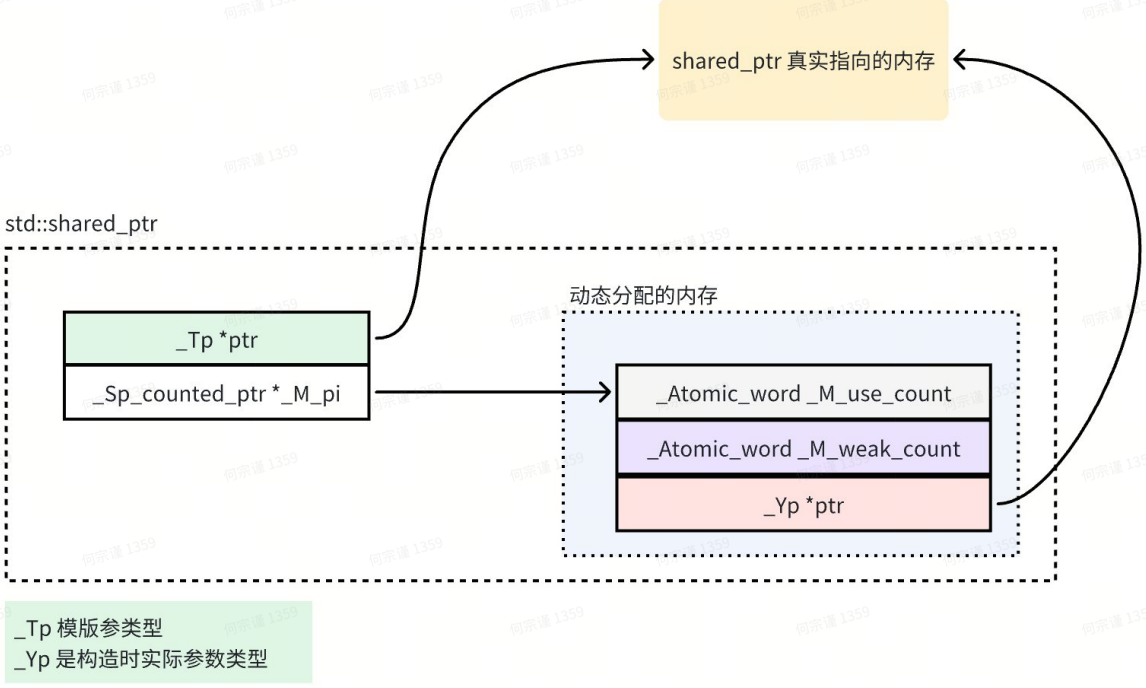
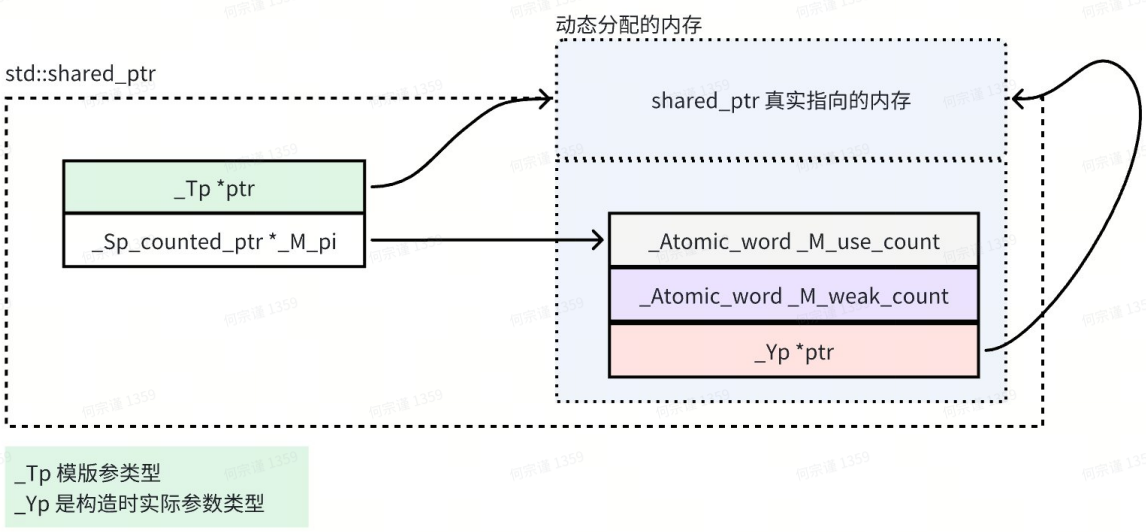
stl shared_ptr 实现思路和相关优化

