C++43期第2组本周情况总结

组长：高孝国

组员：吴杨昊，姜山，李慧强，杨先锋，陈彦昌

时间：2022.05.30-2022.06.04

1. 上课打卡情况

姓 名	星期一	星期二	星期三	星期四	星期五	星期六
吴杨昊	✓	✓	✓	✓	✓	✓
高孝国	✓	✓	✓	✓	✓	✓
姜山	✓	✓	✓	✓	✓	✓
李慧强	✓	✓	✓	✓	✓	✓
杨先锋	✓	✓	✓	✓	✓	✓
陈彦昌	✓	✓	✓	✓	✓	✓

2. 作业完成情况

截至2022-06-04 23:30（具体情况以实际为准）

姓 名	星期一	星期二	星期三	星期四	星期五	星期六
吴杨昊	✓	✓	✓	✓	✓	✓
高孝国	✓	✓	✓	✓	✓	✓
姜山	✓	✓	✓	✓	✓	✓
李慧强	✓	✓	✓	✓	✓	✓
杨先锋	✓	✓	✓	✓	✓	✓
陈彦昌	✓	✓	✓	✓	✓	✓

3. 组内分享内容摘要

主讲人：吴杨昊

会议记录：吴杨昊

字符串Hash与HashMap

什么是字符串Hash
hash，其实就是将一个东西映射成另一个东西，类似Map，key对应value。

那么字符串Hash，其实就是：构造一个数字使之唯一代表一个字符串。但是为了将映射关系进行一一对应，也就是，一个字符串对应一个数字，那么一个数字也对应一个字符串。

用字符串Hash的目的是，我们如果要比较一个字符串，我们不直接比较字符串，而是比较它对应映射的数字，这样子就知道两个“子串”是否相等。从而达到，子串的Hash值的时间为O(1)，进而可以利用“空间换时间”来节省时间复杂的。

我们希望这个映射是一个单射，所以问题就是如何构造这个Hash函数，使得他们成为一个单射。不用担心，接下来的内容正要讲解。

构造字符串Hash

```
1    假如给你一个数字1166，形式上你只知道它只是1和6的组合，但你知道它代表的实际大
     1*10^3+1*10^2+6*10^1+6*10^0。
```

同理，给你一个字符串，要把它转换为数字，就可以先把每一个字符都先对应一个数字，然后把它们按照顺序乘以进制（Base）的幂进行相加，然后这个数可能很大，所以一般会取余数（MOD）。

根据上面的理解，其实将字符串映射成数字，和我们平时的将一个 某Base进制数，变为一个十进制数，相类似。

我们先定义以下：

给定一个字符串$S = s_1 s_2 s_3 ... s_n$，对于每一个s_i就是一个字母，那么我们规定$idx(s_i) = s_i -' a' + 1$。（当然也可以直接用其ASCII值）

构造字符串Hash总共有三种方法。每一种方法，主要都是用使用 Base 和 MOD（都要求是素数），一般都是 Base < MOD，同时将Base和MOD尽量取大即可，这种情况下，冲突（即不同字符串却有着相同的hash值）的概率是很低的。

1）自然溢出方法
对于自然溢出方法，我们定义 Base，而MOD对于自然溢出方法，就是 unsigned long long 整数的自然溢出（相当于MOD 是$2^{64} - 1$）

```
1  #define ull unsigned long long
2
3  ull Base;
4  ull hash[MAXN], p[MAXN];
5
6  hash[0] = 0;
7  p[0] = 1;
```

定义了上面的两个数组，首先$hash[i]$ 表示$[0, i]$ 字串的hash 值。而 $p[i]$表示 $Base^i$，也就是底的 i 次方。

那么对应的 Hash 公式为：

$$hash[i] = hash[i - 1] * Base + idx(s[i])$$

(类似十进制的表示，14，一开始是 0，然后 0 * 10 + 1 = 1，接着 1*10 + 4 = 14)。

2）单Hash方法
同样的，定义了 Base 和 MOD，有了对应的要求余 MOD。所以一般用 long long 就可以了。

```
#define ll long long

ll Base;
ll hash[MAXN], p[MAXN];

hash[0] = 0;
p[0] = 1;
```

定义了上面的两个数组，首先 $hash[i]$ 表示前 i 个字符的字串的hash 值。而 $p[i]$表示$Base^i$，也就是底的 i次方。

那么对应的 Hash 公式为：

$$hash[i] = (hash[i-1] * Base + idx(s[i]))\%MOD$$

对于此种Hash方法，将Base和MOD尽量取大即可，这种情况下，冲突的概率是很低的。

举例

```
如取Base = 13, MOD=101, 对字符串abc进行Hash
hash[0] = 0    (相当于 0 字串)
hash[1] = (hash[0] * 13 + 1) % 101 = 1
hash[2] = (hash[1] * 13 + 2) % 101 = 15
hash[3] = (hash[2] * 13 + 3) % 101 = 97
```

