多线程

# 1：线程同步

答：

关于在多线程中使用共享资源的常用技术；

## 线程同步：

当一个线程执行递增和递减操作时，其他线程需要依次等待的行为；

### 如何实现线程同步：

如果不需要共享对象，就不需要进行线程同步的操作；

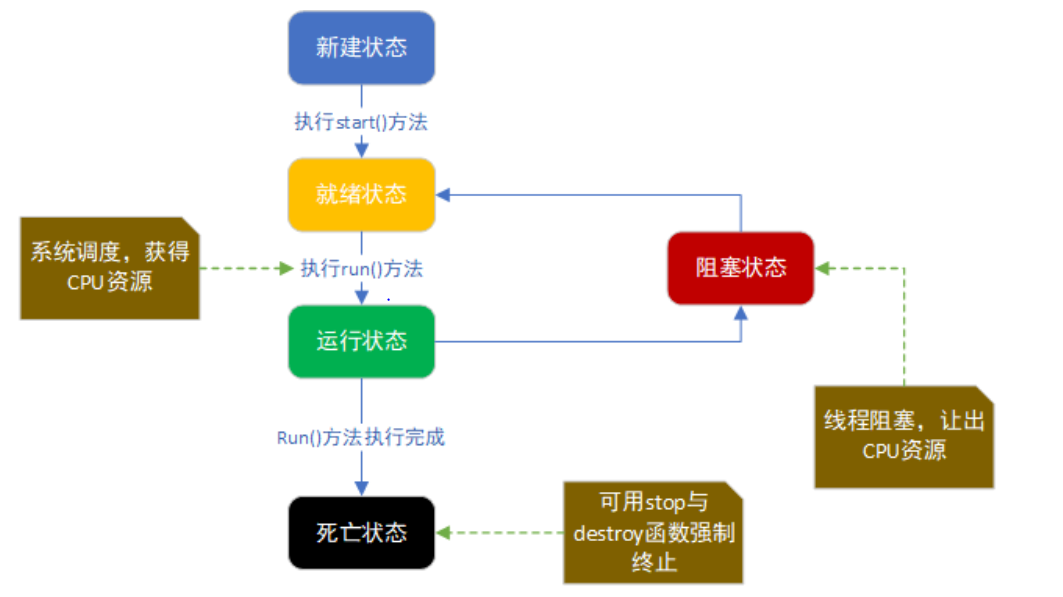
* 只使用原子操作：

意思是一个操作只占用一个量子时间，即一次就可以完成。所以只有当现在这个操作完成之后，其他的线程才能执行其他的操作，因此其他的线程就不需要等待这个操作完成，这样就避免了使用锁，也不会造成死锁。

* 内核模式（切换线程：等待->阻塞->执行）：

将等待线程置为阻塞状态，当线程处于阻塞状态时，只会尽可能少的占用CPU的时间。但是，这样就代表了需要进行至少一次的上下文切换（操作系统的线程调度器）。调度器会保存等待的线程状态，并切换到另一个线程，依次恢复等待的线程状态。这中操作需要消耗很多的资源，但是如果等待时间长的话也会比较值得。

只有操作系统的内核才能阻止线程使用CPU时间。



* 用户模式：

线程一直等待，这种方式只会在等待时浪费CPU时间，减少了上下文切换花费的时间。

好处：时非常轻量，速度快。

坏处：如果等待时间长就会耗费大量的CPU时间。

* 混合模式：

先尝试使用用户模式进行等待，如果等待的时间长，就会使用内核模式进行等待。

## 执行基本的原子操作

通过Interlocked类来实现原子操作。Interlocked类提供了递增，递减，相加等基本的数学操作的原子方法，从而不需要使用锁就可以达到线程同步

## Mutex类

使用mutex类来同步两个单独的程序。Mutex是一种原始的同步方式，只对一个线程赋予独占访问共享资源的权利。

启动主程序时，需要定义一个指定了名称的互斥量。并且设置initialOwner标志为false（false表示互斥量已经被创建，允许程序获取这个名称的互斥量）

注：互斥量是一个全局的操作系统对象，需要正确的关闭互斥量

## SemaphoreSlim类

这个类限制了同时访问同一个资源的线程数目。如果同时访问的线程数目超过了指定的数目，其余的线程就需要等待。

SemaphoreSlim semaphoreSlim = new SemaphoreSlim(4); //参数就表示同时等够又多少个线程访问资源

semaphoreSlim.Wait(); //来等待进入资源的机会，如果已经有指定数目个线程在访问资源

semaphoreSlim.Release(); //线程操作完成之后释放，等待的线程才能进入

注：由于semaphoreslim类并没有使用windows内核信号量，也不支持进程间的同步。如果要进程间同步，就要使用semaphore这个类（创建一个指定名字的对象，就像mutex一样，可以在不同的程序中同步线程）

