3.2 程序设计语言中的OOP机制

3.2.2 对象

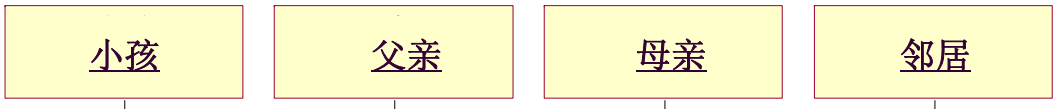
- 对象:
  - 某个特定类的实例。
  - 用同一个类产生的对象具有相同的数据结构、操作集合和能见度，不同的标识，相同或不同的初始状态，拥有和保持不同的运行状态。
  - 对象之间通过消息传递方式进行交互。
- 对象与一般数据类型之实例的区别:
  - 对象是主动的数据，对象之间通过消息传递方式进行通信，而一般数据只能被动地由过程来加工。

3.2 程序设计语言中的OOP机制

3.2.2 对象

例1:

- 主动数据
- 同步消息传递

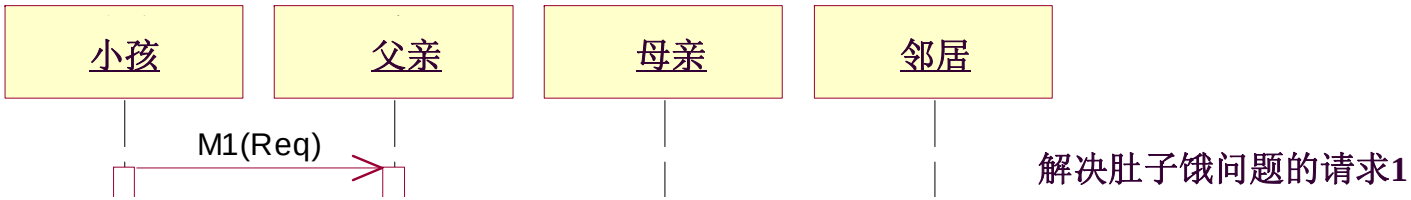


3.2 程序设计语言中的OOP机制

3.2.2 对象

例1:

- 主动数据
- 同步消息传递

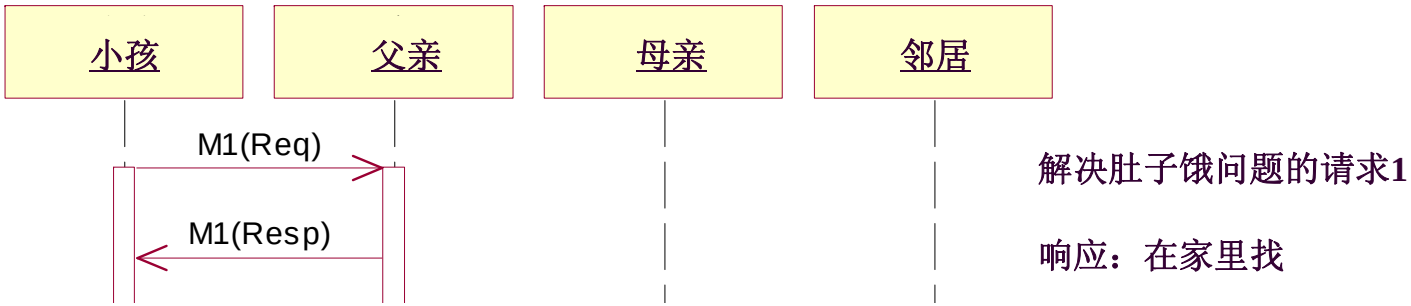


3.2 程序设计语言中的OOP机制

3.2.2 对象

例1:

- 主动数据
- 同步消息传递

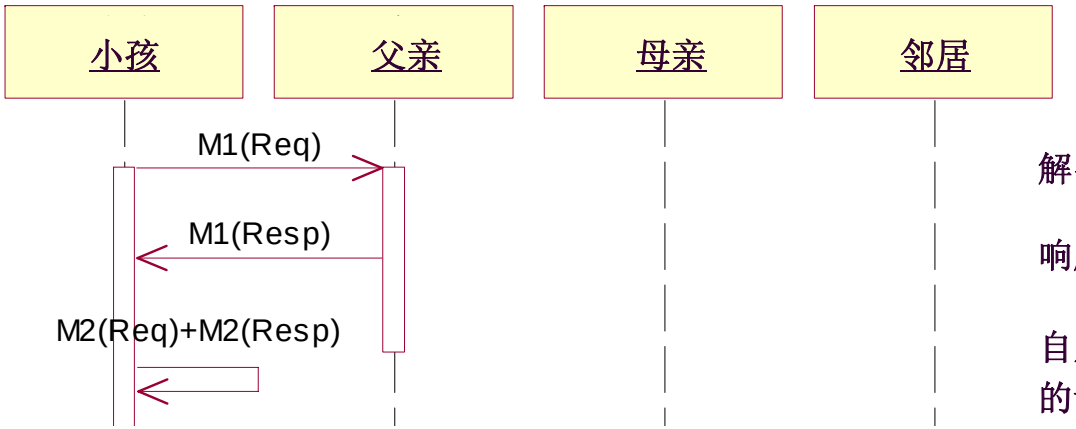


3.2 程序设计语言中的OOP机制

3.2.2 对象

例1:

- 主动数据
- 同步消息传递



解决肚子饿问题的请求1

响应：在家里找

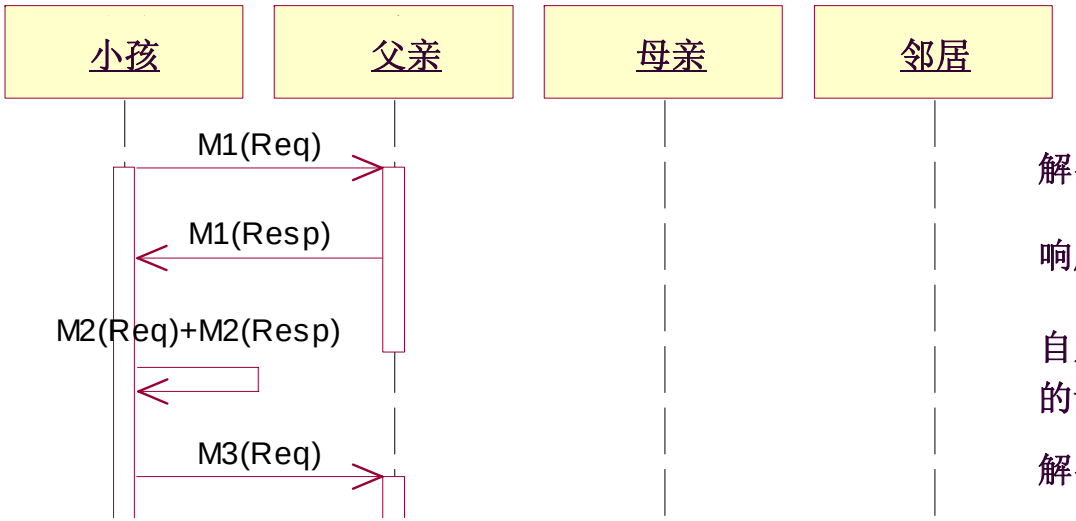
自发自收解决肚子饿问题的请求和响应（失败）

3.2 程序设计语言中的OOP机制

3.2.2 对象

例1:

- 主动数据
- 同步消息传递



解决肚子饿问题的请求1

响应：在家里找

自发自收解决肚子饿问题的请求和响应（失败）

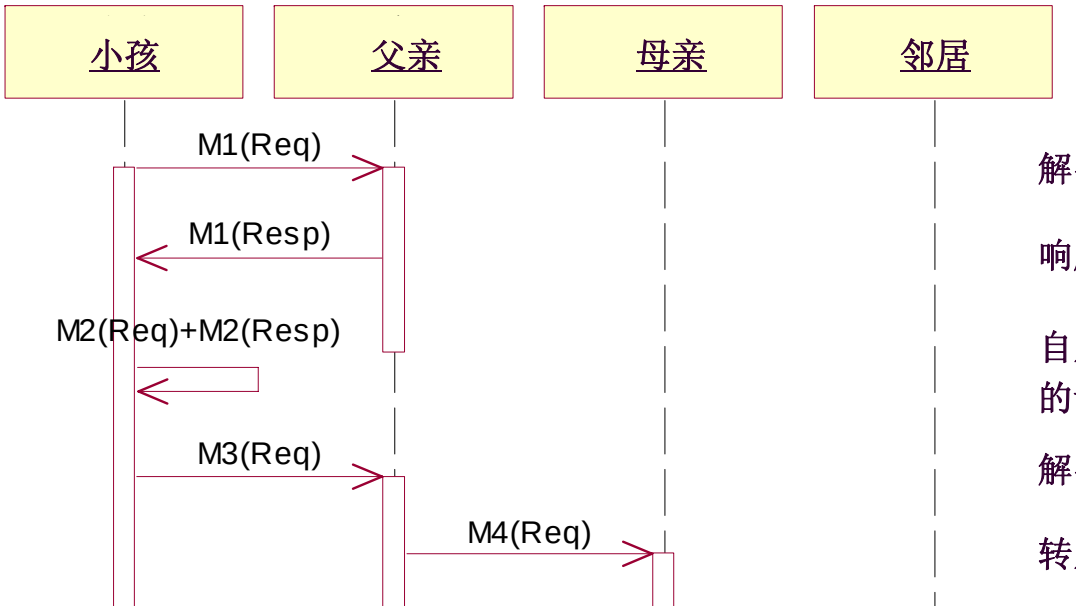
解决肚子饿问题的请求2

3.2 程序设计语言中的OOP机制

3.2.2 对象

例1:

