计算机程序设计语言

单继承与多继承的语言支持

张 天

软件工程组

计算机科学与技术系

南京大学

本章要点

- 本章主要讨论面向对象范型中的继承机制，重点针对单继承和多继承在各种语言中实现的差异性进行分析和比较
  - 理解继承
  - 单继承与多继承
  - 不同语言中的支持与实现

软件固有的复杂性

爱因斯坦认为自然界必定存在着简单的解释，因为上帝不是反复无常或随心所欲的。软件工程师没有这样的信仰来提供安慰。许多必须控制的复杂性是随心所欲的复杂性。

——Frederick Brooks

为什么复杂

- Brooks指出，“软件的复杂性是一个基本特征，而不是偶然如此”，但是为什么呢？
- Booch的回答：
  - 我们认为这种固有的复杂性有4个原因：**问题**域的复杂性、管理开发**过程**的困难性、通过软件可能**实现**的灵活性，以及刻画**离散**系统行为的问题。

注意：仔细琢磨体会软件技术的发展，背后就是这些原因造成的！

问题域复杂性的表现

- 领域知识和计算机知识之间有鸿沟
- 对软件的需求本身就会非常复杂
- 用户的需求是不断变化的

管理开发过程的困难性

- 几十年前，只有几千行的汇编程序就已经是软件工程能力的极限了。但时至今日，交付系统的代码规模常常达到几十万甚至几百万行（而且是高级语言编写）

- 围绕这样复杂系统的开发团队，如何有效沟通、协同、过程控制等，是一个涉及到多方面的复杂问题！

软件实现上的灵活性

- 建筑公司不会为了盖楼自己造钢铁厂，也不会因为需要木料而自己经营林场，但遗憾的是，在软件行业，这种事情经常发生！
- 原因：软件提供了非常大的灵活性，所以开发者有可能表达任何形式的抽象。这带来了巨大的诱惑，但也迫使开发者打造几乎所有的初级构造模块。
- 建筑行业对原材料的品质有统一的标准，但软件行业很少有这种标准。结果是，软件行业还是劳动密集型的产业！

刻画离散系统行为的问题

- 连续系统
  - “当我们说系统是有连续函数描述时，我们是说它不会包含任何隐含的惊奇。输入的小变化总是会导致输出中相应的小变化”
- 离散系统
  - 离散系统中的状态转换不能用连续函数来建模。软件系统之外的每个事件都有可能让系统进入一个新的状态，而且，状态间的变化关系并非总是确定的！
  - 在离散系统中，所有的外部事件都有可能影响系统内部状态的任何部分，而在连续系统中，这是不太可能发生的！

复杂系统的5个属性

- 层次结构
- 相对本原
  - *复杂系统是由基础组成部分构成的，但这些基础组成部分是哪些，这是相对的，是可以因考虑的视角变化而变化的*
- 分离关注
  - *组件的划分是可以根据其行为或自身特点找出差异性的*
- 共同模式：*层次机构的组成方式往往有共同的模式*
- 稳定的中间结构
  - *复杂系统的演化性，存在中间的稳定状态，使演化得不断从简单到复杂再到更复杂*

分离关注

- “**组件内**的联系通常比**组件间**的联系更强。这一事实实际上讲组件中高频的动作（涉及组件的内部结构）和低频的动作（涉及组件之间的相互作用）分离开来。”
- 就是系统本身具有这种特点，是我们可以以相对隔离的方式研究每个部分

控制复杂性

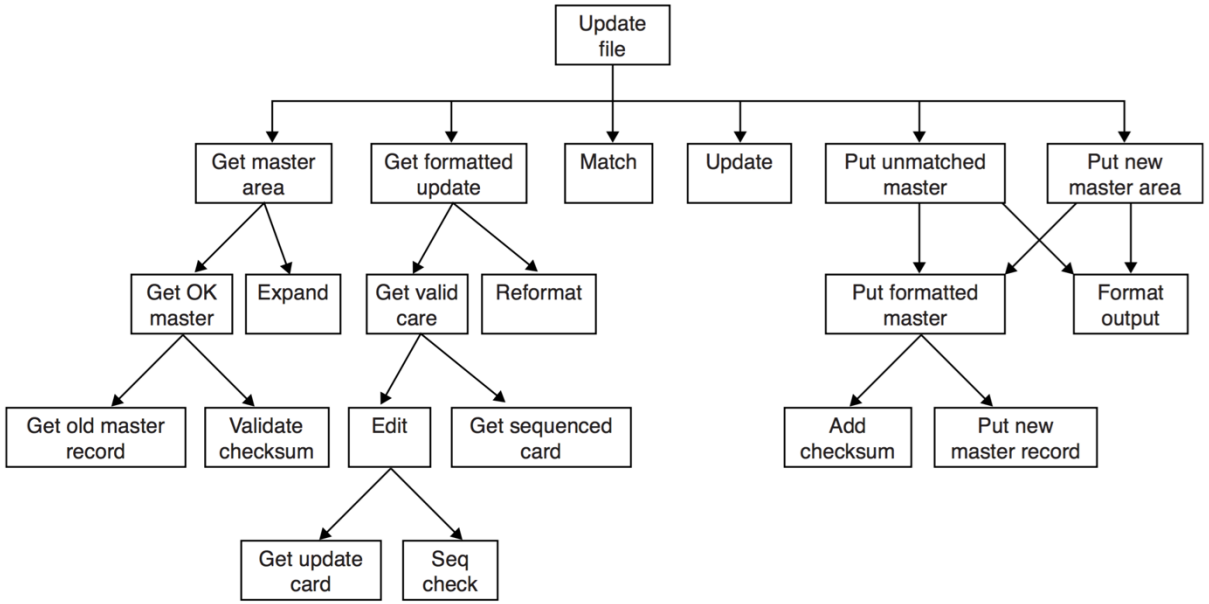
- 既然复杂系统有如上5个属性，那么我们再重新审视复杂的软件系统，应该就有了一些思路：
  - 分而治之的思路
  - 借助抽象的力量
  - 利用层次结构

分而治之

- “控制复杂性的技巧我们从远古就知道了，即分而治之。”—Edsger W. Dijkstra（结构化设计的提出者）
- 两种重要的分解方式
  - 算法分解
  - 面向对象分解

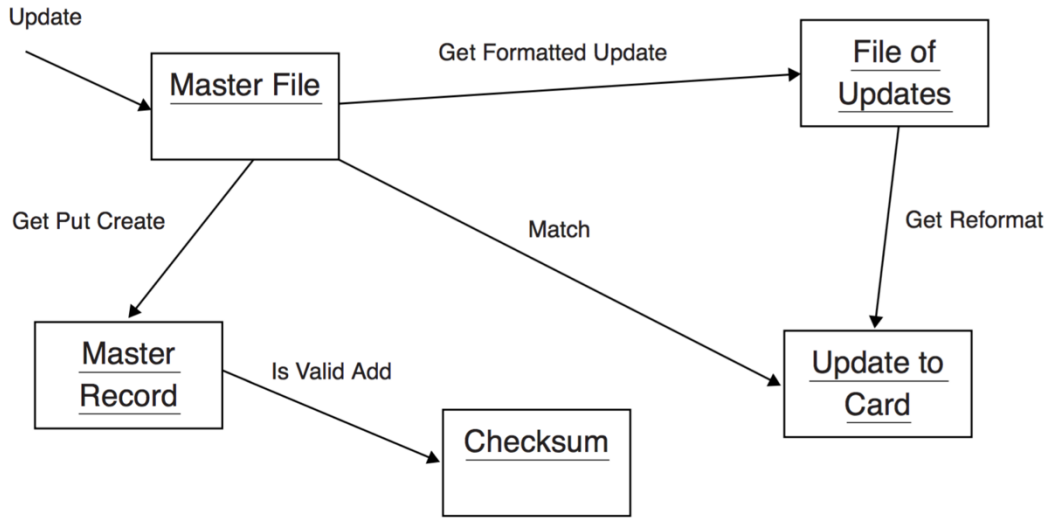
算法分解

结构化设计是基于算法（功能）分解进行分而治之



更新主控文件的功能模块结构图

面向对象分解



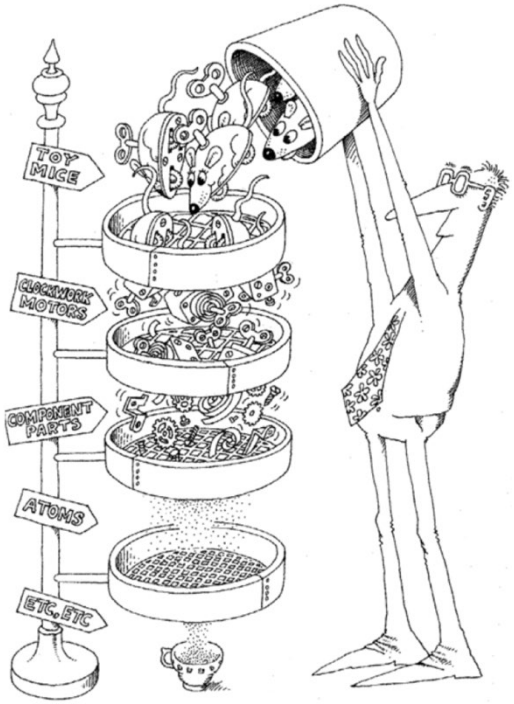
- 面向对象分解将世界看成是一组自动化的代理（对象），它们相互协作，执行某种高级的行为。
- 此例中，“Get Formatted Update”不是存在于一个独立的算法中，而是与“Files of Updates”对象相关联的一个操作。

