnsureDirectory

确保文件夹存在，如果文件夹不存在就创建它。

一个使用mkdir的demo如下：

```
#include <iostream>
#include <string>
#include <vector>
#include <sys/stat.h>
int main() {
    std::string directory_path = "/home/shiqiang/temp_ws/shishisfile/";
    for (size_t i = 1; i < directory_path.size(); ++i) {
        if (directory_path[i] == '/') {
            directory_path[i] = 0;
            std::cout<<"directory_path:"<<directory_path.c_str()<<std::endl;
            if (mkdir(directory_path.c_str(), S_IRWXU) != 0) { //如果没有成创建
                if (errno != EEXIST) { //如果没有成功创建的原因不是因为文件夹已经存在
                    return 0;
                }
            }
            directory_path[i] = '/';
        }
    }
}
```

### o) 函数CopyDir

拷贝目录。

用struct dirent和readdir()的一个demo：

```
#include <sys/stat.h>
#include <iostream>
#include <string>
#include <vector>
#include <dirent.h>

int main() {
    std::string from=".";
    DIR *directory = opendir(from.c_str());
    struct dirent *entry;
    while ((entry = readdir(directory)) != nullptr) {
        std::cout<<"d_name: "<<entry->d_name<<", d_type: "<<(int)entry-
>d_type<<std::endl; //文件和目录的d_type是不一样的
    }
}
```

```
    closedir(directory);  
    return 0;  
}
```

#### p) 函数Copy

拷贝，如果是文件就拷贝文件，如果是目录就拷贝目录。

#### q) 函数RemoveAllFiles

删除目录下的文件。

#### r) 函数ListSubPaths

删除目录下的文件。

#### s) 函数ListSubPaths

返回当前目录下的所有子目录。

#### t) 函数GetFileName

再看看吧。

#### u) 函数GetCurrentPath

获取当前目录。一个demo：

```
#include <unistd.h>  
#include <iostream>  
#include <string>  
  
int main() {  
    char tmp[4096];  
    getcwd(tmp, sizeof(tmp));  
    std::cout<<tmp<<std::endl;  
    return 0;  
}
```

#### v) 函数DeleteFile

删除文件或者目录

#### w) 函数CreateDir

创建目录。

## ./03-apollo/学习apollo之cyber-commo4.md

---

### cyber

---

#### 一: common

##### 10. global\_data.h&global\_data.cc

###### a) BUILD

```
cc_library(  
    name = "global_data",  
    srcs = ["global_data.cc"],  
    hdrs = ["global_data.h"],  
    data = [  
        "//cyber:cyber_conf",  
    ],  
    deps = [  
        "//cyber/base:atomic_hash_map",  
        "//cyber/base:atomic_rw_lock",  
        "//cyber/common:environment",  
        "//cyber/common:file",  
        "//cyber/common:macros",  
        "//cyber/common:util",  
        "//cyber/proto:cyber_conf_cc_proto",  
    ],  
)
```

###### b) 类GlobalData

global\_data.h&global\_data.cc含有一个类。

这种using的用法可以关注一下：`using ::apollo::cyber::base::AtomicHashMap;`

###### c) 类析构函数GlobalData::~~GlobalData

常规函数。

###### d) 类函数GlobalData::ProcessId

返回类成员process\_id\_。

###### e) 类函数GlobalData::SetProcessGroup

设置字符串process\_group\_。

###### f) 类函数GlobalData::ProcessGroup

返回process\_group\_。

**g) 类函数GlobalData::ProcessGroup**

返回process\_group\_。

**h) 类函数GlobalData::SetComponentNums**

设置component\_nums\_。

**i) 类函数GlobalData::ComponentNums**

返回component\_nums\_。

**j) 类函数GlobalData::SetSchedName**

设置sched\_name\_。

**k) 类函数GlobalData::SchedName**

返回sched\_name\_。

**l) 类函数GlobalData::HostIp**

返回host\_ip\_。

**m) 类函数GlobalData::HostName**

返回host\_name\_。

**n) 类函数GlobalData::EnableSimulationMode**

设置run\_mode\_，run\_mode\_ = RunMode::MODE\_SIMULATION。

**o) 类函数GlobalData::DisableSimulationMode**

设置run\_mode\_，run\_mode\_ = RunMode::MODE\_REALITY。

**p) 类函数GlobalData::IsRealityMode**

设置run\_mode\_，run\_mode\_ = RunMode::MODE\_REALITY。

**q) 类函数GlobalData::IsMockTimeMode**

Is Mock Time Mode：是模拟时间模式

设置clock\_mode\_，clock\_mode\_ = ClockMode::MODE MOCK。

**r) 类函数GlobalData::InitHostInfo**

- 初始化一些东西吧。
- 测试`gethostname`的一个demo：

```
#include <unistd.h>
#include <iostream>
#include <string>

int main() {
    char host_name[1024];
    gethostname(host_name, sizeof(host_name)); //返回hostname，我的是
    carbon
    std::cout<<host_name<<std::endl;
    return 0;
}
```

- `unistd.h`为Linux/Unix系统中内置头文件，包含了许多系统服务的函数原型，例如`read`函数、`write`函数和`getpid`函数等。其作用相当于windows操作系统的`windows.h`，是操作系统为用户提供统一API接口，方便调用系统提供的一些服务。
- 赋值`host_ip_`，如果环境变量`CYBER_IP`有值，把`CYBER_IP`赋给`host_ip_`；如果果环境变量`CYBER_IP`无值，当前设备IP赋给`host_ip_`。只支持IPv4(?)。

#### s) 类函数`GlobalData::InitConfig`

读取`cyber.pb.conf`赋给`config_`。

#### t) 类函数`GlobalData::Config`

返回`config_`。

#### u) 类函数`GlobalData::RegisterNode`

- 由`node_name(string)`计算哈希值，更新哈希值与`node_name`的映射表。
- 以哈希值为`id`。

#### v) 类函数`GlobalData::GetNodeById`

- 由`id`找`node_name`。

#### w) 类函数`GlobalData::RegisterChannel`

- 逻辑与u) 基本一样，只不过`node_name`换成了`channel`。

#### x) 类函数`GlobalData::GetChannelById`

- 由`id`找`channel`。

#### y) 类函数`GlobalData::RegisterService`



- 逻辑与u) 基本一样，只不过node\_name换成了service。

#### z) 类函数GlobalData::GetServiceById

- 由id找service。

#### aa) 类函数GlobalData::RegisterTaskName

- 逻辑与u) 基本一样，只不过node\_name换成了task\_name。

#### ab) 类函数GlobalData::GetTaskNameById

- 由id找task\_name。

•

#### ac) 函数program\_path

- 返回当前程序的绝对路径。
- /proc/self/exe是一个符号链接，代表当前程序的绝对路径.用readlink读取/proc/self/exe可以获取当前程序的绝对路径。一个demo：

```
#include <unistd.h>
#include <iostream>
int main(int argc, const char* argv[]) {
    char path[1025];
    auto len = readlink("/proc/self/exe", path, sizeof(path) - 1);
    if (len == -1) {
        return 0;
    }
    path[len] = '\0';
    std::cout << path << std::endl; //
/home/shiqiang/apollo/with_apollo/temp/a.out
    return 0;
}
```

会打印：

```
shiqiang@carbon:~/apollo/with_apollo/temp$ ./a.out
/home/shiqiang/apollo/with_apollo/temp/a.out
```

#### ad) 类函数GlobalData::GlobalData

初始化许多东西。

## 11. file\_test.cc

## 12. environment\_test.cc

# ./03-apollo/学习apollo之cyber-common1.md

---

## cyber

---

### 一: common

cyber的地位相当于ros的ros.h，common是cyber的一个子包。

#### 1. util.h

##### a) BUILD

BUILD中是这样控制它的：

```
cc_library(  
    name = "util",  
    hdrs = ["util.h"],  
)
```

它会生成链接库。

##### b) 内联函数Hash

用到了std::hash, 哈希运算应该是生成一个数字签名，这里就是给出一个字符串的数字签名。

std::size\_t是unsigned long。

为什么会有这种表达方法？std::hash<std::string>{}(key); 注意里边的大括号。删除又会报错。

##### c) 模板函数ToInt

这个函数用到了C++11的新特性：尾置类型。

它相当于：

```
template <typename Enum>  
typename std::underlying_type<Enum>::type ToInt(Enum const value) {  
    return static_cast<typename std::underlying_type<Enum>::type>(value);  
}
```

另外，这个函数的功能是返回当前枚举变量的值，如：

```
enum Week {Sun, Mon, Tue, Wed, Thu, Fri, Sat}; // 定义枚举类型week
Week today=Week::Fri;
std::cout << "today: " << apollo::cyber::common::ToInt(today) << std::endl;
```

会返回：

```
today: 5
```

为什么它要把简单问题复杂化呢？

## 2. types.h

### a) BUILD的控制方法：

```
cc_library(
    name = "types",
    hdrs = ["types.h"],
)
```

生成一个链接库。

### b) 空类NullType

里边啥都没有。

### c) 枚举变量ReturnCode

常规枚举变量。

### d) 枚举变量Relation

enum可以指定底层类型：

```
enum Relation : std::uint8_t {
    NO_RELATION = 0,
    DIFF_HOST,   // different host
    DIFF_PROC,   // same host, but different process
    SAME_PROC,   // same process
};
```

注意这个std::uint8\_t，是强制底层类型，如：

```

#include <iostream>
#include <string>
#include <typeinfo>

enum class Relation1 : std::uint8_t {
    NO_RELATION = 0,
    DIFF_HOST,
    DIFF_PROC,
    SAME_PROC,
};

enum class Relation2{
    NO_RELATION = 0,
    DIFF_HOST,
    DIFF_PROC,
    SAME_PROC,
};

int main(int argc, char* argv[]) {
    std::cout << typeid(std::underlying_type<Relation1>::type).name() <<
    std::endl;
    std::cout << typeid(std::underlying_type<Relation2>::type).name() <<
    std::endl;

    system("pause");
}

```

会输出：

```

h
i
sh: 1: pause: not found

```

不用管第三行，**h**应该是**unsigned int**，**i**应该是**int**（可能是编译器的问题）。

这个博客讲枚举变量的值得作用域的，讲得不错。注意**enum class**和**enum struct**。

<https://blog.csdn.net/dataset/article/details/82773937>

**e) 两个char[]常量：SRV\_CHANNEL\_REQ\_SUFFIX[]和SRV\_CHANNEL\_RES\_SUFFIX[]**

### 3. time\_conversion.h

这个是讲时间转换的。

**a) 常量std::vector<std::pair<int32\_t, int32\_t>> LEAP\_SECONDS**

描述UNIX时间和GPS时间的闰秒映射关系。

**b) 常量UNIX\_GPS\_DIFF**

描述UNIX时间和GPS时间常量映射关系。

**c) 常量ONE\_MILLION**

一百万

**d) 常量ONE\_BILLION**

一亿

**e) 模板函数UnixToGpsSeconds**

UNIX时间（秒）转GPS时间（秒）。

**f) 函数UnixToGpsMicroseconds**

UNIX时间（毫秒）转GPS时间（毫秒）。

**g) 函数UnixToGpsNanoseconds**

UNIX时间（微秒）转GPS时间（微秒）。

**h)**

**i) 模板函数GpsToUnixSeconds**

GPS时间（秒）转UNIX时间（秒）。

**j) 函数GpsToUnixMicroseconds**

GPS时间（毫秒）转UNIX时间（毫秒）。

**k) 函数GpsToUnixNanoseconds**

GPS时间（微秒）转UNIX时间（微秒）。

**l) 函数GpsToUnixMicroseconds**

无符号类型，j的重载。

**m) 函数GpsToUnixNanoseconds**

无符号类型，k的重载。

**n) 函数StringToUnixSeconds**

string转UNIX时间，默认string格式"%Y-%m-%d %H:%M:%S"，返回类型uint64\_t。

**o) 函数UnixSecondsToString**

UNIX转string时间，默认string格式"%Y-%m-%d %H:%M:%S"。

## ./03-apollo/学习apollo之cyber-common2.md

### cyber

#### 一: common

#### 4. macros.h

##### a) BUILD

BUILD中的控制方法：

```
cc_library(  
    name = "macros",  
    hdrs = ["macros.h"],  
    deps = [  
        "//cyber/base:macros",  
    ],  
)
```

