```cpp
//二：unique_ptr常用操作
//（2.1）unique_ptr不支持的操作 （独占式）
//unique_ptr<string> ps1(new string("I Love China!"));
//unique_ptr<string> ps2(ps1); //该智能指针不支持拷贝动作（定义时初始化）
//unique_ptr<string> ps3 = ps1;  //该智能指针不支持拷贝动作（定义时初始化）
//unique_ptr<string> ps4;
//ps4 = ps1; //独占式智能指针不支持赋值操作

//（2.2）移动语义
unique_ptr<string> ps1(new  string("I Love China!"));
unique_ptr<string> ps2 = std::move(ps1); //移动完后，ps1为空，ps2指向原来ps1所指；

//（2.3）release()：放弃对指针的控制权（切断了智能指针和其所指向的对象之间的联系）。
    //返回裸指针，将该智能指针置空。 返回的这个裸指针我们可以手工delete来释放，也可以用来初始化另外一个智能指针，或者给另外一个智能指针赋值
unique_ptr<string> ps1(new  string("I Love China!"));
unique_ptr<string> ps2(ps1.release());
if (ps1 == nullptr)
{
    cout << "ps1被置空" << endl;
}
//ps2.release();  //导致内存泄漏
//string *tempp = ps2.release();  //auto tempp = p2.release();
//delete tempp; //人工（手工）delete释放

//（2.4）reset()
//reset()不带参数情况：释放 智能指针所指向的对象，并将智能指针置空。
unique_ptr<string> ps1(new  string("I Love China1!"));
/*ps1.reset();
if (ps1 == nullptr)
{
    cout << "ps1被置空" << endl;
}*/
//reset()带参数的情况：释放 智能指针所指向的对象，并让该智能指针指向新对象
unique_ptr<string> ps2(new string("I Love China2!"));
ps1.reset(ps2.release());  //reset释放ps1指向的对象内存，让ps1指向ps2所指向的内存，同时ps2被置空

ps1.reset(new string("I Love China3!"));
```

需要注意的是：我们常常需要对动态数组中的某一个元素进行操作，但shared_ptr没有提供[]操作符。

不过我们可以使用 sp.get()先获取原始指针，再对原始指针进行下标操作。

而unique_ptr对动态数组提供了支持，指定删除器是一个可选项。也可以直接使用下标操作：

```
//（2.5） = nullptr: 释放智能指针所指向的对象，并将智能指针置空
//unique_ptr<string> ps1(new string("I Love China!"));
//ps1 = nullptr; //释放ps1所指向的对象，并将ps1置空

//(2.6) 指向一个数组
//unique_ptr<int[]> ptrarray(new int[10]); //注意，数组这里要跟上[]
//ptrarray[0] = 12;
//ptrarray[1] = 24;
//这里不要忘记A[]，否则如果有自己的析构函数，则会报异常
//unique_ptr<A[]> ptrarray(new A[10]); //vector,string ,
```

```
//（2.7）get() ：返回智能指针中保存的裸指针
//考虑到有些函数参数需要的是内置裸指针，所以引入该函数；
unique_ptr<string> ps1(new string("I Love China!"));
string *ps = ps1.get();
*ps = "This is a test!";
//delete ps; //不要这么干，否则产生不可预料的后果；
```

```
//（ 2.9）swap(): 交换两个智能指针所指向的对象；
unique_ptr<string> ps1(new string("I Love China1!"));
unique_ptr<string> ps2(new string("I Love China2!"));
std::swap(ps1, ps2);
ps1.swap(ps2);
```

```
//unique_ptr<int[]> pt2(new int[10]); //对于定义的内容是数组，是没有*解引用运算符的；
```

```
//（2.10）智能指针名字作为判断条件
unique_ptr<string> ps1(new string("I Love China1!"));
if (ps1)  //if(ps1 != nullptr)
{
```

```
//(2.11)转换成shared_ptr类型：如果unique_ptr为右值，就可以将它赋值给shared_ptr
//因为shared_ptr包含一个显式构造函数，可用于将右值unqiue_ptr转换为shared_ptr，shared_ptr将接管原来归unique_ptr所拥有的对象；
```

```
unique_ptr<string> ps(new string("I Love China"));
shared_ptr<string> pss1 = std::move(ps); //左值转右值 ，执行后ps为空，pss1就是shared_ptr
```

```
unique_ptr<string> tuniqp()
{
    //unique_ptr<string> pr(new string("I Love China!"));
    //return pr; //从函数返回一个局部的unique_ptr对象。三章十四节讲过移动构造函数
                 //返回这种局部对象，导致系统给我们生成一个临时unique_ptr对象，调用unique_ptr的移动构造函数。
    return unique_ptr<string>(new string("I Love China!"));
}
```