这样，我们就认为字符串abc当做97，即97就是abc 的hash值。

3）双Hash方法

用字符串Hash，最怕的就是，出现冲突的情况，即不同字符串却有着相同的hash值，这是我们不想看到的。所以为了降低冲突的概率，可以用双Hash方法。

将一个字符串用不同的Base和MOD，hash两次，将这两个结果用一个二元组表示，作为一个总的Hash结果。

相当于我们用不同的Base和MOD，进行两次 单Hash方法 操作，然后将得到的结果，变成一个二元组结果，这样子，我们要看一个字符串，就要同时对比两个 Hash 值，这样子出现冲突的概率就很低了。

那么对应的 Hash 公式为：

$$hash1[i] = (hash1[i-1] * Base1 + idx(s[i]))\%MOD1$$
$$hash2[i] = (hash2[i-1] * Base2 + idx(s[i]))\%MOD2$$

映射的Hash结果为：

$$< hash1[i], hash2[i] >$$

这种Hash很安全。

4）三种不同的构造方法的对比
其实，自然溢出方法，说到底就是单Hash方法，只是把MOD变成了自动溢出，也就是
从速度上来看，应该是： 自然溢出 > 单Hash > 双Hash。（也就是自然溢出 时间更小）。
从安全性上来看，应该：双Hash方法 > 单Hash方法。因为双Hash方法相当于是用两次 单Hash的结果来比较，这样子冲突的概率会变得更低。

一、什么是SimHash

SimHash算法是Google在2007年发表的论文《Detecting Near-Duplicates for Web Crawling》中提到的一种指纹生成算法，被应用在Google搜索引擎网页去重的工作之中。

对于文本去重这个问题，常见的解决办法有余弦算法、欧式距离、Jaccard相似度、最长公共子串等方法。但是这些方法并不能对海量数据高效的处理。

比如说，在搜索引擎中，会有很多相似的关键词，用户所需要获取的内容是相似的，但是搜索的关键词却是不同的，如“北京好吃的火锅“和”哪家北京的火锅好吃“，是两个可以等价的关键词，然而通过普通的hash计算，会产生两个相差甚远的hash串。而通过SimHash计算得到的Hash串会非常的相近，从而可以判断两个文本的相似程度。

二、局部性敏感哈希

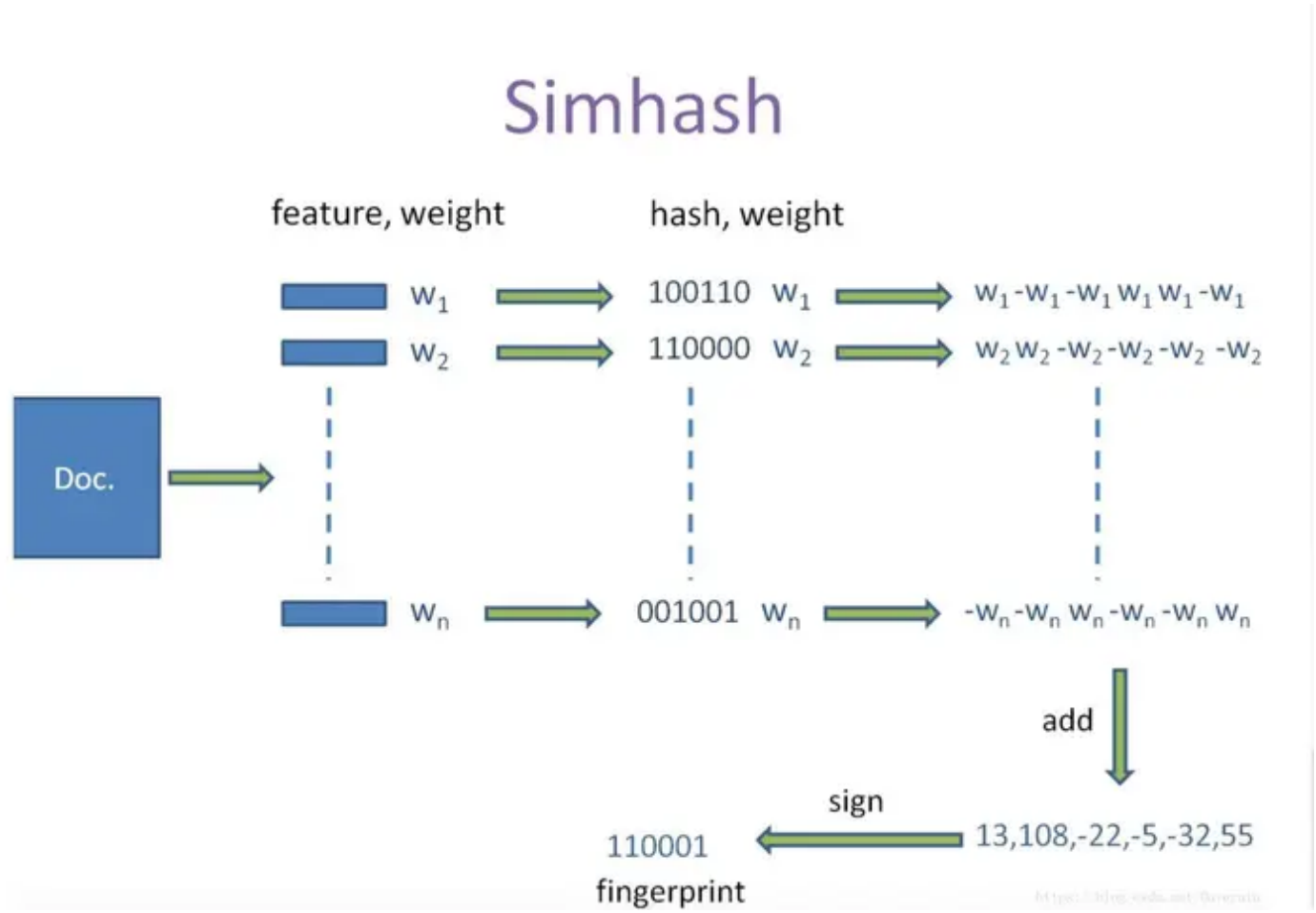
说到hash可能我们第一个想到的是md5这种信息摘要算法，可能两篇文本只有一个标点符号的差距，但是两篇文本A和B的md5值差异就非常大，感兴趣的可以试验一下看看。