两种分解的差异

- 对于复杂系统的分解，哪一种是正确的方法？
  - 这是一个带有欺骗性的问题，因为正确的答案是两种观点都有各自的重要性。
- 算法分解的观点强调了事件的顺序
- 面向对象分解强调了一些代理，他们要么发出动作，要么是这些操作执行的对象

抽象的作用

- 抽象是“穿越”的桥梁
  - 想象自己从最上层的问题领域，一直穿越到最下层的机器实现（冯诺依曼体系结构上的指令系统）
  - 你会经历很多个层次，而每一个都是最终实现的一种抽象描述



层次结构的作用

- “另一种增加单块信息的语义内容的方法，是在复杂软件系统中显示的组织类和对象层次结构”—Grady Booch
  - 注：显然Booch认为对于复杂系统的层次结构属性而言，算法分解的思路已经不太合适了！

- 面向对象的分解自然引发了从方法学到各种高级语言的革命性发展

面向对象编程

面向对象编程（OOP）是一种实现的方法，在这种方法中，程序被组织成很多相互协助的对象，每个对象代表某个类的一个实例，而类则属于一个通过继承关系形成的层次结构。

—— Grady Booch，OOAD

类和对象

- 对层次结构的解释（Booch）
  - 层次结构是抽象的一种分级或排序
- 在复杂系统中，最重要的两种层次结构是类结构（“是一种”的层次结构）和对象结构（“组成部分”层次结构）
- *思考：实例化（类-对象之间）又是一种什么关系？它跟继承关系有和不同？[“是一种”和“是一个”]*

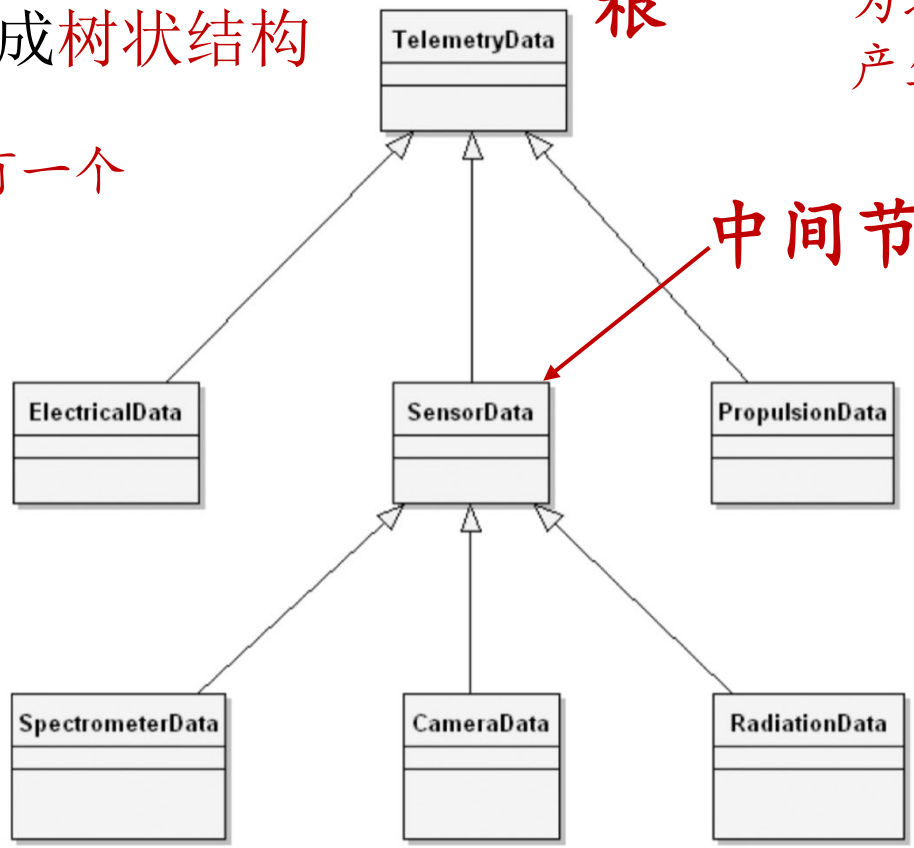
继承

- 继承是最重要的“是一种”层次结构
- 封装、继承
  - 通过继承，封装可以通过三种方式被打破：“子类可以访问其超类的实例变量，可以调用其超类的私有操作，或者直接引用其超类的超类”（Barbara Liskov）
  - 不同的程序设计语言在支持封装和继承方面以不同的方式进行了**折中**

单继承

单继承会形成树状结构

每个类只能有一个父类



根

非叶子类可以设计为抽象类，即不能产生实例的类

中间节点

叶子类

叶子类一定是具体类，即能产生实例的类

思考：从任何一个叶子到根只能有一条路径，这意味着什么？

单继承的局限

- 继承意味着**复用**
- 在单继承中，每个子类都只有一个超类
  - 也就是说，从根到任意层次的子类都只有一条路径
  - 这意味着，在整个继承的层次结构中，重用只能发生在一个继承路径上
  - 再进一步说，无法重用来自不同继承路径上的代码
- 多继承可以解决这个问题

单继承的解决方式

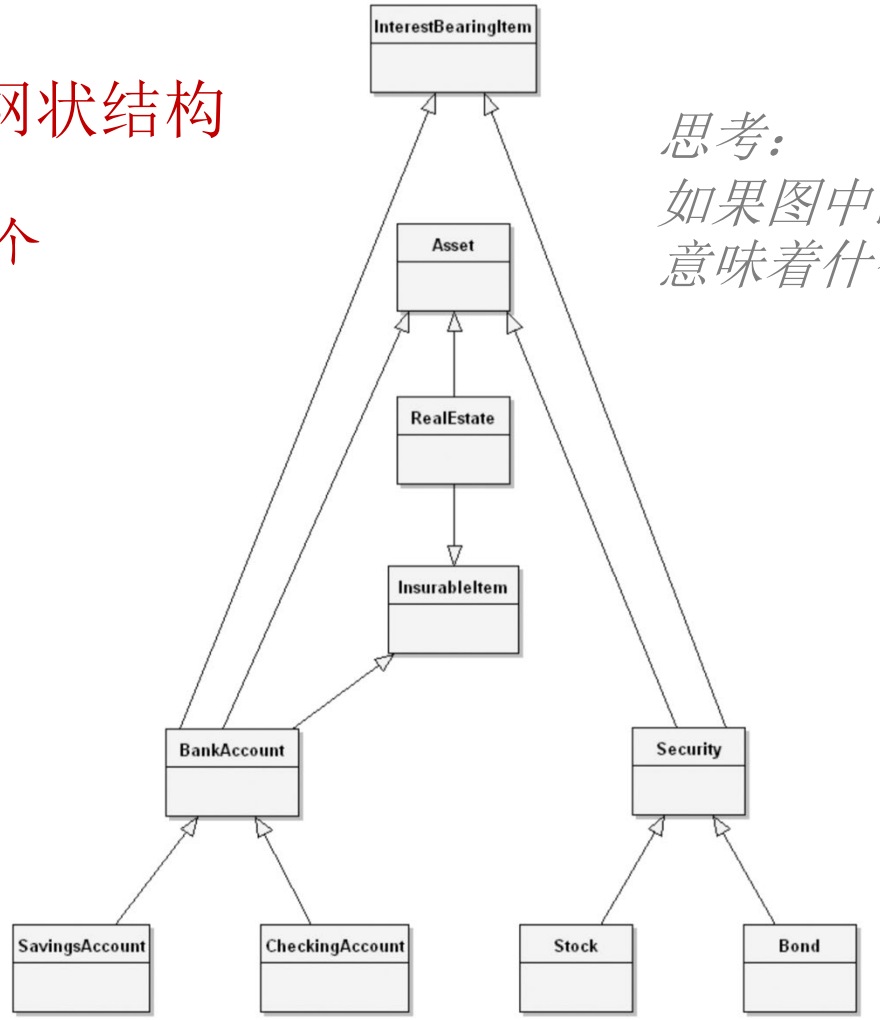
- 当有两个希望复用的类时：
  - 单继承“迫使程序员从两个差不多有吸引力的类中选择一个来继承。这限制了预定义类的实用性，常常需要重复代码。例如，没有办法派生出既是一个圆也是一个图画的图形，程序员必须从一个类派生，然后重新实现另一个类中的功能。”（John Vlissides ）

