C++ 11中的智能指针

导语：

初入Android开发领域，但在过往的开发经历中没有接触过Java使用JNI调用C++，企业微信项目中微盘、微文档模块大量使用了C++智能指针。谨以此文作为记录学习

动态内存

要了解智能指针，首先要了解C++中的动态内存，这里主要介绍两个部分：

- 栈(stack)：函数调用过程中产生的局部变量和函数参数的内存区域。满足后进先出，离开作用域后自动释放掉内存
- 堆(heap)：由程序员动态分配内存，需要手动释放，否则会造成内存泄漏

举一个简单的例子直观感受一下：

```
class Person{
    public:
        Person(string name, int age) : mName(name), mAge(age){
            cout <<"Person()"<<endl;
        }

        ~Person(){
            cout <<"~Person()"<<endl;
        }

        string toString(){
            return "name:" + mName + "age:" + to_string(mAge);
        }

    private:
        string mName;
        int mAge;
};
```

```
int main() {
    Person person("ton",10);
    cout <<person.toString()<<endl;
    return 0;
    //Person()
    //name:ton,age:10
    //~Person()
}
```

```
int main() {
    Person * person = new Person("ton",10);
    cout <<person->toString()<<endl;
    return 0;
    //Person()
    //name:ton,age:10
}
```

可以看到，通过new关键字动态分配内存，在离开main函数作用域后并不会自动释放，需要手动**delete person**。实际上在函数内**没有特殊需求下**应尽量使用栈内存分配。然而在实际项目中堆内存的使用还是十分普遍的。

为什么要使用堆内存?

- 栈空间相对堆空间很小
- 有时候并不希望某一块内存在离开作用域后就被释放
- 因为栈的特性，释放次序是分配次序的反序,无法自定义顺序

总之，使用堆内存分配，给了开发者更大的灵活性，同时也容易导致内存泄漏问题。

为什么要引入智能指针

首先看看普通指针的使用过程中会存在什么问题

1. 当有多个指针指向同一块内存，其中一个指针指向的内存被释放，其他指针成为了悬挂指针，访问内存的时候就会出现无法预测的结果。其次，谁来释放也是一个问题，当有多个指针指向同一块内存，应该是最后一个访问该内存的代码块来释放。

   ```
   void fun1(Person *p) {
       p->toString();
       delete p;
   }

   void fun2(Person *p) {
       cout << p->toString() << endl;
       delete p;
   }
   int main() {
       Person *person = new Person("tom", 10);
       fun1(person);
       //...
       fun2(person);//无法预测的结果
       cout << person->toString() << endl;//无法预测的结果
       return 0;
   }
   ```

2. delete并非总能按理想情况执行

```
int main() {
    Person *person = new Person("tom", 10);
    //....
    delete person;
    return 0;
}
```

从分配内存到释放内存中间这一段程序，可能会提前return，或者发生异常。这两种情况下都不会正常执行到delete。另外，直接忘记delete也是有可能的。

以上问题通过智能指针可以轻松的解决了,C++中存在auto_ptr、unique_ptr、shared_ptr、weak_ptr四种智能指针。尽管名为智能指针，其实是一个类，包装裸指针后实现自动释放动态分配的对象。

auto_ptr

C++98提供第一种智能指针，auto_ptr。

创建方式：

```
auto_ptr<Person> person(new Person("tom",10));
```

auto-ptr在离开作用域后自动释放内存，省去了delete操作。即使在正常离开作用域之前有异常发生,仍然不会影响它释放内存。

```
void test(){
    cout <<"test throw exception"<<endl;
    throw "exception";
}
int main() {
    auto_ptr<Person> person(new Person("tom",10));
    try{
       test();
    } catch (...) {
    }
    return 0;
}
//Person()
//test throw exception
//~Person()
```

auto_ptr解决了上述第二个问题，面对第一个问题的时候仍然存在问题。anto_ptr创建时会拥有所有权，当把auto_ptr1赋给另一个auto_ptr2。所有权就发生了转移,auto_ptr失去所有权，再次访问auto_ptr关联的内存就会发生异常。