有时候我们希望的是原本相同的文章做了微小改动之后的哈希值也是相似的，这种哈希算法称为局部敏感哈希LSH(Locality Sensitive Hashing)，这样我们就能从哈希值来推断相似的文章。

局部敏感哈希算法使得在原来空间相似的样本集合，进行相关运算映射到特定范围空间时仍然是相似的，这样还不够，还需要保证原来不相似的哈希之后仍然极大概率不相似，这种双向保证才让LSH的应用成为可能。

三、simhash的基本过程

SimHash算法主要有五个过程：分词、Hash、加权、合并、降维。
借用一张网络上经典的图片来描述整个过程：



1.分词（可以用一些分词工具来实现）

使用分词手段将文本分割成关键词的特征向量，分词方法有很多，我用了jieba分词来实现，你可以先去除停用词，当然也可以不去除，根据自己的需求选择，假设分割后的特征实词如下：

```
12306 服务器 故障 车次 加载失败 购买 候补订单 支付 官方 消费者 建议 卸载 重装 切换网络 耐心 等待
```

目前的词只是进行了分割，但是词与词含有的信息量是不一样的，比如`12306 服务器 故障`这三个词就比`支付 卸载 重装`更能表达文本的主旨含义，这也就是所谓信息熵的概念。

为此我们还需要设定特征词的权重，方法有很多，简单一点的可以使用TF-IDF来实现。

2.Hash

前面我们使用分词方法和权重分配将文本就分割成若干个带权重的实词，比如权重使用1-5的数字表示，1最低5最高，这样我们就把原文本处理成如下的样式：

```
12306(5) 服务器(4) 故障(4) 车次(4) 加载失败(3) 购买(2) 候补订单(4) 支付(2) 官方(2) 消费者(3) 建议(1) 卸载(3) 重装(3) 切换网络(2) 耐心(1) 等待(1)
```

然后，通过hash函数对每一个词向量进行映射，产生一个n位二进制串，一般常用的位数为32、64、128。

3.加权

前面的计算我们已经得到了每个词向量的Hash串和该词向量对应的权重，这一步我们计算权重向量W=hash*weight。

具体的计算过程如下：hash二进制串中为1的，w = 1 * weight，二进制串中为0的，w = weight * -1.

举个例子，12306的带权重哈希值为 [5 -5 -5 5 5 5 -5 -5]，服务器的带权重哈希值为 [-4 4 4 4 -4 4 -4 4]

4.合并

对于一个文本，我们计算出了文本分词之后每一个特征词的权重向量，在合并这个阶段，我们把文本所有词向量的权重向量相累加，得到一个新的权重向量，假定最终结果为 [18 9 -6 -9 22 -35 12 -5]

5.降维

对于前面合并后得到的文本的权重向量，大于0的位置1，小于等于0的位置0，就可以得到该文本的SimHash值，以上面提到的 [18 9 -6 -9 22 -35 12 -5] 为例，我们得到 [1 1 0 0 1 0 1 0] 这个bit串，也就是论文中提及的该文本的指纹。
到此为止，我们已经计算出了一个文本的SimHash值。那么，如何判断两个文本是否相似呢？我们要用到海明距离。