依赖base下变的macros，生成链接库。

##### b) 宏函数DEFINE\_TYPE\_TRAIT

在base/macros下声明。

它展开应该是这个样子的:

```
template <typename T>  
struct HasShutdown {  
    template <typename Class>  
    static constexpr bool Test(decltype(&Class::Shutdown)*) {  
        return true;  
    }  
    template <typename>  
    static constexpr bool Test(...) {  
        return false;  
    }  
  
    static constexpr bool value = Test<T>(nullptr);  
};  
  
template <typename T>  
constexpr bool HasShutdown<T>::value;
```

需要说明的是：

`constexpr` 常量修饰符，作用有二,1告诉编译器做优化；2初始化后限制修改。

讲 `decltype` 的，很有意思，<https://blog.csdn.net/helloworld19970916/article/details/82935374>

类的静态成员的使用要注意初始化，否则编译不通过。

```
#include <iostream>
#include <string>
#include <typeinfo>
class ClassWithShutdown {
public:
    static int foo_;
    static void set_foo(int val) {
        foo_ = val;
    }
};
//编译通过
int ClassWithShutdown::foo_ = 0;
int main(int argc, char* argv[]) {
    ClassWithShutdown::set_foo(1);
}
```

```
#include <iostream>
#include <string>
#include <typeinfo>
class ClassWithShutdown {
public:
    static int foo_;
    static void set_foo(int val) {
        foo_ = val;
    }
};
//编译不通过
// int ClassWithShutdown::foo_ = 0;
int main(int argc, char* argv[]) {
    ClassWithShutdown::set_foo(1);
}
```

`decltype` 判断类型的用法，和类、类的成员混用下。

```
#include <iostream>
#include <string>
#include <typeinfo>

class AClass {
public:
```

```

    int a_public_int;
    double a_public_double;
    std::string a_public_string;
    void a_public_function() { std::cout<<"I'm a public function."
<<std::endl; }
    static int a_static_public_int;
    static double a_static_public_double;
    static std::string a_static_public_string;
    static void a_static_public_function() { std::cout><<"I'm a static public
function."<<std::endl; }
private:
    int a_private_int;
    double a_private_double;
    std::string a_private_string;
    void a_private_function() { std::cout><<"I'm a private function."
<<std::endl; }
    static int a_static_private_int;
    static double a_static_private_double;
    static std::string a_static_private_string;
    static void a_static_private_function() { std::cout><<"I'm a static
private function."<<std::endl; }
};
int a_normal_function(){
    return 1;
}
int a_normal_function2(double a_d,double a_d2){
    return 1;
}
// 静态成员必须在定义类的文件中对静态成员变量进行初始化，否则会编译出错。
int AClass::a_static_public_int = 0;
double AClass::a_static_public_double = 0.0;
std::string AClass::a_static_public_string = "0";
int AClass::a_static_private_int = 0;
double AClass::a_static_private_double = 0.0;
std::string AClass::a_static_private_string = "0";
int main(int argc, char* argv[]) {
    std::cout><<typeid(int).name()<<std::endl; //i
    std::cout><<typeid(int *).name()<<std::endl; //Pi
    std::cout><<typeid(double).name()<<std::endl; //d
    std::cout><<typeid(double *).name()<<std::endl; //Pd
    std::cout><<typeid(std::string).name()<<std::endl;
//NSt7__cxx112basic_stringIcSt11char_traitsIcESaIcEEE
    std::cout><<typeid(std::string *).name()<<std::endl;
//PNSt7__cxx112basic_stringIcSt11char_traitsIcESaIcEEE
    std::cout><<typeid(AClass).name()<<std::endl; //6AClas
    std::cout><<typeid(decltype(&AClass::a_public_int) ).name()<<std::endl;
//M6AClassi
    std::cout><<typeid(decltype(&AClass::a_public_int)*).name()<<std::endl;
//PM6AClassi
    std::cout><<typeid(decltype(&AClass::a_public_double) ).name()
<<std::endl; //M6AClassd
    std::cout><<typeid(decltype(&AClass::a_public_double)*).name()
<<std::endl; //PM6AClassd
    std::cout><<typeid(decltype(&AClass::a_public_string) ).name()

```



```

<<std::endl;
//M6AClassNST7__cxx1112basic_stringIcSt11char_traitsIcESaIcEEE
    std::cout<<typeid(decltype(&AClass::a_public_string)*).name()
<<std::endl;
//PM6AClassNST7__cxx1112basic_stringIcSt11char_traitsIcESaIcEEE
    std::cout<<typeid(decltype(&AClass::a_public_function) ).name()
<<std::endl; //M6AClassFvve
    std::cout<<typeid(decltype(&AClass::a_public_function)*).name()
<<std::endl; //PM6AClassFvve
    std::cout<<typeid(decltype(&AClass::a_static_public_int) ).name()
<<std::endl; //Pi
    std::cout<<typeid(decltype(&AClass::a_static_public_int)*).name()
<<std::endl; //PPi
    std::cout<<typeid(decltype(&AClass::a_static_public_double) ).name()
<<std::endl; //Pd
    std::cout<<typeid(decltype(&AClass::a_static_public_double)*).name()
<<std::endl; //PPd
    std::cout<<typeid(decltype(&AClass::a_static_public_string) ).name()
<<std::endl; //PNSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEEE
    std::cout<<typeid(decltype(&AClass::a_static_public_string)*).name()
<<std::endl; //PPNSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEEE
    std::cout<<typeid(decltype(&AClass::a_static_public_function) ).name()
<<std::endl; //PFvve
    std::cout<<typeid(decltype(&AClass::a_static_public_function)*).name()
<<std::endl; //PPFvve
    std::cout<<typeid(decltype(AClass::a_static_public_function)*).name()
<<std::endl; //PFvve
    // std::cout<<typeid(decltype(&AClass::a_private_int) ).name()
<<std::endl; //编译不通过
    // std::cout<<typeid(decltype(&AClass::a_private_int)*).name()
<<std::endl; //编译不通过
    // std::cout<<typeid(decltype(&AClass::a_private_double) ).name()
<<std::endl; //编译不通过
    // std::cout<<typeid(decltype(&AClass::a_private_double)*).name()
<<std::endl; //编译不通过
    // std::cout<<typeid(decltype(&AClass::a_private_string) ).name()
<<std::endl; //编译不通过
    // std::cout<<typeid(decltype(&AClass::a_private_string)*).name()
<<std::endl; //编译不通过
    // std::cout<<typeid(decltype(&AClass::a_private_function) ).name()
<<std::endl; //编译不通过
    // std::cout<<typeid(decltype(&AClass::a_private_function)*).name()
<<std::endl; //编译不通过
    // std::cout<<typeid(decltype(&AClass::a_static_private_int) ).name()
<<std::endl; //编译不通过
    // std::cout<<typeid(decltype(&AClass::a_static_private_int)*).name()
<<std::endl; //编译不通过
    // std::cout<<typeid(decltype(&AClass::a_static_private_double) ).name()
<<std::endl; //编译不通过
    // std::cout<<typeid(decltype(&AClass::a_static_private_double)*).name()
<<std::endl; //编译不通过
    // std::cout<<typeid(decltype(&AClass::a_static_private_string) ).name()
<<std::endl; //编译不通过
    // std::cout<<typeid(decltype(&AClass::a_static_private_string)*).name()

```

```
<<std::endl; //编译不通过
// std::cout<<typeid(decltype(&AClass::a_static_private_function)
).name())<<std::endl; //编译不通过
// std::cout<
<<typeid(decltype(&AClass::a_static_private_function)*).name())<<std::endl;
//编译不通过
std::cout<<typeid(decltype(a_normal_function)).name())<<std::endl;
//FivE
std::cout<<typeid(decltype(a_normal_function2)).name())<<std::endl;
//FiddE
}
```

C++11中constexpr的限制：

```
constexpr int func (int x)
{
    if (x>0)
        x = -x;
    return x; // 编译不通过
}
```

```
constexpr int func (int x) { return x < 0 ? -x : x; } //编译通过
```

此限制已在C++14中接解除。参考 <https://www.qedev.com/dev/120195.html>

该宏的用处应该是判断是否有对应的函数(`Shutdown()`),但是底层逻辑还是搞不懂。一个可以参考的测试demo如下：

```
#include <iostream>
#include <string>
#include <typeinfo>
template <typename T>
struct HasShutdown {
    template <typename Class>
    static constexpr bool Test(decltype(&Class::Shutdown)*) {
        return true;
    }
}
template <typename>
static constexpr bool Test(...) {
    return false;
}
static constexpr bool value = Test<T>(nullptr);
};
template <typename T>
constexpr bool HasShutdown<T>::value;

class ClassWithShutdown {
```

```

public:
    void Shutdown() { set_foo(1);}

    static int foo() { return foo_; }
    static void set_foo(int val) { foo_ = val; }
private:
    static int foo_;
};

class ClassWithoutShutdown {
public:
    void Shutdown_invalid() { set_foo(1);}

    static int foo() { return foo_; }
    static void set_foo(int val) { foo_ = val; }

private:
    static int foo_;
};

int main(int argc, char* argv[]) {
    std::cout<<HasShutdown<ClassWithShutdown>::Test<ClassWithShutdown>
(nullptr)<<std::endl;
    std::cout<<HasShutdown<ClassWithShutdown>::value<<std::endl;
    std::cout<<HasShutdown<ClassWithoutShutdown>::Test<ClassWithoutShutdown>
(nullptr)<<std::endl;
    std::cout<<HasShutdown<ClassWithoutShutdown>::value<<std::endl;
}

```

会输出：

```

1
1
0
0

```

### c) 模板函数 `CallShutdown`

`typename` 的用法：

1. 在模板定义语法中关键字 `class` 与 `typename` 的作用完全一样。
2. `typename T::const_iterator it(proto.begin());` 告诉编译器 `T::const_iterator` 是类型而不是变量。

参考 <https://blog.csdn.net/lyn631579741/article/details/110730145>

这个函数的用法应该是如果类型中有 `Shutdown`，调用它。如果没有，编译报错。

一个demo：

```

#include <iostream>
#include <string>
#include <typeinfo>

template <typename T>
struct HasShutdown {
    template <typename Class>
    static constexpr bool Test(decltype(&Class::Shutdown)*) {
        return true;
    }
};

template <typename>
static constexpr bool Test(...) {
    return false;
};

static constexpr bool value = Test<T>(nullptr);
};

template <typename T>
constexpr bool HasShutdown<T>::value;

class ClassWithShutdown {
public:
    void Shutdown() { std::cout<<"Shutdown"<<std::endl; }

    static int foo() { return foo_; }
    static void set_foo(int val) { foo_ = val; }
private:
    static int foo_;
};

class ClassWithoutShutdown {
public:
    void Shutdown_invalid() { std::cout><<"Shutdown invalid"<<std::endl; }

    static int foo() { return foo_; }
    static void set_foo(int val) { foo_ = val; }

private:
    static int foo_;
};

template <typename ClassWithShutdown>
typename std::enable_if<HasShutdown<ClassWithShutdown>::value>::type
CallShutdown(ClassWithShutdown *instance) {
    instance->Shutdown();
}

int main(int argc, char* argv[]) {
    ClassWithShutdown class_with_shutdown;
    CallShutdown(&(class_with_shutdown)); //可以通过编译, 运行会打印 Shutdown
    ClassWithoutShutdown class_without_shutdown;
    // CallShutdown(&(class_without_shutdown)); //不会通过编译
}