Case1:

```
int main() {
    auto_ptr<Person> person(new Person("tom",10));
    auto_ptr<Person> person1 = person;
    cout <<person->toString()<<endl;
    return 0;
    //Person() 没有执行析构函数
    //Process finished with exit code 11
}
```

Case2:

```
void print(auto_ptr<Person> p) {
    cout << p->toString() << endl;
}
int main() {
    auto_ptr<Person> person(new Person("tom",10));
    print(person);
    cout <<person->toString()<<endl;
    return 0;
}
```

Case2和case1发生了同样的问题（exit code 11），因为print函数的参数是person的copy，同样发生了一次所有权转移。但person的析构函数是正常执行了的，p离开print函数作用域自动释放内存。

此外，auto_ptr不能作为STL容器元素，不能指向一组对象。基于这些原因，在C++11中又推出了unique_ptr、shared_ptr、weak_ptr三种智能指针。

Unique_ptr

创建方式：

```
unique_ptr<Person> p(new Person("tom",10));
unique_ptr<Person> person = make_unique<Person>(Person("tom",10));//推荐，更高效
```

作为auto_ptr的继任者，和auto_ptr最大的区别是它的所有权是独占的，不能赋值的方式转移所有权，如下是错误的。

```
unique_ptr<Person> p1 = make_unique<Person>(Person("sad",10));
unique_ptr<Person> p2 = p1; //error
```

这在一定程度上减少了auto_ptr因为所有权产生的问题，但unique仍然提供了转移所有权的方式,因此所有权转移后仍有可能发生auto_ptr的错误。

```
unique_ptr<Person> p1 = make_unique<Person>(Person("sad",10));
unique_ptr<Person> p2 = move(p1); //move转移所有权
p2->toString();
p1->toString();//错误，p1所有权已转移
```

```
unique_ptr<Person> p1 = make_unique<Person>(Person("sad",10));
unique_ptr<Person> p2(p1.release());//释放所有权，并返回指针。转移所有权
cout <<p2->toString();
return 0;
```

```
unique_ptr<Person> p1 (new Person("sad",10));
p1.reset(); // 释放所有权并且释放资源
p1->toString();//error

p1 = nullptr;// 等效于reset
```

unique_ptr还支持管理动态数组，离开作用域时，用delete[]代替delete

```
unique_ptr<int[]> p1 (new int[3]);
```

另外，由于unique_ptr不可拷贝和赋值的特性，当函数参数为unique_ptr时，不能直接传参，要指定函数参数为引用类型。

```
void print(unique_ptr<Person> &person) {
    cout << person->toString() << endl;
}
int main() {
    unique_ptr<Person> p1 (new Person("sad",10));
    print(p1);
    return 0;
}
```

Shared_ptr

上面说到，unique_ptr和auto_ptr在所有权问题上都存在问题。shared_ptr则真正解决了这个问题。

Shared_ptr采用引用计数的方式,允许同一块内存有多个引用，即可以理解多个shared_ptr可以共享所有权。通过拷贝赋值等方式，引用计数增加，当引用计数为0时才释放内存。

创建方式：

```
shared_ptr<Person> p = make_shared<Person>(Person("tom",10));//推荐
shared_ptr<Person> p1(new Person("tom",10));
cout <<p.use_count()<<endl;//use_count获取当前引用计数
```

shared_ptr很好的解决了所有权问题，以下程序运行正确

```
shared_ptr<Person> p = make_shared<Person>(Person("tom",10));
shared_ptr<Person> p1 = p;
shared_ptr<Person> p2(p1);
cout <<p.use_count()<<endl;//3
return 0;
```

在加上一个函数调用试试，依然运行正确

```
void print(shared_ptr<Person> person) {
    cout << person->toString() << endl;
    cout <<"函数体内，当前引用计数:"<<person.use_count()<<endl;
}
int main() {
    shared_ptr<Person> p = make_shared<Person>(Person("tom",10));
    shared_ptr<Person> p1 = p;
    shared_ptr<Person> p2(p1);
    print(p); //引用计数：4
    cout <<p.use_count()<<endl;//引用计数:3，print函数结束，copy的p指针离开函数作用域，引用计数减1
    return 0;
}
```

shared_ptr同样支持管理动态数组，与unique_ptr不同的是，析构是默认是使用的delete。需要自己指定析构策略。

```
shared_ptr<Person> p(new Person[4], [](Person *p) {
    delete[] p;
});
return 0;
```