精简版

c++ stl 的shared_ptr 的内存布局如下



- 每次创建一个shared_ptr 都会 **new 一块24字节**长度的内存。
- shared_ptr 自身存放了一个指针指向外部内存，同时，在动态分配的内存里也存放了外部内存的地址，区别在于那边的类型是真实对象的类型。也就是**shared_ptr 里面同时存放了两个地址一样的指针**。具体原因见后面的分析。
- 当外部内存是动态分配的场景（本文后面例子），使用std::make_shared 能减少一次内存分配，因为两块内存合并到了一起，同时对于访存更友好，所以**尽可能用std::make_shared**



背景

在 【服务端性能优化】C/C++ 高性能代码实践（二） 谢更新 介绍了 shared_ptr 对性能可能造成的影响，在pycton项目中，发现下面这段代码，c++的shared_ptr **能够正确调用到虚函数**，对其内在机制实现产生的兴趣。

```
1   struct A {
```

评论（10）

[画板]

高昌利 3月10日 21:00
能吐槽你这个图和一般的成员变量表示顺序不一致么？一般来说是从上往下是地址从低到高吧？

张青山 3月10日 21:07
额，是不严谨… 有个是在基类

张青山 3月11日 10:47
@高昌利 我认真看了下实现，atomic 变量放在基类 _Sp_counted_base, ptr 指针放在 子类 _Sp_counted_ptr， 从对象模型上，这个布局应该没问题。不确定是否理解了你的问题。 地址高低理论上这张图不关心，属于更底层的实现。。

高昌利 3月11日 10:49
这个图不是特别明显，下面 make shared 的那个比较明显？我记得 count 和 ptr 是放在 make_shared 内存的低地址的。

张青山 3月11日 10:54
Good catch

shared_ptr 里面同时存放了两个地址一…

张振晖 3月11日 15:10
提一个点，这个结构导致 shard_ptr 不是线程安全的；我发现好多同学会把 shard_ptr 当线程安全来用，频繁切换其管理对象

能减少一次内存分配

张振晖 3月11日 14:28
不光是减少一次内存分配，内存是连续的话后续调用的性能也会更好

```
  int a;
  ~A() { printf("~A\n"); }
};
int main() {
  std::shared_ptr<void> f(new A());
}
```

如果让我们来实现，最简单的实现方式如下

```
template <typename T>
class shared_ptr {
T* _data;
public:
  shared_ptr(T *p) : _data(p) {}
  ~shared_ptr() { delete _data; }
};
```

但是按照以上的实现方式，T的类型是 void, 无法**调用到 A 的析构函数**。

改进

为了能调用到A的析构函数，我们必须保存下来所要delete的指针的类型信息。也就是data的类型信息必须是 构造函数传进来的。

```
template <typename T>
class shared_ptr {
T* _data;
public:
  template <typename U>
  shared_ptr(U *p) : ? {}
  ~shared_ptr() { delete _data; }
};
```

由于U的类型必须在构造的时候才能够确定，所以 data 的类型无法直接放在类 shared_ptr 的成员变量中。因此可以进一步改进成下面的方式

```
template <typename T>
class helper {
T * _data;
public:
  helper(T *p) : _data(p) {}
  ~helper() { delete _data; }
};

template <typename T>
class shared_ptr {
?* _helper;
public:
  template <typename U>
  shared_ptr(U *p) { _helper = new helper(p); }
  ~shared_ptr() { delete _helper; }
};
```

> void

张兴锐 3月10日 20:24

这里也挺好奇的，为什么这里要用 void

张青山 3月10日 20:35

我们有个 python 和 c++结合的编译项目里面有这个场景..

