CPP_BASIC

jask

2024-08-11

重读 《C++ 程序设计语言 》

注：由于是重读，这里大多是一些拾遗。

采用格式：章节 + 页码 + 内容

41 章并发

P280

“还请注意，只要你不向其他线程传递局部数据的指针，你的局部数据就不存在这里讨论的诸多问题。”

语言标准

concept

例子：

```
template <typename Lock>
concept is_lockable=requires(Lock &&lock){
  lock.lock();
  lock.unlock();
  {lock.try_lock()}->std::convertible_to<bool>;
};
```

is_lockable 概念可以用来限制模板，使得只有那些具备 lock()、unlock() 和 try_lock() 成员函数，并且 try_lock() 返回类型可以转换为 bool 的类型才可以作为模板参数传递。这个概念通常用于多线程编程中，以确保模板只接受那些具有锁定功能的类型，比如 std::mutex 或 std::recursive_mutex。

限定类型。

std::ref

std::ref 相当于告诉编译器这里要使用引用而不是按值传递，比如 boost::asio::context 就需要按引用。

std::ref 的作用是显式地告诉编译器传递引用，而不是尝试通过值传递。

asio::io_context 的拷贝构造函数被删除，不能被拷贝，只能通过引用或指针传递。

通过 std::ref，你成功避免了编译器试图调用已删除的拷贝构造函数。

std::cref

用于生成一个 std::reference_wrapper 对象，类似于 std::ref，但它为常量对象创建引用。

std::cref 允许你创建对 const 类型对象的引用包装器。

这个包装器可以用于那些期望传递对象而实际上需要传递常量引用的场景。

std::ignore

可用来忽略返回值，cppref 解释：

```
1) An object such that any value can be assigned to it with no effect.
2) The type of std::ignore.
```

```cpp
#include <iostream>
#include <set>
#include <string>
#include <tuple>

[[nodiscard]] int dontIgnoreMe()
{
    return 42;
}

int main()
{
    std::ignore = dontIgnoreMe();

    std::set<std::string> set_of_str;
    if (bool inserted{false};
        std::tie(std::ignore, inserted) = set_of_str.insert("Test"),
        inserted)
        std::cout << "Value was inserted successfully.\n";
}
```

std::move_only_function::move_only_function

std::atomic

C++ 标准库中提供的原子类型模板，允许你对某些基本数据类型进行原子操作，确保在多线程环境中对这些变量的操作不会产生竞争条件（race condition）。std::atomic 提供了一系列原子操作，避免了对锁（如 std::mutex）的依赖，从而提高了并发性能。

基本操作

load()：读取变量的值。
store()：存储一个新值。
exchange()：交换变量值并返回旧值。
fetch_add() 和 fetch_sub()：原子加减操作。
compare_exchange_strong() 和 compare_exchange_weak()：比较并交换值。

> compare_exchange_strong 和 compare_exchange_weak 用于原子的比较并交换操作。它们的用途是当且仅当原子的当前值与给定的预期值相等时，才更新为新值。否则，不做更新。

内存顺序

std::atomic 操作可以指定内存顺序，以控制操作的内存可见性。这些顺序包括：
std::memory_order_relaxed：无同步或顺序保证。
std::memory_order_acquire：确保此操作之前的所有读操作都不会被重排序。
std::memory_order_release：确保此操作之后的所有写操作都不会被重排序。
std::memory_order_acq_rel：结合了 acquire 和 release 的保证。
std::memory_order_seq_cst：最强的内存顺序，提供全局顺序保证。

std::chrono

获取当前时间点：system_clock::now() 返回当前的时间点，类型为 std::chrono::system_clock::time_point。

计算时间差：time_since_epoch() 返回自系统纪元以来到当前时间点的持续时间，类型为 system_clock::duration，表示的时长单位可能是秒、毫秒、微秒等。

时间单位转换：duration_cast(d) 将时间差 d 转换为以微秒为单位的持续时间。duration_cast 是一个模板函数，用于将时间持续时间转换为另一种时间单位。

获取微秒数：.count() 返回以 microseconds 表示的数值。

std::recursive_mutex

允许同一线程多次对互斥量加锁而不会导致死锁。它的主要用途是在递归函数或涉及到多次调用同一资源锁定的场景中。

典型用途：

递归函数中需要加锁的场景：如果一个递归函数需要对某个共享资源进行加锁，而递归调用过程中又需要再次加锁，这时使用普通的 std::mutex 会导致死锁，而使用 std::recursive_mutex 则可以避免这种情况。

对象方法互相调用的场景：如果类的多个方法之间相互调用，并且这些方法都需要对某些共享资源进行加锁，使用 std::recursive_mutex 可以避免同一线程在调用链中多次锁定同一互斥量时出现死锁。

std::chrono::high_resolution_clock

std::chrono::high_resolution_clock 是 C++ 标准库中的一个时钟类型，定义在头文件中。它用于测量时间点，并提供高分辨率的时间精度，通常用于精确的时间测量，比如计算代码执行时间、性能分析等。