四、相似度判断

对于两个文本的SimHash的相似度判断，我们使用海明距离来计算。

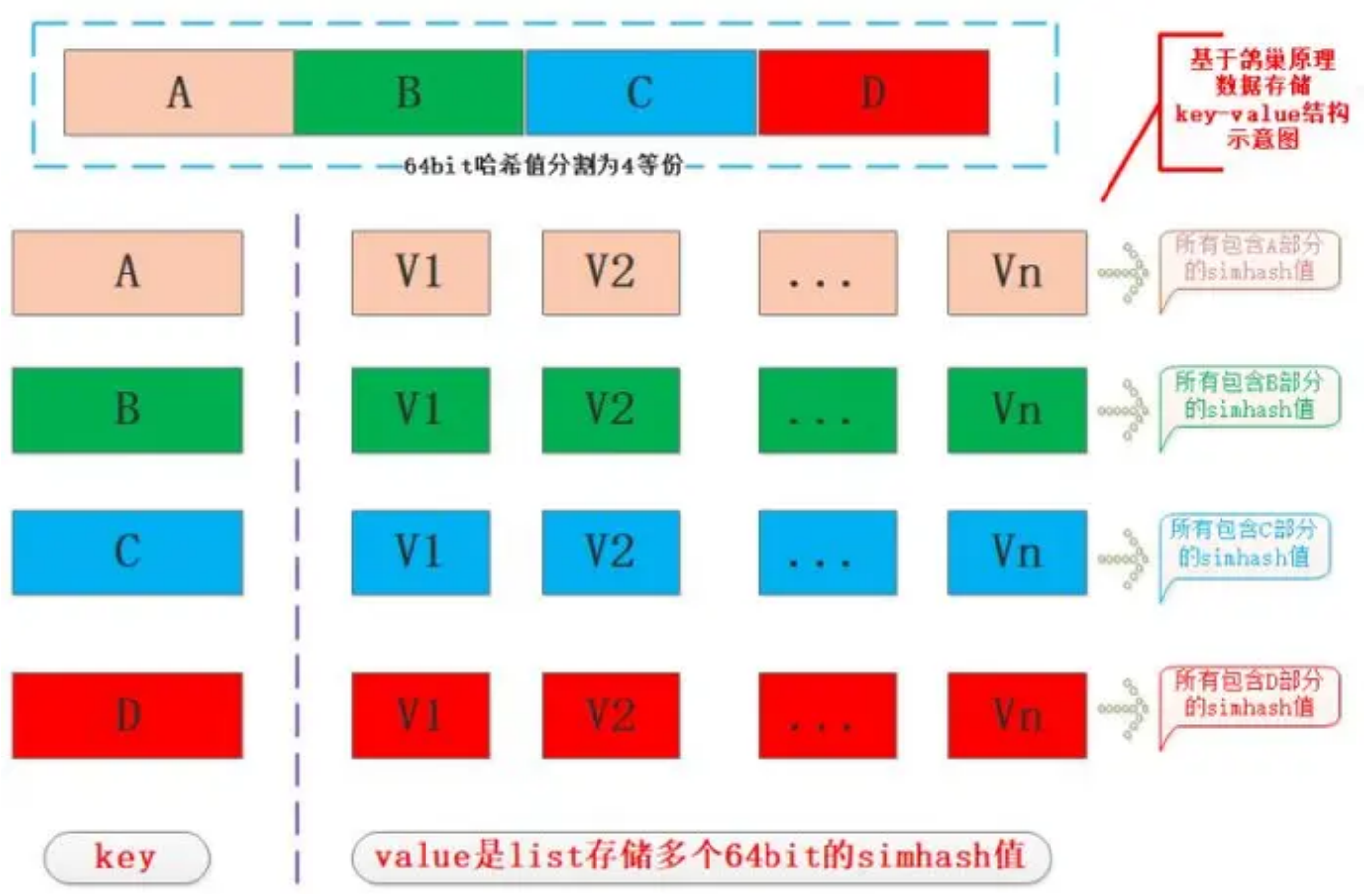
什么是海明距离呢?

简单的说，海明距离可以理解为，两个二进制串之间相同位置不同的个数。举个例子，[1，1，1，0，0，0]和[1，1，1，1，1，1]的海明距离就是3。

在处理大规模数据的时候，我们一般使用64位的SimHash，正好可以被一个long型存储。这种时候，海明距离在3以内就可以认为两个文本是相似的。

五、大规模数据下的海明距离计算

我们在存储时将64bit simhash值均分为4份每份16bit长，然后使用每一份作为key，value是每一份simhash值对应的二进制向量，如下图所示：



当新来一个文本生成**哈希值S'**之后，按照相同的规则生成abcd四部分，之后逐个进行哈希对比，这个时间复杂度是O(1)：

> - 如果abcd四个作为key都不存在，那么可以认为S'没有相似的文本；
> - 如果abcd四个key中有命中，那么就开始遍历对应key的value，查看是否满足<=3的海

> 明距离确定相似性；（一般64位编码的simhash值对应的海明距离阈值设为3）
> - 如果上一个命中的key未找到相似文本，则继续遍历剩下的key，重复相同的过程，直至所有的key全部遍历完或者命中相似文本，则结束。