如此，当shared_ptr 析构的时候，可以调用到helper的析构，而helper本身模版参数为实际参数的类型（也就是例子中的A），当它析构的时候，就能够调用到 A的析构。

现在唯一要解决的是 **_helper 变量的类型**。

解决办法是再给他加个基类 helper_base

```
class helper_base {
public:
  virtual ~helper_base() = default;
};

template <typename T>
class helper : public helper_base {
T * _data;
public:
  helper(T *p) : _data(p) {}
  virtual ~helper() { delete _data; }
};

template <typename T>
class shared_ptr {
helper_base* _helper;
public:
  template <typename U>
  shared_ptr(U *p) { _helper = new helper(p); }
  ~shared_ptr() { delete _helper; }
};
```

如此，就完成了一个 shared_ptr 的实现。而shared_ptr 本身还有个计数功能，可以一起放在helper 中实现。也就是

```
class helper_base {
public:
  virtual ~helper_base() = default;
};

template <typename T>
class helper : public helper_base {
T * _data;
std::atomic<long> ref;

public:
  helper(T *p) : _data(p) {}
  virtual ~helper() { if (ref == 0) delete _data; }
};

template <typename T>
class shared_ptr {
helper_base* _helper;
public:
  template <typename U>
  shared_ptr(U *p) { _helper = new helper(p); }
  ~shared_ptr() { delete _helper; }
};
```

优化

对于shared_ptr, 访问所指向内存指针是一个高频的操作

```
class helper_base {
public:
  virtual ~helper_base() = default;
};

template <typename T>
class helper : public helper_base {
T * _data;
std::atomic<long> ref;

public:
  helper(T *p) : _data(p) {}
  virtual ~helper() { if (ref == 0) delete _data; }
};

template <typename T>
class shared_ptr {
helper_base* _helper;
public:
  template <typename U>
  shared_ptr(U *p) { _helper = new helper(p); }
  ~shared_ptr() { delete _helper; }

  T *get() const { return static_cast<T*>(_helper->_data); }
};
```

每次都需要做一次访存操作，可以通过增加内存换取性能。也就有了最终的版本

```
class helper_base {
public:
  virtual ~helper_base() = default;
};

template <typename T>
class helper : public helper_base {
T * _data;
std::atomic<long> ref;

public:
  helper(T *p) : _data(p) {}
  virtual ~helper() { if (ref == 0) delete _data; }
};

template <typename T>
class shared_ptr {
T *_data;
helper_base* _helper;
public:
  template <typename U>
  shared_ptr(U *p) : _data(p) { _helper = new helper(p); }
  ~shared_ptr() { delete _helper; }

  T *get() const { return _data; }
};
```

> _helper->_data

朱欣 3月10日 21:36

这里 _helper 不需要 static_cast 一下吗才能获取到 _data 吗

张青山 3月10日 22:24

真实的实现里面封装了好多层，这边是简化的写法，我晚点补充下细节…

张青山 3月11日 10:49

@朱欣 这个写法是如果我们去 helper 里面读 data 的写法，不是标准当前的实现。严谨上确实是需要 cast 下（helper crtp 下），这边只是一个示例表达意思。