```

另外；

`typename std::enable_if<bool, T>::type` 的类型是 `T`，`typename std::enable_if<bool, >::type` 的类型是 `void`，所以 `typename std::enable_if<HasShutdown><ClassWithShutdown>::value>::type` 的类型是 `void`。

#### d) 模板函数 `CallShutdown`

c) 的重载。

这个函数的用法应该是如果类型中有 `Shutdown`，编译报错。如果没有，把一个类转移成 `void`，销毁它？

#### e) 一些宏定义取消

#### f) 宏函数 `#define UNUSED(param) (void)param`

转为 `void` 类型到底意味着什么？

#### g) 宏函数 `#define DISALLOW_COPY_AND_ASSIGN(classname)`

下代码指禁止通过拷贝初始化对象，编译时期检查，两种拷贝方式：`a(b)`和`a=b`。

```
classname(const classname &) = delete;
classname &operator=(const classname &) = delete;
```

一个demo：

```
class classname1 {
public:
    int a_num;
    classname1() {}
};

class classname2 {
public:
    int a_num;
    classname2() {}
    classname2(const classname2 &) = delete;
};

class classname3 {
public:
    int a_num;
    classname3() {}
    classname3(const classname3 &) = delete;
    classname3 &operator=(const classname3 &) = delete;
};
```

```
int main(int argc, char *argv[]) {
    classname1 ca1;
    classname2 ca2;
    classname3 ca3;

    classname1 ca1_copy(ca1);
    // classname2 ca2_copy(ca2); //无法通过编译
    // classname3 ca3_copy(ca3); //无法通过编译

    classname1 ca1_copy2;
    ca1_copy2 = ca1;
    classname2 ca2_copy2;
    ca2_copy2 = ca2;
    classname3 ca3_copy2;
    // ca3_copy2 = ca3; //无法通过编译
}
```

所以这个函数的功能如其名称，禁止拷贝或者拷贝形式的初始化。

## ./03-apollo/学习apollo之cyber-data1.md

### data1.cache\_buffer.h

- 详细方法参考: 学习apollo之cyber-data1.html
- 维护了一个类: CacheBuffer, 它的写法与C++的容器很接近

#### ①. CacheBuffer(const CacheBuffer& rhs)

- `std::lock_guard<std::mutex> lg(*rhs*.mutex_);`这个方法之中为什么要加线程锁? 而且还是加的是源的锁?

#### ②.void Fill(const T& value)

- 如果(!fusion\_callback\_), 该方法指的是向队列中填值, 从该方法可以看出, 这是一个用vector实现了的循环队列: 若容量未满,它是:

0	1	2	3	4	5	6	...	n
data0/head_	data1	data2	data3	data4/tail_	NaN	NaN	NaN	NaN

若容量已满,它是:

0	1	2	3	4	5	6	...	n
datan-2	datan-1	datan/tail_	data0/head_	data1	data2	data3	...	datan-3

- 可以永远最多保存过去capacity\_帧的过去的数据.
- 这个方法写的有点乱, Fill单纯填值就行, 为什么要用fusion\_callback\_截断它的功能? 每次分析到这儿的时候我都要紧盯着fusion\_callback\_有没有赋值.

## ./03-apollo/学习apollo之cyber-data3.md

---

### data3.channel\_buffer.h

- 详细方法参考: 学习apollo之cyber-data3.html
- 维护了一个类: ChannelBuffer,

①.`bool Fetch(uint64_t* index, std::shared_ptr<T>& m);`

- Fetch实现的功能: `buffer_`是一个队列, 假设队列长度是10的话, 可以是`buffer_.head_=0, buffer_tail_=5`(未满了); 可以是`buffer_.head_=0, buffer_tail_=9`(满了), 可以是`buffer_.head_=10, buffer_tail_=19`(不仅满了, 而且还新值替换旧值了),
- 对分析`*index`的情形: 其实还是可以理解啦, `*index == 0`返回队尾; `*index == buffer_>Tail() + 1`超出队列当然返回false; `*index < buffer_>Head()`表示太陈旧的值被替换了, 所以会报一个提醒, 同时返回队尾; 当然剩余的情况就是 `*index`在队列中, 所以正常返回它代表的值.
- 这个方法显得乱的原因是它的`*index`含义不清楚, 到底是偏移值还是绝对值; 某些情况转化了`*index`
- `else if (*index == buffer_>Tail() + 1)`处是否应该是`(*index >= buffer_>Tail() + 1)?`

## ./03-apollo/学习apollo之cyber-data4.md

---

### data3.data\_dispatcher.h

- 详细方法参考: 学习apollo之cyber-data4.html
- 维护了一个类: DataDispatcher,

①.`using BufferVector =`

`std::vector<std::weak_ptr<CacheBuffer<std::shared_ptr<T>>>>`

- 要说的是`std::weak_ptr`, 它在`bool DataDispatcher<T>::Dispatch(const uint64_t channel_id, const std::shared_ptr<T>& msg)`也出现了`lock()`方法.
- `std::weak_ptr`是智能指针的一种
- 多线程智能指针必须加锁: <https://www.cnblogs.com/wang1994/p/10765974.html>
- 单纯用`std::shared_ptr`可能造成内存释放失败:

```
#include <iostream>
#include <memory> // for std::shared_ptr
#include <string>
class Person{
public:
    std::string m_name;
    std::shared_ptr<Person> m_partner;
    Person(const std::string &name): m_name(name){
        std::cout << m_name << " created"<<std::endl;
    }
    ~Person(){
        std::cout << m_name << " destroyed"<<std::endl;
    }
};
```

```

    }
};
int main(){
    auto lucy = std::make_shared<Person>("Lucy"); // create a Person named
"Lucy"
    auto ricky = std::make_shared<Person>("Ricky"); // create a Person named
"Ricky"
    lucy->m_partner = ricky;
    ricky->m_partner = lucy;
    std::cout << lucy.use_count() << std::endl;
    std::cout << ricky.use_count() << std::endl;
    return 0;
}

```

用std::weak\_ptr可能修复这种BUG.

```

#include <iostream>
#include <memory> // for std::shared_ptr
#include <string>
class Person{
public:
    std::string m_name;
    std::weak_ptr<Person> m_partner; //这里修改
    Person(const std::string &name): m_name(name){
        std::cout << m_name << " created"<<std::endl;
    }
    ~Person(){
        std::cout << m_name << " destroyed"<<std::endl;
    }
};
int main(){
    auto lucy = std::make_shared<Person>("Lucy"); // create a Person named
"Lucy"
    auto ricky = std::make_shared<Person>("Ricky"); // create a Person named
"Ricky"
    lucy->m_partner = ricky;
    ricky->m_partner = lucy;
    std::cout << lucy.use_count() << std::endl;
    std::cout << ricky.use_count() << std::endl;
    return 0;
}

```

- 另,智能指针怎样判断先析构那个对象呢? 注意到下两个例子的输出是不一样的. 程序如何判断这种不同呢?

```

#include <iostream>
#include <memory> // for std::shared_ptr
#include <string>
class Person{

```



```

public:
    std::string m_name;
    std::shared_ptr<Person> m_partner;
    Person(const std::string &name): m_name(name){
        std::cout << m_name << " created"<<std::endl;
    }
    ~Person(){
        std::cout << m_name << " destroyed"<<std::endl;
    }
};

int main(){
    auto lucy = std::make_shared<Person>("Lucy"); // create a Person named
"Lucy"
    auto ricky = std::make_shared<Person>("Ricky"); // create a Person named
"Ricky"
    lucy->m_partner = ricky;
    // ricky->m_partner = lucy;
    return 0;
}

```

```

#include <iostream>
#include <memory> // for std::shared_ptr
#include <string>
class Person{
public:
    std::string m_name;
    std::shared_ptr<Person> m_partner;
    Person(const std::string &name): m_name(name){
        std::cout << m_name << " created"<<std::endl;
    }
    ~Person(){
        std::cout << m_name << " destroyed"<<std::endl;
    }
};

int main(){
    auto lucy = std::make_shared<Person>("Lucy"); // create a Person named
"Lucy"
    auto ricky = std::make_shared<Person>("Ricky"); // create a Person named
"Ricky"
    // lucy->m_partner = ricky;
    ricky->m_partner = lucy;
    return 0;
}

```

- ?

来分析下data\_dispatcher\_test.cc

```

#include "cyber/data/data_dispatcher.h"
#include <memory>
#include <vector>
#include "gtest/gtest.h"
#include "cyber/common/util.h"

namespace apollo {
namespace cyber {
namespace data {

template <typename T>
using BufferVector =
std::vector<std::weak_ptr<CacheBuffer<std::shared_ptr<T>>>>>;
// 它是DataDispatcher<T>的buffers_map_(键值对)的值的类型 不知道为什么会在这里
auto channel0 = common::Hash("/channel0");
// std::size_t(unsigned long)
auto channel1 = common::Hash("/channel1");
// std::size_t(unsigned long)
TEST(DataDispatcher, AddBuffer) {
    auto cache_buffer1 = new CacheBuffer<std::shared_ptr<int>>>(2);
    /*
    cache_buffer1->{
        T=shared_ptr<int>, head_=0, tail_=0, fusion_callback_=nullptr,
capacity_=3(2+1), buffer_.size=3(capacity_)
    }
    */
    auto buffer0 = ChannelBuffer<int>(channel0, cache_buffer1);
    /*
    buffer0={
        T=int, channel_id_=channel0,buffer_(BufferType(std::shared_ptr<int>)的
智能指针)=cache_buffer1
    }
    */
    auto cache_buffer2 = new CacheBuffer<std::shared_ptr<int>>>(2);
    /*
    cache_buffer2->{
        T=shared_ptr<int>, head_=0, tail_=0, fusion_callback_=nullptr,
capacity_=3(2+1), buffer_.size=3(capacity_)
    }
    */
    auto buffer1 = ChannelBuffer<int>(channel1, cache_buffer2);
    /*
    buffer0={
        T=int, channel_id_=channel1,buffer_(BufferType(std::shared_ptr<int>)的
智能指针)=cache_buffer2
    }
    */
    auto dispatcher = DataDispatcher<int>::Instance();
    /* dispatcher是一个单例
    dispatcher->{
        notifier_(也是一个单例)->{
            notifies_map_=空
        }
    }

```

```

        buffers_map_=空
    }
    */
    dispatcher->AddBuffer(buffer0);
    /* dispatcher是一个单例
    dispatcher->{
        notifier_(也是一个单例)->{
            notifies_map_=空
        }
        buffers_map_={channel0->{cache_buffer1} }
    }
    */
    dispatcher->AddBuffer(buffer1);
    /* dispatcher是一个单例
    dispatcher->{
        notifier_(也是一个单例)->{
            notifies_map_=空
        }
        buffers_map_={
            channel0->{cache_buffer1},
            channel1->{cache_buffer2}
        }
    }
    */
}

TEST(DataDispatcher, Dispatch) {
    auto cache_buffer = new CacheBuffer<std::shared_ptr<int>>(10);
    /*
    cache_buffer->{
        T=shared_ptr<int>, head_=0, tail_=0, fusion_callback_=nullptr,
        capacity_=11(10+1), buffer_.size=11(capacity_)
    }
    */
    auto buffer = ChannelBuffer<int>(channel0, cache_buffer);
    /*
    buffer={
        T=int, channel_id_=channel0, buffer_(BufferType(std::shared_ptr<int>)的
        智能指针)=cache_buffer
    }
    */
    auto dispatcher = DataDispatcher<int>::Instance();
    /* dispatcher是一个单例, gtest中这种单例模式 不同的TEST()会有干涉吗, 如果把它视作
    一个全局变量的话会有干涉的, 这里我先假设有干涉
    /* dispatcher是一个单例
    dispatcher->{
        notifier_(也是一个单例)->{
            notifies_map_=空
        }
        buffers_map_={
            channel0->{cache_buffer1},
            channel1->{cache_buffer2}
        }
    }
    */
}