主讲人：高孝国

会议记录：高孝国

SBOSP(la Sainte Bible Of Smart Pointer)

shared_ptr

```
shared_ptr<string> p1; //空智能指针，可以指向类型为string的对象
unique_ptr<T> up;
shared_ptr<list<int>> p2;

p1 //将p1用作一个类型，若p指向一个对象则为true
*p1 //解引用
if (p1 && p1->empty()) { *p1 = "hi"; } //如果 p1指向一个空 string，解引用p1，将一个新值赋与string

p1.get() //返回p中保存的指针

make_shared<T>(args) //返回一个shared_ptr，指向一个动态分配的类型为T的对象，使用args初始化此对象

shared_ptr<T>p(q) //p是shared ptr q 的拷贝；此操作会递增q中的计数器。q中的指针必须能转换为T*（参见4.11.2节，第143页）

p = q //p和q都是 shared ptr，所保存的指针必须能相互转换。此操作会递减p的引用计数，递增q的引用计数：若p的引用计数变为0，则将其管理的原内存释放

p.unique() //若p.use_count() 为1，返回true；否则返回false
p.use_count() //返回与p共享对象的智能指针数量；可能很慢，主要用于调试
```

make_shared 函数

最安全的分配和使用动态内存的方法是调用一个名为 make_shared 的标准库函数。此函数在动态内存中分配一个对象并初始化它，返回指向此对象的 shared_ptr。与智能指针一样，make_shared 也定义在头文件 memory 中。

```cpp
shared_ptr<int> p3 = make_shared<int>(42);

shared_ptr<string> p4 = make_shared<string>(10, '9');

//指向一个值初始化的int，即为0
shared_ptr<int> p5 = make_share<int>();

//指向一个动态分配的空vector<string>
auto p6 = make_shared<vector<string>>();

auto q(p); //p和q指向相同对象，此对象有两个引用者
auto r = make_shared<int> 42;
r = q;


shared_ptr<Foo> factory(T arg) {
    //恰当的处理arg
    //shared_ptr负责释放内存
    return make_shared<Foo>(arg);
}
```

Shared_ptr和new结合使用

```cpp
shared_ptr<double> p1;
share_ptr<int> p2(new int(42));

shared_ptr<int> p1 = new int(1024); //错误
shared_ptr<int> p2(new int(1204));

shared_ptr<int> clone(int p) {
    return shared_ptr<int>(new int(p));
}

shared_ptr<T> p(q) //p管理内置指针q所指向的对象：q必须指向new分配的内存，且能够转换为T*类型
shared_ptr<T> p(u) //p从 unique_ptr u那里接管了对象的所有权；将u置为空
shared_ptr<T> p(q, d) //p接管了内置指针q 所指向的对象的所有权。q必须能转换为T*类型（参见 4.11.2节，第143 页）。p将使用可调用对象d（参见10.3.2 节，第346 页）来代替 delete

shared_ptr<T> p(p2, d);

```

```
/*
若p 是唯一指向其对象的 shared ptr， reset 会释放此对
象。若传递了可选的参数内置指针 q，会令p指向 q，否则会
将p置为空。若还传递了参数 d，将会调用d而不是 delete
来释放q
*/
 p.reset(); // 析构
 p.reset(q); //
 p.reset(q, d);
```

不要混合使用普通指针和智能指针

```
void process(shared_ptr<int> ptr) {
    //使用ptr
} //ptr离开作用域，被销毁

//正确
shared_ptr<int> p(new int(42));
process(p); //正确
process(shared_ptr<int>(p)); //正确
process(shared_ptr<int>(p.get())); //错误
int i = *p;

//错误
int* x(new int(1024));
process(x); //错误
process(shared_ptr<int>(x));
int j = *x; //为定义的，x已经是野指针


```

不要使用get初始化另一个智能指针或为智能指针赋值

```
shared_ptr<int> p(new int(42));
int* q = p.get();  //正确：但使用q时要注意，不要让它管理的指针被释放
{
shared_ptr<int> p1(q); //注意这里书上的代码有误。
}
int foo = *p; //未定义：p指向的内存已经被释放了,而且会doublefree
```

其他shared_ptr操作

```
p.reset(new int(1024));
if (!p.unique()) p.reset(new string(*p));

*p +=. newVal;
```

智能指针和异常

如果使用智能指针，即使程序块过早结束，智能指针类也能确保在内存不再需要时将其释放。

```
//行
void f() {
    shared_ptr<int> sp(new int(42));
}

//不行
void f() {
    int* ip = new int(42);
    delete ip;
}
```

删除器

```
struct destination;
struct connection;
connection connect(destination*);
void disconnect(connection);
void f(destination& d) {
    connection c = connect(&d);
    //如果没有disconnect就无法关闭c
}

//函数方式
void end_connection(connection* p) { disconnect(*p); }

void f(destination& d) {
    connection c = connect(&d);
    shared_ptr<connection> p(&c, end_connection);
}

//对象方式
```

```
struct EndConn {
  void operator()(connection* p) const {
      if(p) {
          disconnect(*p);
      }
  }
};

void f(destination& d) {
    connection c = connect(&d);
    shared_ptr<connection> p(&c, EndConn());
}

//附unique_ptr
unique_ptr<connection, EndConn> up(&c);
```