John Vlissides：Design Patterns作者之一, 在2005年11月因脑瘤过世，年仅44岁

多继承

多继承会形成网状结构

每个类可以有多个父类



思考：
如果图中出现菱形结构，意味着什么？

多继承的问题

- 当我们采用多继承时，会遇到两个问题：
  - 如何处理来自不同超类的名字冲突
  - 如何重复的继承

用Git类比

- 多继承所带来的问题是显而易见的，尝试用Git中的merge来思考这个问题：
  - 当对多个分支（相当于多个父类）进行合并（相当于多继承）时，如果多条分支上出现对同一个文件（相当于继承到的相同部分）的修改，就会出现冲突

名字冲突

- 当两个或多个不同超类对接口中的某些元素使用相同的名字时，就可能发生名字冲突，如同名的实例**变量**或**方法**
  - 例如，A类和B类都有value属性，但分别表示其大小（为int类型）和内容（为string类型），那么如果C继承自A和B的话，将得到哪个属性呢？
  - 但在另一种情况下，如果A和B都有name属性表示名称，也都是string类型，那么C就不用担心冲突了
- 名字冲突可能导致多继承的子类的二义性行为

解决名字冲突

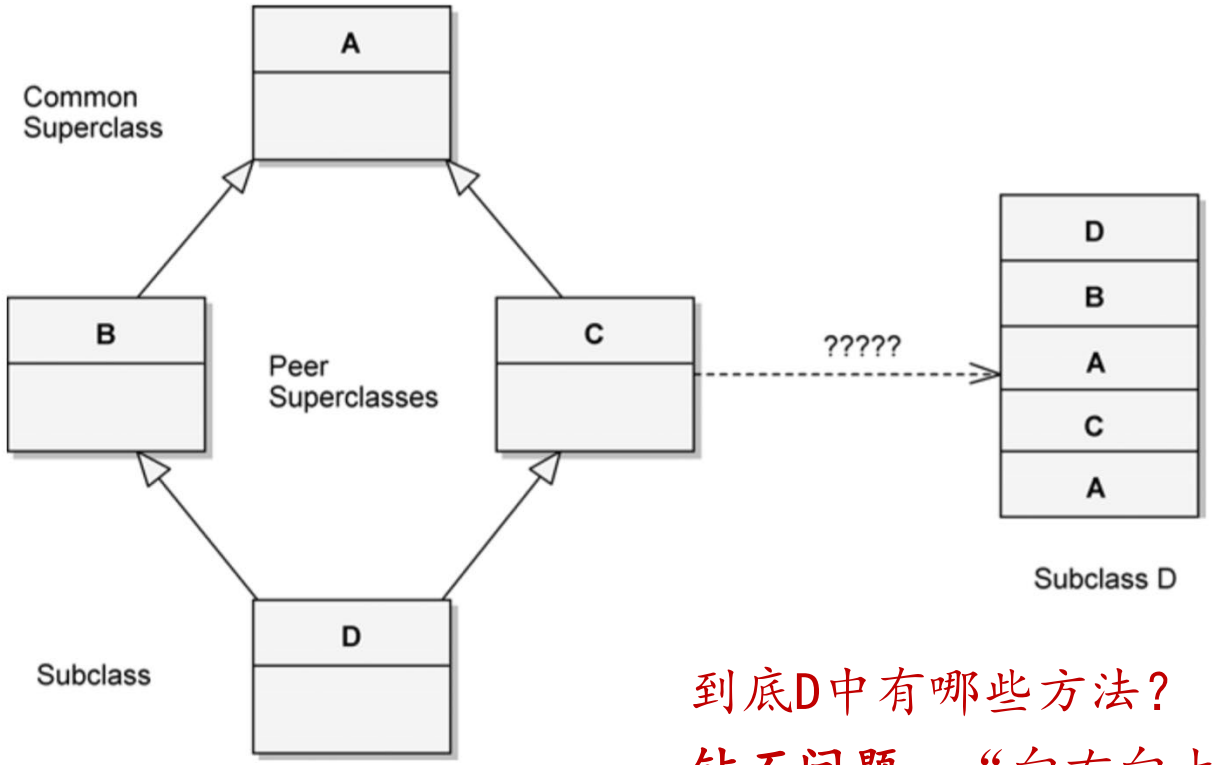
- 有三种方法可以解决名字冲突问题
  - 首先，编程语言可以强制规定出现冲突的继承是违法的
  - 其次，编程语言规定，通过不同类所引入的相同名字就是指同一个元素（会互相覆盖，需要设置优先级策略）
  - 第三，编程语言的语义运行这种冲突，但需要所有对这个名字的引用都有完整的限定符，表明声明它的位置

重复继承

- Meyer是这样描述该问题的：
  - “由于多继承而引起的一个微妙的问题就是，如果一个类通过多个途径成为另一个类的祖先，……那么迟早会有人写出D类有两个父类B和C，而B和C又都以A为父类，……或者其他情况，使得D从A继承了两次（或更多）”
- 思考：子类中保留重复继承父类的多个拷贝，还是一个拷贝？
- *注：重复继承给语言实现带来的问题更甚于软件设计层面上的问题*

菱形继承结构

- 重复继承的一个直观的情况是继承结构中出现菱形



到底D中有哪些方法？

钻石问题：“向右向上”匹配算法出现问题！

解决重复继承

- 有几种常用的方法来解决重复继承的问题
  - 首先，在编程语言中不支持重复继承
  - 其次，编程语言允许重复继承，但要求使用完整的限定名来引用成员的具体拷贝
  - 第三，将对同一个类的多次引用视为代表相同的类
- 不同的OOPL在其中选择自己的实现策略

混入类

- 多重继承的问题促使了混入类（mixin）的出现
  - 最早引入mixin的是**Flavors**语言(1986年)：“混入类在语法上等同于一个正常的类，但它的目的是不同的。这种类的目的只是……向其它的Flavors类添加功能，开发者永远不能创建混入类的实例。”
  - *这里先简单了解，后面再深入讨论*

实例研究：Python

- 通过对Python语言中多继承的具体实现机制剖析，来理解编程语言对多继承支持的问题
  - Python中的OOP
  - Python对继承的支持
  - Python中钻石问题的解决方案

