# 04\_weak\_ptr

## 前言

std::weak\_ptr是一个不控制资源生命周期的智能指针，是对对象的一种弱引用，只是提供了对其管理的资源的一个访问手段，引入它的目的为协助std::shared\_ptr工作。

## weak\_ptr的使用

std::weak\_ptr可以从一个std::shared\_ptr或另一个std::weak\_ptr对象构造，std::shared\_ptr可以直接赋值给std::weak\_ptr，也可以通过std::weak\_ptr的lock()函数来获得std::shared\_ptr。它的构造和析构不会引起引用计数的增加或减少。std::weak\_ptr可用来解决std::shared\_ptr相互引用时的死锁问题（即两个std::shared\_ptr相互引用，那么这两个指针的引用计数永远不可能下降为0，资源永远不会释放）。

#include <iostream>

#include <memory>

int main()

{

//创建一个std::shared\_ptr对象

std::shared\_ptr<int> sp1(new int(123));

std::cout << "use count: " << sp1.use\_count() << std::endl;

//通过构造函数得到一个std::weak\_ptr对象

std::weak\_ptr<int> sp2(sp1);

std::cout << "use count: " << sp1.use\_count() << std::endl;

//通过赋值运算符得到一个std::weak\_ptr对象

std::weak\_ptr<int> sp3 = sp1;

std::cout << "use count: " << sp1.use\_count() << std::endl;

//通过一个std::weak\_ptr对象得到另外一个std::weak\_ptr对象

std::weak\_ptr<int> sp4 = sp2;

std::cout << "use count: " << sp1.use\_count() << std::endl;

return 0;

}

运行结果：

use count: 1

use count: 1

use count: 1

use count: 1

## weak\_ptr的特点

无论通过何种方式创建std::weak\_ptr都不会增加资源的引用计数，因此每次输出引用计数的值都是1。

既然，std::weak\_ptr不管理对象的生命周期，那么其引用的对象可能在某个时刻被销毁了，如何得知呢？std::weak\_ptr提供了一个expired()方法来做这一项检测，返回true，说明其引用的资源已经不存在了；返回false，说明该资源仍然存在，这个时候可以使用std::weak\_ptr的lock()方法得到一个std::shared\_ptr对象然后继续操作资源，以下代码演示了该用法：

// tmpConn\_ 是一个 std::weak\_ptr<TcpConnection> 对象

// tmpConn\_ 引用的TcpConnection已经销毁，直接返回

if (tmpConn\_.expired())

return;

std::shared\_ptr<TcpConnection> conn = tmpConn\_.lock();

if (conn)

{

//对conn进行操作，省略...

}

既然使用了std::weak\_ptr的expired()方法判断了对象是否存在，为什么不直接使用std::weak\_ptr对象对引用资源进行操作呢？实际上这是行不通的，std::weak\_ptr类没有重写operator->和operator\*方法，因此不能像std::shared\_ptr或std::unique\_ptr一样直接操作对象，同时std::weak\_ptr类也没有重写operator!操作，因此也不能通过std::weak\_ptr对象直接判断其引用的资源是否存在。

因此，std::weak\_ptr的正确使用场景是那些资源如果可用就使用，如果不可使用则不用的场景，它不参与资源的生命周期管理。

std::weak\_ptr的应用场景，经典的例子是订阅者模式或者观察者模式中。这里以订阅者为例来说明，消息发布器只有在某个订阅者存在的情况下才会向其发布消息，而不能管理订阅者的生命周期。

class Subscriber

{

};

class SubscribeManager

{

public:

void publish()

{

for (const auto &iter : m\_subscribers)

{

if (!iter.expired())

{

//TODO：给订阅者发送消息

}

}

}

private:

std::vector<std::weak\_ptr<Subscriber>> m\_subscribers;

};

## std::weak\_ptr常用函数

1. swap()

void swap (weak\_ptr& x)

将当前 weak\_ptr 对象的内容与 x 的内容交换。

1. reset()

void reset()

将当前 weak\_ptr 对象管理的指针和计数器变成空的，就像默认构造的一样。

1. use\_count()

long int use\_count()

返回与当前 weak\_ptr 对象在同一指针上共享所有权的 shared\_ptr 对象的数量。

1. expired()

bool expired()

检查是否过期，返回 weak\_ptr 对象管理的指针为空，或者和他所属共享的没有更多 shared\_ptr。lock 函数一般需要先调用 expired 判断，如果已经过期，就不能通过 weak\_ptr 恢复拥有的 shared\_ptr。此函数应返回与（use\_count() == 0）相同的值，但是它可能以更有效的方式执行此操作。

1. lock()

shared\_ptr<element\_type> lock()

如果它没有过期，则返回一个 shared\_ptr，其中包含由 weak\_ptr 对象保留的信息。如果 weak\_ptr 对象已经过期，则该函数返回一个空的 shared\_ptr（默认构造一样）。因为返回的 shared\_ptr 对象也算作一个所有者，所以这个函数锁定了拥有的指针，防止它被释放（至少在返回的对象没有释放它的情况下）。 此操作以原子方式执行。

一旦智能指针对象接管了你的资源，所有对资源的操作都应该通过智能指针对象进行，不建议再通过原始指针进行操作了。

当然，除了 std::weak\_ptr 之外，std::unique\_ptr 和 std::shared\_ptr 都提供了获取原始指针的方法——get() 函数。

## 总结

以上就是今天要讲的内容，后续会有更多内容。

## 参考资料

版权声明：本文参考了其他资料和CSDN博主的文章，遵循CC 4.0 BY-SA版权协议，现附上原文出处链接及本声明。

1. https://blog.csdn.net/haiguangboy/article/details/122278954