智能指针陷阱

不使用相同的内置指针值初始化（或 reset）多个智能指针。

不 delete get（）返回的指针。

不使用 get()初始化或 reset 另一个智能指针。

如果你使用 get()返回的指针，记住当最后一个对应的智能指针销毁后，你的指针就变为无效了。

如果你使用智能指针管理的资源不是 new 分配的内存，记住传递给它一个删除器（参见12.1.4节，第415页和12.1.5节，第419页）。

陷阱例子

```
//shared_ptr 被提前释放:
void process(shared_ptr<int> svp) {}
int main() {
 int* vp = new int(10);
 process(shared_ptr<int>(vp));//被释放
}

//使用栈对象初始化智能指针，造成悬挂指针
auto process() {
 int v=10;
 int* vp = &v;
```

```
 return shared_ptr<int>(vp);
}
int main() {
 cout << *process() << endl;
}

//循环引用
struct C {
    ~C() { cerr << "destructor" << endl; }
    shared_ptr<C> sp;
};
int main() {
 auto p = make_shared<C>();
 auto q = make_shared<C>();
 p->sp = q;
 q->sp = p;
}

//不同的智能指针托管同一个裸指针（堆空间）
void test3() {
 Point* pt = neew Point(1,2);
 shared_ptr<Point> sp(pt);
 shared_ptr<Point> sp2(sp);
 shared_ptr<Point> sp3(pt); //不会报错但是不行
}

void test() {
 Point* pt = neew Point(1,2);
 unique_ptr<Point> up(pt);
 unique_ptr<Point> up2(pt);
}

void test2() { //shared_ptr同理
 unique_ptr<Point> up(new Point(1,2));
 unique_ptr<Point> up2(new Point(3,4));
 up.reset(up2.get());
}
```

enable_shared_from_this

在很多场合，经常会遇到一种情况，**如何安全的获取对象的this指针**，一般来说我们不建议直接返回this指针，可以想象下有这么一种情况，返回的this指针保存在外部一个局部/全局变量，当对象已经被析构了，但是外部变量并不知道指针指向的对象已经被析构了，如果此时外部使用了这个指针就会发生程序奔溃。既要像指针操作对象一样，又能安全的析构对象，很自然就想到，智能指针就很合适！

```
class Point
: public std::enabled_shared_from_this<Point> {
    shared_ptr<Point> addPoint(Point* pt) {
     //
     return shared_from_this();
 }
}
```

使用情境

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

class Widget{
public:
 Widget(){
     std::cout << "Widget constructor run" << std::endl;
 }
 ~Widget(){
     std::cout << "Widget destructor run" << std::endl;
 }

 std::shared_ptr<Widget> GetSharedObject(){
     return std::shared_ptr<Widget>(this);
 }
};

int main()
{
 std::shared_ptr<Widget> p(new Widget());
 std::shared_ptr<Widget> q = p->GetSharedObject();

 std::cout << p.use_count() << std::endl;
 std::cout << q.use_count() << std::endl;
```

```
25
26     return 0;
27   }
28
29   /*
30   Widget constructor run
31   1
32   1
33   Widget destructor run
34   Widget destructor run
35   22:06:45: 程序异常结束。
36   */
```

从输出我们可以看到，调用了一次构造函数，却调用了两次析构函数，很明显这是不正确的。而std::enable_shared_from_this正是为了解决这个问题而存在。

首先来看一个例子：

```
1   void test1() { //正确的做法
2     auto sp1 = make_shared<int>(5);
3     weak_ptr<int> wp1(sp1);
4     cout << "sp1.use_count: " << sp1.use_count() << endl;
5     cout << "wp1.use_count: " << wp1.use_count() << endl;
6     auto sp2 = shared_ptr<int>(wp1);
7     cout << "sp2.use_count: " <<sp2.use_count() << endl;
8     cout << "sp1.use_count: " << sp1.use_count() << endl;
9   }
10
11  void test2() { //错误的做法
12    auto sp1 = make_shared<int>(6);
13    shared_ptr<int> sp2(sp1.get());
14
15    cout << sp1.use_count() << endl; //输出仍是1，一个裸指针被托管两次
16  }
17
18  //可以说这是shared_ptr的一个致命缺点。
```

源码

```
1   template<class _Tp>
2   class _LIBCPP_TEMPLATE_VIS enable_shared_from_this
3   {
4     mutable weak_ptr<_Tp> __weak_this_; //数据成员
```