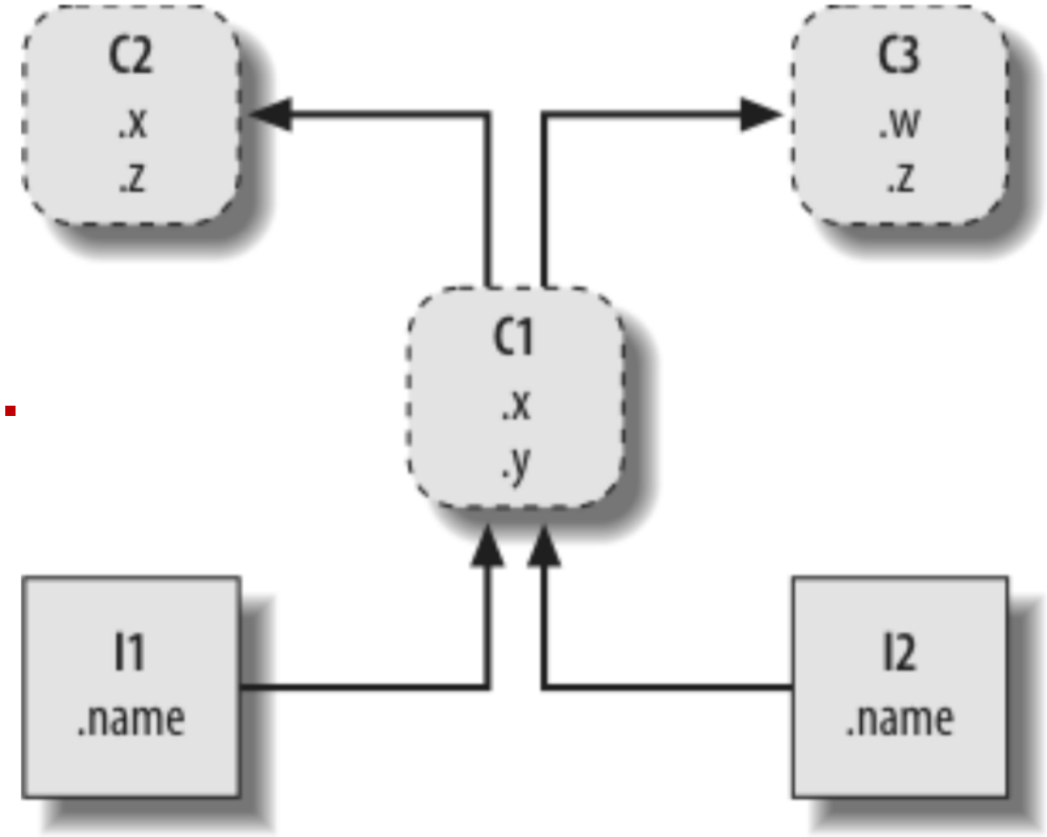
Python支持的风格

- Python是一种混合风格的语言
  - 支持过程式开发和面向对象开发
  - 事实上，Python还可以支持函数式编程（将函数最为第一类对象使用）
- Mark Lutz：
  - 从根本上讲，Python是一种面向对象语言。Python的函数也是对象，可以用常规的对象工具来处理函数

继承属性的搜索

- 读取属性和调用方法：object.attribut
  - 当我们对class语句产生的对象使用这种方式时，这个表达式会在Python中启动搜索，即搜索对象链接的树，来寻找attribute首次出现的地方
  - 具体地说，先搜索object对象自身，然后是该对象“之上”的所有类，并按照由下之上，由左至右的顺序
  - 注意：该示例既适用于属性，也适用于方法

对象树示例



定义部分：

class C1 (C2, C3): …

I1和I2都是实例

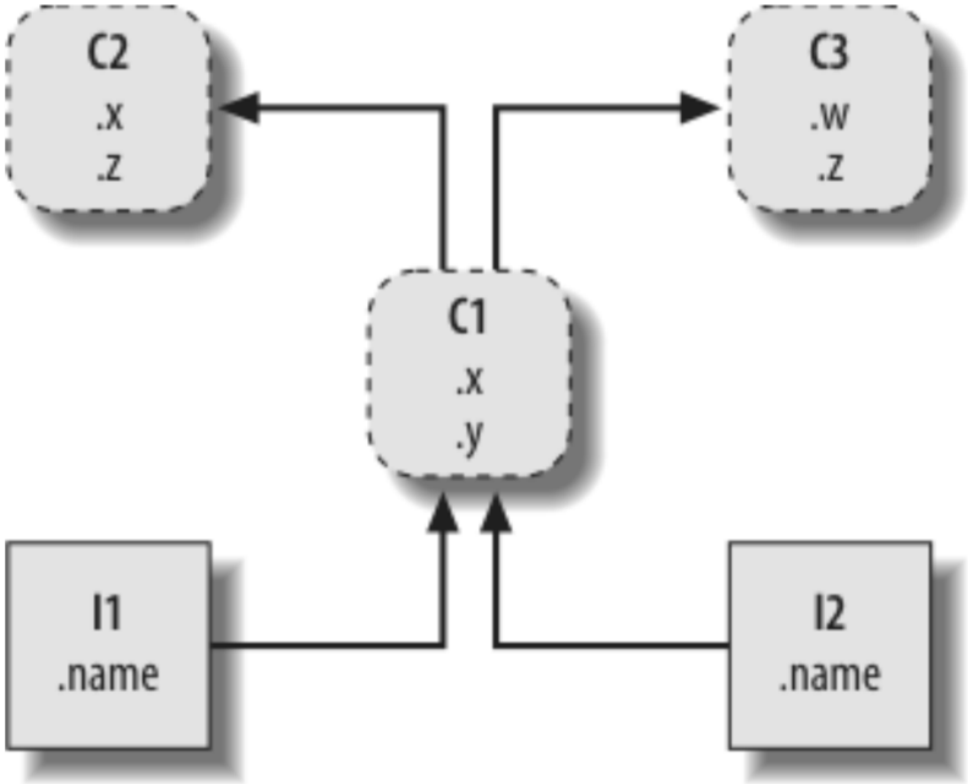
程序实验：
这些属性的检索顺序和返回：

- I1.**x** and I2.**x**?
- I1.y and I2.y?
- I1.**z** and I2.**z**?
- I2.name?

对象树示例

定义部分：

class C1(C2, C3): …



- I1.**x** and I2.**x** both find x in C1 and stop because C1 is lower than C2.
- I1.y and I2.y both find y in C1 because that's the only place y appears.
- I1.**z** and I2.**z** both find z in C2 because C2 is further to the left than C3.
- I2.name finds name in I2 without climbing the tree at all.

Python继承机制

- 超类列在类定义开头的括号中
- 类从其超类中继承属性
- 实例会继承所有可读取类的属性
- 每个object.attribute都会开启新的独立搜索
- 逻辑的修改是通过创建子类，而不是修改超类
  - 这一条是因为Python属于动态语言，允许修改类

类和实例

- 可以将Python的类理解为是一种命名空间工具
- 类和实例的差别在于，类是一种产生实例的工厂
- 特别值得注意的是：
  - 在Python的和继承相关的树中，类和实例几乎完全相同，每种类型的主要用途都是用来作为另一种类型的命名空间（变量的封装，也就是我们可以附加属性的地方）
  - 因此，这些树的名称也可以称为类树和命名空间树

钻石继承

- 一个重要的原因是3.x中的类模式进行了调整，所有的类都显示或隐式地派生在类object
  - 这个变化的结果就是，Python3的多继承必会出现钻石问题
- 在Python2.x和3.x中，钻石继承（菱形继承）的搜索顺序发生了重大变化

建议大家动手做一下实验看看效果

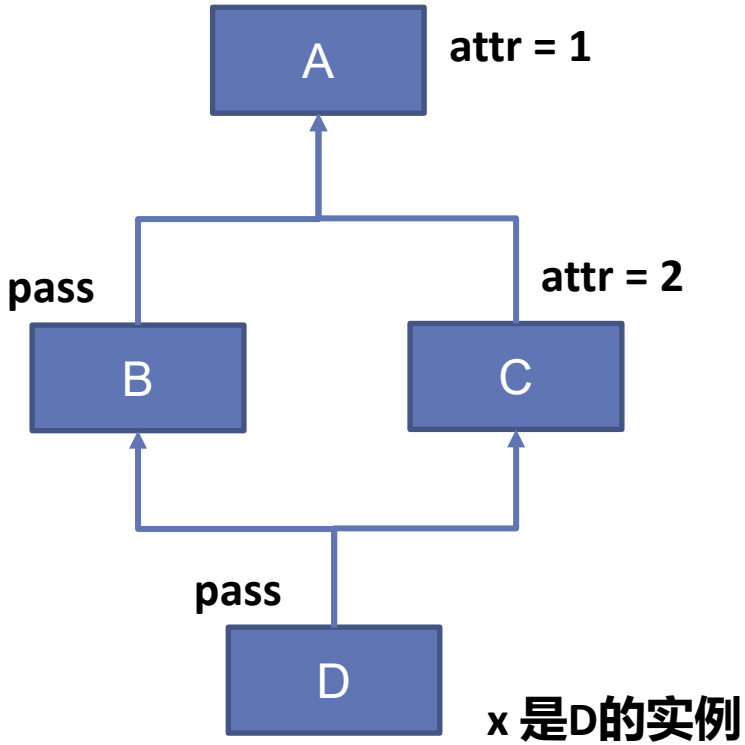
钻石继承变动

- **经典的类继承搜索策略**
  - **深度优先**，然后才是由左向右，即先一路向上搜索，返回后再向右侧以同样方式搜索下一个
- Python3对多继承的搜索策略进行了调整
  - **宽度优先**，搜索过程先水平向右，然后向上移动一层