- 主动数据
- 同步消息传递



解决肚子饿问题的请求1

响应：在家里找

自发自收解决肚子饿问题的请求和响应（失败）

解决肚子饿问题的请求2

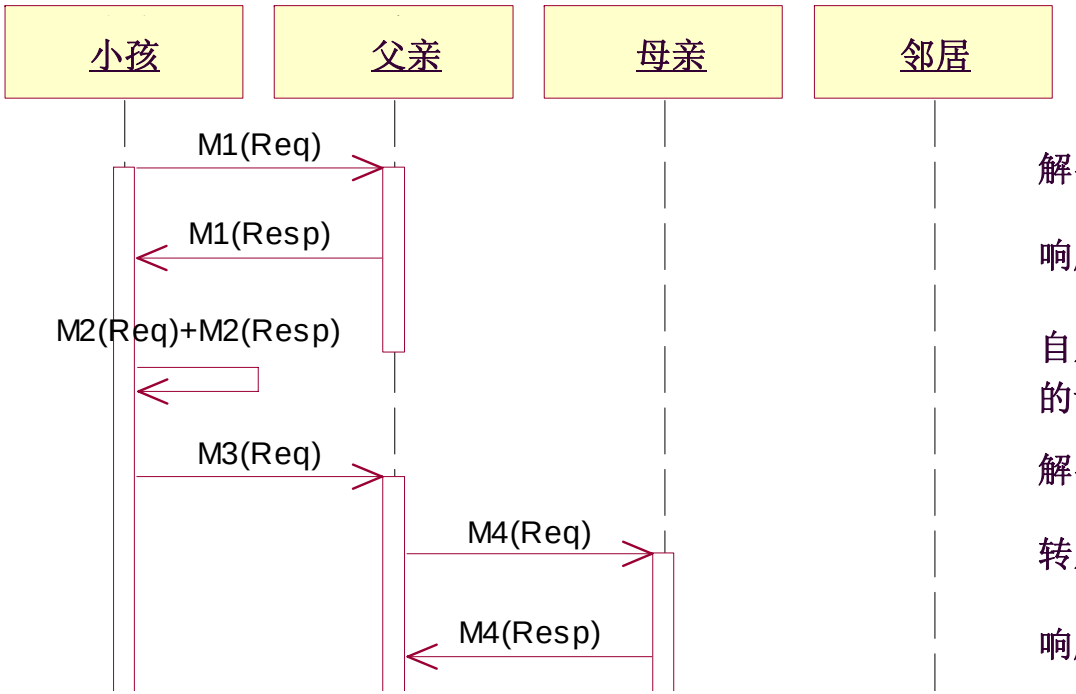
转发请求

3.2 程序设计语言中的OOP机制

3.2.2 对象

例1:

- 主动数据
- 同步消息传递



解决肚子饿问题的请求1

响应：在家里找

自发自收解决肚子饿问题的请求和响应（失败）

解决肚子饿问题的请求2

转发请求

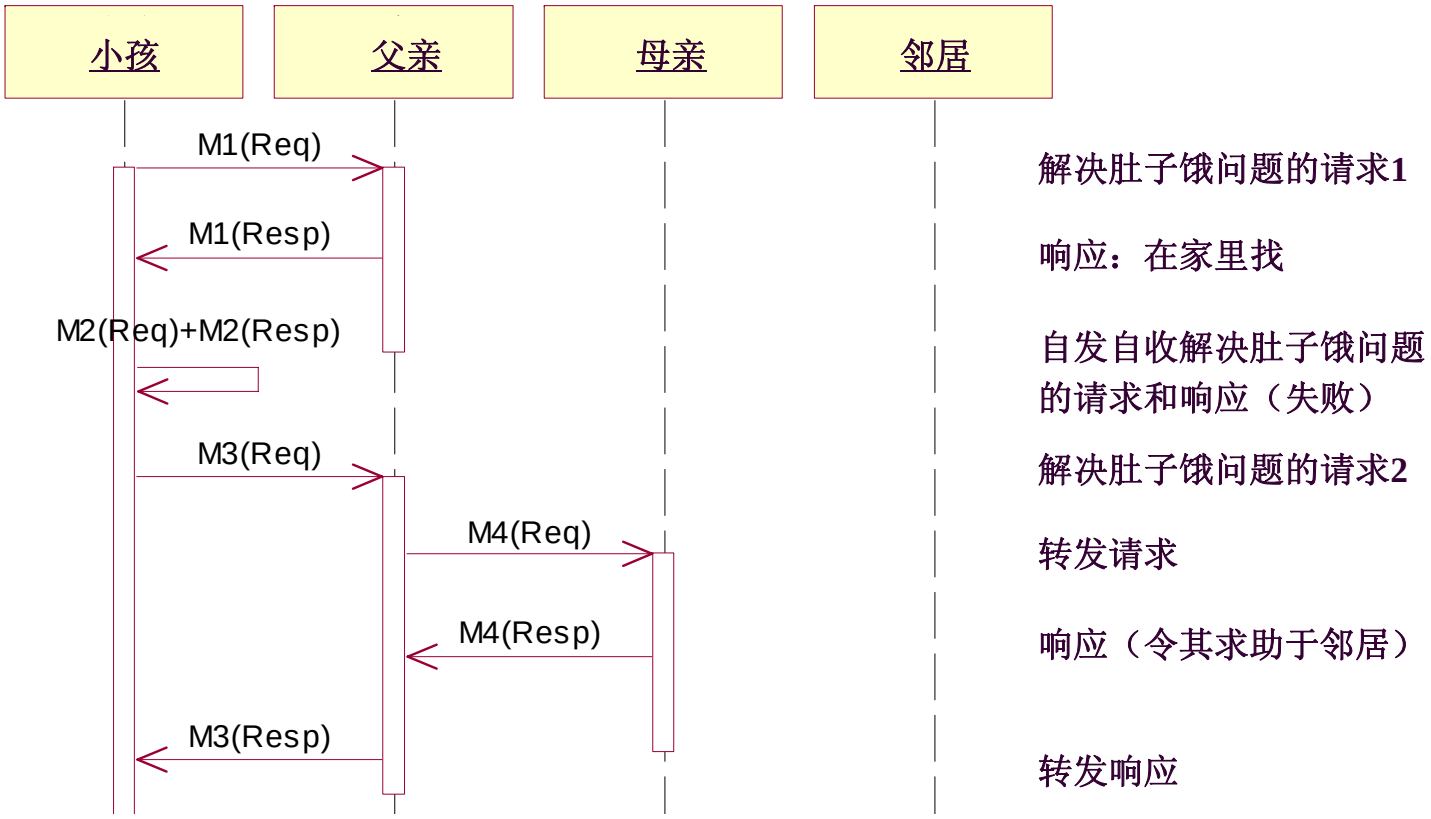
响应（令其求助于邻居）

3.2 程序设计语言中的OOP机制

3.2.2 对象

例1:

- 主动数据
- 同步消息传递

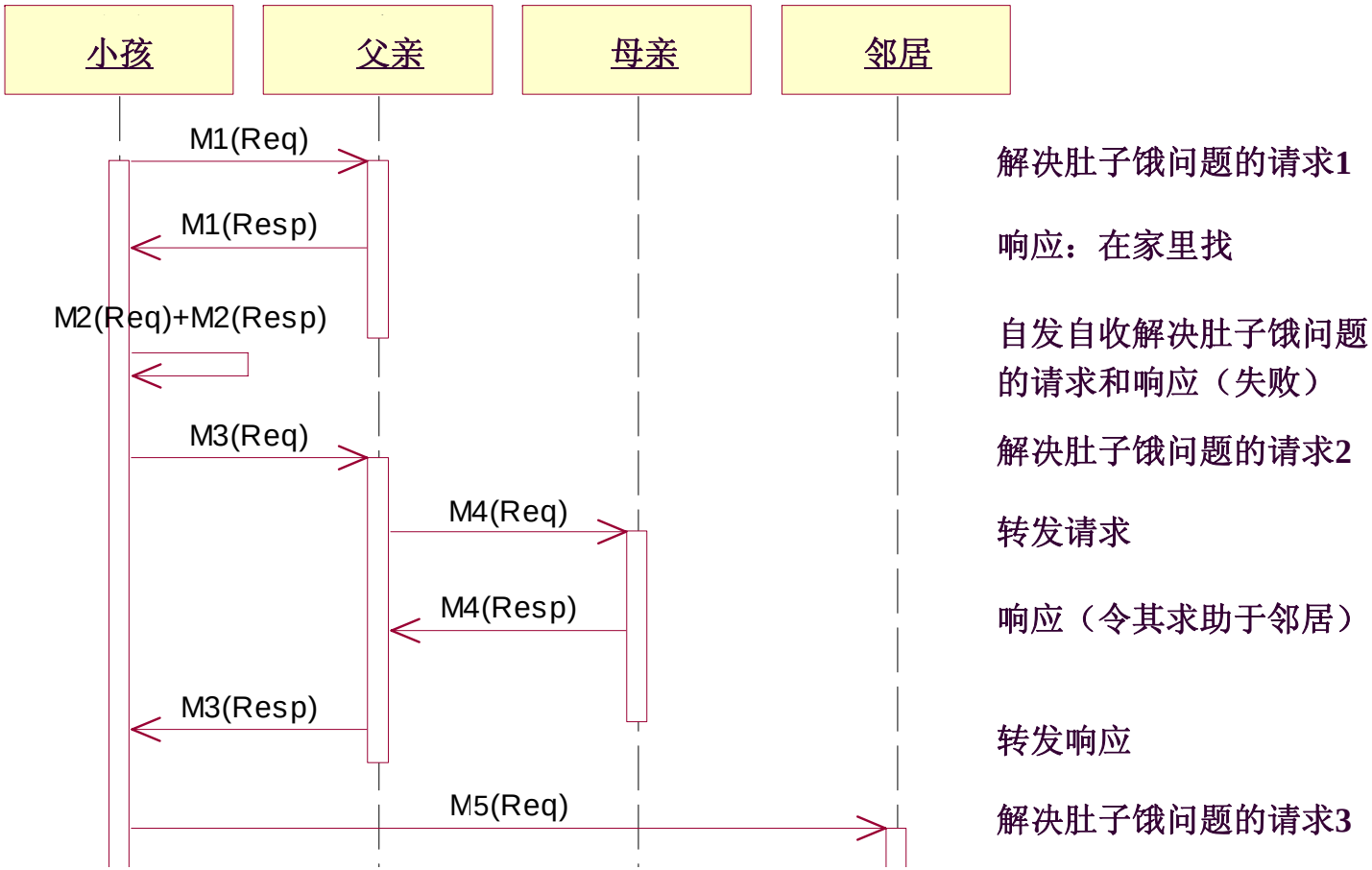


3.2 程序设计语言中的OOP机制

3.2.2 对象

例1:

- 主动数据
- 同步消息传递

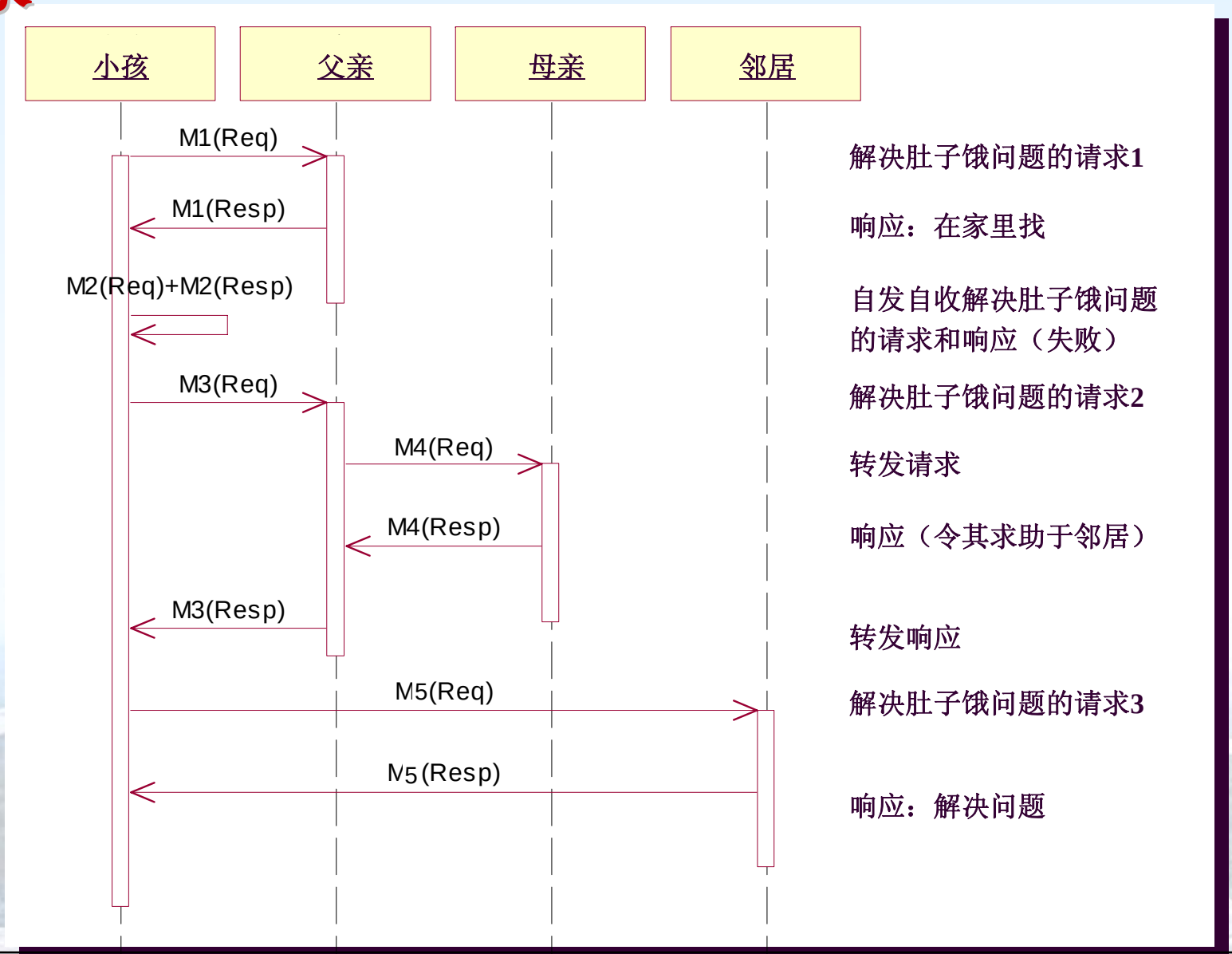


3.2 程序设计语言中的OOP机制

3.2.2 对象

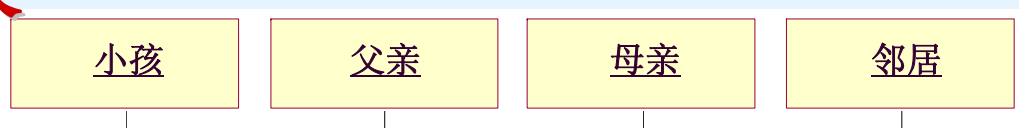
例1:

- 主动数据
- 同步消息传递



3.2 程序设计语言中的OOP机制

3.2.2 对象



例2:

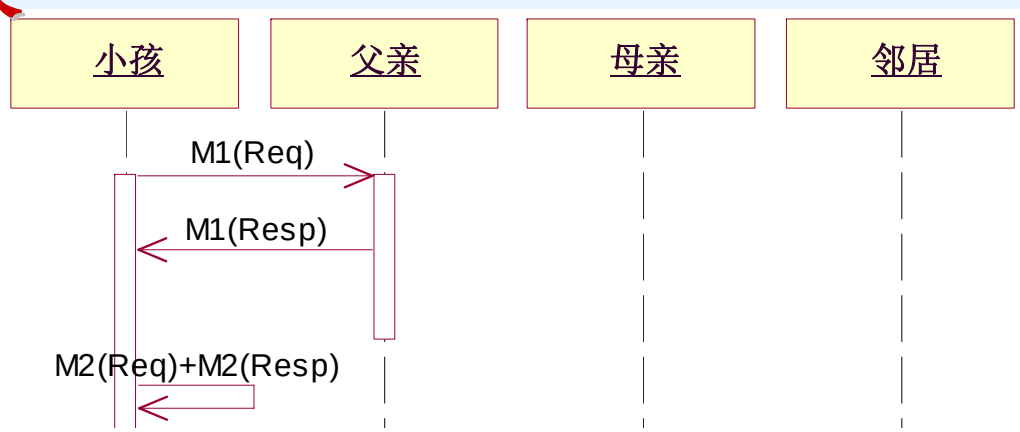
- 主动数据
- 异步消息传递
- 状态保持

3.2 程序设计语言中的OOP机制

3.2.2 对象

例2:

- 主动数据
- 异步消息传递
- 状态保持



解决肚子饿问题的请求1

响应：在家里找

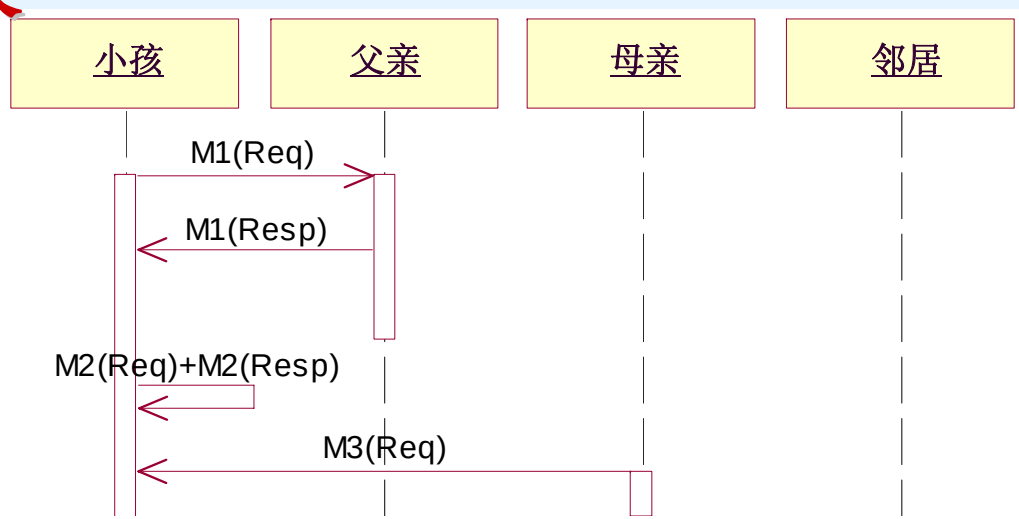
自发自收解决肚子饿问题的请求和响应（失败）

3.2 程序设计语言中的OOP机制

3.2.2 对象

例2:

- 主动数据
- 异步消息传递
- 状态保持



解决肚子饿问题的请求1

响应：在家里找

自发自收解决肚子饿问题的请求和响应（失败）

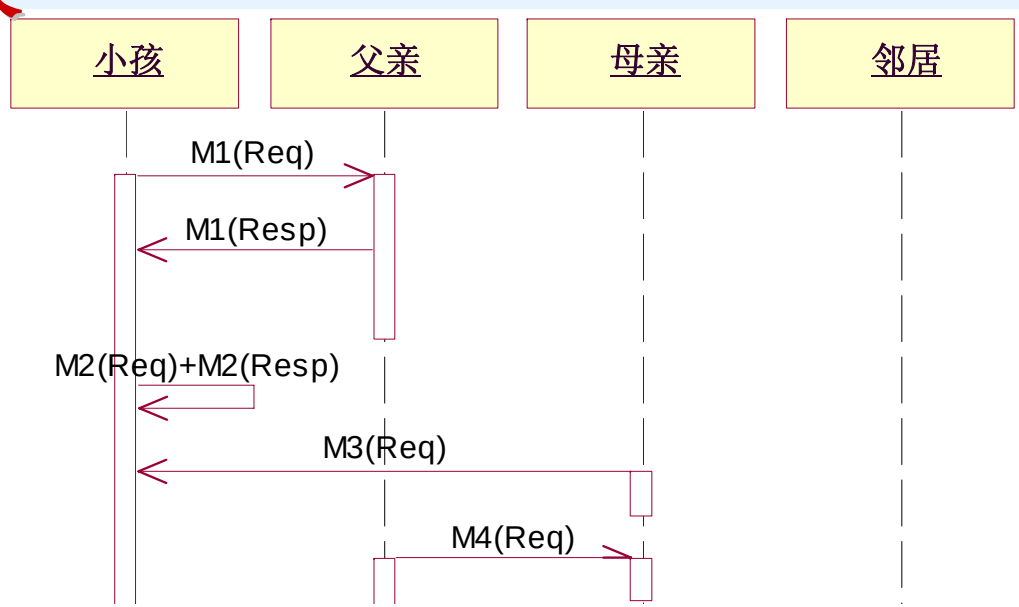
主动询问是否肚子饿？

3.2 程序设计语言中的OOP机制

3.2.2 对象

例2:

- 主动数据
- 异步消息传递
- 状态保持



解决肚子饿问题的请求1

响应：在家里找

自发自收解决肚子饿问题的请求和响应（失败）

主动询问是否肚子饿？

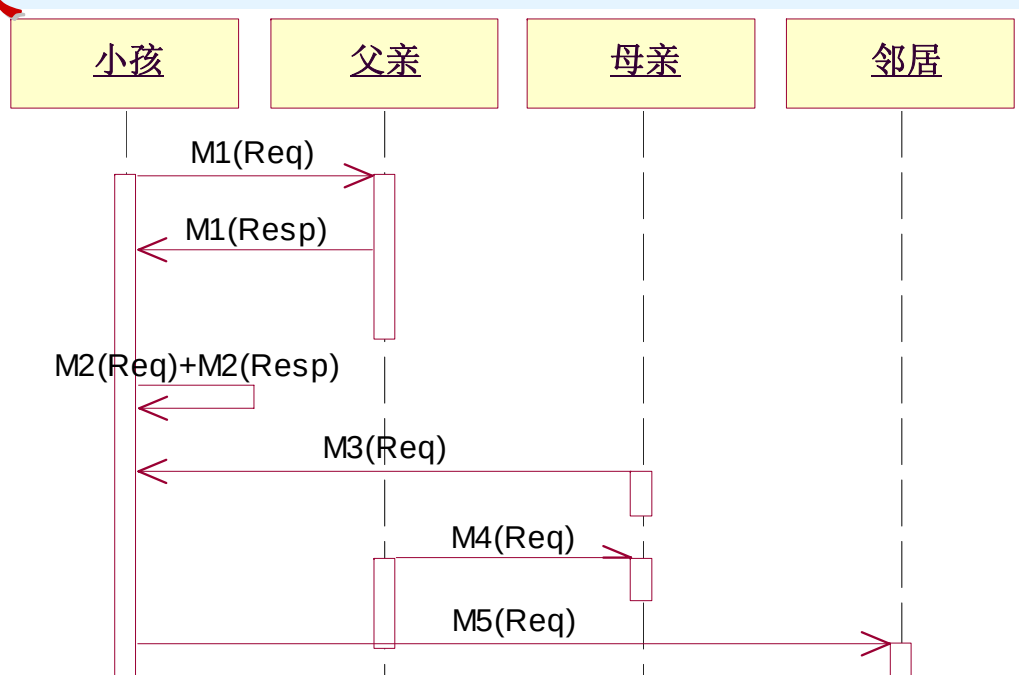
异步请求：如何是好？

3.2 程序设计语言中的OOP机制

3.2.2 对象

例2:

- 主动数据
- 异步消息传递
- 状态保持



解决肚子饿问题的请求1

响应：在家里找

自发自收解决肚子饿问题的请求和响应（失败）

主动询问是否肚子饿？

异步请求：如何是好？

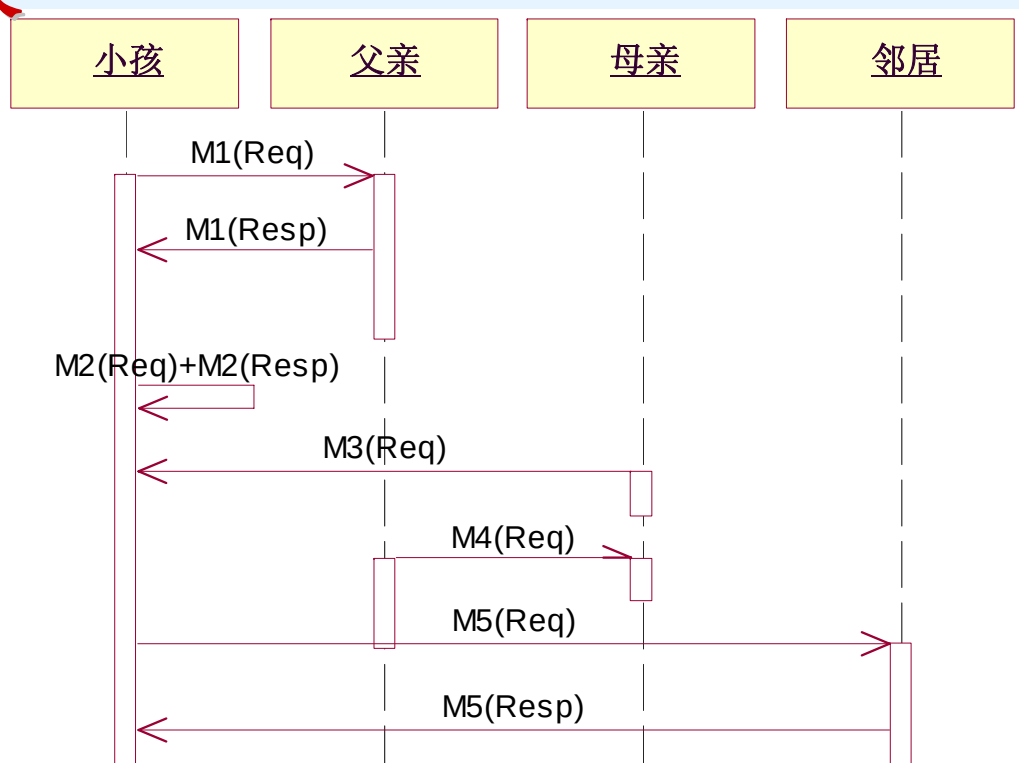
解决肚子饿问题的请求2

3.2 程序设计语言中的OOP机制

3.2.2 对象

例2:

- 主动数据
- 异步消息传递
- 状态保持



解决肚子饿问题的请求1

响应：在家里找

自发自收解决肚子饿问题的请求和响应（失败）

主动询问是否肚子饿？

异步请求：如何是好？

解决肚子饿问题的请求2

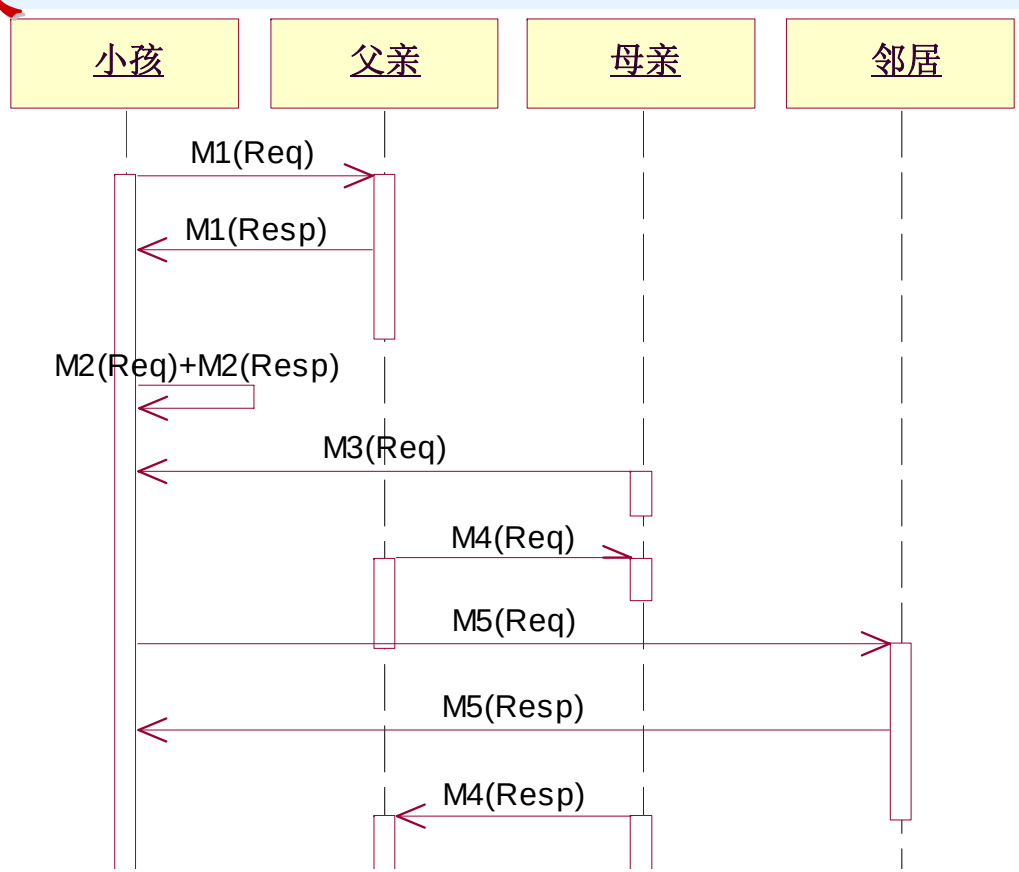
响应：解决问题

3.2 程序设计语言中的OOP机制

3.2.2 对象

例2:

- 主动数据
- 异步消息传递
- 状态保持



解决肚子饿问题的请求1

响应：在家里找

自发自收解决肚子饿问题的请求和响应（失败）

主动询问是否肚子饿？

异步请求：如何是好？

解决肚子饿问题的请求2

响应：解决问题

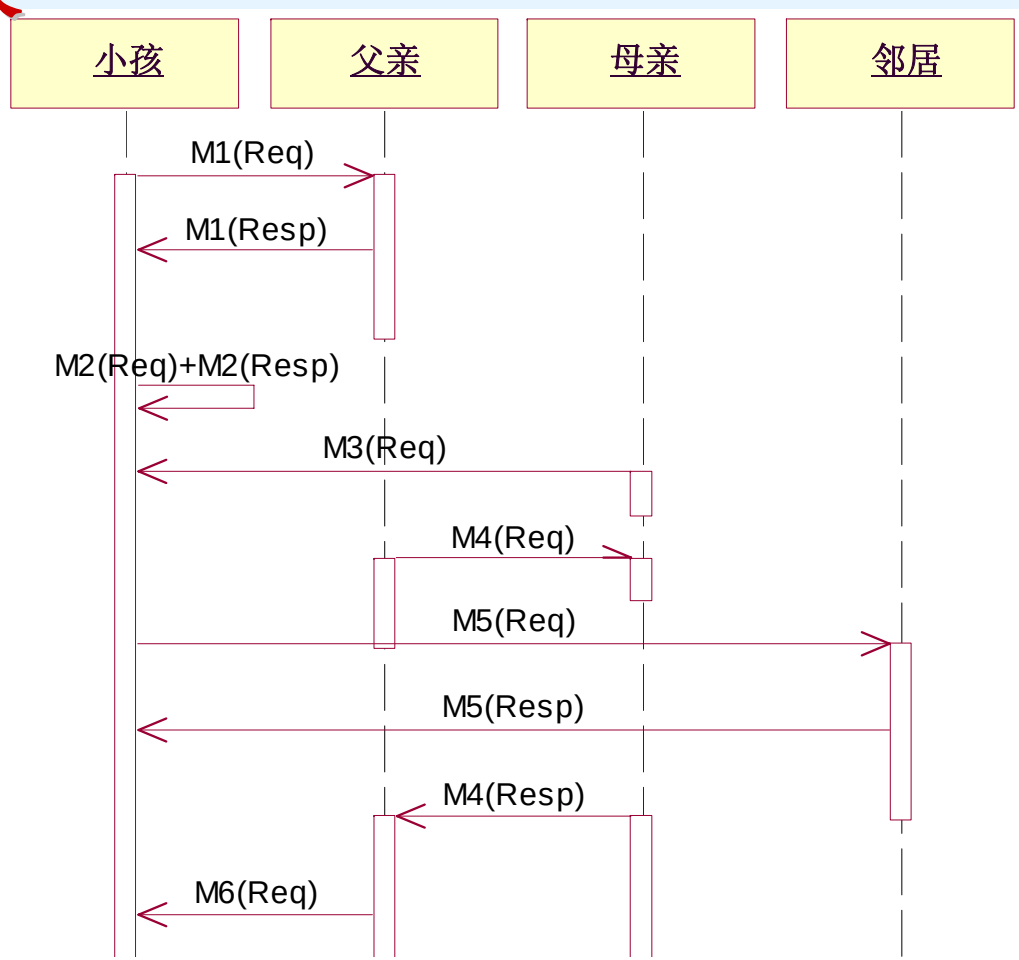
对异步请求的响应：已询问、需落实

3.2 程序设计语言中的OOP机制

3.2.2 对象

例2:

- 主动数据
- 异步消息传递
- 状态保持



解决肚子饿问题的请求1

响应：在家里找

自发自收解决肚子饿问题的请求和响应（失败）

主动询问是否肚子饿？

异步请求：如何是好？

解决肚子饿问题的请求2

响应：解决问题

对异步请求的响应：已询问、需落实

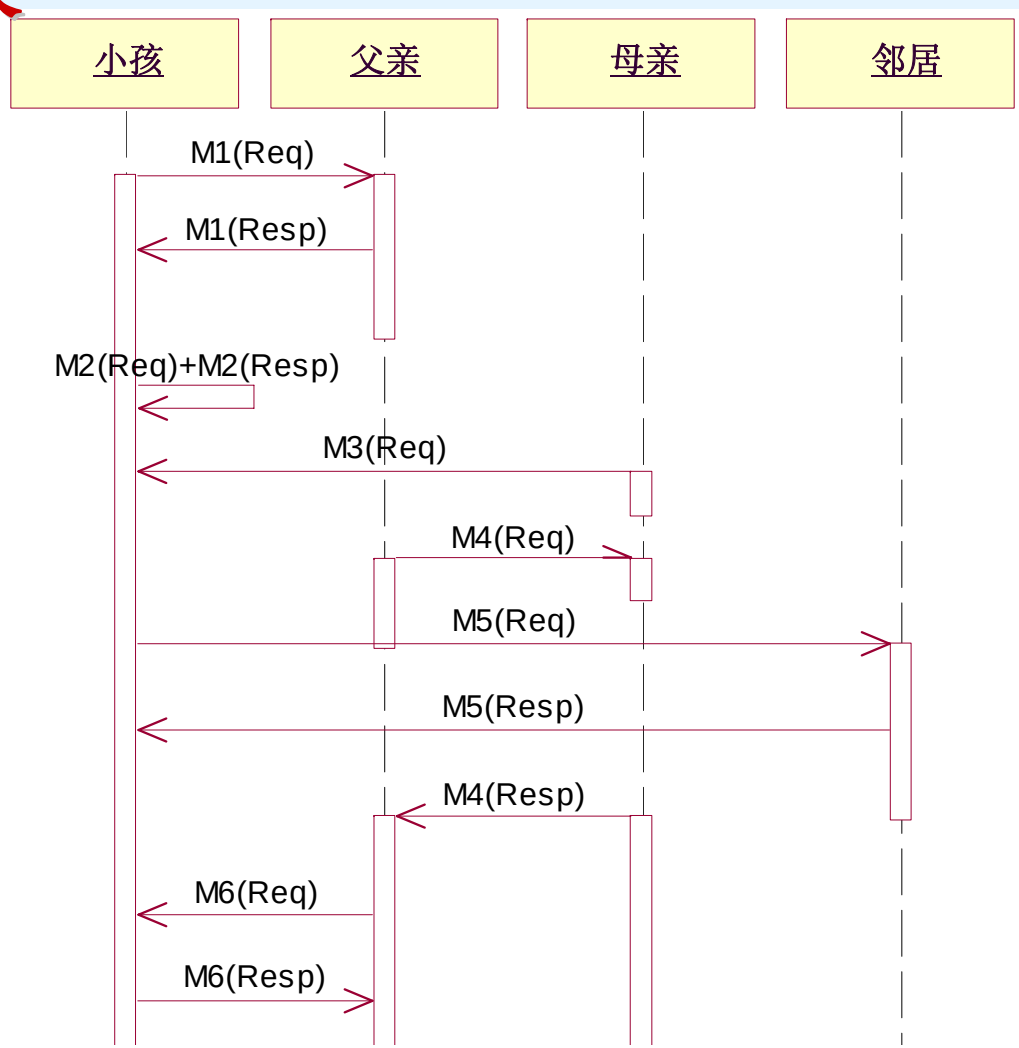
询问

3.2 程序设计语言中的OOP机制

3.2.2 对象

例2:

- 主动数据
- 异步消息传递
- 状态保持



解决肚子饿问题的请求1

响应：在家里找

自发自收解决肚子饿问题的请求和响应（失败）

主动询问是否肚子饿？

异步请求：如何是好？

解决肚子饿问题的请求2

响应：解决问题

对异步请求的响应：已询问、需落实

询问

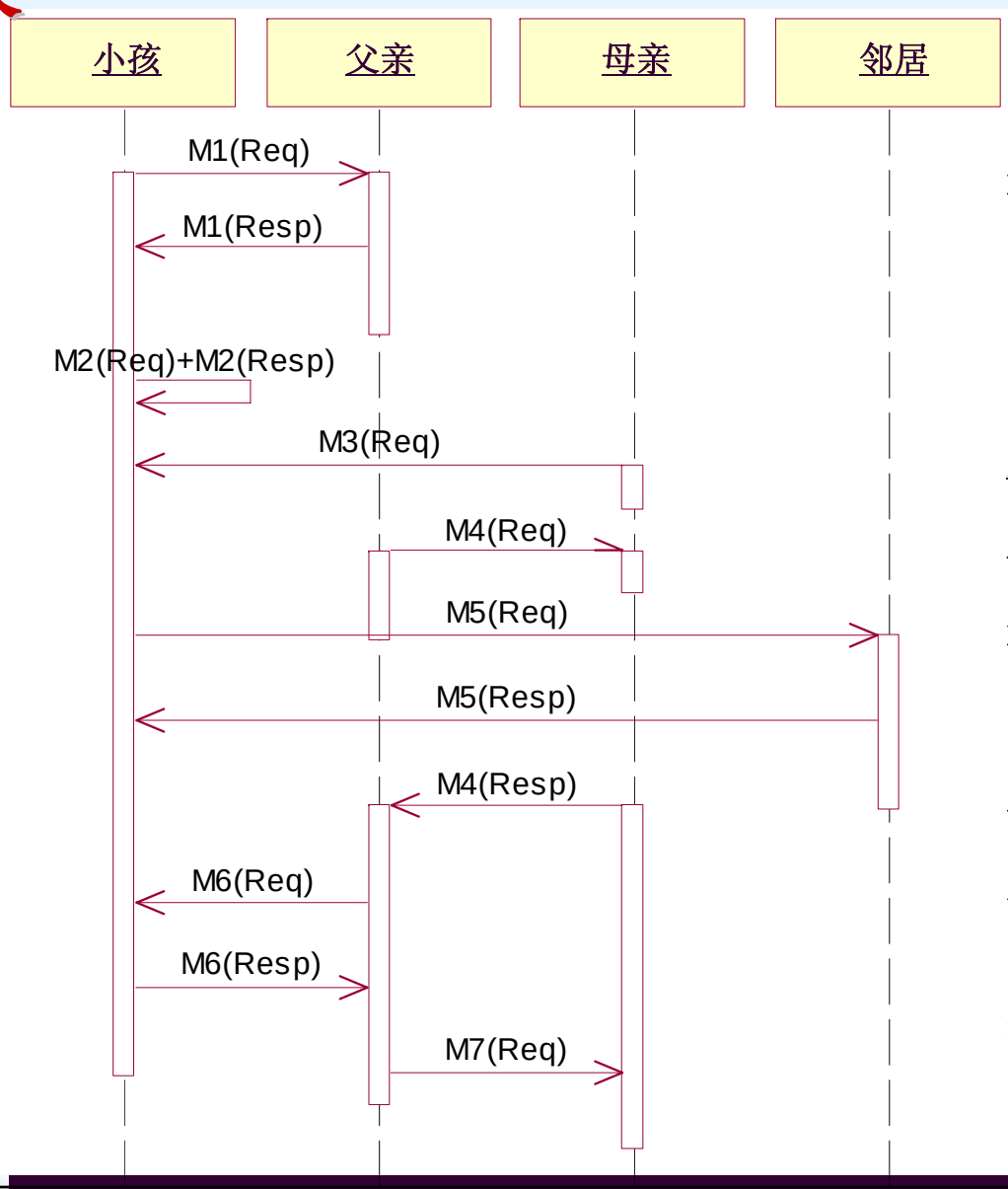
响应（已解决，状态保

3.2 程序设计语言中的OOP机制

3.2.2 对象

例2:

- 主动数据
- 异步消息传递
- 状态保持



解决肚子饿问题的请求1

响应：在家里找

自发自收解决肚子饿问题的请求和响应（失败）

主动询问是否肚子饿？

异步请求：如何是好？

解决肚子饿问题的请求2

响应：解决问题

对异步请求的响应：已询问、需落实

询问

响应（已解决，状态保持）

回复

3.2 程序设计语言中的OOP机制

3.2.2 对象

- 大部分面向对象语言令每个对象有独立的数据空间（以保持各自的状态），同一个类的对象共享操作代码（以节省代码空间、保证操作一致性）。
- 大部分面向对象语言支持类属性（即同一个类的对象共享的数据），如 C++ 中的静态数据成员。
- 大部分面向对象语言采用为对象连续分配数据空间的策略（以支持继承性的实现）。

3.2 程序设计语言中的OOP机制

3.2.2 对象

- 对象之间的关联是设计的难点，一般可以通过以下手段实现:
  - 对象自己的数据结构（例如指针类型的数据成员）。
  - 消息传递的参数。
  - 在起组织和控制作用的其他类型的对象中以关联表方式定义的数据成员以及相应的对象管理操作。
  - 全局的关联表以及相应的对象管理操作。

3.2 程序设计语言中的OOP机制

3.2.3 消息传递和方法

- 使得对象成为主动的数据，依赖于语言中消息传递机制和方法的支持。
- 在一个基于消息传递的通信系统中，实体之间通过消息来进行交互。
- 在两个实体之间通信，其必要条件是:
  （1）它们之间至少存在一条信道;
  （2）遵循同一种通信协议。
- 发送一条消息时，须指明信道或给出决定信道的方法，最常见的是用接收方的标识（如名字）来命名信道。

3.2.3 消息传递和方法

- 发送一条消息的典型方式:

send <*expression-list*> *to* <*destination-designator*>

其中，send … to … 是消息传递通信原语，发送的消息值在 <expression-list> 中，而 <destination-designator> 则指明了接收方。

3.2.3 消息传递和方法

与过程调用相比，消息传递的区别是:

- 必须给出关于信道的信息，通常要显式指明接收方。
  其优点是:
  - 可以对消息的传递范围加以约束，使接收方显式地表示为消息所规定的操作承受对象（过程调用所使用的信道则是隐含的，只要接口语法合法，便无条件地被调用）。
  - 可以适用于并发和分布环境，可以用接收方集合来（可以是动态地）约束合法消息传递的范围。

3.2 程序设计语言中的OOP机制

3.2.3 消息传递和方法

与过程调用相比，消息传递的区别是:

- 认为接收方具有状态保持能力，即消息有后效，而过程调用则一般是没有后效的。
- 消息传递可以是异步的，而过程调用则只能是同步的。

3.2.3 消息传递和方法

- 对于一个类来说，
  - 它关于方法接口的定义，就规定了外界与其实例的消息传递协议，
  - 而这个类本身则（由其类型以及与其他类型的关联关系）决定了消息传递的合法范围（即：使用这组消息传递协议的接收方只能是该类及其子类的实例）。
- 由于类是先于对象构造而成的，所以一个类为它的实例提供了可以预知的对外通信方式。

3.2 程序设计语言中的OOP机制

3.2.3 消息传递和方法

```
class Employee {
  char *Name;
  int   Age;
public:
  void changeAge(int newAge);
  void retire();
  Employee( char *n, int a );
  ~Employee();
};
```

```
#include "Employee.h"
void Employee::changeAge(int newAge)
{ Age = newAge; }
/* ... */
Employee::Employee( char *n, int a )
{ Name = new char[25];
  strncpy(Name, n, 24);
  Age = a;
}
Employee::~Employee()
{ delete [] Name; }
```

```
#include "Employee.h"
void userFunc()
{ Employee e1("张三",24),
     *e2 = new Employee("李四",60);
  e1.changeAge(25);
  e2->changeAge(61);
  delete e2;
}
```

消息传递

实例生成

实例消除 e1

实例消除 *e2

3.2 程序设计语言中的OOP机制

3.2.3 消息传递和方法

- 在上例中，消息传递 e1.changeAge(25); 的语义是:
  - 消息的接收方：类型为 Employee 的对象 e1;
  - 消息值：用参数 25 来调用方法 changeAge。
  - 符号“.”：相当于一种 send to … with … 形式的通信原语。
- 如果不采用消息传递方式，则需要将对象用参数代入普通 C 函数进行加工，如:

```
changeAgeInC(&e1, 25); /* 将不得不解除对e1的封装 */
```

3.2 程序设计语言中的OOP机制

3.2.3 消息传递和方法

- 方法体虽然也是一段程序代码，但与传统的过程体有一些本质差别：
  - 方法体中将直接引用对应类中定义的数据成员，这些数据成员被该类中的所有方法所共享。
  - 定义时必须把所定义的操作抽象于接收消息的具体对象（因为在定义类的时候还无法预知、也无法穷举那些与特定的方法相关的对象）。
  - 使用时必须把所定义的操作关联于接收消息的具体对象（因为必须通过具体的对象才能体现特定的方法所定义的对象行为）。

3.2 程序设计语言中的OOP机制

3.2.3 消息传递和方法

是什么机制支撑了
这样的双重作用？

抽象于具体对象

```
#include "Employee.h"
void Employee::changeAge(int newAge)
{ Age = newAge; }
/* ... */
Employee::Employee( char *n, int a )
{ Name = new char[25];
  strncpy(Name, n, 24);
  Age = a;
}
Employee::~Employee()
{ delete [] Name; }
```

关联于具体对象

```
#include "Employee.h"
void userFunc()
{ Employee e1("张三",24),
     *e2 = new Employee("李四",60);
  e1.changeAge(25);
  e2->changeAge(61);
  delete e2;
}
```

3.2 程序设计语言中的OOP机制

3.2.3 消息传递和方法

- 这样的机制是面向对象语言中的一种特有结构：对象的自身引用（*self-reference*）。
- 这种结构在不同的面向对象语言中有不同的名称，在 C++ 中称为this，在 **Smalltalk-80**、**Objective-C** 等语言中则称为 self。
- 以 C++ 为例，对于类 c 和方法 c::m，在 c::m 的方法体中出现的、c 的成员（包括数据成员和成员函数）名 n，将被编译程序按 this->n 来对待。这里的 this 是一个类型为 c* 的指针，它的值由语言中的消息传递机制提供，它指向当前接收以调用 c::m 为消息内容的那个对象。

3.2 程序设计语言中的OOP机制

3.2.3 消息传递和方法

对象自身引用的双重作用：

- 其值使得方法体中引用的成员名与特定的对象相关。
- 其类型决定了共享方法体的对象范围。

```
#include "Employee.h"
void userFunc()
{ Employee e1("张三",24),
     *e2 = new Employee("李四",60);
  e1.changeAge(25); // this = &e1
  e2->changeAge(61);// this = e2
  delete e2;
}
```

```
#include "Employee.h"
void Employee::changeAge(int newAge)
{ this->Age = newAge; }
/* ... */
Employee::Employee( char *n, int a )
{ this->Name = new char[25];
  strncpy(this->Name, n, 24);
  this->Age = a;
}
Employee::~Employee()
{ delete [] this->Name; }
```

提示：设法让一个程序成分的值和类型同时起不同的、相关的作用，是软件技术中常见的做法。

3.2 程序设计语言中的OOP机制

3.2.3 消息传递和方法

this的值是怎么传进来的?

```
#include "Employee.h"
void Employee::changeAge(int newAge)
{ this->Age = newAge; }
/* ... */
Employee::Employee( char *n, int a )
{ this->Name = new char[25];
  strncpy(this->Name, n, 24);
  this->Age = a;
}
Employee::~Employee()
{ delete [] this->Name; }
```

```
#include "Employee.h"
void userFunc()
{ Employee e1("张三",24),
    *e2 = new Employee("李四",60);
  e1.changeAge(25); // this = &e1
  e2->changeAge(61);// this = e2
  delete e2;
}
```

- 一般采用编译时方法换名和自动插入参数的做法。

3.2 程序设计语言中的OOP机制

3.2.3 消息传递和方法

- 对象自身引用所提供的另一种支持是将对象按整体对待。

```
class dlink{
  dlink *pre;
  dlink *suc;
public:
  void append(dlink *p);
  dlink(){ pre = suc = NULL; }
  // ...
};
```

```
void dlink::append(dlink *p)
{ p->suc = suc;
  p->pre = this;  // 必须整体引用
  if (suc != NULL)
    suc->pre = p;
  suc = p;
}
```

3.2 程序设计语言中的OOP机制

3.2.4 继承性与类层次结构

- 孤立的类只能描述实体集合的特征同一性，而客观世界中实体集合的划分通常还要考虑实体特征方面有关联的相似性。“相似”无非是既有共同点，又有差别。
  - 内涵的相似性：在客观世界中具有“一般－特殊”的关系。例如：雇员和经理。
  - 结构的相似性：具有相似的表示。例如在 **Smalltalk** 类库中，类 Bag 利用类 HashTable 作为基本存储结构。

3.2 程序设计语言中的OOP机制

3.2.4 继承性与类层次结构

- 两种极端的做法：
  - 将相似的事物用不同的类型来表示，自然能够表示其差别，但体现不了它们之间存在共性的事实，且共性的表示也可能不一致，当扩充维护过程中需要对其共性部分进行修改时，就面临着保持一致性的问题。
  - 将相似的事物用相同的类型来表示（例如把可能的特征都定义上去，再根据标识性信息进行投影），自然能够表示其共同点，但体现其差别就十分困难，且失去了类型化的支持。一旦需要扩充和修改（哪怕只是涉及差别部分），也将影响利用此种类型表示的所有其他事物。

3.2.4 继承性与类层次结构

- 继承性：在类之间既能体现其共性和差别，又能给出其间存在共性和差别关系的信息，还能将这样的关系按照需要进行传递的类型化机制。
  - 继承性是面向对象技术实现基于差别的开发的一种主要机制。
  - 继承的传递性将增加类之间的耦合度，故应防止滥用继承。
  - 重视多重继承带来的问题，使用多重继承不可能代替对象组装的设计。