```

```

*/
auto msg = std::make_shared<int>(1);
// 智能指针
EXPECT_FALSE(dispatcher->Dispatch(channel0, msg));
/* dispatcher是一个单例，它有更新，更新了cache_buffer1，但dispatcher-
>notifier_为空，所以返回false
dispatcher->{
    notifier_(也是一个单例)->{
        notifies_map_=空
    }
    buffers_map_={
        channel0->{cache_buffer1},
        channel1->{cache_buffer2}
    }
}
*/
/*
cache_buffer1->{
    T=shared_ptr<int>, head_=0, tail_=1, fusion_callback_=nullptr,
capacity_=3(2+1), buffer_.size=3, buffer_={0, msg, 0, ...})
}
*/
dispatcher->AddBuffer(buffer);
/* dispatcher是一个单例，它有更新，
dispatcher->{
    notifier_(也是一个单例)->{
        notifies_map_=空
    }
    buffers_map_={
        channel0->{cache_buffer1, buffer},
        channel1->{cache_buffer2}
    }
}
*/
EXPECT_FALSE(dispatcher->Dispatch(channel0, msg));
/* dispatcher是一个单例，它有更新，更新了buffer
dispatcher->{
    notifier_(也是一个单例)->{
        notifies_map_=空
    }
    buffers_map_={
        channel0->{cache_buffer1, cache_buffer},
        channel1->{cache_buffer2}
    }
}
cache_buffer->{
    T=shared_ptr<int>, head_=0, tail_=0, fusion_callback_=nullptr,
capacity_=11(10+1), buffer_.size=11(capacity_), buffer_={0, msg, 0, ...}
}
*/
auto notifier = std::make_shared<Notifier>();
// 智能指针
DataNotifier::Instance()->AddNotifier(channel0, notifier);
/* DataNotifier::Instance()(和dispatcher->notifier_绑定)->{

```

```

    notifies_map_={channel0->{notifier}}
}
*/
EXPECT_TRUE(dispatcher->Dispatch(channel0, msg));
/* 更新dispatcher, 更新了cache_buffer1和cache_buffer
dispatcher是一个单例, 它有更新, 更新了buffer
dispatcher->{
    notifier_(也是一个单例)->{
        notifies_map_=空
    }
    buffers_map_={
        channel0->{cache_buffer1, cache_buffer},
        channel1->{cache_buffer2}
    }
}
cache_buffer1->{
    T=shared_ptr<int>, head_=0, tail_=0, fusion_callback_=nullptr,
    capacity_=3(2+1), buffer_.size=3(capacity_), buffer_={0, msg, msg, ...}
}
cache_buffer->{
    T=shared_ptr<int>, head_=0, tail_=0, fusion_callback_=nullptr,
    capacity_=11(10+1), buffer_.size=11(capacity_), buffer_={0, msg, msg, ...}
}
但dispatcher->notifier_[channel0]有值, 所以返回true
*/
}

} // namespace data
} // namespace cyber
} // namespace apollo

```

## ./03-apollo/学习apollo之cyber-data5.md

### data5. data\_visitor\_base.h

- 详细方法参考: 学习apollo之cyber-data5.html
- 维护了一个类: DataDispatcher,

#### ①.using BufferVector =

**std::vector<std::weak\_ptr<CacheBuffer<std::shared\_ptr<T>>>>**

- 要说的是std::weak\_ptr, 它在bool DataDispatcher<T>::Dispatch(const uint64\_t channel\_id, const std::shared\_ptr<T>& msg)也出现了lock()方法.
- std::weak\_ptr是智能指针的一种
- 多线程智能指针必须加锁: <https://www.cnblogs.com/wang1994/p/10765974.html>
- 单纯用std::shared\_ptr可能造成内存释放失败:

```

#include <iostream>
#include <memory> // for std::shared_ptr

```

```

#include <string>
class Person{
public:
    std::string m_name;
    std::shared_ptr<Person> m_partner;
    Person(const std::string &name): m_name(name){
        std::cout << m_name << " created"<<std::endl;
    }
    ~Person(){
        std::cout << m_name << " destroyed"<<std::endl;
    }
};
int main(){
    auto lucy = std::make_shared<Person>("Lucy"); // create a Person named
"Lucy"
    auto ricky = std::make_shared<Person>("Ricky"); // create a Person named
"Ricky"
    lucy->m_partner = ricky;
    ricky->m_partner = lucy;
    std::cout << lucy.use_count() << std::endl;
    std::cout << ricky.use_count() << std::endl;
    return 0;
}

```

用std::weak\_ptr可能修复这种BUG.

```

#include <iostream>
#include <memory> // for std::shared_ptr
#include <string>
class Person{
public:
    std::string m_name;
    std::weak_ptr<Person> m_partner; //这里修改
    Person(const std::string &name): m_name(name){
        std::cout << m_name << " created"<<std::endl;
    }
    ~Person(){
        std::cout << m_name << " destroyed"<<std::endl;
    }
};
int main(){
    auto lucy = std::make_shared<Person>("Lucy"); // create a Person named
"Lucy"
    auto ricky = std::make_shared<Person>("Ricky"); // create a Person named
"Ricky"
    lucy->m_partner = ricky;
    ricky->m_partner = lucy;
    std::cout << lucy.use_count() << std::endl;
    std::cout << ricky.use_count() << std::endl;
    return 0;
}

```

- 另,智能指针怎样判断先析构那个对象呢? 注意到下两个例子的输出是不一样的. 程序如何判断这种不同呢?

```
#include <iostream>
#include <memory> // for std::shared_ptr
#include <string>
class Person{
public:
    std::string m_name;
    std::shared_ptr<Person> m_partner;
    Person(const std::string &name): m_name(name){
        std::cout << m_name << " created"<<std::endl;
    }
    ~Person(){
        std::cout << m_name << " destroyed"<<std::endl;
    }
};
int main(){
    auto lucy = std::make_shared<Person>("Lucy"); // create a Person named
"Lucy"
    auto ricky = std::make_shared<Person>("Ricky"); // create a Person named
"Ricky"
    lucy->m_partner = ricky;
    // ricky->m_partner = lucy;
    return 0;
}
```

```
#include <iostream>
#include <memory> // for std::shared_ptr
#include <string>
class Person{
public:
    std::string m_name;
    std::shared_ptr<Person> m_partner;
    Person(const std::string &name): m_name(name){
        std::cout << m_name << " created"<<std::endl;
    }
    ~Person(){
        std::cout << m_name << " destroyed"<<std::endl;
    }
};
int main(){
    auto lucy = std::make_shared<Person>("Lucy"); // create a Person named
"Lucy"
    auto ricky = std::make_shared<Person>("Ricky"); // create a Person named
"Ricky"
    // lucy->m_partner = ricky;
    ricky->m_partner = lucy;
}
```

```
    return 0;
}
```

- ?

## ./03-apollo/学习apollo之cyber-data6.md

### data6. data\_fusion.h

- 维护了模板类: DataFusion 和它的重载?
- 没有对应的html.

#### ④. 模板类的默认类型

- 模板类可以预定义模板类型, 如:

```
•
#include <iostream>
#include <typeinfo>
template <typename T>
class GetType {
public:
    static void judgeType() {
        if (typeid(T) == typeid(int)) {
            std::cout << "It's a int." << std::endl;
        } else if (typeid(T) == typeid(float)) {
            std::cout << "It's a float." << std::endl;
        } else if (typeid(T) == typeid(double)) {
            std::cout << "It's a double." << std::endl;
        } else if (typeid(T) == typeid(char)) {
            std::cout << "It's a char." << std::endl;
        } else {
            std::cout << "I dont know what it is." << std::endl;
        }
    }
};

template <typename M0, typename M1 = int, typename M2 = int, typename
M3 = int>
class AC {
public:
    bool testTemplate(M0 m0, M1 m1, M2 m2, M3 m3) {
        std::cout << "-----" << std::endl;
        GetType<M0>::judgeType();
        GetType<M1>::judgeType();
        GetType<M2>::judgeType();
        GetType<M3>::judgeType();
        std::cout << "+++++" << std::endl;
    }
};

int main() {
    AC<int> ac0; //编译通过, 默认展开成AC<int, int, int, int> ac0
```



```

    ac0.testTemplate(1, 1, 1, 1);
    AC<int, float> ac1; //编译通过, 默认展开成AC<int, float, int, int> ac1
    ac1.testTemplate(1, 1, 1, 1);
    AC<int, float, double> ac2; //编译通过, 默认展开成AC<int, float,
double, int> ac2
    ac2.testTemplate(1, 1, 1, 1);
    AC<int, float, double, char> ac3; //编译通过, 默认展开成AC<int, float,
double, char> ac3
    ac3.testTemplate(1, 1, 1, 1);
}

```

## ②. 模板类的重载

- 下边是一个例子, 很神奇的写法, 目的是为了降低冗余的输出吗?

```

#include <iostream>
#include <typeinfo>
template <typename T>
class GetType {
public:
    static void judgeType() {
        if (typeid(T) == typeid(int)) {
            std::cout << "It's a int." << std::endl;
        } else if (typeid(T) == typeid(float)) {
            std::cout << "It's a float." << std::endl;
        } else if (typeid(T) == typeid(double)) {
            std::cout << "It's a double." << std::endl;
        } else if (typeid(T) == typeid(char)) {
            std::cout << "It's a char." << std::endl;
        } else {
            std::cout << "I dont know what it is." << std::endl;
        }
    }
};
// 类一
template <typename M0, typename M1 = int, typename M2 = int, typename M3 =
int>
class AC {
public:
    bool testTemplate(M0 m0, M1 m1, M2 m2, M3 m3) {
        std::cout << "-----" << std::endl;
        GetType<M0>::judgeType();
        GetType<M1>::judgeType();
        GetType<M2>::judgeType();
        GetType<M3>::judgeType();
        std::cout << "+++++" << std::endl;
    }
};
// 类二
// AC<M0, M1, M2, M3>的重载, 如果实例化一个类AC<int, float, double, char>, 和下面
的重载类没有关系; 但如果实例化一个类AC<int, float, double>, 本来要找 AC<int,

```

float, double, int>实例化的, 但下边的重载类存在, 则会实例化一个AC<int, float, double>, 另外需要注意的是`class AC<M0, M1, M2, int>`这里的int要和上面的int对应的上, 否则编译会报错.

```
template <typename M0, typename M1, typename M2>
class AC<M0, M1, M2, int> {
public:
    bool testTemplate(M0 m0, M1 m1, M2 m2) {
        std::cout << "-----" << std::endl;
        GetType<M0>::judgeType();
        GetType<M1>::judgeType();
        GetType<M2>::judgeType();
        std::cout << "+++++" << std::endl;
    }
};
// 类三
// 并且还可以套娃
template <typename M0, typename M1>
class AC<M0, M1, int, int> {
public:
    bool testTemplate(M0 m0, M1 m1) {
        std::cout << "-----" << std::endl;
        GetType<M0>::judgeType();
        GetType<M1>::judgeType();
        std::cout << "+++++" << std::endl;
    }
};
int main() {
    AC<int> ac0; //类三, 编译通过
    // ac0.testTemplate(1, 1, 1); //编译报错
    ac0.testTemplate(1, 1);
    AC<int, float> ac1; //类三, 编译通过,
    // ac1.testTemplate(1, 1, 1); //编译报错
    ac1.testTemplate(1, 1);
    AC<int, float, double> ac2; //类二, 编译通过
    ac2.testTemplate(1, 1, 1);
    AC<int, float, double, char> ac3; //类一, 编译通过
    ac3.testTemplate(1, 1, 1, 1);
}
```

## ./03-apollo/学习apollo之cyber-data7.md

### data7.all\_latest.h

- 三个类, 也是重载类? 参考 data6.data\_fusion.h
- 如果理解代码含义的话, 只看第一个类, 应该就可以了.

```
template <typename M0, typename M1 = NullType, typename M2 = NullType,
          typename M3 = NullType>
class AllLatest : public DataFusion<M0, M1, M2, M3>
```