## AutoResetEvent类

从一个线程向另一个线程发送通知。这个类可以通知等待的线程有某事件发生。

AutoResetEvent workerEvent = new AutoResetEvent(false); //构造方法传入false:初始状态为unsignaled，表示任何线程调用这两个对象中的任何一个的waitone()都将会被阻塞，直到调用了set()；如果传入true：初始状态为signaled，表示如果线程调用waitone()会被立即执行，然后状态自动变为unsignaled，所以需要再对该实例调用set()方法，以便让其他的线程使用该实例的waitone()继续执行。

AutoResetEvent类使用的是内核时间模式，所以等待时间不能太长。

## ManualResetEventSlim类

在线程间传递信号。

//若要将初始状态设置为终止，则为 true；若要将初始状态设置为非终止，则为 false。

ManualResetEventSlim mainEvent1 = new ManualResetEventSlim(false);

mainEvent1.Set(); //将对象设置为有信号，表示其余等待的线程可以执行；简单的理解为打开了大门，允许通过

mainEvent1.Reset(); //将对象设置为非终止的状态，表示其余等待线程需要继续等待；简单的理解为关上了大门，不允许通过

## CountDownEvent类

等待一定数量的线程完成后继续执行。

CountDownEvent countdown = new CountDown(2); //参数表示要完成2次操作才会从等待的状态中退出。

Countdown.Signal(); //操作完成发出信号

Countdown.Wait(); //等待操作完成

优点：针对需要等待多个异步操作完成的情况，这个类会非常轻便。

缺点：如果调用Signal()方法没有达到规定的次数，这个类将一直阻塞等待。

## Barrier类

用于组织多个线程及时在某个时刻碰面。该类提供了一个回调函数，每次线程调用了SignalAndWait()后该回调函数会被执行。

Barrier barier = new Barrier(2,b=>WriteLine(“hello”+b)); //2：表示想要同步的线程数目；回调就是要调用的回调函数

Barrier.SignalAndWait(); //调用此方法来执行回调

优点：在多个线程迭代运算中非常有效，在每一次迭代完成之后执行一次回调。

不会阻塞主线程。

## ReaderWriterLockSlim类

创建一个线程安全的机制，在多线程中队一个集合进行读写操作。ReaderWriterLockSlim代表了一个管理资源访问的锁，允许多个线程同时读取资源，但是只有一个线程可以独占写操作时的资源。

ReaderWriterLockSlim rw = new ReaderWriterLockSlim(); //创建实例

Rw.EnterReadLock(); //进入读锁

Rw.ExitReadLock(); //退出读锁

Rw.EnterWriteLock(); //进入写锁

Rw.ExitWriteLock(); //退出写锁

读锁允许多线程来同时读取数据，写锁在被释放前会阻塞其他线程的所有操作（包括读操作）。

由于在获取读锁时，即从集合中读取数据时，会根据当前数据而决定是否获取一个写锁修改数据集合。一旦得到了写锁，就会阻塞其他的读取操作，而浪费了大量的时间。所以为了尽可能的减小阻塞的时间，可以使用EnterUpgradeReadLock()来获取读锁，在获取了读锁之后，如果发现真的需要修改数据集合，才通过EnterWriteLock()来获取写锁。然后释放写锁，最后通过ExitUpgradeReadLock()来释放读锁。

## SpinWait类

Var spinWait = new SpinWait(); //创建实例

spinWait.NextSpinWillYield(); //下个操作是否产生处理器，同时触发上下文切换（我的理解是：获取下一个操作是否会造成CPU占用，同时转换为内模式）

# 2：使用线程池

创建线程是昂贵的操作，所以为每个短暂的异步操作创建线程会产生显著的开销。为了解决这个问题，就出现了一种叫做池的方式。

线程池可以成功的适应任何需要大量短暂的开销大的资源的情形。将预先分配的资源放入池中，需要的时候在线程池里面拿，之后再放回。

.NET线程池是这种概念的一种实现，线程池是受CLR管理的，这意味着每个CLR都有一个线程池实例。ThreadPool类型拥有一个QueueUserWorkItem静态方法（接受一个委托，表示用户的一个异步操作）。如果将委托放入线程池中时，还没有任何线程，就会创建一个新的工作者线程并这姓这个委托线程。

