```
//三：lambda表达式延迟调用易出错细节分析
//int x = 5;
////auto f = [=] //当遇到auto这一行，也就是在捕获的这个时刻，x的值就已经被赋值到了这个f中了。
//auto f = [&]
//{
//  return x;
//};
//x = 10;
//cout << f() << endl; //我们认为是10但实际是5
//也就是说，凡是按值捕获的外部变量，在lambda表达式定义的这个时刻，所有这些外部变量值就被复制了一份存储在lambda表达式变量中。
//解决办法就是按引用绑定
```

```
//解决办法就是按引用绑定      //auto f = [&]
```

```
std::vector<std::function<bool(int)>> gv; //全局变量，每个元素都是个function，每个function给进去的int,返回的是
void myfunc()
{
    srand((unsigned)time(NULL));
    int tmpvalue = rand() % 6; //产生一个0-5之间的随机数
    gv.push_back(
        [&](int tv) {
        if (tv %tmpvalue == 0) //如果tv是tmpvalue的倍数
        {
            return true;
        }
        return false;
    });
}
```

```
int main()
{
    //一：捕获列表中的&：捕获外部作用域中所有变量，并作为引用在lambda表达式中使用；
    //按照引用这种捕获方式，会导致lambda表达式包含绑定到局部变量的引用
    myfunc();
    cout << gv[0](10) << endl; //非法调用，会出现不可预料的问题；

    //二：形参列表可以使用auto
    //c++14允许在lambda表达式的形参列表中使用auto;
    //引用捕获方式超出范围的情形也叫做“引用悬空”
```

```
//三：成员变量的捕获问题
AT *pat = new AT();
pat->addItem();
delete pat;
cout << gv[0](10) << endl; //7,0
//结论：lambda表达式的执行正确与否，取决于pat对象是否存在，只有pat对象存在，这个lambda表达式执行才正确；
```

我们大家首先要明确一点：捕获这个概念，只针对于在创建lambda表达式的作用域内可见的 非静态 局部变量（包括形参）；

```
gv.push_back(
    //=是按值捕获的意思；
    //我们会认为这个[=]是按值捕获，使用我们能够访问成员变量m_tmpvalue
    //所以我们顺理成章的认为，这里这个lambda表达式是所使用的m_tmpvalue是按值捕获的；
```

```
//m_tmpvalue并不是非静态局部变量，m_tmpvalue是A1类的成员变量，成员变量是不能被捕获到的；
//this:指向对象本身；  所以这里你用[=]捕捉的是this指针值；
```

```
//[=](auto tv) {    //=就等价于有this    ,用了=,  用了&就默认等于添加了this这里的[=]就等价于[this]
[this](auto tv) {
cout << m_tmpvalue << endl; //this->m_tmpvalue
if (tv % m_tmpvalue == 0)  //this->m_tmpvalue
{
    return true;
}
```

```
//五：静态局部变量：捕获这种事，是不包括静态局部变量的，也就是说，静态局部变量是不能被捕获的，
//但是可以在lambda中使用；另外，静态局部变量是保存在静态存储区，它的有效期一直到程序结束。
//但是这种对static变量的使用有点类似于按 引用  捕获这种效果；
```

```
//四：广义lambda捕获 :c++14 : generalized lambda capture:
```

```
gv.push_back(
    [abc = m_tmpvalue](auto tv) { //将m_tmpvalue复制到闭包里来
    cout << abc << endl;
    if (tv % abc == 0)
    {
        return true;
    }
    return false;
});
```