## ④. `std::tuple`

- c++元组. c++11中引入的新的类型, 可类比`std::pair`。但是`std::pair`只能支持两个元素。理论上, 元组支持0~任意个元素。如下:

```
#include <tuple>
#include <iostream>
int main() {
    std::tuple<char, int, long, std::string> a_tuple('A', 2, 3, "4");
    int index = 0;
    std::cout << index++ << " = " << std::get<0>(a_tuple) << "\n";
    std::cout << index++ << " = " << std::get<1>(a_tuple) << "\n";
    std::cout << index++ << " = " << std::get<2>(a_tuple) << "\n";
    std::cout << index++ << " = " << std::get<3>(a_tuple).c_str() <<
    "\n";
    return 0;
}
```

这里有一个很有意思的知识点, 就是数值型模板, 之前用的比较多的是类型型模板. 一个用数值类型模板的例子:

```
#include <iostream>
class AC{
public:
    int a_int_array[7]={0,1,2,3,4,5,6};
    template <unsigned int N>
    void accessElementByIndex(){
        std::cout<<N<<": "<<a_int_array[N]<<std::endl;
    }
};
int main() {
    AC a_c;
    a_c.accessElementByIndex<1>();
    a_c.accessElementByIndex<5>();
    a_c.accessElementByIndex<6>();
}
```

阅读下[all\\_latest\\_test.cc](#)代码:

```
#include <memory>
#include <string>
#include <vector>
#include "gtest/gtest.h"

#include "cyber/common/log.h"
#include "cyber/cyber.h"
```

```

#include "cyber/data/data_visitor.h"
#include "cyber/data/fusion/all_latest.h"

namespace apollo {
namespace cyber {
namespace data {

using apollo::cyber::message::RawMessage;
using apollo::cyber::proto::RoleAttributes;

// using FusionDataType = tuple<shared_ptr<RawMessage>,
shared_ptr<RawMessage>>
std::hash<std::string> str_hash;
// hash返回的是一个size_t(unsigned int) 对任何对象操作, 返回一个无符号整型.

TEST(AllLatestTest, two_channels) {
    auto cache0 = new CacheBuffer<std::shared_ptr<RawMessage>>(10);
    // cache0->{T=shared_ptr<RawMessage>, head_=0, tail_=0,
fusion_callback_=nullptr, capacity_=11(10+1), buffer_.size=10(capacity_)}
    auto cache1 = new CacheBuffer<std::shared_ptr<RawMessage>>(10);
    // cache1->{T=shared_ptr<RawMessage>, head_=0, tail_=0,
fusion_callback_=nullptr, capacity_=11(10+1), buffer_.size=10(capacity_)}
    ChannelBuffer<RawMessage> buffer0(static_cast<uint64_t>(0), cache0);
    // buffer0={T=RawMessage, channel_id_=0, buffer_(和cache0绑定)}
    ChannelBuffer<RawMessage> buffer1(static_cast<uint64_t>(1), cache1);
    // buffer1={T=RawMessage, channel_id_=0, buffer_(和cache1绑定)}
    std::shared_ptr<RawMessage> m;
    // m=nullptr
    std::shared_ptr<RawMessage> m0;
    // m0=nullptr
    std::shared_ptr<RawMessage> m1;
    // m1=nullptr
    uint64_t index = 0;
    // index=0
    fusion::AllLatest<RawMessage, RawMessage> fusion(buffer0, buffer1);
    /* fusion={
        M0=RawMessage, M1=RawMessage, buffer_m0_=buffer0, buffer_m1_=buffer1,
        buffer_fusion_={
            T=FusionDataType, channel_id_=0,
            buffer_->{
                T=shared_ptr<FusionDataType>, head_=0, tail_=0,
fusion_callback_=nullptr, capacity_=11, buffer_.size=10
            }
        },
        buffer_m0_.buffer(和cache0绑定)->fusion_callback_={
            函数体, 输入一个const shared_ptr<RawMessage>& m0, 如果
buffer_m1_.buffer_(和cache1绑定)为空, 返回void; 如果buffer_m1_.buffer_(和
cache1绑定)不为空, 获取buffer_m1_.buffer_(和cache1绑定)最后一个元素m1, 执行
buffer_fusion_.Buffer()->Fill(make_shared<FusionDataType>(m0, m1))
        }
    }
    */
    EXPECT_FALSE(fusion.Fusion(&index, m0, m1));
    // 返回false, 因为fusion.buffer_fusion_.buffer_->Empty()==0, 不更改任何对象的

```

值

```
cache0->Fill(std::make_shared<RawMessage>("0-0"));
// 因为cache0->fusion_callback_不为nullptr, 所以执行cache0-
>fusion_callback_(make_shared<RawMessage>("0-0")). 对于cache0-
>fusion_callback_(make_shared<RawMessage>("0-0")), 此时buffer_m1_.buffer_(和
cache1绑定)为空, 所以返回void. 所以该步其实没有更改任何对象的值
```

```
EXPECT_FALSE(fusion.Fusion(&index, m0, m1));
```

```
// 返回false, 因为fusion.buffer_fusion_.buffer_->Empty()==0, 不更改任何对象的
值
```

```
cache1->Fill(std::make_shared<RawMessage>("1-0"));
```

```
// 因为cache1->fusion_callback_=nullptr, 所以该步是在cache1->buffer_处填值,
操作后: cache1->{T=shared_ptr<RawMessage>, head_=0, tail_=1,
fusion_callback_=nullptr, capacity_=11, buffer_.size=10, buffer_={0,
make_shared<RawMessage>("1-0"), 0, ...}}
```

```
EXPECT_FALSE(fusion.Fusion(&index, m0, m1));
```

```
// 返回false, 因为fusion.buffer_fusion_.buffer_->Empty()==0, 不更改任何对象的
值
```

```
cache0->Fill(std::make_shared<RawMessage>("0-1"));
```

```
// 因为cache0->fusion_callback_不为nullptr, 所以执行cache0-
>fusion_callback_(make_shared<RawMessage>("0-1")). 对于cache0-
>fusion_callback_(make_shared<RawMessage>("0-1")), 此时buffer_m1_.buffer_(和
cache1绑定)不为空, 执行buffer_fusion_.buffer_(和cache1绑定)-
>Fill(make_shared<FusionDataType>(m0, m1)), m0是make_shared<RawMessage>("0-
1"), m1是buffer_m1_.buffer_(和cache1绑定)最后一个元素, 对于
buffer_fusion_.buffer_(和cache1绑定)->Fill(make_shared<FusionDataType>(m0,
m1)), buffer_fusion_.buffer_->fusion_callback_=nullptr, 所以给
buffer_fusion_.buffer_填值, 操作后:
```

```
/* fusion={
```

```
    M0=RawMessage, M1=RawMessage, buffer_m0_=buffer0(和之前的fusion相比, 该成
员变量没有更新), buffer_m1_=buffer1(和之前的fusion相比, 该成员变量有更新),
```

```
    buffer_fusion_(和之前的fusion相比, 该成员变量有更新)={
```

```
        T=FusionDataType, channel_id_=0,
```

```
        buffer_(智能指针)->{
```

```
            T=shared_ptr<FusionDataType>, head_=0, tail_=1,
```

```
fusion_callback_=nullptr, capacity_=11, buffer_.size=10, buffer_={0,
make_shared<FusionDataType>("0-1", "1-0"), 0, ...}
```

```
        }
```

```
    },
```

```
    buffer_m0_.buffer(和cache0绑定)->fusion_callback_={
```

```
        函数体, 输入一个const shared_ptr<RawMessage>& m0, 如果
```

```
buffer_m1_.buffer_(和cache1绑定)为空, 返回void; 如果buffer_m1_.buffer_(和
cache1绑定)不为空, 获取buffer_m1_.buffer_(和cache1绑定)最后一个元素m1, 执行
buffer_fusion_.Buffer()->Fill(make_shared<FusionDataType>(m0, m1))
```

```
    }
```

```
}
```

```
*/
```

```
EXPECT_TRUE(fusion.Fusion(&index, m0, m1));
```

```
// fusion.buffer_fusion_.buffer_不为空, 执行完毕:*index=1,
```

```
m0=make_shared<RawMessage>("0-1"), m1=make_shared<RawMessage>("1-0")
```

```
index++;
```

```
// index=2
```

```
EXPECT_EQ(std::string("0-1"), m0->message);
```

```
// 对的
```

```
EXPECT_EQ(std::string("1-0"), m1->message);
```

```

// 对的
EXPECT_FALSE(fusion.Fusion(&index, m0, m1));
// fusion.buffer_fusion_.buffer_不为空, index=2, 所以在
fusion.buffer_fusion_.Fetch(index, fusion_data)中*index == buffer_->Tail()
+ 1, 所以返回false
cache0->Fill(std::make_shared<RawMessage>("0-2"));
// 因为cache0->fusion_callback_不为nullptr, 所以执行cache0-
>fusion_callback_(make_shared<RawMessage>("0-2")). 对于cache0-
>fusion_callback_(make_shared<RawMessage>("0-2")), 此时buffer_m1_.buffer_(和
cache1绑定)不为空, 执行buffer_fusion_.buffer_(和cache1绑定)-
>Fill(make_shared<FusionDataType>(m0, m1)), m0是make_shared<RawMessage>("0-
2"), m1是buffer_m1_.buffer_(和cache1绑定)最后一个元素, 对于
buffer_fusion_.buffer_(和cache1绑定)->Fill(make_shared<FusionDataType>(m0,
m1)), buffer_fusion_.buffer_->fusion_callback_=nullptr, 所以给
buffer_fusion_.buffer_填值, 操作后:
/* fusion={
    M0=RawMessage, M1=RawMessage, buffer_m0_=buffer0(和之前的fusion相比, 该成
员变量没有更新), buffer_m1_=buffer1(和之前的fusion相比, 该成员变量没有更新),
    buffer_fusion_(和之前的fusion相比, 该成员变量有更新)={
        T=FusionDataType, channel_id_=0,
        buffer_(智能指针)->{
            T=shared_ptr<FusionDataType>, head_=0, tail_=2,
            fusion_callback_=nullptr, capacity_=11, buffer_.size=10, buffer_={0,
            make_shared<FusionDataType>("0-1", "1-0"), make_shared<FusionDataType>("0-
            2", "1-0"), 0, ...}
        }
    },
    buffer_m0_.buffer(和cache0绑定)->fusion_callback_={
        函数体, 输入一个const shared_ptr<RawMessage>& m0, 如果
        buffer_m1_.buffer_(和cache1绑定)为空, 返回void; 如果buffer_m1_.buffer_(和
        cache1绑定)不为空, 获取buffer_m1_.buffer_(和cache1绑定)最后一个元素m1, 执行
        buffer_fusion_.Buffer()->Fill(make_shared<FusionDataType>(m0, m1))
    }
}
*/
EXPECT_TRUE(fusion.Fusion(&index, m0, m1));
// fusion.buffer_fusion_.buffer_不为空, index=2,
fusion.buffer_fusion_.buffer_->tail_=2, 执行完毕: m0=make_shared<RawMessage>
("0-2"), m1=make_shared<RawMessage>("1-0")
index++;
// index=3
EXPECT_EQ(std::string("0-2"), m0->message);
// 对的
EXPECT_EQ(std::string("1-0"), m1->message);
// 对的
EXPECT_FALSE(fusion.Fusion(&index, m0, m1));
// fusion.buffer_fusion_.buffer_不为空, index=3, 所以在
fusion.buffer_fusion_.Fetch(index, fusion_data)中*index == buffer_->Tail()
+ 1, 所以返回false
cache1->Fill(std::make_shared<RawMessage>("1-1"));
// 因为cache1->的fusion_callback_=nullptr, 所以该步是在cache1->buffer_处填值,
操作后: cache1->{T=shared_ptr<RawMessage>, head_=0, tail_=2,
fusion_callback_=nullptr, capacity_=11, buffer_.size=10, buffer_={0,
make_shared<RawMessage>("1-0"), make_shared<RawMessage>("1-1"), 0, ...}}