注：在这个过程中保持线程操作是短暂的是非常重要的。不要在线程池中放入长时间运行的操作，或者阻塞工作者线程。这将导致工作者线程变得非常繁忙，无法服务用户操作。

当停止向线程池中放置新的操作时，线程池最终会删除一定时间后过期的不再使用的线程，这将释放那些不再使用的系统资源

## 在线程池中使用委托

重构

# 5：重构列表

重构手法含有五个部分：

名称：建造一个重构词汇表。

概要：重构手法的使用场景，以及要做的事情。可以更快的找到需要的重构手法。

动机：为什么要这样重构。

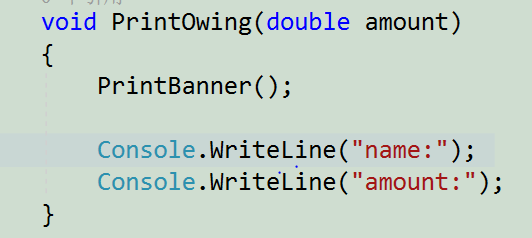
做法：如何进行重构。

范例：简单的实例。

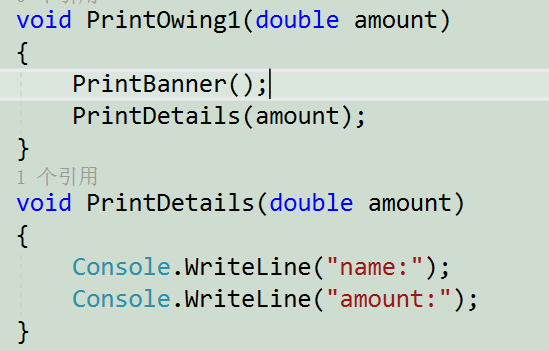
# 6：重新组织函数

很多重构手法中都是通过对函数进行整理，使之更恰当的包装代码。

**提炼函数(Extract Method)**



**↓**



把一段代码从原先的函数中提取出来，放进一个单独的函数中，再以简单明了的命名来解释函数的用途。

简单明了的函数名有以下好处：

* 每个函数的粒度都很小，那么重用的几率就会更大（比较灵活）。
* 会使高层函数读起来像是一系列注释。
* 函数粒度较小，函数的覆写也会比较容易。

在提取函数的过程中会有以下几种情况：

* 无局部变量

直接提取代码块到另一个函数。

* 有局部变量

局部变量包括传进源函数的参数和源函数临时声明的变量。局部变量的作用域仅限于源函数，如何在提取函数时处理这些局部变量？

最简单的情况：被提取的代码块只读取局部变量的值，不修改。这样就直接把他们当作参数传进提取的函数内。

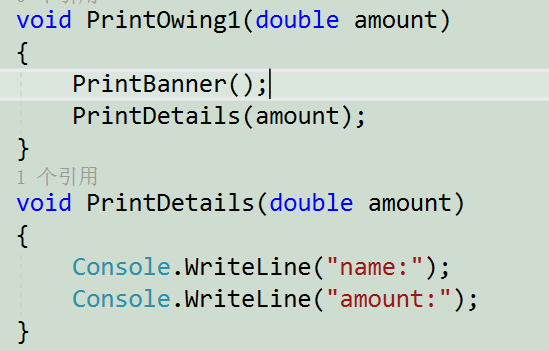
其次是：对局部变量再赋值，如果发现了对函数的赋值，因采取移除对参数的赋值的方式来解除这种操作。

如果是只在被提取的代码段中对局部变量赋值：可以将这个变量的声明转移到提取的函数中。

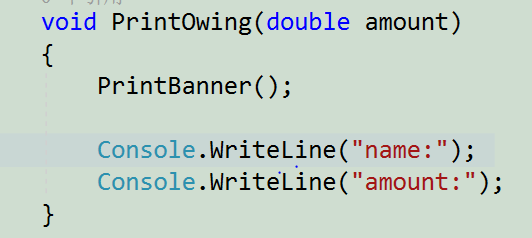
如果在被提取的代码段之后还使用了这个变量：要将变量作为返回值返回。如果有多个变量可以使用传出参数或者多写几个函数。