实际上是接管匿名的移动构造对象

```
int main()
{
    //一：返回unique_ptr    :
    //虽然unique_ptr智能指针不能拷贝，但是，当这个unique_ptr将要被销毁，是可以拷贝的。最常见用法就是从函数返回一个unique_ptr
    unique_ptr<string>ps;
    ps = tuniqp(); //可以用ps来接，则临时对象直接构造在ps里，如果不接，则临时对象会被释放，同时会释放掉所指向的对象的内存
```

```
void mydeleter(string *pdel)
{
    delete pdel;
    pdel = nullptr;
    //这里可以打印一下日志。
}
```

一、decltype意义

有时我们希望从表达式的类型推断出要定义的变量类型，但是不想用该表达式的值初始化变量（如果要初始化就用auto了）。为了满足这一需求，C++11新标准引入了decltype类型说明符，它的作用是选择并返回操作数的数据类型，在此过程中，编译器分析表达式并得到它的类型，却不实际计算表达式的值。

```
    int tempA = 2;

    /*1.dclTempA为int*/
    decltype(tempA) dclTempA;
    /*2.dclTempB为int，对于getSize根本没有定义，但是程序依旧正常，因为decltype只做分析，并不调用getSize，*/
    decltype(getSize()) dclTempB;

    return 0;
}
```

```
//二：指定删除器 ，delete:默认删除器：
//a)指定删除器
//格式： unique_ptr<指向的对象类型,删除器的类型> 智能指针变量名：
//删除器，可调用对象,比如函数，类重载了()。
//我们学习过了shared_ptr删除器,比较简单 shared_ptr<int> p(new int(),mydelete);
//unique_ptr删除器相对复杂一点，多了一步，先要在类型模板参数中传递进去类型名，然后在参数中再给具体的删除其函数名；
typedef void(*fp)(string *); //定义一个函数指针类型，类型名为fp
unique_ptr<string, fp> ps1(new string("I Love China!"), mydeleter);
```

```
using fp2 = void(*)(string *);  //用using定义一个函数指针类型，类型名为fp2
unique_ptr<string, fp2> ps1(new string("I Love China!"), mydeleter);
```

```
typedef decltype(mydeleter)* fp3; //这里多了一个*,因为decltype返回的是函数类型void(string *)
                                  //加*表示函数指针类型，现在fp3应该 void *(string *);
unique_ptr<string, fp3> ps1(new string("I Love China!"), mydeleter);
```

```
//a.5)用lambda表达式看看写法,lambda表达式可以理解成带有operator()类类型对象。 //把lambda表达式理解成一个class;
//所以  decltype(mydella)  = class{.....}
auto mydella = [](string *pdel) {
    delete pdel;
    pdel = nullptr;
    //可以打印日志
};
unique_ptr<string, decltype(mydella)> ps5(new string("I Love China!"), mydella);
```

```
//b)指定删除器额外说明
//shared_ptr:就算两个shared_ptr指定的删除器不相同，只要他们所指向的对象相同，那么这两个shared_ptr也属于同一个类型。
//但是unique_ptr不一样，指定unique_ptr中的删除器会影响unique_ptr的类型,所以从灵活性来讲，shared_ptr设计的更灵活；
//咱们在讲解shared_ptr的时候，删除器不同，但指向类型一样的shared_ptr，可以放到同一个容器里，vector<shared_ptr...
//unique_ptr如果，删除器不同，那么久等于整个unique_ptr类型不同。这种类型不同的unique_ptr智能指针是没有办法放到同一个容器里的；
```

```
//三：尺寸问题：通常情况下，unique_ptr尺寸跟裸指针一样：
//string *p;
//int ilenp = sizeof(p); //4字节
//unique_ptr<string> ps1(new string("I Love China!"));
//int ilen = sizeof(ps1); //4字节
//如果你增加了自己的删除器，则unique_ptr的尺寸可能增加，也可能不增加
//a)如果lambda表达式这种删除器，尺寸就没变化
```

```
//增加字节对效率有影响，所以自定义删除器要慎用；
//shared_ptr，不管你指定什么删除器，shared_ptr的尺寸(大小)都是裸指针的2倍；
```