```

```

EXPECT_FALSE(fusion.Fusion(&index, m0, m1));
// fusion.buffer_fusion_.buffer_不为空, index=3, 所以在
fusion.buffer_fusion_.Fetch(index, fusion_data)中*index == buffer_->Tail()
+ 1, 所以返回false
for (int i = 0; i < 100; i++) {
    cache0->Fill(std::make_shared<RawMessage>(std::string("0-") +
  std::to_string(2 + i + 1)));
}
// 因为cache0->fusion_callback_不为nullptr, 所以执行cache0-
>fusion_callback_(make_shared<RawMessage>("0-#(2+i+1)")). 对于cache0-
>fusion_callback_(make_shared<RawMessage>("0-#(2+i+1)")), 此时
buffer_m1_.buffer_(和cache1绑定)不为空, 执行buffer_fusion_.buffer_(和cache1绑
定)->Fill(make_shared<FusionDataType>(m0, m1)), m0是make_shared<RawMessage>
("0-#(2+i+1)"), m1是buffer_m1_.buffer_(和cache1绑定)最后一个元素, 对于
buffer_fusion_.buffer_(和cache1绑定)->Fill(make_shared<FusionDataType>(m0,
m1)), buffer_fusion_.buffer_->fusion_callback_=nullptr, 所以给
buffer_fusion_.buffer_填值, for循环操作后:
/* fusion={
    M0=RawMessage, M1=RawMessage, buffer_m0_=buffer0(和之前的fusion相比, 该成
员变量没有更新), buffer_m1_=buffer1(和之前的fusion相比, 该成员变量有更新),
    buffer_fusion_(和之前的fusion相比, 该成员变量有更新)={
        T=FusionDataType, channel_id_=0,
        buffer_(智能指针)->{
            T=shared_ptr<FusionDataType>, head_=92, tail_=102,
fusion_callback_=nullptr, capacity_=11, buffer_.size=10, buffer_={
                make_shared<FusionDataType>("0-99", "1-1"),
                make_shared<FusionDataType>("0-100", "1-1"),
                make_shared<FusionDataType>("0-101", "1-1"),
                make_shared<FusionDataType>("0-102", "1-1"),
                make_shared<FusionDataType>("0-92", "1-1"),
                make_shared<FusionDataType>("0-93", "1-1"),
                make_shared<FusionDataType>("0-94", "1-1"),
                make_shared<FusionDataType>("0-95", "1-1"),
                make_shared<FusionDataType>("0-96", "1-1"),
                make_shared<FusionDataType>("0-97", "1-1"),
                make_shared<FusionDataType>("0-98", "1-1"),
            }
        }
    },
    buffer_m0_.buffer_(和cache0绑定)->fusion_callback_={
        函数体, 输入一个const shared_ptr<RawMessage>& m0, 如果
buffer_m1_.buffer_(和cache1绑定)为空, 返回void; 如果buffer_m1_.buffer_(和
cache1绑定)不为空, 获取buffer_m1_.buffer_(和cache1绑定)最后一个元素m1, 执行
buffer_fusion_.Buffer()->Fill(make_shared<FusionDataType>(m0, m1))
    }
}
*/
EXPECT_TRUE(fusion.Fusion(&index, m0, m1));
// fusion.buffer_fusion_.buffer_不为空, *index=3,
fusion.buffer_fusion_.buffer_->head_=92, 所以*index < buffer_->Head(), 执行
完毕: *index=102, m0=make_shared<RawMessage>("0-102"),
m1=make_shared<RawMessage>("1-1")
index++;
//index=103

```



```

    EXPECT_EQ(std::string("0-102"), m0->message);
    // 对的
}
// 下边的TEST(AllLatestTest, three_channels)和TEST(AllLatestTest,
four_channels)与上边及其类似。不再分析

TEST(AllLatestTest, three_channels) {
    auto cache0 = new CacheBuffer<std::shared_ptr<RawMessage>>(10);
    auto cache1 = new CacheBuffer<std::shared_ptr<RawMessage>>(10);
    auto cache2 = new CacheBuffer<std::shared_ptr<RawMessage>>(10);
    ChannelBuffer<RawMessage> buffer0(0, cache0);
    ChannelBuffer<RawMessage> buffer1(1, cache1);
    ChannelBuffer<RawMessage> buffer2(2, cache2);
    std::shared_ptr<RawMessage> m;
    std::shared_ptr<RawMessage> m0;
    std::shared_ptr<RawMessage> m1;
    std::shared_ptr<RawMessage> m2;
    uint64_t index = 0;
    fusion::AllLatest<RawMessage, RawMessage, RawMessage> fusion(buffer0,
  buffer1,
  buffer2);

    // normal fusion
    EXPECT_FALSE(fusion.Fusion(&index, m0, m1, m2));
    cache0->Fill(std::make_shared<RawMessage>("0-0"));
    EXPECT_FALSE(fusion.Fusion(&index, m0, m1, m2));
    cache1->Fill(std::make_shared<RawMessage>("1-0"));
    EXPECT_FALSE(fusion.Fusion(&index, m0, m1, m2));
    cache2->Fill(std::make_shared<RawMessage>("2-0"));
    EXPECT_FALSE(fusion.Fusion(&index, m0, m1, m2));
    cache0->Fill(std::make_shared<RawMessage>("0-1"));
    EXPECT_TRUE(fusion.Fusion(&index, m0, m1, m2));
    index++;
    EXPECT_EQ(std::string("0-1"), m0->message);
    EXPECT_EQ(std::string("1-0"), m1->message);
    EXPECT_EQ(std::string("2-0"), m2->message);
}

TEST(AllLatestTest, four_channels) {
    auto cache0 = new CacheBuffer<std::shared_ptr<RawMessage>>(10);
    auto cache1 = new CacheBuffer<std::shared_ptr<RawMessage>>(10);
    auto cache2 = new CacheBuffer<std::shared_ptr<RawMessage>>(10);
    auto cache3 = new CacheBuffer<std::shared_ptr<RawMessage>>(10);
    ChannelBuffer<RawMessage> buffer0(0, cache0);
    ChannelBuffer<RawMessage> buffer1(1, cache1);
    ChannelBuffer<RawMessage> buffer2(2, cache2);
    ChannelBuffer<RawMessage> buffer3(3, cache3);
    std::shared_ptr<RawMessage> m;
    std::shared_ptr<RawMessage> m0;
    std::shared_ptr<RawMessage> m1;
    std::shared_ptr<RawMessage> m2;
    std::shared_ptr<RawMessage> m3;
    uint64_t index = 0;
    fusion::AllLatest<RawMessage, RawMessage, RawMessage, RawMessage> fusion(

```



```

        buffer0, buffer1, buffer2, buffer3);

// normal fusion
EXPECT_FALSE(fusion.Fusion(&index, m0, m1, m2, m3));
cache0->Fill(std::make_shared<RawMessage>("0-0"));
EXPECT_FALSE(fusion.Fusion(&index, m0, m1, m2, m3));
cache1->Fill(std::make_shared<RawMessage>("1-0"));
EXPECT_FALSE(fusion.Fusion(&index, m0, m1, m2, m3));
cache2->Fill(std::make_shared<RawMessage>("2-0"));
EXPECT_FALSE(fusion.Fusion(&index, m0, m1, m2, m3));
cache3->Fill(std::make_shared<RawMessage>("3-0"));
EXPECT_FALSE(fusion.Fusion(&index, m0, m1, m2, m3));
cache0->Fill(std::make_shared<RawMessage>("0-1"));
EXPECT_TRUE(fusion.Fusion(&index, m0, m1, m2, m3));
index++;
EXPECT_EQ(std::string("0-1"), m0->message);
EXPECT_EQ(std::string("1-0"), m1->message);
EXPECT_EQ(std::string("2-0"), m2->message);
EXPECT_EQ(std::string("3-0"), m3->message);
}

} // namespace data
} // namespace cyber
} // namespace apollo

```

## ./03-apollo/学习apollo之cyber-data8.md

### data7. data\_visitor\_test.h

阅读下data\_visitor\_test.cc代码:

```

#include "cyber/data/data_visitor.h"
#include <memory>
#include <string>
#include <vector>
#include "gtest/gtest.h"
#include "cyber/common/log.h"
#include "cyber/cyber.h"
#include "cyber/message/raw_message.h"

namespace apollo {
namespace cyber {
namespace data {

using apollo::cyber::message::RawMessage;
using apollo::cyber::proto::RoleAttributes;
std::hash<std::string> str_hash;

auto channel0 = str_hash("/channel0");
// 类型是uint64_t

```

```

auto channel1 = str_hash("/channel1");
// 类型是uint64_t
auto channel2 = str_hash("/channel2");
// 类型是uint64_t
auto channel3 = str_hash("/channel3");
// 类型是uint64_t
void DispatchMessage(uint64_t channel_id, int num) {
    for (int i = 0; i < num; ++i) {
        auto raw_msg = std::make_shared<RawMessage>();
        // 创建一个智能指针
        DataDispatcher<RawMessage>::Instance()->Dispatch(channel_id, raw_msg);
        // 单例模式, 键值查找DataDispatcher<RawMessage>::Instance()->buffers_map_
        如果查找到对应的值(类型, 可以视作vector<CacheBuffer<T>>), 对于
        vector<CacheBuffer<T>>的每个元素: CacheBuffer<T>, 执行Fill(msg), Fill的功能再说
        一次, 如果fusion_callback_有值, 执行fusion_callback_; 如果fusion_callback_没值,
        向CacheBuffer<T>的buffer_填值. 再执行notifier_->Notify(channel_id), 是执行键值
        查找notifier_->notifies_map_, 如果找到, 执行值的每个元素(值是一个vector)的
        callback.
    }
}

std::vector<VisitorConfig> InitConfigs(int num) {
    std::vector<VisitorConfig> configs;
    configs.reserve(num);
    for (int i = 0; i < num; ++i) {
        uint64_t channel_id = str_hash("/channel" + std::to_string(i));
        uint32_t queue_size = 10;
        configs.emplace_back(channel_id, queue_size);
    }
    return configs;
}
// 一个VisitorConfig vector

TEST(DataVisitorTest, one_channel) {
    // 测试开始
    auto channel0 = str_hash("/channel");
    // channel0类型是uint64_t
    auto dv = std::make_shared<DataVisitor<RawMessage>>(channel0, 10);
    // dv是一个智能指针, 上述构造函数还操作了单例:
    DataDispatcher<RawMessage>::Instance()
    /*
    记b1(CacheBuffer)={
        T=shared_ptr<RawMessage>, head_=0, tail_=0, fusion_callback_=nullptr,
        capacity_=11(10+1), buffer_.size=10(capacity_)
    }
    dv->{
        T=RawMessage,
        buffer_(它是一个ChannelBuffer)={
            T=RawMessage,
            channel_id_=channel0,
            buffer_(它是一个CacheBuffer)->b1
        },
        notifier_(智能指针, 在data_visitor_base.h中, 被派生而来, 在基类的构造函数中被
        创建)->{