**内联函数(Inline Method)**

在函数调用点插入函数的代码块，移除调用函数。



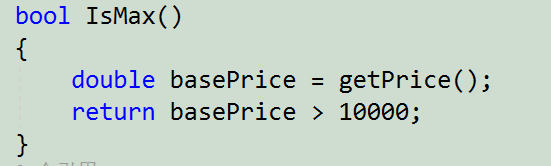
**↓**



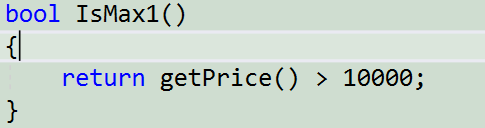
目的是让函数之间的委托变得简单，减少无用的间接层的存在。让代码变得更加清晰。

**内联临时变量(Inline Temp)**

将所有对该变量的引用动作，替换为对他赋值的那个表达式自身。



**↓**

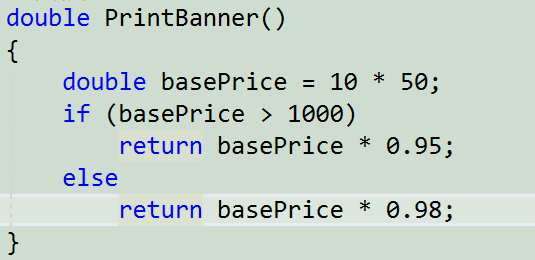


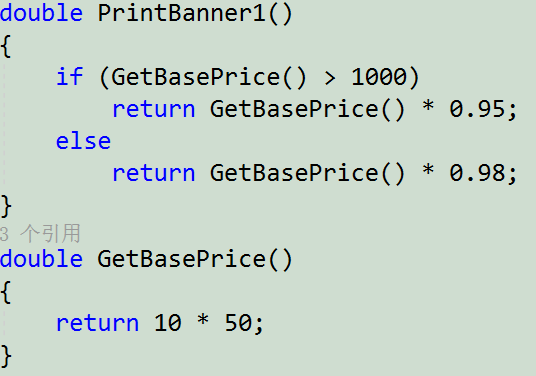
内联临时变量多半是作为以查询代替临时变量的一部分使用。

一般单独使用内联临时变量的情况是：作为某个函数调用的返回值来使用，一般这个变量是不会产生任何影响的，除非是在提取方法时的用到的变量，此时就需要把他内联化。

**以查询代替临时变量(replace temp with query)**

将这个表达式提炼到一个独立的函数中。将这个临时变量的所有引用替换为函数的调用。这样，新的函数也可以被其他函数调用。

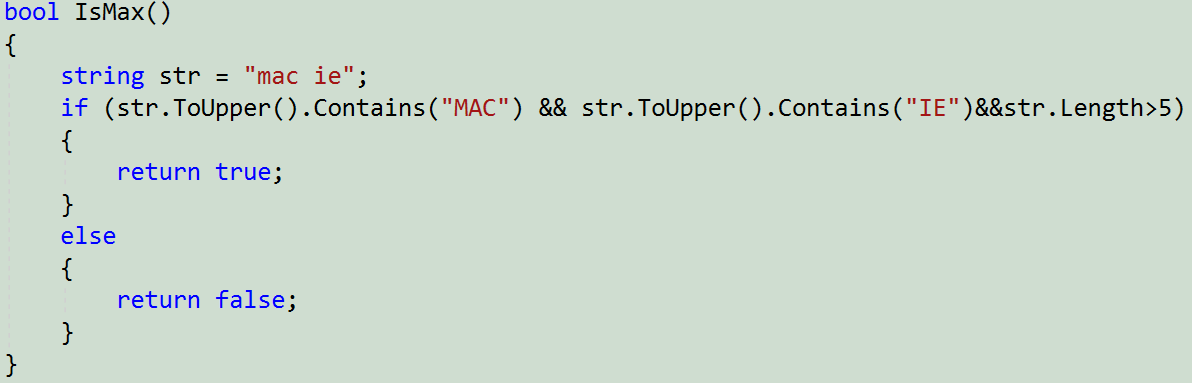


