### Explicit表明构造函数是显示的，防止构造函数隐式自动转换，构造函数后面的冒号是初始化表达式，函数后面加override表示将重写这个函数

class C\_tick\_wakeup : public EventCallback {

AsyncConnectionRef conn;

public:

explicit C\_tick\_wakeup(AsyncConnectionRef c): conn(c) {}

void do\_request(uint64\_t fd\_or\_id) override {

conn->tick(fd\_or\_id);

}

};

### 多重继承，但后面public Counter<CDentry>是什么鬼？模板类？

class CDentry : public MDSCacheObject, public LRUObject, public Counter<CDentry> {

class CInode : public MDSCacheObject, public InodeStoreBase, public Counter<CInode> {

### 引用的本质在C++中就是指针常量（常指针）

void Server::handle\_client\_getattr(MDRequestRef& mdr, bool is\_lookup)

const cref\_t<MClientRequest> &req = mdr->client\_request;

em.first->second = std::make\_unique<Batch\_Getattr\_Lookup>(this, mdr);

### pair是将2个数据组合成一个数据

map<pair<uint64\_t,string>,uint64\_t>

for (map<**int**,string>::iterator it=mapA.begin(); it!=mapA.end(); ++it)

### std::optional

**std::optional 是在 C++ 17 中引入到标准库中的,C++ 17 之前的版本可以通过 boost::optional 实现几乎相同的功能。**

objecter->get\_fs\_stats(stats, std::optional<int64\_t>(), &cond);

作用是什么，不太清楚

### 命名空间、类和函数

**Class::Method 是类 Class 中的方法 Method 的意思，常用于在 class 块外定义方法。**

**Class::Class 是类 Class 的**[**构造函数**](https://so.csdn.net/so/search?q=%E6%9E%84%E9%80%A0%E5%87%BD%E6%95%B0&spm=1001.2101.3001.7020)**的意思。**

**Handle是CephFuse中的类，下面是Handle类的定义**

class CephFuse::Handle { //这是什么写法

### Boost

Boost**是一个功能强大、构造精巧、跨平台、开源并且完全免费的**C++**程序库。**

typedef boost::intrusive\_ptr<MDRequestImpl> MDRequestRef;

### std:: 是个名称空间标示符

std**:: 是个名称空间标示符，**C++**标准库中的函数或者对象都是在命名空间**std**中定义的**

inline std::ostream& operator<<(std::ostream &out, const MutationImpl &mut)

std::set<std::string> str\_keys;

std::map<dirfrag\_t, dirlump> lump\_map;

### std::move

**std::move作用主要可以将一个左值转换成右值引用，从而可以调用C++11右值引用的拷贝构造函数 不是很明白**

session->con->send\_message2(std::move(r));

什么是移动构造函数？什么是移动赋值函数？

### auto

**auto在这里的作用是什么？**

auto r = build\_client\_request(request);

### 接口类

**接口描述了类的行为和功能,而不需要完成类的特定实现。接口类只提供方法的声明,而自身不提供方法定义的抽象类。接口类自身是不能实例化,只能由接口类的子类来完成。**

class MDSRankDispatcher : public MDSRank MDSRankDispatcher 是 MDSRank 与 MDSDaemon 的接口类

intrusive\_ptr是一种引用计数型智能指针

boost::intrusive\_ptr<CdsContext> global\_init(const std::map<std::string, std::string> \*defaults,

uint32\_t module\_type,

code\_environment\_t code\_env, int flags,

bool run\_pre\_init)

### get\_instance

可以直接指定通过如下方法调用类的成员函数？get\_instance方法将返回一个UmdsMdService对象？

UmdsMdService\* UmdsMdService::instance = NULL;

class UmdsMdService {

static UmdsMdService\* instance;

}

UmdsMdService::get\_instance()->get\_meta\_obj\_handle(request, true, con\_handle, obj\_handle);

### encode编码

如何理解如下代码？

class UmdPlacementInfo{

void encode(bufferlist& bl) const {

ENCODE\_START(1, 1, bl);

encode(name, bl);

encode(meta\_pool, bl);

encode(rep\_num, bl);

encode(create\_time, bl);

encode(data\_pools, bl);

encode(data\_pools\_desp, bl);

encode(description, bl);

encode(meta\_redundant, bl);

encode(meta\_pool\_container, bl);

encode(storage\_name\_id, bl);