```

```

        callback=nullptr
    },
    data_notifier_(单例, 在data_visitor_base.h中, 被派生而来)->{
        notifies_map_={
            [channel0->{notifier_,}](键值对),
        }
    },
    next_msg_index_(在data_visitor_base.h中, 被派生而来)=0,
}
DataDispatcher<RawMessage>::Instance()->{
    buffers_map_={
        [channel0->{{->b1(一个智能指针指向b1)},}](键值对),
    }
}
*/
DispatchMessage(channel0, 1);
// 更新了DataDispatcher<RawMessage>::Instance(), 其实也就是b1填了一个值
/*
b1={
    T=shared_ptr<RawMessage>, head_=0, tail_=1, fusion_callback_=nullptr,
    capacity_=11(10+1), buffer_.size=10(capacity_, buffer_={0, raw_msg, 0,
...})(raw_msg是一个RawMessage的智能指针)
}
DataDispatcher<RawMessage>::Instance()->{
    buffers_map_={
        [channel0->{{->b1(一个智能指针指向b1)},}](键值对),
    }
}
*/
std::shared_ptr<RawMessage> msg;
// 创建一个智能指针
EXPECT_TRUE(dv->TryFetch(msg));
/* dv->buffer_.buffer_->不为空, 所以是true, 执行后:
dv->{
    T=RawMessage,
    buffer_(它是一个ChannelBuffer)={
        T=RawMessage,
        channel_id_=channel0,
        buffer_(它是一个CacheBuffer)->b1
    }
    data_notifier_(单例, 在data_visitor_base.h中, 被派生而来)->{
        notifies_map_={
            [channel0->{{callback=nullpr},}](键值对),
        }
    }
    next_msg_index_=2
}
msg=raw_msg
*/
EXPECT_FALSE(dv->TryFetch(msg));
/* 虽然dv->buffer_.buffer_->不为空, 但next_msg_index_越界了
(next_msg_index_==dv->buffer_.buffer_.tail_+1==3), 在Fetch()中判断为false, 所以返回false
    不改变任何对象的值
*/

```

```

    */
    DispatchMessage(channel0, 10);
    // 更新了DataDispatcher<RawMessage>::Instance(), 其实也就是b1填值
    /*
    b1={
        T=shared_ptr<RawMessage>, head_=1, tail_=11,
        fusion_callback_=nullptr, capacity_=11(10+1), buffer_.size=10(capacity_,
        buffer_={raw_msg, raw_msg, raw_msg1, ...})(raw_msg是一个RawMessage的智能指针)
    }
    DataDispatcher<RawMessage>::Instance()->{
        buffers_map_={
            [channel0->{{->b1(一个智能指针指向b1)},,}](键值对),
        }
    }
    */
    for (int i = 0; i < 10; ++i) {
        EXPECT_TRUE(dv->TryFetch(msg));
        // 关注next_msg_index_, 它如果越界, 就返回false, 因为上边b1又重新填了10个值,
        // 所以这里可以取10个值
    }
    EXPECT_FALSE(dv->TryFetch(msg));
    // 同样地, 第12次取失败了
}
// 下边的分析和上边的极其类似
TEST(DataVisitorTest, two_channel) {
    auto dv =
        std::make_shared<DataVisitor<RawMessage, RawMessage>>
        (InitConfigs(2));

    std::shared_ptr<RawMessage> msg0;
    std::shared_ptr<RawMessage> msg1;
    DispatchMessage(channel0, 1);
    EXPECT_FALSE(dv->TryFetch(msg0, msg1));
    DispatchMessage(channel1, 1);
    EXPECT_FALSE(dv->TryFetch(msg0, msg1));
    DispatchMessage(channel0, 1);
    EXPECT_TRUE(dv->TryFetch(msg0, msg1));
    DispatchMessage(channel0, 10);
    for (int i = 0; i < 10; ++i) {
        EXPECT_TRUE(dv->TryFetch(msg0, msg1));
    }
    EXPECT_FALSE(dv->TryFetch(msg0, msg1));
}

TEST(DataVisitorTest, three_channel) {
    auto dv = std::make_shared<DataVisitor<RawMessage, RawMessage,
    RawMessage>>(
        InitConfigs(3));

    std::shared_ptr<RawMessage> msg0;
    std::shared_ptr<RawMessage> msg1;
    std::shared_ptr<RawMessage> msg2;
    DispatchMessage(channel0, 1);
    EXPECT_FALSE(dv->TryFetch(msg0, msg1, msg2));
}

```

```

DispatchMessage(channel1, 1);
EXPECT_FALSE(dv->TryFetch(msg0, msg1, msg2));
DispatchMessage(channel2, 1);
EXPECT_FALSE(dv->TryFetch(msg0, msg1, msg2));
DispatchMessage(channel0, 1);
EXPECT_TRUE(dv->TryFetch(msg0, msg1, msg2));
DispatchMessage(channel0, 10);
for (int i = 0; i < 10; ++i) {
    EXPECT_TRUE(dv->TryFetch(msg0, msg1, msg2));
}
EXPECT_FALSE(dv->TryFetch(msg0, msg1, msg2));
}

TEST(DataVisitorTest, four_channel) {
    auto dv = std::make_shared<
        DataVisitor<RawMessage, RawMessage, RawMessage, RawMessage>>(
            InitConfigs(4));

    std::shared_ptr<RawMessage> msg0;
    std::shared_ptr<RawMessage> msg1;
    std::shared_ptr<RawMessage> msg2;
    std::shared_ptr<RawMessage> msg3;
    DispatchMessage(channel0, 1);
    EXPECT_FALSE(dv->TryFetch(msg0, msg1, msg2, msg3));
    DispatchMessage(channel1, 1);
    EXPECT_FALSE(dv->TryFetch(msg0, msg1, msg2, msg3));
    DispatchMessage(channel2, 1);
    EXPECT_FALSE(dv->TryFetch(msg0, msg1, msg2, msg3));
    DispatchMessage(channel3, 1);
    EXPECT_FALSE(dv->TryFetch(msg0, msg1, msg2, msg3));
    DispatchMessage(channel0, 1);
    EXPECT_TRUE(dv->TryFetch(msg0, msg1, msg2, msg3));
    DispatchMessage(channel0, 10);
    for (int i = 0; i < 10; ++i) {
        EXPECT_TRUE(dv->TryFetch(msg0, msg1, msg2, msg3));
    }
    EXPECT_FALSE(dv->TryFetch(msg0, msg1, msg2, msg3));
}

} // namespace data
} // namespace cyber
} // namespace apollo

```

asd

## ./03-apollo/学习apollo之cyber-event1.md

### x1.perf\_event.h

- 链接库
- 详细方法参考: 学习apollo之cyber-event1.html
- 不论是基类还是派生类,都维护了简单的成员变量/读写方法,似乎维护着一些标志位

#### ①.类之中成员变量直接赋值: `int cr_state_ = 1;`

- 有这种写法,下局一个例子:

```
#include<iostream>
#include<string>
class AC{
public:
    std::string a_public_string="a_public_string";
    std::string a_public_string_by_constructor="a_public_string";
    AC(std::string apsb): a_public_string_by_constructor(apsb){}
    void printAllString(){
        std::cout<<"a_public_string: "<<a_public_string<<std::endl;
        std::cout<<"a_public_string_by_constructor: "
<<a_public_string_by_constructor<<std::endl;
        std::cout<<"a_private_string: "<<a_private_string<<std::endl;
    }
private:
    std::string a_private_string="a_private_string";
};
int main(void)
{
    AC ac("modified by constructor");
    ac.printAllString();
    // 输出:
    // a_public_string: a_public_string
    // a_public_string_by_constructor: modified by constructor
    // rivate_string: a_private_string
}
```

- 用C11之前的也会输出正确结果,但编译会有warning.
- 可以看出变量声名出赋值`std::string a_public_string="a_public_string";`会被构造器赋值覆盖掉.

#### ②.common::GlobalData::GetTaskNameById(cr\_id\_)

- 调用的是global\_data.h&global\_data.cc中的方法, 其中维护了一个  
task\_id\_map\_(AtomicHashMap<uint64\_t, std::string, 256>)(global\_data.cc中的一个全局变量),简单理

解是一个uint64\_t-std::string的键值对, 通过键cr\_id找值(std::string).

### ③.common::GlobalData::GetChannelById(channel\_id\_)

- 调用的是global\_data.h&global\_data.cc中的方法, 其中维护了一个channel\_id\_map\_(AtomicHashMap<uint64\_t, std::string, 256>)(global\_data.cc中的一个全局变量), 简单理解是一个uint64\_t-std::string的键值对, 通过键channel\_id找值(std::string).

### ④.uint64\_t channel\_id\_ = std::numeric\_limits<uint64\_t>::max()

- 方法:std::numeric\_limits<uint64\_t>::max()在<limit>中, 提供基本算数类型的极值信息, <https://blog.csdn.net/fengbingchun/article/details/77922558>

## ./03-apollo/学习apollo之cyber-event2.md

### x2.perf\_event\_cache.h&.cc

- 链接库
- 详细方法参考: 学习apollo之cyber-event2.html
- 维护了一个类, 该类在读取了一些配置量后操作Event\_queue\_?

### ①.类之中成员变量直接赋值: PerfEventCache::PerfEventCache()

- .cc文件中: 为什么在头文件中没有声明, 这里却可以直接定义?
- global\_conf = GlobalData::Instance()->Config()在global\_data.h&cc中, 返回的是一个proto(perf\_conf.proto)对象, 下代码global\_data.h&cc中是读取proto(config\_)的对应的代码, 默认读取路径是\$WorkRoot/conf/cyber.pb.conf,

- ```
auto config_path = GetAbsolutePath(WorkRoot(), "conf/cyber.pb.conf");
if (!GetProtoFromFile(config_path, &config_)) {...}
```

- 奇特的是如果打开对应的proto文件, 内容是:

- ```
# transport_conf {
#   shm_conf {
#       # "multicast" "condition"
#       notifier_type: "condition"
#       # "posix" "xsi"
#       shm_type: "xsi"
#       shm_locator {
#           ip: "239.255.0.100"
#           port: 8888
#       }
#   }
#   participant_attr {
#       lease_duration: 12
```

```
#      announcement_period: 3
#      domain_id_gain: 200
#      port_base: 10000
#    }
#    communication_mode {
#      same_proc: INTRA
#      diff_proc: SHM
#      diff_host: RTPS
#    }
#    resource_limit {
#      max_history_depth: 1000
#    }
# }
run_mode_conf {
  run_mode: MODE_REALITY
  clock_mode: MODE_CYBER
}
scheduler_conf {
  routine_num: 100
  default_proc_num: 16
}
```

- 并没有`if (global_conf.has_perf_conf())`中需要的`perf_conf`字段, 难道它加载的是别的文件?

## ②flush()

- 参考<https://blog.csdn.net/caoshangpa/article/details/78920823>
- 文件的写过程: 缓冲区->存储, `flush()`的过程是把数据从缓冲区写到存储, 按`kFlushSize`调用`flush()`的好处是降低 读写的CPU占用?