> 可以通过增加内存换取性能

张兴锐 3月10日 20:21

原来是这样，我之前也很疑惑为什么两边有同一个指针

黄兢成 3月10日 22:32

@张兴锐 std::shared_ptr 有些时候两边的指针并不相等的，如支持 Aliasing constructor 的用法。

黄兢成 3月10日 22:35 （编辑过）

或者多继承情况下，Base 和 Derived 的转换需要地址偏移。

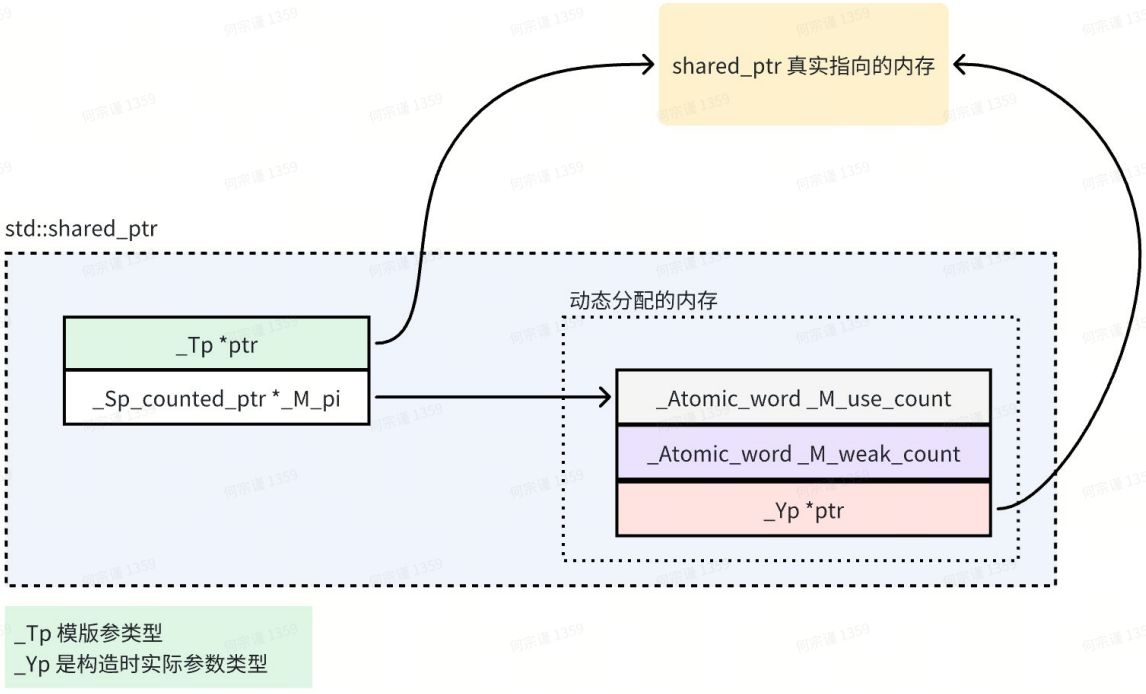
这其实就是stl里面shared_ptr **实现的基本思路**

```
class _Sp_counted_ptr_base {};
template <typename _Ptr>
class _Sp_counted_ptr : public _Sp_counted_ptr_base {
  void
  _M_release() noexcept
  {
    delete _M_ptr;
  }
  _Ptr _M_ptr; // A *
};

template <typename _Tp>
class shared_ptr {
  template <typename _Yp>
  shared_ptr(_Yp *__p) : _M_ptr(__p){ _M_pi = new
_Sp_counted_ptr<_Yp>(__p) }

  _Tp* _M_ptr; // void *_M_ptr;
  _Sp_counted_ptr_base *_M_pi;       // 用真实类型实例化
};
```

内存布局如下



进一步优化

在我们的例子中

```
struct A {
  int a;
  ~A() { printf("~A\n"); }
};
int main() {
  std::shared_ptr<void> f(new A());
}
```

意味着 外部的内存也是动态分配的，所以我们的例子中就会出现两次内存分配

std::shared_ptr<Base> ptr(new Derived()); 此时也会一边存储 Base*，另一边控制块存储 Derived*。

张青山 3月11日 10:52

@黄兢成 严谨上确实有可能存在地址不一致的可能（主要是编译器会自己根据对象模型调整偏移），但这部分其实都是编译器偷偷干的，对于用户而言可以不感知。但确实是有可能。