ENCODE\_FINISH(bl);

}

}

nullptr为关键字，表示空指针，而NULL只是一个宏定义

### C++的标准库

图示

描述已自动生成

C++的标准库分类

1. C库
2. 标准的C++库：包括C++ I/O库、String库、数值库等
3. 标准模板库（STL），它包括五大类组件：算法、容器、迭代器、函数对象、适配器。

1.stl是指的标准模板库,是Standard Template Library的简称。在C++标准中，STL被组织为下面的13个头文件：<algorithm>、<deque>、<functional>、<iterator>、<vector>、<list>、<map>、<memory>、<numeric>、<queue>、<set>、<stack>和<utility>。

2.std是命名空间的名字,是Standard的简称,是模板库(包括stl)的设计者,特意在库文件里面加上的命名空间的名字。

标准库连网络框架、json解析都没有，字符串处理停留在原始时代，没有官方的开发范式，c++是到了c++11才在标准库里有了线程，Cpp一直没有包管理机制（c++20才有了Modules的概念）。几个成熟的c/c++的通用库：

boost: boost是试验版，冒险版，激进版的stl；xml及json解析,asio网络模块

folly:c++14的库

glib:Gtk+库和Gnome的基础

ACL:协程模块fiber

C++标准库的内容分为10类：

C1.语言支持 C2.输入/输出 C3.诊断 C4.一般工具 C5.字符串

C6.容器 C7.迭代器支持 C8.算法 C9.数值操作 C10.本地化

|  |  |  |
| --- | --- | --- |
|  |  |  |
| c1 | cstddef |  |
| c2 | limits |  |
| c3 |  |  |
| c3 |  |  |
| c5 |  |  |
| c6 |  |  |
| c7 |  |  |
| c8 |  |  |
| c9 |  |  |
| c10 |  |  |

std::chrono是C++11引入的日期时间处理库。

std::thread是 C++ 11 新引入的标准线程库。

### 智能指针

boost::intrusive\_ptr

std::unique\_ptr

std::shared\_ptr

shared\_ptr<NSInfo> ns\_info = nullptr;

typedef boost::intrusive\_ptr<UmdsOpContext> UmdSrvRequestRef;

### 匿名函数

如何理解： 你可以认为就是个匿名函数（金鑫）

auto write\_one = [&](CDentry \*dn) {

…

}

[]：定义匿名函数

[&]：以引用形式捕获所有外部变量，也就是外部变量均可用

|  |
| --- |
| auto toNumber = [&](string const& s) -> unsigned {  ...  }  []：定义匿名函数  [&]：以引用形式捕获所有外部变量，也就是外部变量均可用  (string const &s) ：匿名函数的参数  ->：定义匿名函数  unsigned：函数返回值类型  {...}：函数实现体 |

函数比较轻便，即用即删除，很适合需要完成一项功能，但是此功能只在此一处使用

这个专业名字叫Lambda函数

### 函数冲突

函数定义冲突解决方案

#undef max

### C++中包含c的头文件

觉得很奇怪

#if defined(\_\_cplusplus)

extern "C" {

#endif

#if defined(\_\_cplusplus)

}

#endif

奇技淫巧怎能不提宏

### 好的c++网站

图形用户界面, 应用程序, Word

描述已自动生成

### pair

pair c++ 98就支持了

pair<string,int> p; pair作为函数的返回值，类似c语言的struct 返回？

<int ,pair<int ,int > > （后边的两个 > > 要有空格，否则就会是 >>  位移运算符）

pair <string ,double> p3 = make\_pair("aa",5.0);

每个pair 都有两个属性值  first  和second

 cout<<p1.first<<p1.second;

typedef pair<string, string> author;

std::pair<CDentry \*, CDentry \*>

Server::rdlock\_two\_paths\_xlock\_destdn(MDRequestRef &mdr, bool xlock\_srcdn)

tuple: c++ 11引入

c++支持tuple？

make\_tuple()

std::tuple<int, char> first;

**auto** first = std::make\_tuple (10,'a');

//修改指定的元素

std::get<0>(mytuple) = 100;