## ./03-apollo/安装apollo-2020年12月16日.md

安装apollo-2020年12月16日

### 1 github上下代码

<https://github.com/ApolloAuto/apollo>

下述步骤在文件夹demo\_guide中有：

### 2 dev\_start.sh

进入目录 apollo，执行： `bash docker/scripts/dev_start.sh`

它在下载docker的镜像，有点慢。（太慢了，2020年12月16日）

执行完毕之后会产生如下docker镜像：

REPOSITORY	TAG	SIZE
IMAGE ID	CREATED	



apolloauto/apollo	dev-x86_64-18.04-20201210_1400	
cf71b0496db2	6 days ago	8.95GB
apolloauto/apollo	map_volume-sunnyvale_big_loop-latest	
e7b1a71d5b9d	4 weeks ago	440MB
apolloauto/apollo	yolov4_volume-emergency_detection_model-x86_64-latest	
e3e249ea7a8a	2 months ago	264MB
apolloauto/apollo	faster_rcnn_volume-traffic_light_detection_model-	
x86_64-latest	58537bb25841	3 months ago 170MB
apolloauto/apollo	data_volume-audio_model-x86_64-latest	
17cb2a72a392	3 months ago	194MB
apolloauto/apollo	map_volume-sunnyvale_with_two_offices-latest	
93a347cea6a0	9 months ago	509MB
apolloauto/apollo	map_volume-san_mateo-latest	
48cd73de58ba	14 months ago	202MB
apolloauto/apollo	map_volume-sunnyvale_loop-latest	
36dc0d1c2551	2 years ago	906MB

太多了，不知道都是干啥的。

### 3 dev\_into.sh

再运行

```
bash docker/scripts/dev_into.sh
```

会进入到一个docker里边去。

### 4 bootstrap.sh（又是一条很慢的命令，似乎在下什么东西，似乎是网页端）

安装？

```
bash scripts/bootstrap.sh
```

### 5 demo\_3.5.record

又在下载？


```
python docs/demo_guide/record_helper.py demo_3.5.record
```

### 6 cyber\_recorder

```
cyber_recorder play -f demo_3.5.record --loop
```

## 7 浏览器打开 localhost:8888

然后就可以了

 screenshot from 2020-12-17 11-00-51.png

## ./03-apollo/草稿/template\_1.md

---

data1. `cache_buffer.h`

- CacheBuffer
- 详细方法参考: 学习apollo之cyber-data1.html
- 

①.XXXX

- .c

## ./03-apollo/跟着apollo学习shell-1-2020年12月15日.md

---

## 2020年12月15日

---

文件：docker/scripts/dev\_start.sh, docker/scripts/apollo\_base.sh

2020年12月16日-大致读了一遍-把各个小命令拆分着看了一遍，还可以。

## 脚本里边初始化变量

test1.sh

```
A_VAR="avar"
echo "1"
export ${A_VAR}
echo "2"
export $A_VAR
echo "3"
echo $A_VAR
echo "4"
echo ${A_VAR}
# export 并不会输出内容，不知道为什么，只知道它和echo是不一样的！
```

会输出：

```
1  
2  
3  
avar  
4  
avar
```

## 脚本加函数

```
func1 #同一命令行，第一次运行的话会报错，第二次可以正常运行，说明函数的声明是到命令行的  
环境变量里边去了  
function func1(){  
    echo "in func1"  
}  
func1
```

第一次会输出：

```
func1: command not found  
in func1
```

第二次会输出：

```
in func1  
in func1
```

## 函数加参数

```
function func2(){  
    echo "in func2"  
    echo "$@"  
}  
func2 "im param"
```

会输出：

```
in func2  
im param
```

## 函数调用函数

```
function func1(){  
    echo "in func1"  
}  
function func2(){  
    func1  
    echo "in func2"  
}  
func2
```

会输出：

```
in func1  
in func2
```

## \$()运算

括号里边是执行程序，整句话是把执行程序的返回生成变量，如：

```
echo $(cd / && ls)
```

会输出(\目录下的文件(夹)名称，根据自己的系统来定)：

```
bin boot cdrom core dev etc home initrd.img initrd.img.old lib lib64  
lost+found media mnt opt proc root run sbin snap srv sys tmp usr var  
vmlinuz vmlinuz.old
```

## dirname

字符串操作，语义是输出当前文件的文件夹，如：

```
echo $(dirname "/1/2/3/")
```

会输出：

```
/1/2
```

还有一个明命令叫**basename**：

```
echo $(basename "/1/2/3/")
```

会输出：

```
3
```

## `${BASH_SOURCE[0]}`

会输出当前的脚本名称

如：

```
echo ${BASH_SOURCE[0]} # in test2.sh
```

会输出：

```
test2.sh
```

## `x86_64`

貌似本身就是一个阉割版的命令行

## `uname`

`-a,-m,-s`

## `local`

`local`一般用于局部变量声明，多在在函数内部使用。

- `shell`脚本中定义的变量是`global`的，其作用域从被定义的地方开始，到`shell`结束或被显示删除的地方为止。
- `shell`函数定义的变量默认是`global`的，其作用域从“函数被调用时执行变量定义的地方”开始，到`shell`结束或被显示删除处为止。函数定义的变量可以被显示定义成`local`的，其作用域局限于函数内。但请注意，函数的参数是`local`的。
- 如果同名，`Shell`函数定义的`local`变量会屏蔽脚本定义的`global`变量。

源自:<https://blog.csdn.net/superbfly/article/details/49274889>

如：

```
text="out of func3"
echo $text
function func3()
```

```
{
    local text="in func3" #局部变量
    echo $text
}
func3
echo $text
```

会输出：

```
out of func3
in func3
out of func3
```

```
text="out of func3"
echo $text
function func3()
{
    text="in func3" #局部变量
    echo $text
}
func3
echo $text
```

会输出：

```
out of func3
in func3
in func3
```

## if

文件表达式

- -e filename 如果 filename存在，则为真
- -d filename 如果 filename为目录，则为真
- -f filename 如果 filename为常规文件，则为真
- -L filename 如果 filename为符号链接，则为真
- -r filename 如果 filename可读，则为真
- -w filename 如果 filename可写，则为真
- -x filename 如果 filename可执行，则为真
- -s filename 如果文件长度不为0，则为真
- -h filename 如果文件是软链接，则为真
- filename1 -nt filename2 如果 filename1比 filename2新，则为真。
- filename1 -ot filename2 如果 filename1比 filename2旧，则为真。

### 整数变量表达式

- -q 等于
- -e 不等于
- -t 大于
- -e 大于等于
- -t 小于
- -e 小于等于

### 字符串变量表达式

- [ \$a = \$b ] 如果string1等于string2，则为真 字符串允许使用赋值号做等号
- [ \$string1 != \$string2 ] 如果string1不等于string2，则为真
- [ -n \$string ] 如果string 非空(非0) ，返回0(true)
- [ -z \$string ] 如果string 为空，则为真
- [ \$string ] 如果string 非空，返回0 (和-n类似)

### 逻辑非 ! 条件表达式的相反

- [ ! 表达式 ]
- [ ! -d \$num ] 如果不存在目录\$num

### 辑与 -a 条件表达式的并列

- [ 表达式1 -a 表达式2 ]

### 逻辑或 -o 条件表达式的或

- [ 表达式1 -o 表达式2 ]

源自:<https://www.cnblogs.com/smallredness/p/11054882.html>

如：

```
if [ -e "test2.sh" ]; then #存在test2.sh文件 另外，可以看出if后跟的条件语句是用中括号括起来的
    echo "test2.sh exists!"
fi
```

会输出：

```
test2.sh exists!
```

## function可以用return提前返回

### exit

表示退出命令行，

可后加code，如：

```
exit 1
```

vscode 会报错（似乎命令行不会）：

```
The terminal process "/bin/bash" terminated with exit code: 1.
```

## cat

文件内容写入，具体参见:<https://www.runoob.com/linux/linux-comm-cat.html>

## tee

还有这奇葩命令？？

具体参见:<https://www.runoob.com/linux/linux-comm-tee.html>。如：

```
echo "biubiubiu" | tee -a test.txt
```

屏幕输出biubiubiu同时把biubiubiu增量写入到test.txt

## read

屏幕输入，如：

```
read RP # 命令行输入 biubiubiu  
echo $RP # 命令行会输出biubiubiu
```

参考: <https://www.runoob.com/linux/linux-comm-read.html>

大误：为什么会有个-r？

## echo 1 >> 2.txt

把1作为内容输入到2.txt中去，另一个类似的命令是：

```
cat 1.txt >> 2.txt #把1.txt的内容输入到2.txt中去。
```

==~

正则匹配（相对的是==），如：



```
AVAR=1
[[ $AVAR =~ ^[0-9]+$ ]] && echo "is a number." || echo "not a numer!"
AVAR=b
[[ $AVAR =~ ^[0-9]+$ ]] && echo "is a number." || echo "not a numer!"
```

会输出：

```
is a number.
not a numer!
```

## EOF

换行用的，参考：<https://blog.csdn.net/zongshi1992/article/details/71693045>

## for

摘自：<https://www.cnblogs.com/EasonJim/p/8315939.html>

### 1

```
for((i=1;i<=4;i++));
do
echo $(expr $i \* 3 + 1); # i*3+1
done
```

会输出：

```
4
7
10
13
```

### 2

```
for i in $(seq 1 4)
do
echo $(expr $i \* 3 + 1); # i*3+1
done
```

会输出：

```
4
7
10
13
```

**3**

```
for i in {1..4}
do
echo $(expr $i \* 3 + 1); # i*3+1
done
```

会输出：

```
4
7
10
13
```

**4**

```
for i in `ls`; #执行ls命令
do
echo $i is file name\! ; #输出当前文件夹下可以ls出的文件
done
```

**5**

```
for i in $* ; #放到脚本里执行，后接参数，如biubiubiu,
do
echo $i is input chart\! ; #会输出biubiubiu is input chart!
done
```

会输出：

```
4
7
10
13
```

**6**

```
for i in f1 f2 f3 ;
do
echo $i is appoint ;
done
```

会输出：

```
f1 is appoint
f2 is appoint
f3 is appoint
```

**7**

```
list="rootfs usr data data2" #空格可以作为分隔符
for i in $list;
do
echo $i is appoint ;
done
```

会输出：

```
rootfs is appoint
usr is appoint
data is appoint
data2 is appoint
```

## while

**1**

```
i=1
sum=0
while [ $i -le 10 ] # -le小于等于
do
    let sum=sum+i #等价于let sum=sum+$i； 等价于let sum=$sum+$i； 不等价于let
$sum=$sum+$i；
    #另外若果删除let sum=$sum+$i则会变成字符串操作，你可以试一下
    let i++
done
echo $sum
```

会输出：

```
55
```

2

```
i=1
j=1
while [ $i -le 5 ]
do
    while [ $j -le 5 ]
    do
        echo -n "*"
        let j++
    done
    echo
    let i++
    let j=1
done
```

会输出：

```
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
```

摘自：<https://blog.csdn.net/wdz306ling/article/details/79602739>

和while类似的一个关键词是until，它正好是和while反着来的

## -eq -ne -gt -lt -ge -le

- -eq //等于
- -ne //不等于
- -gt //大于 (greater)
- -lt //小于 (less)
- -ge //大于等于
- -le //小于等于

命令的逻辑关系：

- 在linux 中 命令执行状态：0 为真，其他为假
- 逻辑与：&&

- 第一个条件为假时，第二条件不用再判断，最终结果已经有；
- 第一个条件为真时，第二条件必须得判断；
- 逻辑或：||
- 逻辑非：!

摘自：<https://www.cnblogs.com/jxhd1/p/6274854.html>

## shift

```
# 脚本输入，后接参数： 1 2 3 4
while [ $# -ne 0 ] # $#参数个数 $number 第number个参数
do
echo "第一个参数为：$1 参数个数为：$#"
shift #参数列表左移
done
```

会输出：

```
第一个参数为：1 参数个数为：4
第一个参数为：2 参数个数为：3
第一个参数为：3 参数个数为：2
第一个参数为：4 参数个数为：1
```

## case

```
#判断用户输入

#在屏幕上输出"请选择yes/no"，然后把用户选择赋予变量cho
read -p "Please choose yes/no: " -t 30 cho
case $cho in
#判断变量cho的值
    "yes")
    #如果是yes
        echo "Your choose is yes!"
        #则执行程序1
        ;;
    "no")
    #如果是no
        echo "Your choose is no!"
        #则执行程序2
        ;;
    *)
    #如果既不是yes, 也不是no
        echo "Your choose is error!"
        #则执行此程序
        ;;
esac
```

摘自：<http://c.biancheng.net/view/1003.html>

## eval

解析并执行字符串中的命令

```
name=woodie
cmd="echo Helllo $name\! "
eval $cmd # 会输出：Hello woodie!
```

参考：<https://www.cnblogs.com/triple-y/p/11236082.html>

## nohub

nohub 英文全称 no hang up（不挂起），用于在系统后台不挂断地运行命令，退出终端不会影响程序的运行。

nohub 命令，在默认情况下（非重定向时），会输出一个名叫 nohub.out 的文件到当前目录下，如果当前目录的 nohub.out 文件不可写，输出重定向到 \$HOME/nohup.out 文件中。

参考：<https://www.runoob.com/linux/linux-comm-nohup.html>

## ./0x-template.md

---

### 代码总览

xxxxxx

### 功能/知识

1.

- 
- 

2.

- 
-