黄兢成 3月11日 12:24

@张青山 不仅仅是继承情况下，偏移导致两边地址可能不同。参考我上面评论那图的代码，std::shared_ptr 支持 aliasing constructor。shared_ptr 自身存储的指针，跟控制块（对应文章中的 helper<T>) 存储的指针，地址、类型可以完全不同的。并且用户能显式设置进去。此时从控制块中 get 地址去 static_cast，没有办法还原出 shared_tr 自身存储那地址。控制块存储的指针用于释放，shared_ptr 存储的指针用于使用。两边都存储并非仅仅为了性能。只是 aliasing constructor 较少用。

展开

张青山 3月11日 13:41

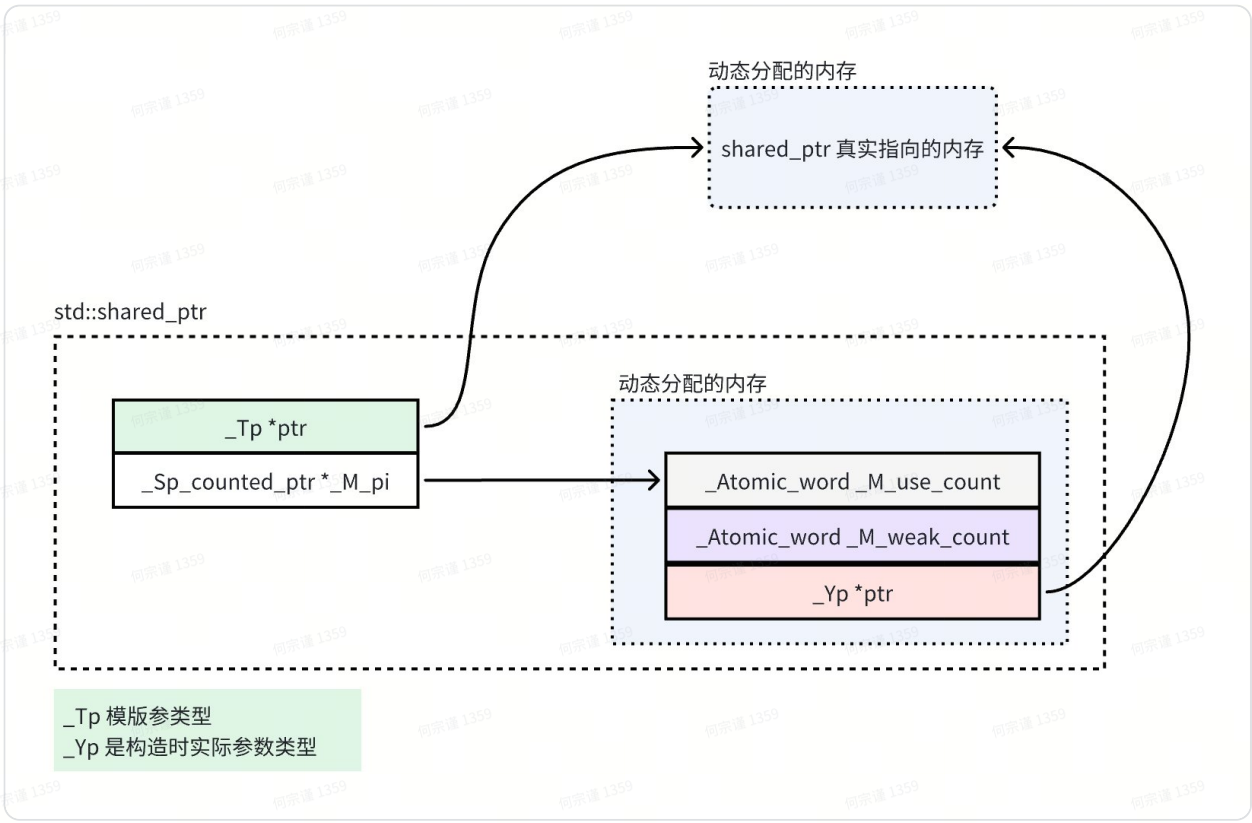
@黄兢成 你说的是有道理的，在这种 case 下，新的 shared_ptr 控制块会直接指向传进来的那个 shared_ptr 的控制块（里面包含 计数 和 需要释放的内存指针），同时自己的指针有初始化成 传进来的那个。。。