3.2 程序设计语言中的OOP机制

3.2.4 继承性与类层次结构

```
// Employee是Manager的父类
class Employee {
  char *Name;
  int   Age;
public:
  void changeAge(int newAge);
  void retire();
  Employee( char *n, int a );
  ~Employee();
};
```

共性描述

```
#include "Employee.h"
// Manager是Employee的子类
class Manager: public Employee {
  int   Level;
public:
  void changeLevel( int l );
  Manager( char *n, int a, int l );
  ~Manager();
};
```

差别定义

继承的方法

```
#include "Manager.h"
void userFunc3()
{ Manager  m1("王五", 35, 2);
    *m2 = new Manager("赵六", 28, 3);
  m1.changeAge(36);
  m2->changeLevel(2);
}
```

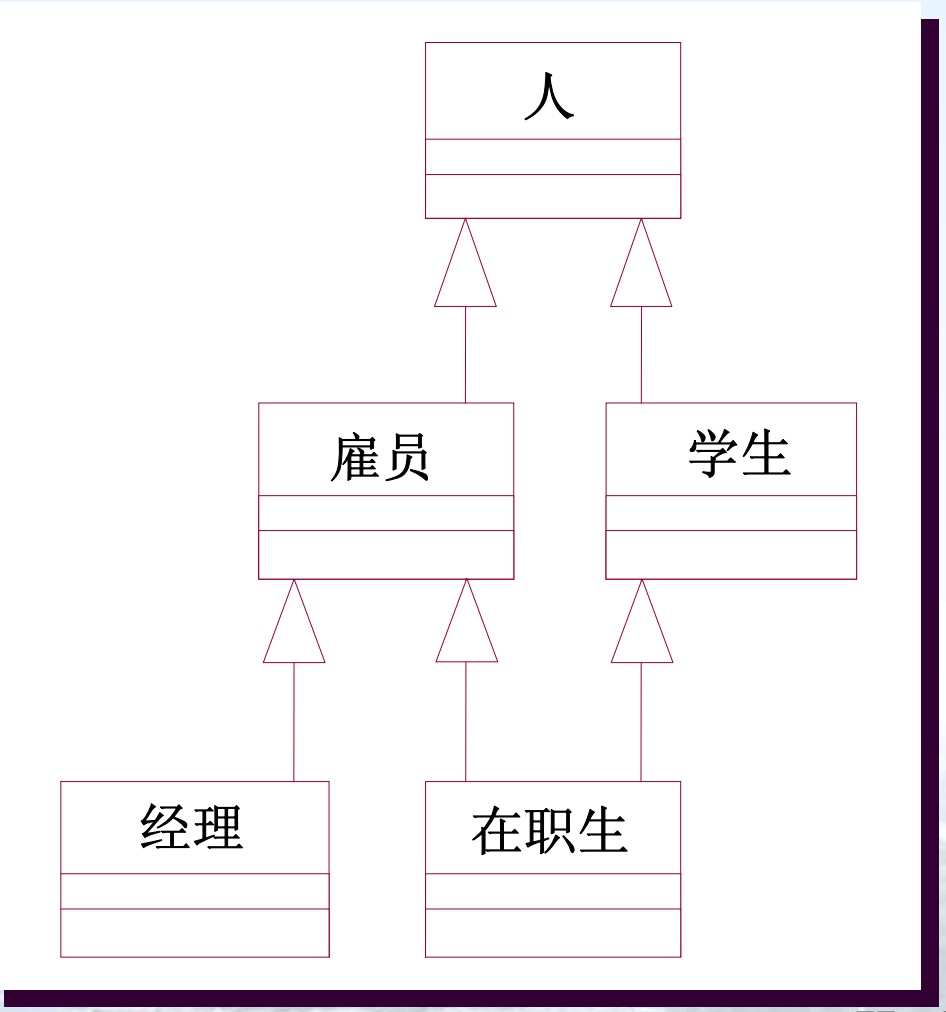
3.2.4 继承性与类层次结构

- 上例中，类 Manager并不是其实例类型的全部定义，另一部分由类Employee定义，体现了类与类型之间的差别。
- 类之间的继承关系是传递的。
- 一个类的所有直接和间接父类（子类）统称为这个类的父类（子类）。
- 只允许一个类最多有一个直接父类－支持单重继承。
- 允许一个类有多个直接父类－支持多重继承。

3.2 程序设计语言中的OOP机制

3.2.4 继承性与类层次结构

- 在一个面向对象的系统中，子类与父类之间的继承关系构成了这个系统的类层次结构，可以用树（对应于单重继承）或格（对应于多重继承）这样的图来描述。



3.2.4 继承性与类层次结构

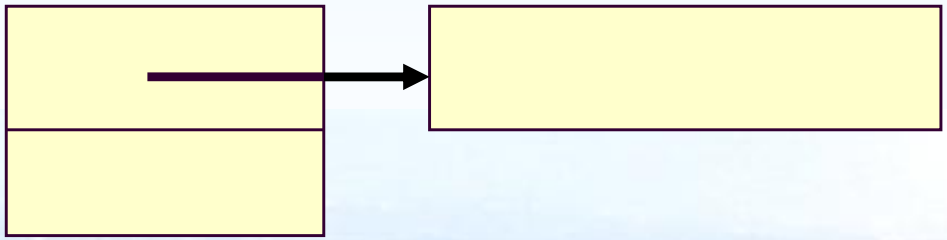
- 由单重继承形成的类层次结构对实例生成和实例消除的作用：
  - 类 c 的实例生成：从类 c 出发，沿继承路径上溯到对应类层次结构分支的顶点（即到达一个没有父类的类），再自上而下地沿该路径逐一调用对应类的 constructor，最后调用 c 的 constructor。
  - 类 c 的实例消除：先调用 c 的 destructor，再从类 c 出发，沿继承路径自下而上逐一调用对应类的 destructor，直到对应类层次结构分支的顶点。