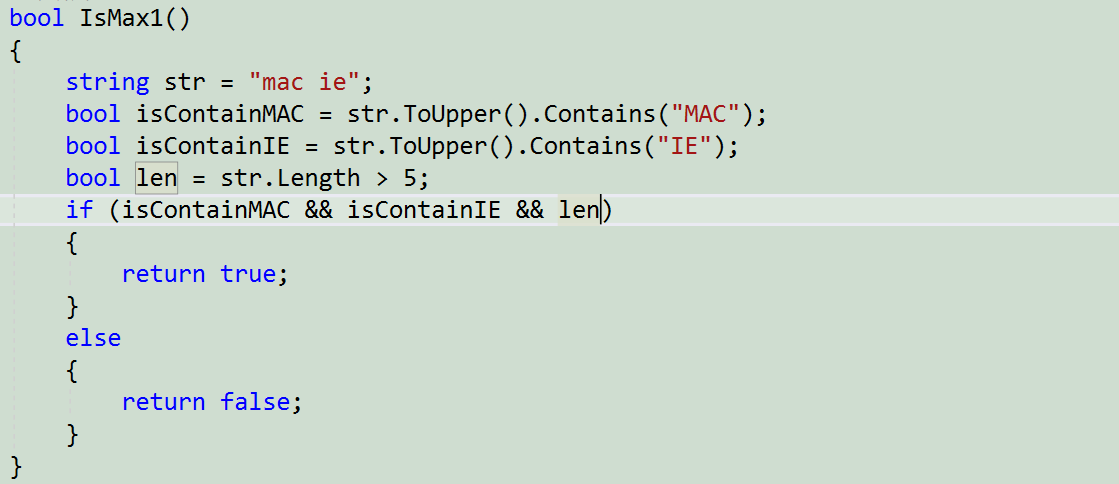


临时变量的问题在于：他们时暂时的，并且只能在所属函数中使用。由于其可见性，可能会导致写很长的函数才能访问到需要的变量。如果把变量替换为一个查询，那么这个新函数（这个函数不i修改所属对象里的任何内容）所属的范围都能使用这个变量，这样就很方便。

**引入解释性变量(Introduce explainning variable)**

将一个复杂表达式（或其中一部分）的结果放进一个临时变量，以此变量名称来解释表达式的用途。





一个表达式可能非常难以理解和阅读，此时引入一个或多个临时变量可以帮助我们理解表达式。

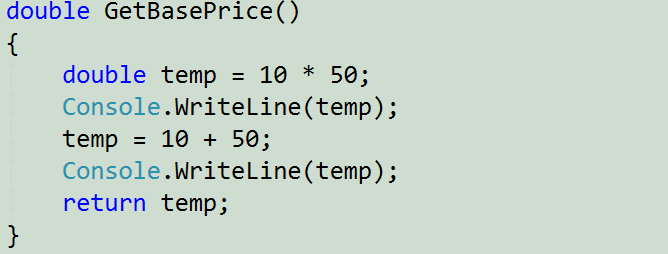
在逻辑表达式中，使用这个重构手法，可以将每个条件子句提取出来，以明确的命名来解释子句的用途。

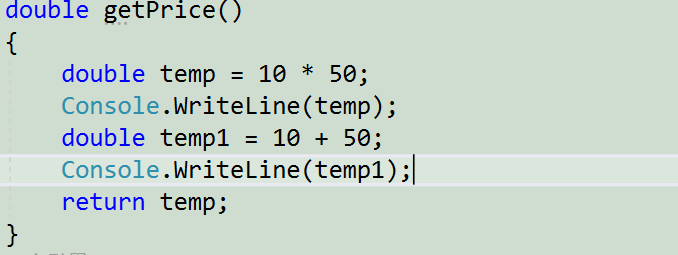
在较长的算法当中，运用临时变量来解释每一步的意义。

此种手法也有一定的缺陷，就是临时变量有自己的作用范围，不一定能够实用，所以应该以具体情况具体分析。能用提取函数就更好了。那什么时候该使用这个手法呢？就是在提取函数需要花费很多时间的时候(以此手法来整理代码，使代码变得明了，在考虑是否能够使用提取函数的手法来再次重构)。