张青山 3月11日 14:03

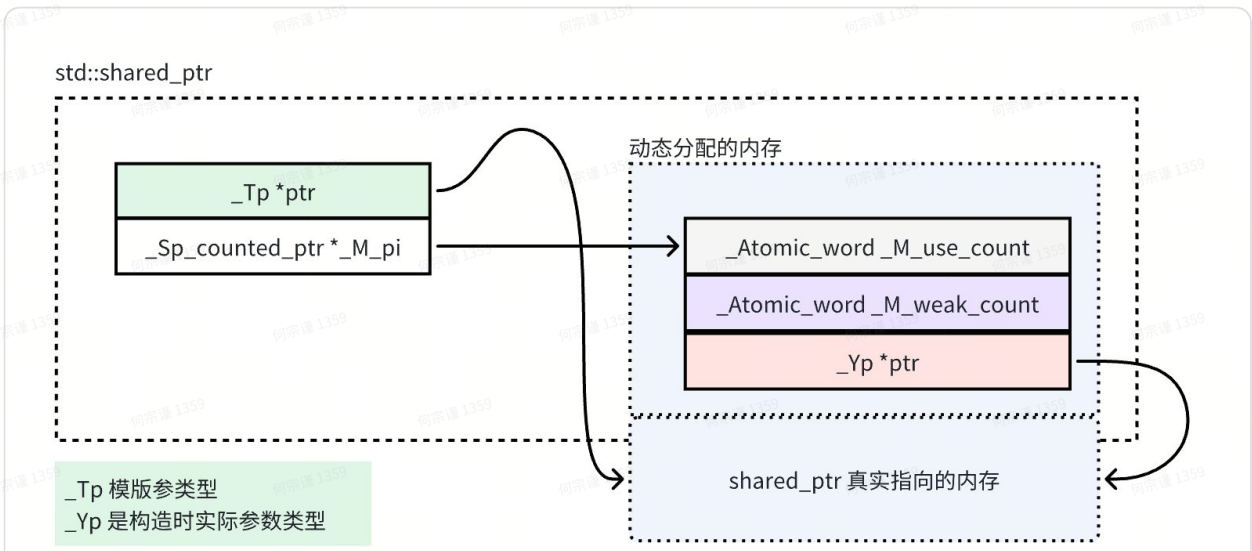
我在文档最后一个章节补充了下这部分， @黄兢成 帮忙看下有么有问题。

黄兢成 3月11日 14:26

@张青山 章节和示意图没有问题，只是最好能简单补充下 aliasing constructor 的动机。有时会在数据嵌套的场合使用。如下例子 VideoFrame 内部保存了 Data。但有某些限制，我们需直接使用（或保存）std::shared_ptr<Data>，于是这个 Data 就需要依赖 VideoFrame 的生命期。为避免复制，就可用 aliasing constructor 了。只是这例子也是我编造的，实际上我还没有在



从原理上，把它们放在一起，一次内存分配就够了



这就是c++ std::make_shared 的作用。

https://en.cppreference.com/w/cpp/memory/shared_ptr/make_shared

> Allocates memory for an object and initialize the object with the supplied arguments. Returns a std::shared_ptr object managing the newly created object
>
> These functions will typically allocate more memory than sizeof(T) to allow for internal bookkeeping structures such as reference counts.
>
> These functions may be used as an alternative to std::shared_ptr<T>(new T(args...)). The trade-offs are:
>
> - std::shared_ptr<T>(new T(args...)) performs at least two allocations (one for the object `T` and one for the control block of the shared pointer), while std::make_shared<T> typically performs only one allocation (the standard recommends, but does not require this; all known implementations do this).

由于 用户分配的内存和shared_ptr 自身的内存放在一起，所以就不能通过传deleter的方式自定义释放内存方法

> - Unlike the std::shared_ptr constructors, `std::make_shared` does not allow a custom deleter.