特点：

高分辨率：它通常提供比 system_clock 更高的精度，可以用于测量短时间间隔，如纳秒或微秒级的时间。

非系统时钟：high_resolution_clock 不是一个与系统时间同步的时钟，因此它通常不用于显示当前日期和时间，而是用于测量时间间隔。

实现细节：在不同的系统上，high_resolution_clock 可能映射到不同的实现。例如：

在某些实现中，它可能是 steady_clock，这意味着它不会受到系统时间调整的影响（如系统时间更新或闰秒）。

在其他实现中，它可能是 system_clock 的高精度版本。

用途：high_resolution_clock 常用于需要高精度的性能分析和时间测量。例如，测量一段代码执行的时间、计算函数的延迟等。

cxxabi abi::__cxa_demangle()

abi::__cxa_demangle() 是一个 GCC 和 Clang 提供的函数，用于将 C++ 编译器修饰过的名称（mangled name）还原为可读的、未修饰的函数名。

如果 abi::__cxa_demangle() 成功，还原后的名称会存储在返回的指针 v 中。

如果解析成功，函数将这个可读的函数名返回；否则返回原始字符串。

一些库

测试

doctest

single-header, 开箱即用

gtest

功能丰富

rpc

json-rpc-cxx

学习 ing, 功能相对简单，依赖简单。

什么是多态?

不针对 C++，而是一种类型论的概念。

Ad-hoc Polymorphism 特设多态

特设多态通过函数重载和运算符重载来实现，同一函数或运算符在不同的类型上有不同的行为。

C++ 中的运算符重载/函数重载，Haskell，都有这个机制。

Parametric Polymorphism 参数多态

参数多态允许函数或数据类型对任何类型的参数进行操作，而不依赖于具体的类型。在参数多态中，类型是参数化的，常用于泛型编程。

经典：C++ 的模板特殊化这样的类型多态（type polymorphism）表面上类似于参数多态并同时引入了特设多态。

Subtype Polymorphism 子类型多态

子类型多态，也称为包含多态，是指对象的某种子类型可以替代父类型使用。这通常涉及继承和接口，允许基于父类的接口调用子类的实现。

有点像里氏替换原则。

Row Polymorphism（行多态）

Row Polymorphism 是一种较为特殊的多态形式，通常用于记录类型（record types）或类似结构中。它允许在具有不完全相同字段的记录类型之间进行操作。不同于参数多态，Row Polymorphism 能够处理部分类型，即它允许记录类型的子集作为参数，而不要求记录类型完全匹配。

Row Polymorphism 允许函数作用于多个具有相同字段的记录类型，即使它们有额外的字段。

语言标准

for co_await

已经被移除标准：

The wording for co_await statement makes assumptions of what future asynchronous generator interface will be. Remove it for now as not to constraint the design space for asynchronous generators.

字符串字面量

对于一个字符串字面量，其类型为 char[]，长度编译期可知。

字符串字面量是静态分配的，因此函数返回字符串字面量是很安全的行为。

C++ 提供了原始字符串，也就是 R’ str’。

指针与 const

constexpr 提供的是（尽可能）编译期求值，指示或确保在编译期求值。

const 提供的是当前作用域内值不发生改变，主要任务是规定接口的不可修改性。

使用 const 会改变一种类型，所谓的改变并不是改变了常量的分配方式，而是限制了他的使用方式。

c++ 允许通过显示类型转换的方式显式地除掉对于指针指向常量的限制。

注意，const 变量的存储是一个 implementation detail.

GCC Compiler 会把 read-only 变量，常数和跳转表放置在.text 段中。

在局部 const 变量的情况下，存储在 stack segment （栈区）的写保护区。

对于全局初始化 const 变量，存储在 data segment 部分。

对于全局未初始化 const 变量，存放在 BSS segment（存放未初始化的全局/静态变量）。

右值引用

C++ 之所以设计了几种不同形式的引用，是为了支持对象的不同用法：

1. 非 const 左值引用所引用的对象可以由用户写入内容。

2.const 左值引用所引用的对象从用户的角度来看是不可修改的。

3. 右值引用对应一个临时对象，用户可以修改这个对象，并且认定这个对象以后不会被用到了。

杂项

forward declaration

为什么 c/c++ 需要我们在调用前声明呢?

比如说：

```
void foo();
void test(){
foo();
}
void foo(){
xxx
}
```

这是历史问题，c/c++ 编译器被设计为 single-pass，当编译器需要链接符号时必须知道这个符号链接的对象是谁。

对于 C#，java 这样的 two pass compiler 来说，就不需要前向声明（但是仍然有两个包互相依赖的问题）