资源管理

在程序编写过程中需要面对大量资源的管理问题，包括内存、文件描述符、互斥锁、图形界面中的字型和笔刷、数据库连接、网络sockets。 经过训练后，基于对象的资源管理办法，几乎可以消除资源管理问题。

条款13: 以对象管理资源

- 为防止资源泄露，请使用 RAII 对象，他们在构造函数中获得资源并在析构函数中释放资源。
- std::shared_ptr 和 std::auto_ptr 是常用的 RAII class。前者是较佳选择，因为其 copy 行为比较直观，后者需要容忍复制行为。

```
class Investment {};
Investment* createInvestment();
void f() {
    Investment* pInv = createInvestment();

    delete pInv;
}
```

上面这段代码展示了通过工厂函数来提供某个特定的 Investment 对象的过程。看起来没有问题，但实际上当有人开始更改这段代 码时就会产生问题，比如有人在 delete 之前添加 当某些条件满足时 return。这将导致 delete 不会被执行，也就造成了资源 泄露的问题。

修改的方法：把资源放进对象内，依赖“析构函数自动调用机制”确保资源被释放。

```
void f() {
    std::auto_ptr<Investment> pInv(createInvestment());
    ...
}
```

这样在运行结束时，auto_ptr 的析构函数会自动删除 pInv。

这种例子展示了这个观点的两个关键想法：

- 获得资源后立即放进管理对象(RAII)
- 管理对象运用自购函数确保资源被释放

需要注意的是别让多个 auto_ptr 同时指向同一个对象。如果是那样，对象会被删除一次以上。而为了预防这个问题，auto_ptr 有 一个不寻常的性质：若通过 copy 构造函数 或 copy assignment 操作符来复制他们，他们会变成 null，而复制所得的指针将 取得资源的唯一拥有权。

```
std::auto_ptr<Investment> pInv1(createInvestment());
std::auto_ptr<Investment> pInv2(pInv1); // 现在 pInv1->nullptr
```

```
pInv1 = pInv2; // 现在 pInv2->nullptr
```

引用计数型智慧指针

有时候需要允许元素完成正常的复制行为，比如 STL 容器。在这种情况下使用 auto_ptr 就不是最佳选项了。使用 std::shared_ptr 可以允许正常的复制行为。

```
void f() {
    std::shared_ptr<Investment> pInv1(createInvestment());
    std::shared_ptr<Investment> pInv2(pInv1);
}
```

值得注意的是，两者在析构函数中做 delete 而不是 delete[] 操作。因此不要处理动态分配而得来的 array。使用 vector 或者 string 来代替他们。

本条建议不止包括使用 std::auto_ptr 和 std::shared_ptr 这些资源管理类来管理资源，还建议面向无法被这些管理类管理的 资源通过手动编写自己的资源管理类来进行管理。

条款14: 在资源管理类中小心 copying 行为

- 复制 RAII 对象必须一并复制它所管理的资源，所以资源的 copying 行为决定 RAII 对象的 copying 行为
- 普遍而常见的 RAII class copying 行为是：抑制 copying、引用计数。

在条款13中所使用的两个智能指针都旨在管理 heap-based 资源，而有时我们需要管理 non-heap-based 资源时就需要自行撰写 管理对象。

一个例子是为 C API 所提供的 Mutex 互斥对象提供资源管理对象，保证解锁的正常进行。

```
class Lock{
public:
    explicit Lock(Mutex* pm) : mutexPtr(pm) { lock(mutexPtr); }
    ~Lock() { unlock(mutexPtr); }
private:
    Mutex* mutexPtr;
};

Mutex m;
{
    Lock m1(&m); // 进入时加锁，离开此块时 m1 的析构函数自动调用解锁
}
```

看上去一起都好，但是当出现复制操作时问题就会变得复杂。

```
Lock m1(&m);
Lock m2(m1);
```

你可以有4种选择：

1. 禁止复制

许多时候对允许 RAII 对象被复制并不合理。这个时候就应该禁止这种行为的发生。条款6给出了方案，使用 private 限定 copy 或者继承含有 private copy 的对象：

```
class Lock : private Uncopyable {
public:
    ...
};
```

2. 对底层资源使用 “引用计数法”