引申

现实工程中碰到过。

> _data(p)
>
> 吴博雅 3月12日 13:18
> p 是 U*的，_data 是 T*的，类型会不会无法隐式转换
>
> 张青山 3月12日 13:41 （编辑过）
> @吴博雅 这是调用者要保证的，类似 T *data = new U()
> 不然编译会报错。

> 是否可以不用在堆上分配 一个指向外部…
>
> 张振晖 3月11日 15:07

当我们限制shared_ptr 的使用场景（比如模版参数和构造函数的参数类型一致），是否可以不用在堆上分配 一个指向外部内存的指针，来达到优化目的?

理论上可行，但是我们无法减少内存分配的次数，最多只是节省了堆上的8字节内存，相对于可用性带来的损失，少用8字节内存未必是一个好的方案。

补充

c++ stl的shared_ptr 的实现（想要支持的功能）比我们想象的还要复杂，**shared_ptr 本身和 动态分配的内存 为什么要同时维护两个相同的指针**，这个问题一方面可以提高get函数的性能，另一方面，经过 黄兢成 同学的提醒，在某些条件下，确实存在不一样的情况，使得实现上不得不提供两个。

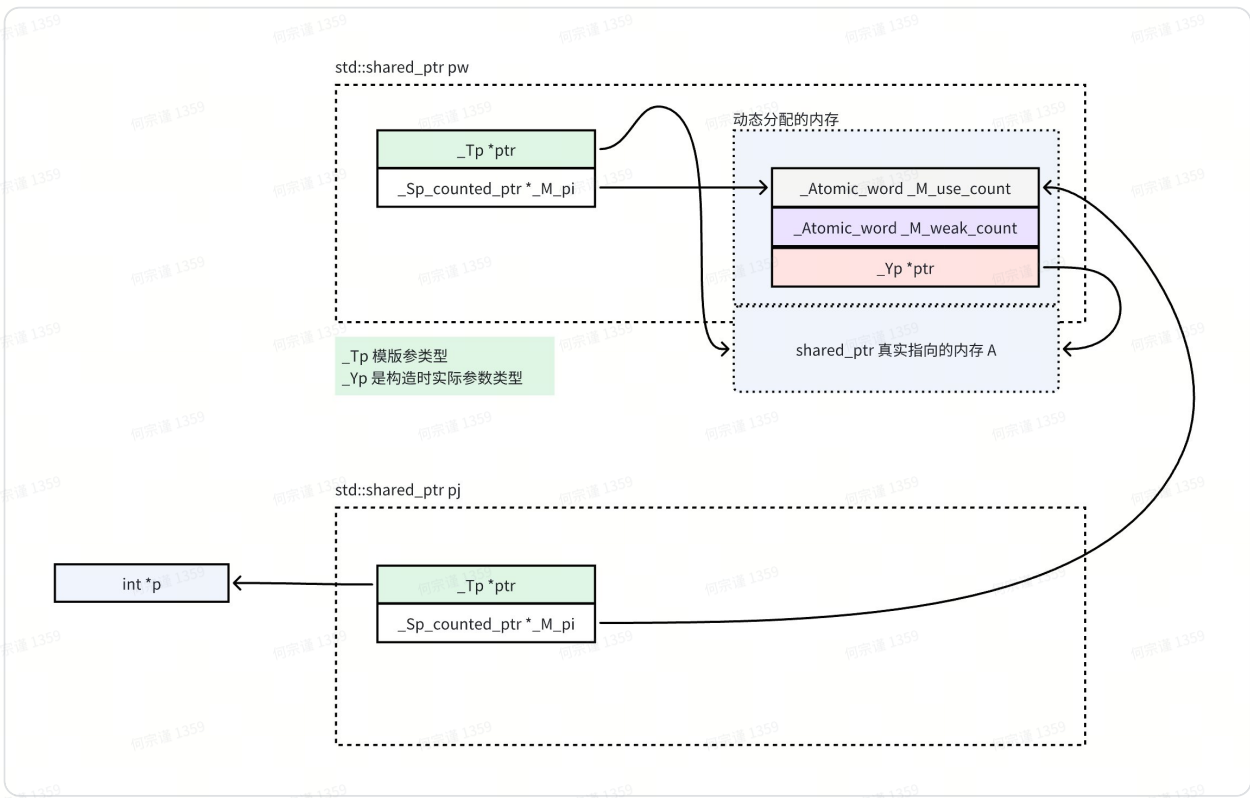
```
template< class Y >
shared_ptr( const shared_ptr<Y>& r, element_type* ptr ) noexcept;   (8)
```