**Python**等语言在一个函数中返回多个值

### c++ 里面的map容器的迭代器first、second用法

例：

map<string, int> m\_stlmap;

m\_stlmap[“xiaomi”] = 88;

auto mpit = m\_stlmap.begin();

first会得到Map中key的有效值，

second会得到Map中value的有效值。

所以

mpit ->first; // 得到是 string 值是 “xiaomi”

mpit ->second; //得到是 int 值是 88

### RAII

RAII（**R**esource **A**cquisition **I**s **I**nitialization）是由c++之父Bjarne Stroustrup（本贾尼·斯特劳斯特卢普）提出的，也称为“资源获取就是初始化”。

RAII是[c++](https://baike.baidu.com/item/c%2B%2B/99272?fromModule=lemma_inlink)等编程语言常用的管理资源、避免内存泄露的方法。它保证在任何情况下，使用对象时先构造对象，最后析构对象。

ScopeGuard 是 RAII 的一个例子，C++ 中对象出了作用域会自动调用其释构函数。

```c++

void func() {

// 申请资源(内存，句柄...)

SCOPE\_GUARD{

//释放资源

};

do\_somthing();

}

```

C11特性，实现对象自动释放（RAll）的方式（ScopeGuard ）

|  |
| --- |
| 1. void\* data = malloc(size); 2. ON\_SCOPE\_EXIT { 3. free(data); 4. }; 6. xxxxx 7. if (xxx) { 8. xxxxx 9. } 10. else { 11. xxxxx 12. } |

C++ 实现用到 C++ 11 的 lamda，定义对象将 lamda 存起来，在析构函数中调用。这个 lamda 在不同的场合也有不同的叫法，比如匿名函数，闭包，代码块，block。

auto sg1 = ScopeGuard([&] {

cparentinode->UnLock();

cparentinode->Put();

});

### C++中的NULL和nullptr

C里面的NULL，它是一个宏，本质上是一个(void \*)0

C++使用NULL的问题：函数重载

避免问题，C++中引入了nullptr（关键字）

nullptr 为指针类型，NULL 则是 int 类型

### 类的私有变量

C++变量前面加下划线和不加下划线都不会影响对变量的定义，只是风格问题，更喜欢将成员变量或者私有成员变量的前面加上下划线。

使用保留标识符是未定义行为，比较流行的私有变量命名方式是以下划线结尾，可以较好地避开上面的规则。

类的私有变量应该带有下划线作后缀。（Option）

但ceph并未遵守。

私有函数貌似在前面加了下划线

\_prepare\_db\_environment

### 类中类

类中再定义类？

类中定义结构体，结构体有继承

class BlueStore

struct BufferSpace {

class GarbageCollector

嵌套类

class A{ class B{}; };

### 宏定义

是否必须抵头写？

#ifdef

#endif

### modernize-use-trailing-return-type

Rewrites function signatures to use a trailing return type (introduced in C++11).

重写函数签名以使用尾随返回类型 （在第C++11中引入）

C++11新标准中， 尾置返回类型(trailing return type)跟在形参列表后面并以一个->符号开头。任何函数都可以使用尾置返回，但返回类型比较复杂的函数使用它最有效，例如返回类型是数组的指针或者数组的引用。

使用尾置返回类型(trailing return type)简化函数声明。

`auto Foo() -> std::string { return ...; }`, we think the code is less readable in this form.

解决问题模板函数返回值不确定，或者写法麻烦的问题。trailing-return-type，又称跟踪返回类型：泛型编程运行期返回值确认。

### decltype

decltype被称作类型说明符，它的作用是选择并返回操作数的数据类型。

decltype，在C++中，作为操作符，用于查询表达式的数据类型。decltype在C++11标准制定时引入，主要是为泛型编程而设计，以解决泛型编程中，由于有些类型由模板参数决定，而难以（甚至不可能）表示之的问题。

“声明类型”（Declared Type）

C++引入模板，以及由标准模板库引领的泛型编程逐渐兴起

主要对编写模板库的开发者有用，从2008年3月5日发布的4.3版开始，[GCC](https://baike.baidu.com/item/GCC?fromModule=lemma_inlink)C++编译器也加入了decltype操作符。

### 智能指针 smart pointer







图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

make\_unique 同 unique\_ptr 、auto\_ptr 等一样，都是 smart pointer，可以取代new 并且无需 delete pointer

**条款21：尽量使用std::make\_unique和std::make\_shared而不直接使用new**

std::make\_shared是c++11的一部分，但很可惜std::make\_unique不是。它是在c++14里加入标准库的。

make函数用来把一个任意参数的集合完美转移给一个构造函数从而生成动态分配内存的对象，并返回一个指向那个对象的灵巧指针。

std::make\_unique

std::make\_shared

std::allocate\_shared

指针永远是C++最难的地方之一，所以引入了智能指针来更自动地管理内存

文本

描述已自动生成

boost::Intrusive\_ptr 半自动指针？

### 屏蔽lint

// NOLINTBEGIN

...