早期Python2.x示例

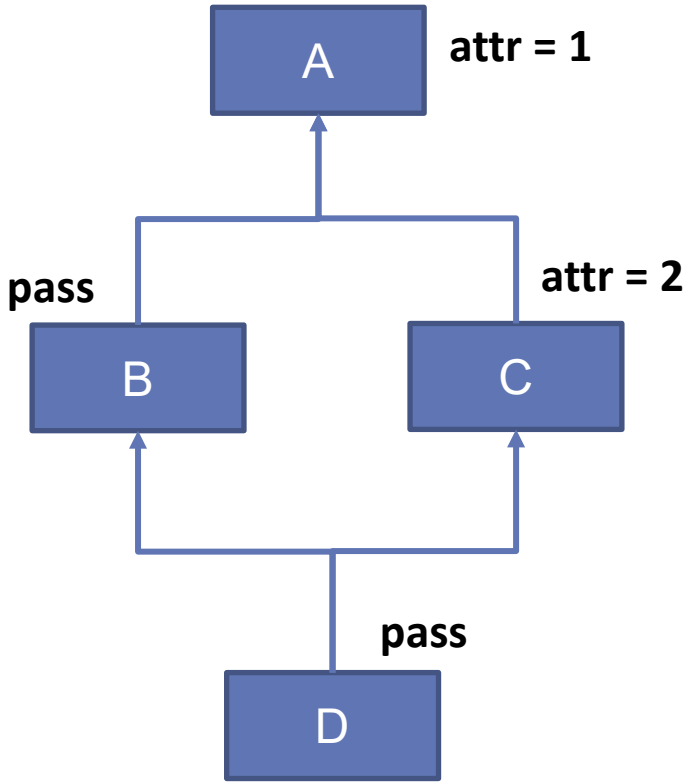
```
>>> class A:
        attr = 1    # Classic (Python 2.x)

>>> class B(A):     # B and C 都指向A
        pass

>>> class C(A):
        attr = 2

>>> class D(B, C):
        pass        # 先找A再找C

>>> x = D()
>>> x.attr          # 查找顺序：x, D, B, A

1
```

注意，Python把类作为命名空间工具



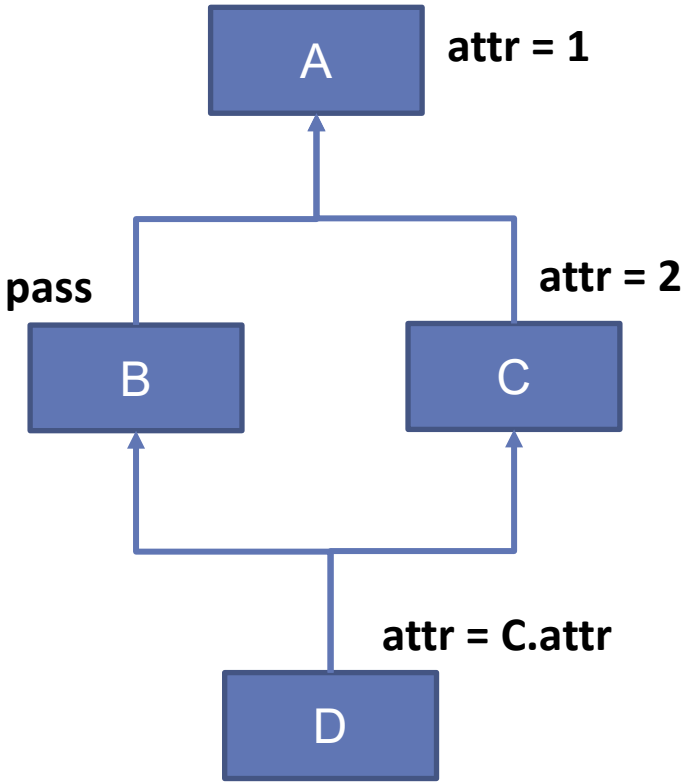
Python3.x示例

```
>>> class A(object):
        attr = 1    # Classic (Python 3.x)

>>> class B(A):
        pass

>>> class C(A):
        attr = 2

>>> class D(B, C):
        pass      # 先找C后找A

>>> x = D()
>>> x.attr          # 查找顺序：  x, D, B, C

2
```



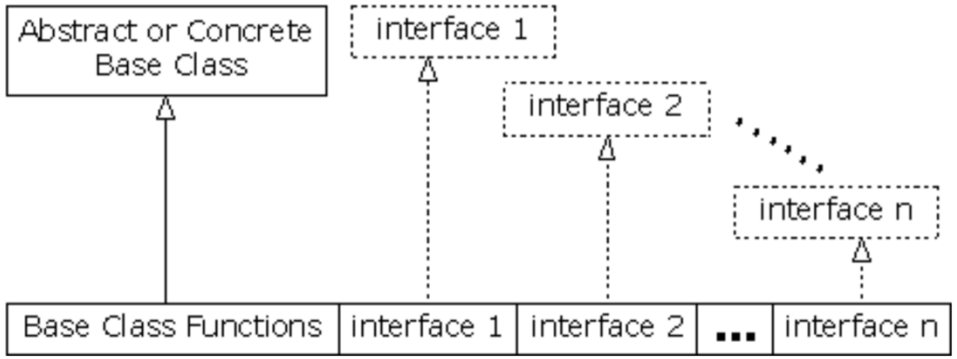
显式解决冲突

```
>>> class A(object):
        attr = 1    # Classic

>>> class B(A):
        pass

>>> class C(A):
        attr = 2

>>> class D(B, C):
        attr = C.attr    # 显式化指向C

>>> x = D()
>>> x.attr      # Works like new-style (all 3.0)
2
```



Java的解决之道

- 在Java语言中，多重继承被禁止，取而代之的是通过“单继承加多接口”方式来模拟多继承
  - 在TIJ4中，Bruce也只是简单的用类似上面的话来解释Java中对多继承的支持方式



思考：Java的这种方式有什么问题？

接口

- 从OOP的角度来看，类能够解决复用的问题，而接口纯粹是一种外部功能契约，不存在代码复用的能力
- Java中虽然使用接口实现了外部功能契约的集成，但实际上和多继承所体现的多复用是不一样的

- *思考：能否从设计模式的角度解决这个问题？*

使用模式求解

- 从效果上来看，多继承可以让子类复用多个父类代码，并且所有父类型都可以接收子类实例

- Java的解决方案：
  - 使用**委派**和**聚合**
  - *个人使用的方案：在Java中，可以选择其一进行继承，然后聚合其它类的实例，同时对各个类抽象出其接口，声明为对其实现*

Java示例

```
interface LockingMixin {
  void lock();
  void unlock();
}

class Lock {
  void lock(){...;};
  void unlock(){...;};
}

class Printer implements LockingMixin {
  final Lock lock = new Lock();
  void lock() {lock.lock();}
  void unlock() {lock.lock();}
  void spool(TextData text){
    this.lock();
    ...
    this.unlock();
  }
}
```

- 用委派来实现多重继承的 Java 程序
- 把从接口调用的方法，都明确地委派给实现共同功能的对象
- 本来在多重继承中可 以自动实现的，现在要通过手工来实现

Java示例

```
interface LockingMixin {
  void lock();
  void unlock();
}

class Lock implements LockingMixin {
  void lock(){};
  void unlock(){};
}

class Printer{
  final Lock lock = new Lock();
  void spool(TextData text){
    this.lock.lock();
    ...
    this.lock.unlock();
  }
}
```

- 不用委派来实现多重继承的 Java 程序
- 把实现共通功能的对象作为成员变量来使用。这样操作对象并不需要直接实现接口，而只是作为属性保存一个实现共通功能的对象，在程序中直接调用该属性的方法
- 没有了委托的方法，这些部分就变得简单明了，但是在调用共通功能的时候，每次都要引用属性加上.lock，会让人觉得不怎么漂亮