**分解临时变量(split temporary variable)**

程序某个临时变量被赋值超过一次，但它既不是循环变量，也不用于收集结果；此时就需要针对每一次赋值，创建一个新的临时变量来使用。

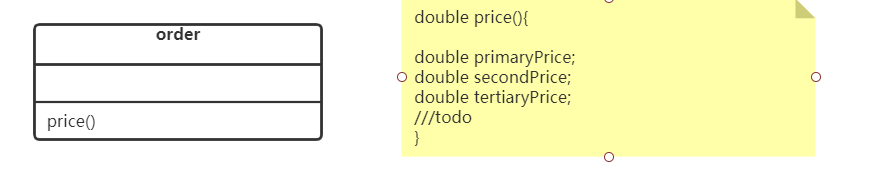


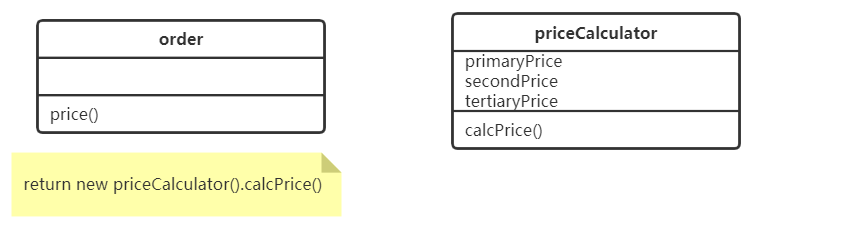


临时变量有很多不同的用途，某些用途就会导致对临时变量的多次赋值，比如：循环变量(for(int i))，结果收集变量(sum +=1)；除了这两种情况，还有其他很多变量是用来保存一段代码或函数的结果，像这种变量应该是只被赋值一次的。如果被赋值多次，就以为这这个变量承担了多个不同的责任，就应该由多个变量来完成，保证其每个变量只做一件事情。

**以函数对象取代函数(replace method with method object)**

如果一个大型函数，对其中的局部变量的使用无法去通过提取函数的手法来进行重构。那么此时将这个函数放进单独的对象中，再将局部变量变为对象里的字段，就可以在这个对象内对这个大型函数进行分解。





如果一个函数之中局部变量很多，那么要分解函数是非常困难的。以函数对象取代函数这一手法会减轻这种困难，此手法会将所有的局部变量都变成函数的属性，再利用这个对象来分解函数。好处就是提取函数更加轻松，代码也变得更加清晰。

**替换算法(substitute algorithm)**

将原来函数替换为思路更清晰的算法。

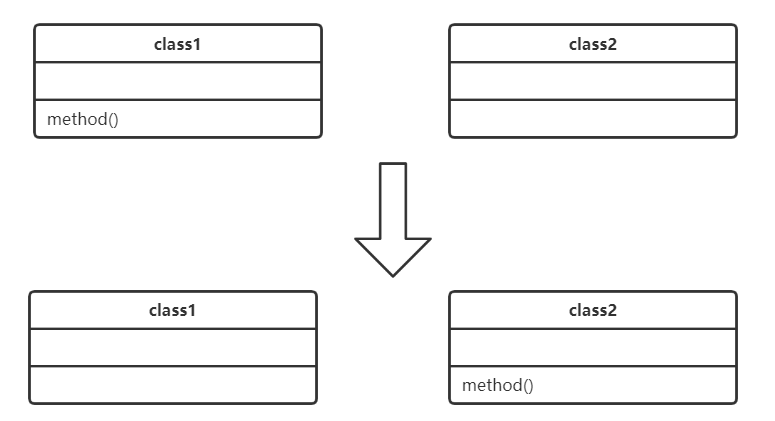
**7：在对象之间搬移特性**

类经常会因为承担过多的责任而变得臃肿复杂，这个时候就需要对类进行简化，往往会使用提炼类的方式将一部分责任分离出去。

如果一个类经过重构之后，这个类变得没有太大用处，这个时候就需要使用将类内联的手法，把这个过于简单的类删除掉。

**搬移函数(move method)**

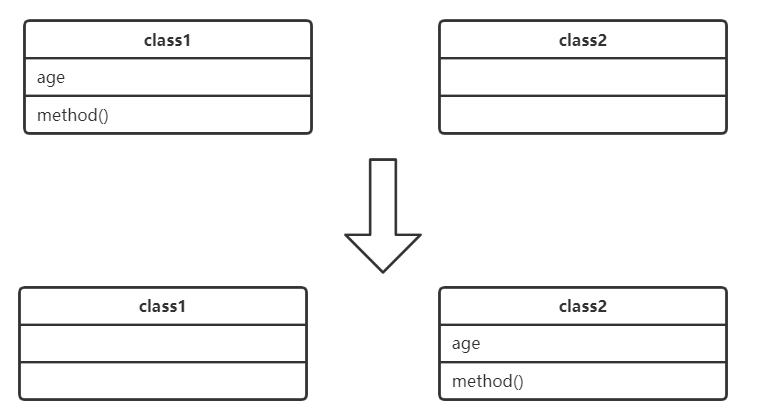
如果处于A类里面的一个函数与B类进行了过多的交互，此时就应该将那个函数移入B类。