// NOLINTEND

### 强制转换

static\_cast和dynamic\_cast：

static\_cast <type\_name> (expression) 将expression类型强制转换为type\_name类型

dynamic\_cast <type\_name> (expression) 使得能够在类层次中进行向上转换

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

在派生关系中, 基类的指针可以直接指向一个派生类的对象

auto serverThreadFunc(void \*arg) -> void \*

{

uint32\_t server\_port = \*static\_cast<uint32\_t \*>(arg);

const\_cast<char \*>(conf\_path)

### C++ RTTI机制

Run-Time Type Identification 运行时类型识别

这个机制是为了让程序在运行时能根据基类的指针或引用来获得该指针或引用所指的对象的实际类型。

C++是一门静态类型语言，其数据类型是在编译期就确定的，不能在运行时更改。然而由于面向对象程序设计中多态性的要求，C++中的指针或引用本身的类型，可能与它实际代表的类型并不一致，有时我们需要将一个多态指针转换为其实际指向对象的类型，就需要知道运行时的类型信息，这就有了运行时类型识别需求。

Java中任何一个类都可以通过反射机制来获取类的基本信息（接口、父类、方法、属性、Annotation等），而且Java中还提供了一个关键字，可以在运行时判断一个类是不是另一个类的子类或者是该类的对象，Java可以生成字节码文件，再由JVM（Java虚拟机）加载运行，字节码文件中可以含有类的信息。

C++通过以下两个关键字提供RTTI功能：

1. typeid：该运算符返回其表达式或类型名的实际类型
2. dynamic\_cast：该运算符将基类的指针或引用安全地转换为派生类类型的指针或引用

### c++ Explicit 关键字

1. 将构造函数标记为显式

### unique\_lock

unique\_lock<std::mutex> lock(g\_mtx);

### 深拷贝和浅拷贝

定义

深拷贝是将一个对象完全复制到另一个对象中，包括所有成员变量和动态分配的内存。

浅拷贝只是简单地复制指针，这样两个对象会共享同一块内存，可能会导致不可预知的错误。

实现

深/浅拷贝的实现在复制构造函数中

MyClass(const MyClass& other)

深/浅拷贝的实现在赋值运算符重载函数

MyClass& operator=(const MyClass& other)

应用

深拷贝通常用于需要复制动态分配内存的对象，例如字符串类、容器类等。

浅拷贝通常用于不需要复制动态分配内存的对象，例如基本数据类型、指针等。

从测试来看：

结构体包含字符数组或string，结构体赋值采用类似深拷贝。

结构体包含指针，结构体赋值采用浅拷贝。

### 未分类

<<标准C++库大全>>,

UmdCliRequestRef request = init\_request<crt\_md\_mk\_in, crt\_md\_mk\_out>(UMD\_OP\_CREATE\_MD, callback, args);

这是什么写法，shit

auto op = registry.create\_operation<T>(std::forward<Args>(args)...);

auto fut = op->start().then([op /\* by copy \*/] {

set(CMAKE\_CXX\_STANDARD 17)

C++纷繁复杂，说一二，做三四。

如何理解

class MDSContext : public Context

{

public:

template<template<typename> class A>

using vec\_alloc = std::vector<MDSContext\*, A<MDSContext\*>>;

using vec = vec\_alloc<std::allocator>;

template<template<typename> class A>

using que\_alloc = std::deque<MDSContext\*, A<MDSContext\*>>;

using que = que\_alloc<std::allocator>;

void complete(int r) override;

virtual MDSRank \*get\_mds() = 0;

};

#### emplace

emplace操作是从C++11开始引入新特性

emplace比insert少一次拷贝构造

#### 发送消息的回调函数不在类中