shared_ptr的其他API

```
shared_ptr<Person> p = make_shared<Person>(Person("tom",10));
shared_ptr<Person> p1 = p;
shared_ptr<Person> p2(p1);
p1.reset(); //引用计数-1 ，p1不能再使用，其他shared_ptr正常使用
cout <<p.use_count()<<endl; //2
cout <<p1->toString()<<endl; //error

Person *person = p.get(); //获取真正的指针
cout <<person->toString()<<endl;

bool unique = p.unique(); //是否唯一引用
cout << unique << endl; //0
```

```
bool isnull = p.operator bool(); //是否指向一个内存，用于判空
cout << isnull << endl; //1
```

使用注意事项

1.尽量不要使用原始指针初始化智能指针，原始指针delete之后，智能指针再去访问产生异常。

```
Person *person = new Person("tom",10);
shared_ptr<Person> p(person);
delete person;
p->toString(); //error
```

2.函数传参时不要直接创建shared_ptr，因使用创建好的

3.避免循环引用

shared_ptr看上去已经足够智能，为什么还会出现另一种weak_ptr呢，原因与上述第3点有关，shared_ptr使用不注意循环引用，最终导致内存泄漏。

Weak_ptr

一个shared_ptr循环引用的例子:

```
class A {
public:
    A() { cout << "A()" << endl; }

    ~A() { cout << "~A()" << endl; }

    shared_ptr<B> m_spb;
};
```

```
class B {
public:
    B() { cout << "B()" << endl; }

    ~B() { cout << "~B()" << endl; }

    shared_ptr<A> m_spa;
};
```

```
int main() {
    shared_ptr<A> spa(new A);
    shared_ptr<B> spb(new B);
    spa->m_spb = spb; //spb.use_count = 2
    spb->m_spa = spa; //spa.use_count = 2;
    return 0;
    //A()
    //B()
}
```

可以看到，在程序结束后，A和B都没有正确的析构。因为发生了循环引用。函数结束前，A对象和B对象的引用计数都是2，函数结束后，A和和B的的引用计数是1，导致内存泄漏。原因是虽然spa和spb离开作用域，引用计数减1，但是A被B强引用，B被A强引用。A要析构需要引用计数变为0，即需要B析构。同理，B也需要A先析构。结果就是A，B都无法析构。

weak_ptr的出现解决了这个问题。

和其他智能指针不同，weak_ptr不能直接创建。需要从shared_ptr创建。

```
shared_ptr<A> spa(new A); //1个强引用计数
weak_ptr<A> wp1(spa); //1个强引用计数，1个弱引用计数
weak_ptr<A> wp2 = wp1; //1个强引用计数，2个弱引用计数
cout << wp2.use_count() << endl;//返回强引用计数1
//A()
//1
//~A()
```

只有强引用计数才作为释放的依据。weak_ptr只会增加弱引用计数，当强引用计数变为0时，weak_ptr就过期了通过expired()判断。weak_ptr不能直接使用，可以通过lock函数得到对应的shared_ptr。此时强引用计数会+1，这一切的前提weak_ptr没有过期

用weak_ptr解决循环引用

把A，B的成员类型都改为weak_ptr类型

```
class B {
public:
    B() { cout << "B()" << endl; }

    ~B() { cout << "~B()" << endl; }

    weak_ptr<A> m_wpa;

    void test() {
        cout << "B class" << endl;
    }
};
```

```
class A{
public:
    A() { cout << "A()" << endl; }

    ~A() { cout << "~A()" << endl; }

    weak_ptr<B> m_wpb;

    void printB(){
        if (!m_wpb.expired()){
            m_wpb.lock()->test();
        }
    }
};
```

```
shared_ptr<A> spa(new A);
shared_ptr<B> spb(new B);
spa->m_wpb = spb;
spb->m_wpa = spa;
spa->printB();
return 0;
//A()
//B()
//B class
//~B()
//~A()
```

A、B都争取析构了。尽管在printB函数内，对B的引用计数+1了，等于2。离开函数作用域后又变为1。
A、B间不存在相互的强引用，解开了循环引用了。