加深对继承的理解

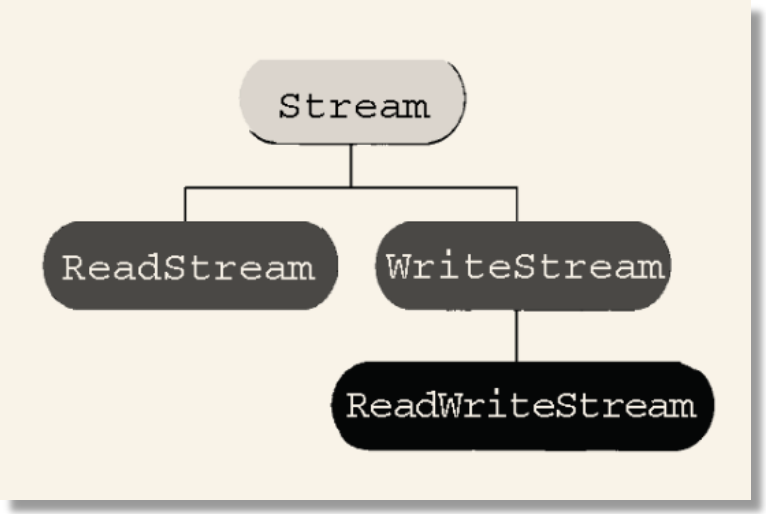
- 通过前面的介绍，已经把多继承的困境和基本解决方式做了简单的解释，现在再进一步看看目前编程语言发展的趋势，通过对混入类（mixin）概念的支持来实现多继承
- 这里，让我们回过头来，再体会一下OOP中的不同继承，还是从多继承的复杂性上来引入混入类

OOP的两种继承

- 接口继承（interface inheritance）
  - 也称**规格继承**，需要父类对象参数的函数，可以接受子类对象参数，并且可以通过父类的接口使用子类对象（调用其方法）

- 实现继承（implementation inheritance）
  - 通过继承在子类中直接复用父类的代码（成员变量和成员函数）

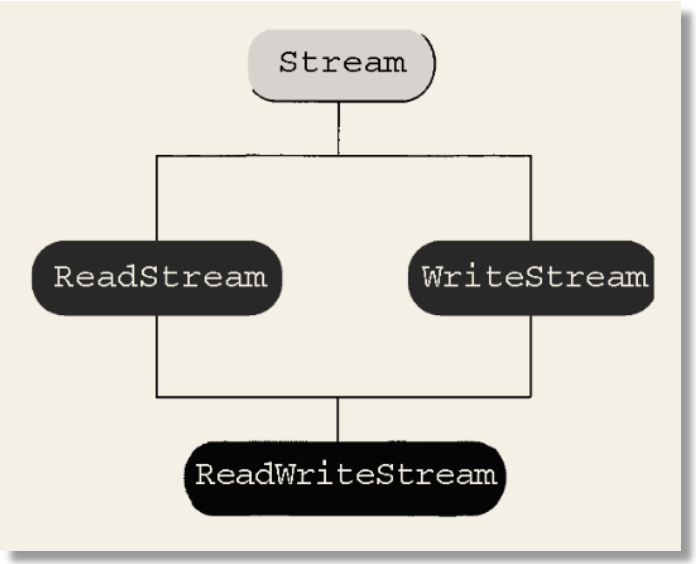
单继承的困境



单一继承的问题：
ReadWriteStream 只能有一个父类，即 WriteStream，而不能同时继承 ReadStream！

- 结果是 ReadWriteStream 继承了 WriteStream 这个类，然后再把 ReadStream 的代码复制过来，实现 ReadStream 的功能
- 从程序维护的观点来看，程序复制是必须禁止的。由于单一继承的限制而导致的程序复制是我们不愿意看到的

多重继承的意义



用多重继承的解决方法：

非常直观和自然，ReadWriteStream 类可以继承两个父类

- 从另外的角度来看，如果有多重继承的话，那么很自然地从 ReadStream 和 WriteStream 继承就可以生成 ReadWriteStream

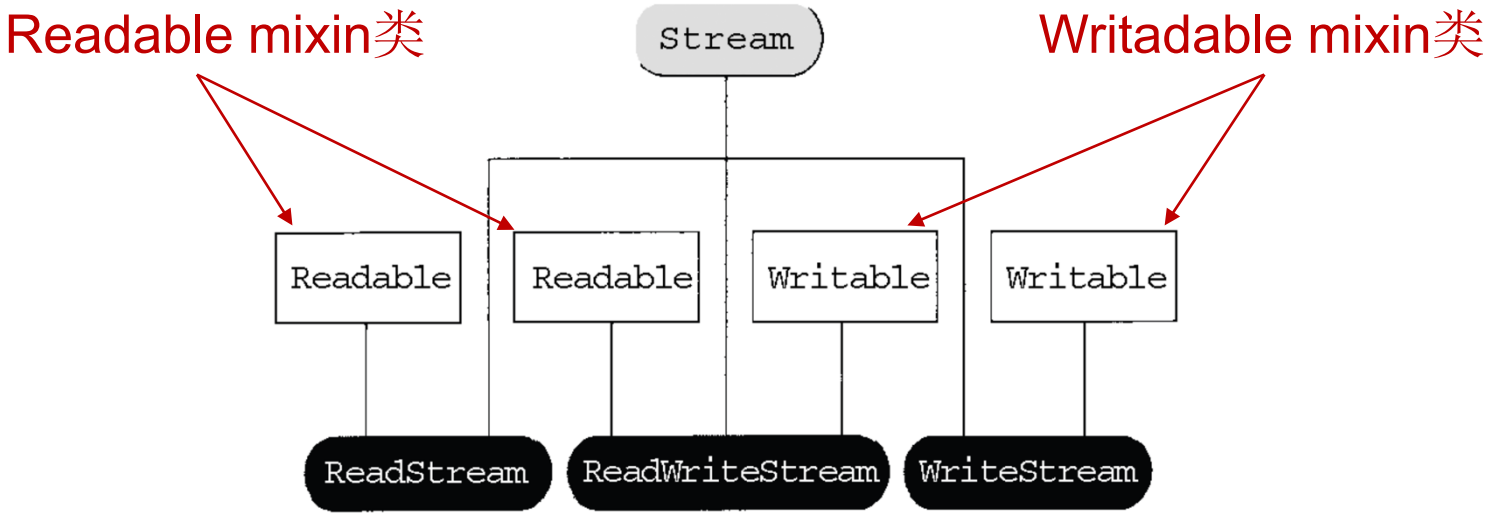
受限的多重继承

- 多重继承到底引入了什么复杂性？
  - **松本行弘**：多继承非常灵活和强大，但就像goto一样，它让继承结构变得复杂，形成所谓“意大利面条继承”。而要想有效使用继承去不为之所累，就需要**控制手段——Mix-in**

Mix-in机制

- Mix-in是降低多重继承复杂性的一个技术，最早在**Flavors**语言中使用
- 实现Mix-in不需要编程语言提供特别的功能，但需要设置一些限制(以Ruby为例)：
  - 默认情况下使用单一继承
  - 第二个及以上的父类就必须是Mix-in抽象类
  - 而Mix-in不能单独生成实例，也不能继承自普通类

Mix-in示例



- 在使用Mix-in的类结构中，Stream只有3个子类
- 实际的输入/输出处理用Readable和Writable两个Mix-in类来实现

结构特点

- 从 Stream 的类层次来看，父类是 Stream，负责输入输出的是 ReadStream、WriteStream 和 ReadWriteStream 这 3 个子类，它们形成了非常清晰的树结构，层次很简单，没有变成网状结构。
- 而且，由于 Mix-in 类实现了共用功能，从而避免了复制程序代码

Ruby对Mix-in的支持

- Ruby在语言层面支持加入了对Mix-in的支持
  - 在Ruby中，Mix-in的单位是模块（module）

```
# Stream 类，Object 的子类
class Stream < Object
 # 这里的定义省略
 ...
end

# 定义输入用的 Mix-in
module Readable
 # 定义输入用的方法
 def read
   ...
 end
end
```

```
# 输出用 Mix-in
module Writable
 # 定义输出用的方法
 def write(str)
   ...
 end
end

# 输入用Stream，Stream 的子类
class ReadStream < Stream
 # 继承输入用的Mix-in
 # Ruby 称为include
 include Readable

end
```

```
# 输出用Stream，Stream 的子类
class WriteStream < Stream

 # 继承输出用Mix-in
 include Writable

end

# 输入输出用Stream，Stream 的子类
class ReadWriteStream < Stream
 # 继承输入用Mix-in
 include Readable

 # 继承输出用Mix-in
 include Writable
end
```