```
protected:
 _LIBCPP_INLINE_VISIBILITY _LIBCPP_CONSTEXPR
 enable_shared_from_this() _NOEXCEPT {}
 _LIBCPP_INLINE_VISIBILITY
 /*这里*/
 enable_shared_from_this(enable_shared_from_this const&) _NOEXCEPT
{}
 _LIBCPP_INLINE_VISIBILITY
 enable_shared_from_this& operator=(enable_shared_from_this const&)
_NOEXCEPT
     {return *this;}
 _LIBCPP_INLINE_VISIBILITY
 ~enable_shared_from_this() {}
public:
 _LIBCPP_INLINE_VISIBILITY
 /*这里*/
 shared_ptr<_Tp> shared_from_this()
     {return shared_ptr<_Tp>(__weak_this_);}
 _LIBCPP_INLINE_VISIBILITY
 shared_ptr<_Tp const> shared_from_this() const
     {return shared_ptr<const _Tp>(__weak_this_);}

#if _LIBCPP_STD_VER > 14
 _LIBCPP_INLINE_VISIBILITY
 weak_ptr<_Tp> weak_from_this() _NOEXCEPT
    { return __weak_this_; }

 _LIBCPP_INLINE_VISIBILITY
 weak_ptr<const _Tp> weak_from_this() const _NOEXCEPT
     { return __weak_this_; }
#endif // _LIBCPP_STD_VER > 14

 template <class _Up> friend class shared_ptr;
};
```

std::enable_shared_from_this是模板类，内部有个 *Tp类型weak_ptr指针，调用shared_from_this成员函数便可获取到* Tp类型智能指针，从这里可以看出，_Tp类型就是我们的目标类型。再来看看std::enable_shared_from_this的构造函数都是protected，因此不能直接创建std::enable_from_shared_from_this类的实例变量，只能作为基类使用。因此使用方法如下代码所示：

```
#include <iostream>
```

```cpp
#include <memory>

class Widget : public std::enable_shared_from_this<Widget>{
public:
 Widget(){
     std::cout << "Widget constructor run" << std::endl;
 }
 ~Widget(){
     std::cout << "Widget destructor run" << std::endl;
 }

 std::shared_ptr<Widget> GetSharedObject(){
     return shared_from_this();
 }
};

int main()
{
 std::shared_ptr<Widget> p(new Widget());
 std::shared_ptr<Widget> q = p->GetSharedObject();

 std::cout << p.use_count() << std::endl;
 std::cout << q.use_count() << std::endl;

 return 0;
}

/*
Widget constructor run
2
2
Widget destructor run
*/
```

这里为什么要创建智能指针p而不是直接创建裸指针p？根本原因在于std::enable_shared_from_this内部的weak_ptr，若只是创建裸指针p，那么p被delete后仍然面对不安全使用内部this指针问题。因此p只能被定义为智能指针。当p被定义为shared_ptr智能指针后，p指针引用计数是1(weak_ptr不会增加引用计数)，再通过shared_from_this获取内部this指针的智能指针，则p的引用计数变为2。

这里就有一个问题了，`shared_ptr<T>`是如何判断出T继承自`enable_shared_from_this`的?

```cpp
#include <stdlib.h>
#include <iostream>

class H {};
class A: public H {};
class B {};

// (1)
template <typename X>
void is_derived_from_h(X* px, H* ph) {
 std::cout << "TRUE" << std::endl;
}

// (2)
void is_derived_from_h(...) {
 std::cout << "FALSE" << std::endl;
}

int main(int argc, char* argv[]) {

 A* pa = new A;
 B* pb = new B;

 is_derived_from_h(pa, pa); // 此处走(1)，因为(1)的条件更加接近
 is_derived_from_h(pb, pb); // 走(2)，因为(1)完全不符合

 delete pa;
 delete pb;

 return EXIT_SUCCESS;
}
```

Unique_ptr

```
unique_ptr<double> p1;
unique_ptr<int> p2(new int(42));
unique_ptr<string> p1(new string("Stegosaurus"));
unique_ptr<string> p2(p1); //错误，不支持拷贝
unique_ptr<string> p3;
p3 = p2; //错误，不支持赋值

unique<T> u1; //空unique_ptr，可以指向类型为T的对象
unique<T, D> u2;
unique<T, D> u(d);

u = nullptr; //释放u指向的对象，将u置为空
T* p = u.release(); //u 放弃对指针的控制权，返回指针，并将u置为空， 返回一个裸指针，不会释放。
//所以不能不接受返回值。

u.reset(); //释放u指向的对象
u.reset(q);
u.reset(nullptr);

unique_ptr<string> p2(p1.release()); //p1--->p2 移交控制权
unique_ptr<string> p3(new string("Trex"));
p2.reset(p3.release()); //reset会释放p2原来指向的内存

//可以拷贝或赋值一个将要被销毁的unique_ptr,执行一种特殊的拷贝
unique_ptr<int> clone(int p) {
    return unique_ptr<int>(new int(p));
}

unique_ptr<int> clone(int p) {
    unique_ptr<int> ret(new int(p));
    return ret;
}
```

删除器