有时候，我们希望保有资源直到它的最后一个使用者被销毁。通常使用 std::shared_ptr 来实现引用计数就可以完成这个任务。 需要注意 shared_ptr 的缺省行为是引用为0时删除对象，而我们则需要为 shared_ptr 指定一个删除器 unlock。

```
class Lock{
public:
    explicit Lock(Mutex* pm) : mutexPtr(pm, unlock) { lock(mutexPtr.get()); }
private:
    std::shared_ptr<Mutex> mutexPtr;
};
```

3. 复制底部资源

有时候需要针对一份资源拥有任意数量的附件。而资源管理类的唯一理由是，当不再需要某个附件时确保释放它。此时当发生复制时 应该使用深拷贝。

4. 转移底部资源的拥有权

某些罕见的场合你可能希望确保永远只有一个 RAII 对象指向一个未加工资源，即使 RAII 对象被复制依然如此。此时，资源的拥 有权会从被复制物转移到目标物。这是 auto_ptr 所奉行的复制意义。

条款15: 在资源管理类中提供对原始资源的访问

- APIs 往往要求访问原始资源，所以每一个 RAII class 都应该提供一个 “取得其所管理资源” 的办法。
- 对原始资源的访问可能经由 显式/隐式 转换。一般而言显式更加安全，隐式更加方便。

资源管理类能够很好的对抗资源泄露，但是有时你不得不直接访问原始资源。auto_ptr 和 shared_ptr 就提供了访问原始资源的 能力。

```
std::shared_ptr<Investment> pInv(createInvestment());
int daysHeld(const Investment* pi);
int days = daysHeld(pInv.get());
```

同时，两者也提供了指针取值操作符(-> *)。

除了这种显示转换方案，也可以通过提供隐式类型转换函数来简化。

```
FontHandle getFont();
void releaseFont(FontHandle fh);
void changeFontSize(FontHandle fh, int size);

class Font {
public:
    explicit Font(FontHandle fh) : f(fh) {}
    ~Font() { releaseFont(f); }
    operator FontHandle() const { return f; } // 隐式转换函数

private:
    FontHandle f;
};

Font f(getFont());
int newFontSize;
changeFontSize(f, newFontSize); // 这里直接使用隐式转换
```

需要注意，隐式转换会增加错误机会。这一点就要求在实现 资源管理对象 时要依据需要完成的工作，谨慎选择使用 显示 或者 隐式 的转换方案。

条款16: 成对使用 new 和 delete 时要采用相同形式

- 如果你在 new 表达式中使用 []，必须在对应的 delete 表达式中使用 []。如果 new 没有使用，delete 也一定不要使用。
- 尽量避免对数组形式做 typedef 动作。

这一点看似简单，但是也有可能出现问题。

```
typedef std::string AddressLines[4];
std::string* pal = new AddressLines;

delete pal; // 错误
delete [] pal; // 正确，因为AddressLines 等价于 string[4]，因此实际上是个数组
```

为了避免此类错误，应该尽量不要对数组形式做 typedefs 动作。

条款17: 以独立语句将 newed 对象置入智能指针

- 以独立语句将 newed 对象存储于智能指针中。如果不这样做，一旦异常被抛出，可能导致难以察觉的资源泄露。

```
int priority();
void processWidget(std::shared_ptr<Widget> pw, int priority);

processWidget(new Widget, priority());
```

上述代码的调用存在两个问题：

1. 禁止隐式转换

上述代码无法通过编译，因为 shared_ptr 构造函数是一个 explicit 的，无法进行隐式转换操作。因此上述代码 Widget* 无法 转换为 shared_ptr。

2. 可能出现资源泄露

c++ 的函数参数的调用次序不是固定的。如果顺序如下：

1. 执行 new Widget
2. 调用 priority()
3. 调用 std::shared_ptr()

且第二部产生异常，那么程序就不得不跳出。而此时，new 已经申请了资源，但并没有放入到 shared_ptr。因此不会自动回收，那 也就意味着资源泄露。

想要解决这两个问题，就直接将这两步进行拆分。

```
std::shared_ptr<Widget> pw(new Widget);
processWidget(pw, priority());
```

这样就避免了发生隐式转换与跨域语句的操作。