模块如何回答多继承问题？

- 思考一个问题：
  - Ruby中的模块实现了Mix-in机制，也体现了对代码复用的支持，但该机制是如何解决多继承的若干问题的？解决的效果呢？
- 模块如何解决：
  - 多继承的命名冲突问题如何解决，即对相同名字的代码块的复用时，如何区别不同代码块？
  - 重复继承问题，即重复对某个类的继承时，子类中该如何放置其拷贝（单独还是多份）？

模块的命名冲突

- 以Ruby为例：
  - 当Ruby的类通过include引入了不同module时，就可能出现命名冲突（本质跟多继承一样）
  - 就像前面用Git类比那样，只要是多个分支合并，就必然存在冲突的可能性（根据墨菲法则，就必然会发生冲突）！
  - Ruby的module使用了加前缀的方式（姑且称之为全路径名策略）：ModuleName::FuncName（跟Python的策略一样）
- 事实上，**全路径名策略**就是目前对此问题的简单而实用的好办法

模块的重复继承

- Ruby中的模块（module）不允许继承自普通类，因此在继承树中，模块根本就没有参与继承树的生成，也就不存在重复继承的问题！

模块的Limitation

- 可以看出，模块实际上是将目标定位在对“实现体的复用”上
  - 通过引入模块，类中就直接集成了模块中实现好的方法体。这有些像聚合关系，但类的聚合还需要实例化才能使用其中的方法体
  - 模块并没有试图在类型体系上进行contribution，这个事情在Ruby中是交给类的继承体系去做的
  - 在本质上，Ruby还是单继承，模块是在解决多继承的第二个父类时开始加入的！
  - 模块并没有丝毫改变多继承命名冲突的本质

混入模块的搜索树

- 当同时存在继承和混入时，方法的搜索策略如下：
  1. Ruby先搜索当前类自己定义的方法
  2. 再搜索该类的混入模块中的方法
  3. 然后再往上一层，搜索父类的（同样顺序）
  4. 对同一层中的多个混入模块，**最后**引入的**先**搜索

再看Mix-in机制

- 从Ruby对Mix-in的支持方式——模块（module）来看，其实Mix-in机制是将多继承带来的好处切分开，分为接口多继承和实现多继承
- Mix-in通过限制实例化和继承能力，使混入类只具备对零碎代码的重用能力，而不是整体的（带类型特征）的重用

思考与讨论

- 最后再思考这样一个问题：
  - 为什么不允许Mix-in具有实例化的能力，这样究竟会带来什么样的问题？

尝试回答此问题

- 在设计Mix-in的时候，其理念是限制多继承的能力。那么是限制哪个能力？
  - 应该是接口多继承的能力
  - 换句话说，就是不允许Mix-in扮演“抽象类型”的角色
- 那么这样做有什么意义？
  - 将能力的关注点局限在实现体的复用上，而不去干扰类型系统
  - 这样类型系统就变成了类继承体系要去操心的事情了

C++中的多继承

- **Bruce Eckel**（TIJ、TIC++作者）：
  - “程序员对多重继承的需求并不是显而易见的，关于在C++中多重继承是否是必要的这一问题存在（现在仍然存在着）着大量的争论。”
  - “不管是什么原因激发我们使用MI，但是要真正使用它将比看上去要难看得多。”
- 从应用的角度来看，C++中的多继承的确实要比前面几种语言难一些

多继承在C++中的位置

- 1989年在AT&T **cfront**发布版（release）2.0中加入了MI，这是C++语言自1.0版以来发生的首次重要变化
- 但从那以后，许多其它特性的加入（最著名的是模板），改变了编程思想，并且使MI的作用处于了次要地位
- 在C++中，多重继承依然属于高级机制，但地位是次要的

接口继承

```
class Printable {
public:
  virtual ~Printable() {}
  virtual void print(ostream&) const = 0;
};
class Intable {
public:
  virtual ~Intable() {}
  virtual int toInt() const = 0;
};
class Stringable {
public:
  virtual ~Stringable() {}
  virtual string toString() const = 0;
};
```

- 定义了三个接口
- 注意，C++中并没有接口关键字，只有接口类（interface class），即仅含有声明，没有数据和函数体
- 除了析构函数外，接口类中都是纯虚函数

```
class Able : public Printable, public Intable,
             public Stringable {
  int myData;
public:
  Able(int x) { myData = x; }
  void print(ostream& os) const { os << myData; }
  int toInt() const { return myData; }
  string toString() const {
    ostringstream os;
    os << myData;
    return os.str();
  }
}
```

- 类Able“实现”了以上3个接口，因为它提供了那些所声明函数的实现
- 此时，Able的实例将具有多“is-a”关系

```
void testPrintable(const Printable& p) { p.print(cout); cout << endl; }
void testIntable(const Intable& n) {  cout << n.toInt() + 1 << endl; }
void testStringable(const Stringable& s) {  cout << s.toString() + "th" << endl; }
int main() {
  Able a(7);
  testPrintable(a);  testIntable(a);  testStringable(a);
} ///:~
```

- 三个测试函数均可以接受Able的对象

- *思考：以上这个示例如果使用模板机制，将会是什么样子？这两种不同的设计方式又有什么差别？*

实现继承(Mix-in)

```
// A "mixin" class.
class Countable {
  long count;
protected:
  Countable() { count = 0; }
  virtual ~Countable() { assert(count == 0); }
public:
  long attach() { return ++count; }
  long detach() {
    return (--count > 0) ? count : (delete this, 0);
  }
  long refCount() const { return count; }
};
```

- 注意，显然这不是一个独立类
- 因为构造函数和析构函数都是保护继承的
- 所以它必须有友元或派生类来使用它

```
class DBConnection : public Database, public Countable {
  DBConnection(const DBConnection&); // Disallow copy
  DBConnection& operator=(const DBConnection&);
protected:
  DBConnection(const string& dbStr) throw(DatabaseError)
  : Database(dbStr) { open(); }
  ~DBConnection() { close(); }
public:
  static DBConnection*
  create(const string& dbStr) throw(DatabaseError) {
    DBConnection* con = new DBConnection(dbStr);
    con->attach();
    assert(con->refCount() == 1);
    return con;
  }
  // Other added functionality as desired...
};
```

- 这里通过继承使用Countable类和其它类没有区别
- C++语言层面并没有Mix-in的概念，即没有该语言机制和语法

C++中的Mix-in

- 从上面的示例可以看出，C++语言并没有在语言层面上直接对Mix-in提供支持
  - 这意味着C++中没有任何语法（如关键字）或机制来体现Mix-in
- Mix-in仅仅是用户根据自己的理解，在具体使用多继承机制时，设计上的技巧

两者的区别

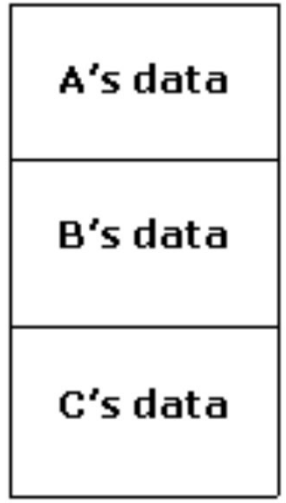
- 关键字支持和用户自己设计上支持有何区别？
  - 关键字的支持：说明编程语言上提供了对这种概念的直接支持（编译器需要负责为此做出努力）
  - 用户自己设计上的支持：只是用户自己在使用该程序设计语言时，遵守了自己设定的规则，从而在使用效果上表现出相同的特征（用户需要自己努力）

重复子对象

- C++在实现继承时，会保留基类的所有数据成员（包括方法和属性）的副本
- 因此派生类的对象，在内存中会包含其基类数据
  - 这与Python、Ruby等动态语言的对象模型不太相同
  - 另外，对于继承树中的虚函数，C++编译器使用了虚函数表来保存，这也同前面提到的语言不太一样

