A.2 删除函数

有时让类去做拷贝是没有意义的。 `std::mutex` 就是一个例子——拷贝一个互斥量，意义何在？ `std::unique_lock<>` 是另一个例子——一个实例只能拥有一个锁；如果要复制，拷贝的那个实例也能获取相同的锁，这样 `std::unique_lock<>` 就没有存在的意义了。实例中转移所有权(A.1.2节)是有意义的，其并不是使用的拷贝。当然其他例子就不一一列举了。

通常为了避免进行拷贝操作，会将拷贝构造函数和拷贝赋值操作符声明为私有成员，并且不进行实现。如果对实例进行拷贝，将会引起编译错误；如果有其他成员函数或友元函数想要拷贝一个实例，那将会引起链接错误(因为缺少实现)：

```
class no_copies
{
public:
  no_copies(){}
private:
  no_copies(no_copies const&);  // 无实现
  no_copies& operator=(no_copies const&);  // 无实现
};

no_copies a;
no_copies b(a);  // 编译错误
```

在C++11中，委员会意识到这种情况，但是没有意识到其会带来攻击性。因此，委员会提供了更多的通用机制：可以通过添加 `= delete` 将一个函数声明为删除函数。

no_copise类就可以写为：

```
class no_copies
{
public:
  no_copies(){}
  no_copies(no_copies const&) = delete;
  no_copies& operator=(no_copies const&) = delete;
};
```

这样的描述要比之前的代码更加清晰。也允许编译器提供更多的错误信息描述，当成员函数想要执行拷贝操作的时候，可将连接错误转移到编译时。

拷贝构造和拷贝赋值操作删除后，需要显式写一个移动构造函数和移动赋值操作符，与 `std::thread` 和 `std::unique_lock<>` 一样，你的类是只移动的。

下面清单中的例子，就展示了一个只移动的类。

清单A.2 只移动类

```
class move_only
{
  std::unique_ptr<my_class> data;
public:
  move_only(const move_only&) = delete;
  move_only(move_only&& other):
    data(std::move(other.data))
  {}
  move_only& operator=(const move_only&) = delete;
  move_only& operator=(move_only&& other)
  {
    data=std::move(other.data);
    return *this;
  }
};

move_only m1;
move_only m2(m1);  // 错误，拷贝构造声明为“已删除”
move_only m3(std::move(m1));  // OK，找到移动构造函数
```

只移动对象可以作为函数的参数进行传递，并且从函数中返回，不过当想要移动左值，通常需要显式的使用 `std::move()` 或 `static_cast<T&&>` 。

可以为任意函数添加 `= delete` 说明符，添加后就说明这些函数是不能使用的。当然，还可以用于很多的地方；删除函数可以以正常的方式参与重载解析，并且如果被使用只会引起编译错误。这种方式可以用来删除特定的重载。比如，当函数以short作为参数，为了避免扩展为int类型，可以写出重载函数(以int为参数)的声明，然后添加删除说明符：

```
void foo(short);
void foo(int) = delete;
```

现在，任何向foo函数传递int类型参数都会产生一个编译错误，不过调用者可以显式的将其他类型转化为short：

```
1 foo(42);  // 错误，int重载声明已经删除
2 foo((short)42);  // OK
```