搬移函数是重构理论的支柱。如果一个类有太多行为，或者一个类与另一个类有太多的交互而形成高耦合，此时就需要搬移函数，来将系统中的类变得更加简单。

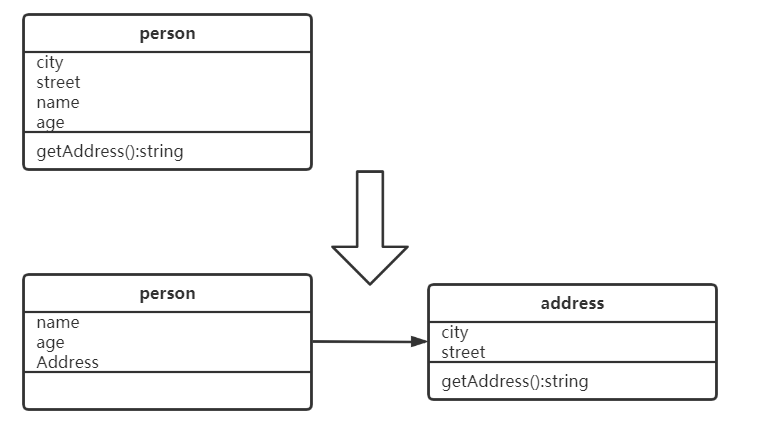
**搬移字段(move filed)**

与搬移函数的意义差不多。



**提炼类(extract class)**

一个类做了两个类的事情，此时就需要新建一个类。一个类应该是一个清除的抽象，处理一些明确的责任。

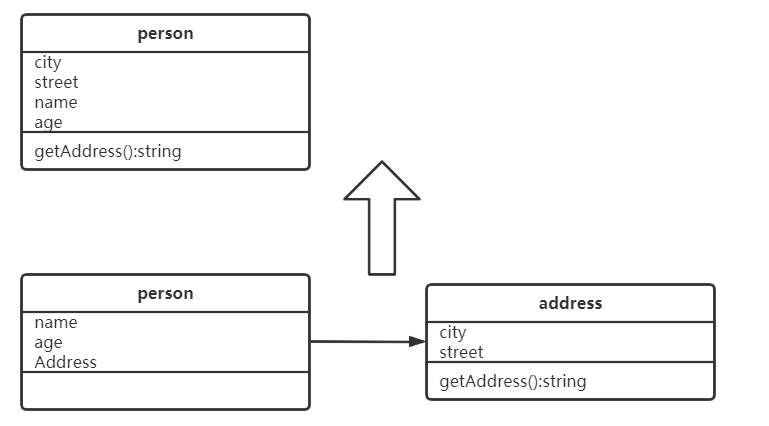


另一种可以提炼类的情形是，在开发后期出现的情况是类的子类化方式，如果发现子类化只影响类的部分特性，或者类的另一些特性需要以另一种子类化的方式来完成，这个时候就需要分解原来的类。