```
void f(destination& d) {
    connection c = connect(&d);
    unique_ptr<connection, decltype(end_connection)*>
        p(&c, end_connection);
}

/*
decltype 表明返回括号里的类型，如：
const int ci = 0, &cj = ci;
decltype(ci) x=0;
decltype (ci) x = 0; //x的类型是 const int
decltype (cj) y = x;  //y的类型是 const int&,y绑定到变量x
decltype (cj) z // 错误：z是一个引用，必须初始化
```

> **为什么share_ptr没有release成员?**
>
> 不支持将实现智能指针向非智能指针的转换。原因不知道。可能是不安全吧。

weak_ptr

不控制所指对象生存期的智能指针，指向一个由shard_ptr管理的对象，不会改变shared_ptr的引用计数，用来防止循环引用。

```
weak_ptr<T> w;
weak_ptr<T> w(sp); //与shared_ptr sp指向相同对象的weak_ptr
weak_ptr<T> w2(w); //也可以拷贝构造
w = p; //p可以是一个shared_ptr或一个weak_ptr
w.reset(); //将w置为空
w.use_count();
w.expired(); //若use_count为0，返回true，否则返回 false
w.lock(); //如果expired 为true，返回一个空shared_ptr;否则返回一个指向w的对象的shared_ptr
```

> 由于对象可能不存在，我们不能使用 weak ptr 直接访问对象，而必须调用 lock。此函数检查 weak_ ptr 指向的对象是否仍存在。如果存在，1ock 返回一个指向共享对象的 shared_ptr。与任何其他 shared_ptr 类似，只要此 shared_ptr 存在，它所指向的底层对象也就会一直存在。例如：
>
> ```
> if (shared_ptr<int> np = wp.lock()) {
> }
> ```

资源管理

RAII

利用栈对象的生命周期管理堆资源。智能指针是一个类，传入原生的裸指针，由智能指针类创建的对象析构时，该指针指向的空间会被自动释放。

> **资源管理RAII(Resource Acquisition Is Initialization)**技术，c++之父Bjarne Stroustrup提出。
>
> 特征：
>
> 1. 在构造函数中初始化资源或托管资源；
> 2. 在析构函数中释放资源；
> 3. 提供若干访问资源的方法；
> 4. 不允许复制与复制（对象语义）：将拷贝函数与赋值运算符函数删除或者设置为私有的；
>
> 本质：利用栈对象的生命周期进行管理资源。

一个自制智能指针的案例

```cpp
#include <iostream>

using std::cout;
using std::endl;

class Point {
public:
    Point(int ix = 0, int iy = 0)
        : _ix(ix), _iy(iy) {
        cout << "Point(int = 0, int = 0)" << endl;
    }

    void print() const {
        cout << "(" << _ix
             << ", " << _iy
             << ")";
    }

    ~Point() {
        cout << "~Point()" << endl;
```

```
21      }
22      friend std::ostream& operator<<(std::ostream& os, const Point&
    rhs);
23
24  private:
25      int _ix;
26      int _iy;
27  };
28
29  std::ostream& operator<<(std::ostream& os, const Point& rhs) {
30      os << "(" << rhs._ix
31         << ", " << rhs._iy
32         << ")";
33      return os;
34  }
35
36  //模板相关的概念
37  template <typename T>
38  class RAII {
39  public:
40      //在构造函数中初始化资源
41      RAII(T* data)
42          : _data(data) {
43          cout << "RAII(T *)" << endl;
44      }
45
46      //在析构函数中释放资源
47      ~RAII() {
48          cout << "~RAII()" << endl;
49          if (_data) {
50              delete _data;
51              _data = nullptr;
52          }
53      }
54
55      //提供若干访问资源的方法
56      T* operator->() { return _data; }
57      T& operator*() { return *_data; }
58      T* get() { return _data; }
59
60      void reset(T* data) {
61          if (_data) {
```

```
62              delete _data;
63              _data = nullptr;
64          }
65
66          _data = data;
67      }
68
69      //不允许复制或者赋值
70      RAII(const RAII& rhs) = delete;
71      RAII& operator=(const RAII& rhs) = delete;
72
73  private:
74      T* _data;
75  };
76
77  int main(int argc, char** argv) {
78      RAII<int> raii(new int(10));  //栈对象的生命周期
79
80      cout << endl;
81      RAII<Point> pt(new Point(1, 3));  //栈对象的生命周期
82      pt->print();
83      (*pt).print();
84      pt.get();
85      pt.reset(new Point(1, 4));
86
87      cout << endl;
88      RAII<Point> pt2 = pt;  // error
89      return 0;
90  }
```

4. 课后疑问

无。

5. 其他意见或建议（没有不填）

分享条件比较恶劣，会议室又闷又热没有网，只能围坐在一块看一个屏幕。

6. 组长总结（2022.06.04）

很认真很好很不错。准备的很充分。