> The *aliasing constructor*: constructs a `shared_ptr` which shares ownership information with the initial value of r, but holds an unrelated and unmanaged pointer ptr. If this `shared_ptr` is the last of the group to go out of scope, it will call the stored deleter for the object originally managed by r

具体用法如下

```
1   std::shared_ptr<A> pw = std::make_shared<A>();
2   int *p;
3   std::shared_ptr<int> pj(pw, p);
```

具体的内存布局如下



pj 这个shared_ptr 自身存放的是传进来的第二个参数 `p` 的值，它不再动态创建 24字节动态内存，而是直接指向传进来的shared_ptr 已经创建好的 堆内存。如此，就实现了文档中描述的功能。

> template< class Y >
>
> shared_ptr(const shared_ptr<Y>& r, element_type* ptr) noexcept;

`shares ownership information with the initial value of r,` 和 `but holds an unrelated and unmanaged pointer ptr. If this shared_ptr is the last of the group to go out of scope, it`

部分场景可行，比如 map<shard_ptr<int>>，如果这个 map 非常大，这个场景能节约大量的内存，是有价值的

std::shared_ptr<int> pj(pw, p);

张振晖 3月11日 14:47
真没见过这个用法，学到了

张振晖 3月11日 14:50
我理解这里是用 pw 中的引用计数来管理 p 是吧，当 pj 或者 pw 共同的引用计数归零，销毁 p?

张青山 3月11日 14:50
管理 new 出来的 A，p 的内存不做管理

张青山 3月11日 14:50
我加个例子

张振晖 3月11日 14:55
哦哦，懂了，相当于把 pj 割裂成两半，引用计数与 pw 共享，但是内存指向 p（int*）；所以通过 get 调用返回的是 p，但是引用计数归零销毁的是 A；在这套逻辑里 p 无法被销毁，其生命周期需要被其他内容管理，比如 A

张青山 3月11日 14:57
对的，我补充了例子，应该更加清晰一点。

`will call the stored deleter for the object originally managed by r`

Aliasing constructor的具体例子

```
#include <memory>
class B {
};

class A {
public:
B b;
};

int main() {
  std::shared_ptr<A> ptr_A = std::make_shared<A>();
  // 由于某些原因，我们想把b 传出去
  std::shared_ptr<B> ptr_B = std::shared_ptr<B>(ptr_A, &ptr_A->b);
  // do something with ptr_B
  // foo(ptr_B)
  return 0;
}
```

某种情况下（比如出于架构的封装等），我们需要把A的成员b 传出去，但我们不希望在这个过程中A的内存被释放了，就可以用aliasing constructor。ptr_B 和 ptr _A 共享对内存 A的所有权，ptr_B.get() 拿到的是 ptr_A->b 的地址。

> std::shared_ptr<B> ptr_B = std::shared···

李旭 昨天 16:28 （编辑过）

一种最常见的例子是在 managed i/o buffer 下（比如 zero-copy/DMA 这种由 driver 预先跟网卡协商 pre-allocate 出来的），buffer 由 I/O driver 的 runtime 管理，但是我们会把部分数据 slicing 交给用户去做读写，这个时候通过一个 proxy 对象把 buffer 扔出来就 ok，同时 runtime 也能感知到外部可能还有用户；另一种是异构抽象，比如在不同的 db driver 中抽象一个统一的 connection 对象，参考 https://codesynthesis.com/~boris/blog/2012/04/25/shared-ptr-aliasing-constructor/。

展开