单继承示例

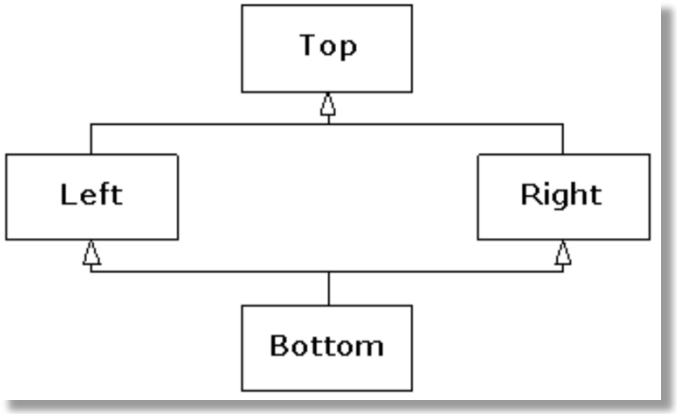
```
class A { int x; };
class B { int y; };
class C : public A, public B { int z; };
int main() {
  cout << "sizeof(A) == " << sizeof(A) << endl;
  cout << "sizeof(B) == " << sizeof(B) << endl;
  cout << "sizeof(C) == " << sizeof(C) << endl;
  C c;
}
// Output:
sizeof(A) == 4
sizeof(B) == 4
sizeof(C) == 12
```



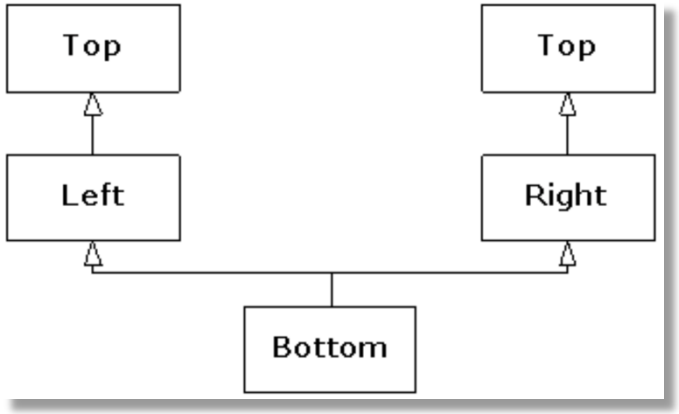
- 对象c以A子对象开头，然后是B子对象部分，最后的数据完全来自C类型本身

多继承示例

```
class Top {
  int x;
public:
  Top(int n) { x = n; }
};
class Left : public Top {
  int y;
public:
  Left(int m, int n) : Top(m) { y = n; }
};
class Right : public Top {
  int z;
public:
  Right(int m, int n) : Top(m) { z = n; }
};
class Bottom : public Left, public Right {
  int w;
public:
  Bottom(int i, int j, int k, int m)
  : Left(i, k), Right(j, k) { w = m; }
};
```



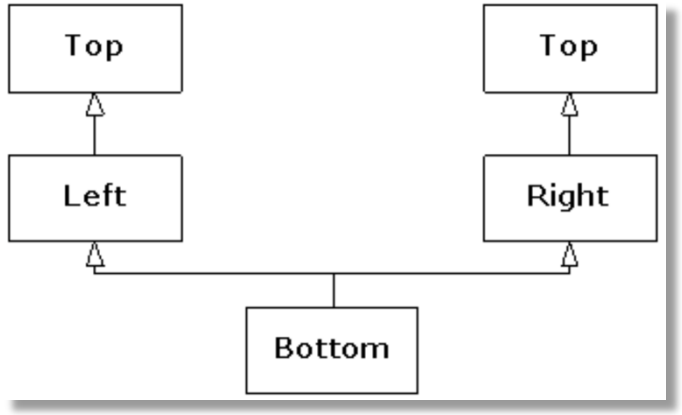
直观上的结构



事实上的结构

问题

```
int main() {
  Bottom b(1, 2, 3, 4);
  cout << sizeof b << endl; // 20
  cout << b.y << endl;  // 3
  cout << b.z << endl;  // 3
  cout << b.w << endl;  // 4
  cout << b.x << endl;  // compiler: error
}
```



- 对象b的长度是20个字节（int为4个字节），所以一个完整的Bottom对象中共有5个int变量
- 这里出现了两个Top的子对象供选择，编译器在b.x位置报错，因为不知道该选择哪一个

虚基类

- 虚基类的设计就是为了解决上面的问题
- 为了实现真正的菱形继承，Left和Right子对象在Bottom对象的内部应该共享一个Top对象
- 通过将Top设计为Left和Right的虚基类，就可以实现这种共享

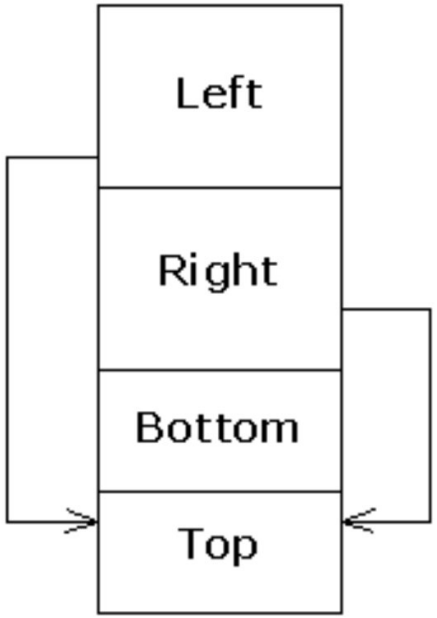
虚基类示例

```
class Top {
protected:
  int x;
public:
  Top(int n) { x = n; }
  virtual ~Top() {}
  friend ostream&
  operator<<(ostream& os, const Top& t) { return os << t.x; }
};
class Left : virtual public Top {
protected:
  int y;
public:
  Left(int m, int n) : Top(m) { y = n; }
};
class Right : virtual public Top {
protected:
  int z;
public:
  Right(int m, int n) : Top(m) { z = n; }
};
```

- 在声明继承Top的头部位置加**virtual**关键字
- 此时，Top为Left和Right的虚基类

虚基类示例

```
class Bottom : public Left, public Right {
  int w;
public:
  Bottom(int i, int j, int k, int m)
     : Top(i), Left(0, j), Right(0, k) { w = m; }
  friend ostream&
  operator<<(ostream& os, const Bottom& b) {
    return os << b.x << ',' << b.y << ',' << b.z
      << ',' << b.w;
  }
};
```



- **特别注意**：此时，虚基类Top的初始化工作只能交给最后派生类（most derived class），即Bottom类

复杂性

- 虚基类的使用明显又增加了C++中对多重继承使用上的复杂性
  - 用户要负责找出菱形继承的相同父类
  - 在所有对该父类派生的继承树分支上使用虚基类
  - 用户还要记住在最后派生类中对虚基类进行初始化

名字冲突

```
class Top {
public:
  virtual ~Top() {}
};
class Left : virtual public Top {
public:
  void f() {}
};
class Right : virtual public Top {
public:
  void f() {}
};
class Bottom : public Left, public Right {
public:
  using Left::f;
};
```

- C++使用基类名来限定有冲突的函数名
- 在同一个层次上的不同分支存在的同名函数会发生冲突

不冲突的情况

```
class Top {
public:
  virtual ~Top() {}
  virtual void f() {}
};

class Left : virtual public Top {
public:
  void f() {}
};

class Right : virtual public Top {};

class Bottom : public Left, public Right {};

int main() {
  Bottom b;
  b.f();   // Calls Left::f()
} ///:~
```

- 此时的函数名f()不存在冲突
- 函数选择的优先级：处于继承层次的子类方向上的优先级更高

作业

- 考察Go语言中对面向对象的支持，尤其是在继承和多态方面
  - 该语言是一种什么风格的语言？
  - 从面向对象的角度来理解，Go语言与Java语言的区别主要有哪些？

阅读资料

1. 人月神话（40周年版），Frederick Brooks，清华大学出版社，2015
2. Python学习手册（第4版），Mark Lutz，机械工业出版社，2011
3. Ruby元编程（第2版），Paolo Perrotta，华中科技大学出版社，2015
4. 松本行弘的程序世界， 松本行弘，人民邮电出版社，2011
5. 编程的修炼，Edsger W. Dijkstra，电子工业出版社，2013
6. 面向对象分析与设计（第3版），Grady Booch等，电子工业出版社，2012
7. Think in Java 4th, Bruce Eckel, 2006
8. Think in C++ 2nd, Bruce Eckel, 2004
9. 深度探索C++对象模型，Stanley Lippman，电子工业出版社，2015

附录1：Python安装

- 下载安装
  - Mac10系统自带的是Python2.7
  - 从Python官网下载安装：https://www.python.org/downloads/
- Sell中使用
  - 启动2.x的命令：python
  - 启动3.x的命令：python3
- 推荐IDE
  - PyCharm：https://www.jetbrains.com/pycharm/