在分解类的时候，尽量不要事先建立从新类通向旧类的连接。并且建立起双向连接时候要检查是否能变成单向连接。

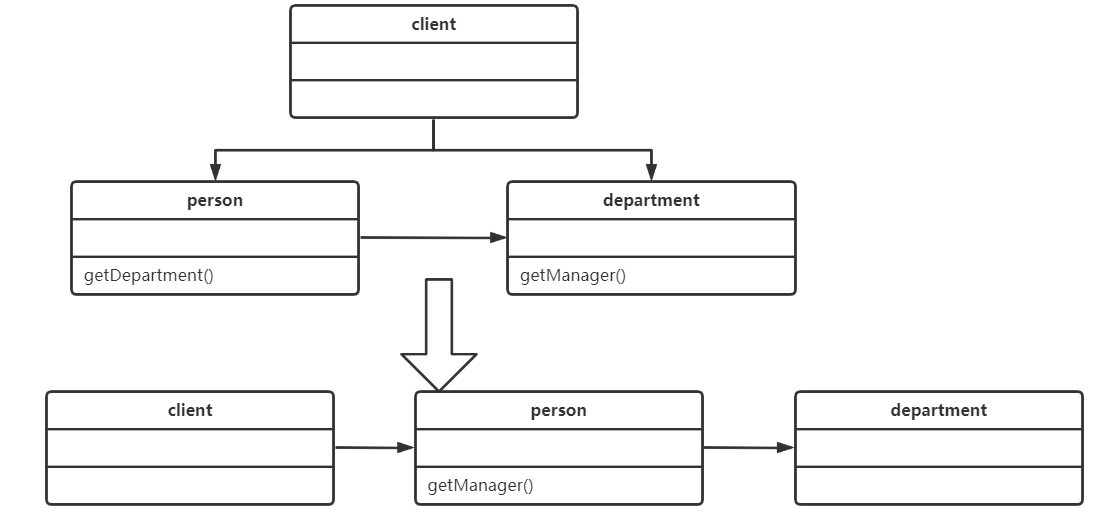
**将类内联化(inline class)**

这个类没有太大的用处，将这个类的内容迁移到另一个类中并且移除原类。



将类内联与提取类正好是一个相反的过程，如果一个类不再承担足够的责任，没有单独存在的理由，就需要删除这个类。

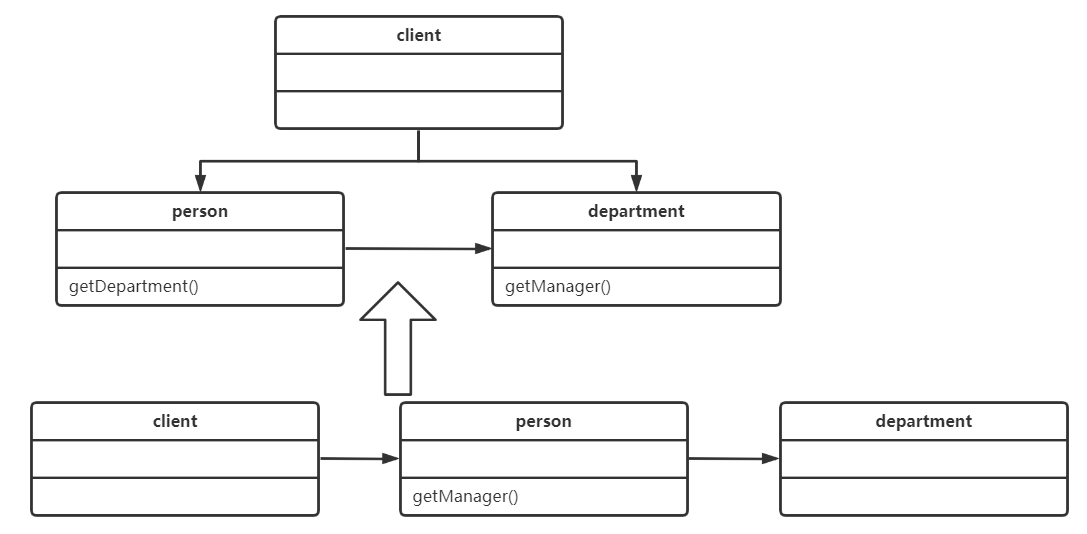
**隐藏委托关系(hide delegate)**



封装即使不是对象的关键特征，也是最关键特征之一。封装意味着每个对象都应该尽可能少的了解系统的其他部分。如此，如果某一关系发生变化，需要修改的地方就会比较少，让修改变得容易。

让用户更少的了解其中的关系，也让交互变得更简单。

**移除中间人(remove middle man)**



减少大量简单的中间操作，改为由客户直接和目标对象进行交互。

隐藏委托关系的好处是让交互变得更加简单。但是这也有一定的坏处：每当客户要使用委托类的新特性的时候，就必须在中间类处新增一个调用。

而移除中间类就是这一个相反的过程。适度的隐藏和现实可以使开发变得更加简单。

**引入外加函数(introduce foreign method)**

在需要为某一类增加一个新的函数，但又无法修改这个类的代码，这个时候就需要使用这一手法来新增一个函数。(类似于扩展函数)

**引入本地扩展(introduce local extension)**

需要为某一个类增加一些函数时，使用。

建立一个新类，使它包含这些额外的函数，并且让这个新类成为不能修改的类的一个子类。