3.2 程序设计语言中的OOP机制

3.2.4 继承性与类层次结构

- 面向对象语言一般采用将子类实例的存储结构“粘接”在父类实例的存储结构之后的存储分配策略，为直接重用父类方法体的二进制码提供了支持。

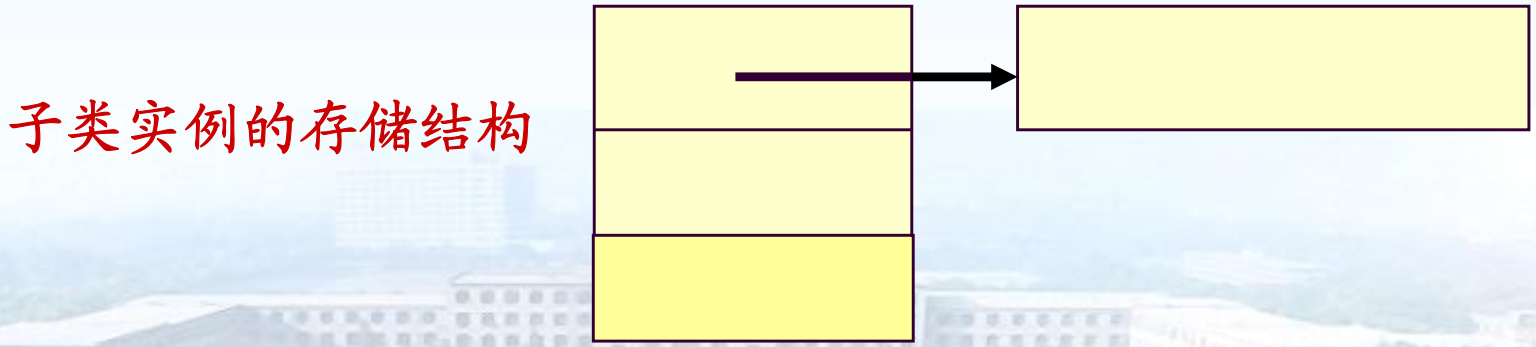
父类实例的存储结构



3.2 程序设计语言中的OOP机制

3.2.4 继承性与类层次结构

- 面向对象语言一般采用将子类实例的存储结构“粘接”在父类实例的存储结构之后的存储分配策略，为直接重用父类方法体的二进制码提供了支持。



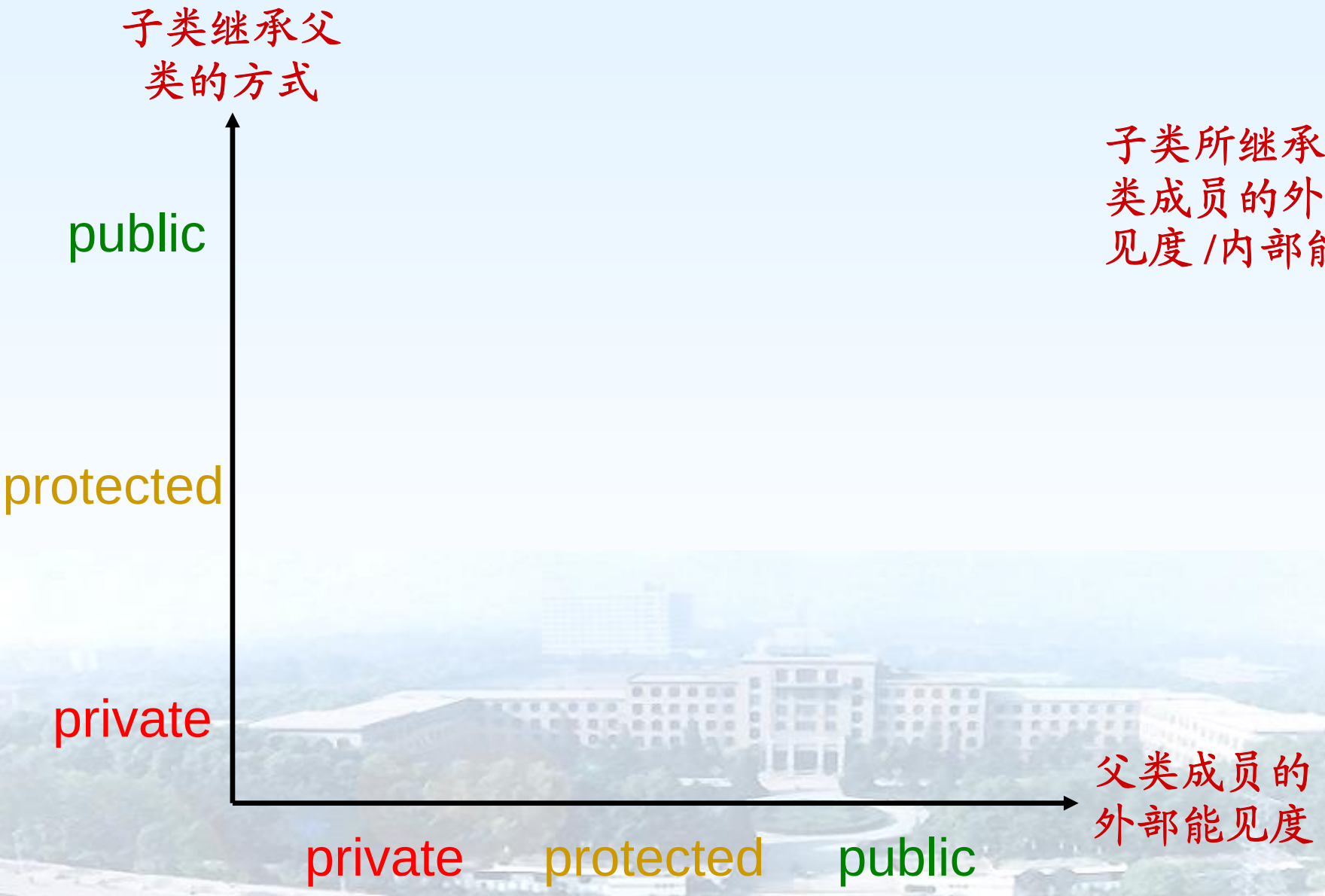
3.2.4 继承性与类层次结构

- 在 C++ 中，类的定义中可以出现三种访问控制程度（能见度）的声明：public、private、protected。
- 继承性使得在父类中定义的能见度也与子类有关，因为父类的成员被子类继承后也是子类的成员，而凡是成员都有能见度的问题。

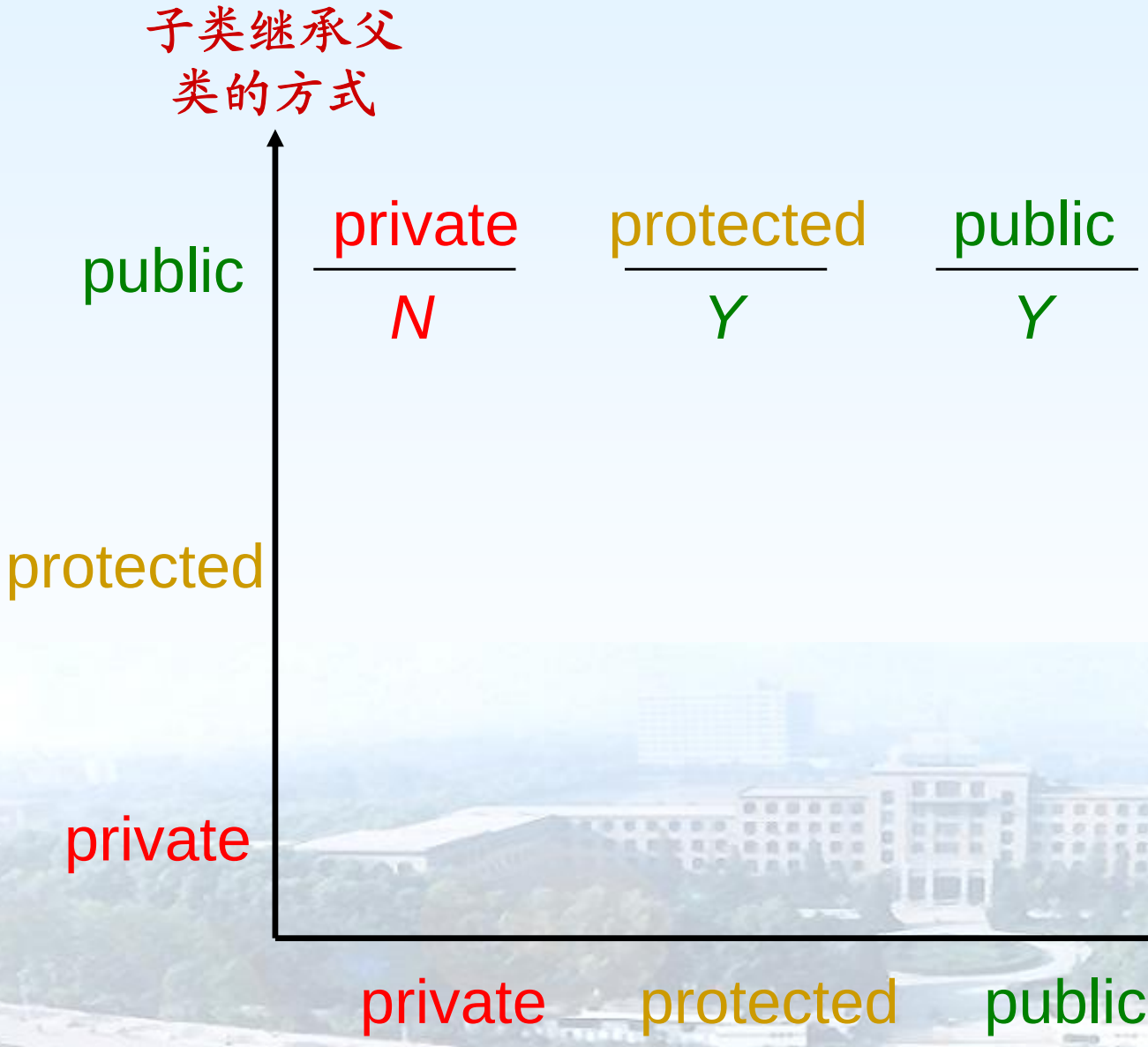
3.2.4 继承性与类层次结构

- 父类的成员在子类中的外部能见度，是指被子类继承的父类成员在子类中的（外部）访问控制程度，也分为public、private、protected 三种。
- 父类的成员在子类中的内部能见度，是指被子类继承的父类成员在子类中定义的方法中的内部访问控制程度，分为 Y（可访问）和 N（不可访问）两种。

3.2 程序设计语言中的OOP机制

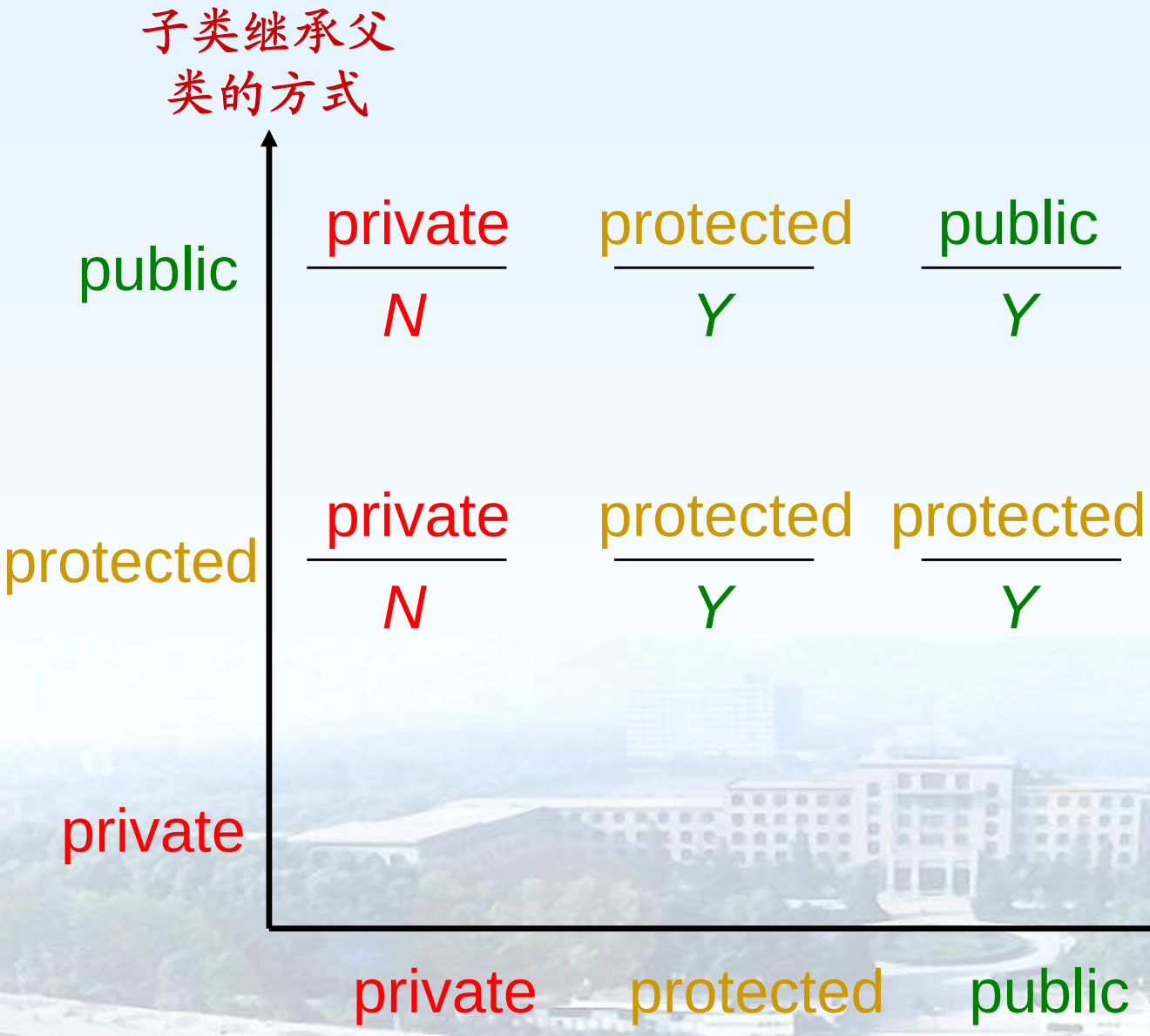


3.2 程序设计语言中的OOP机制



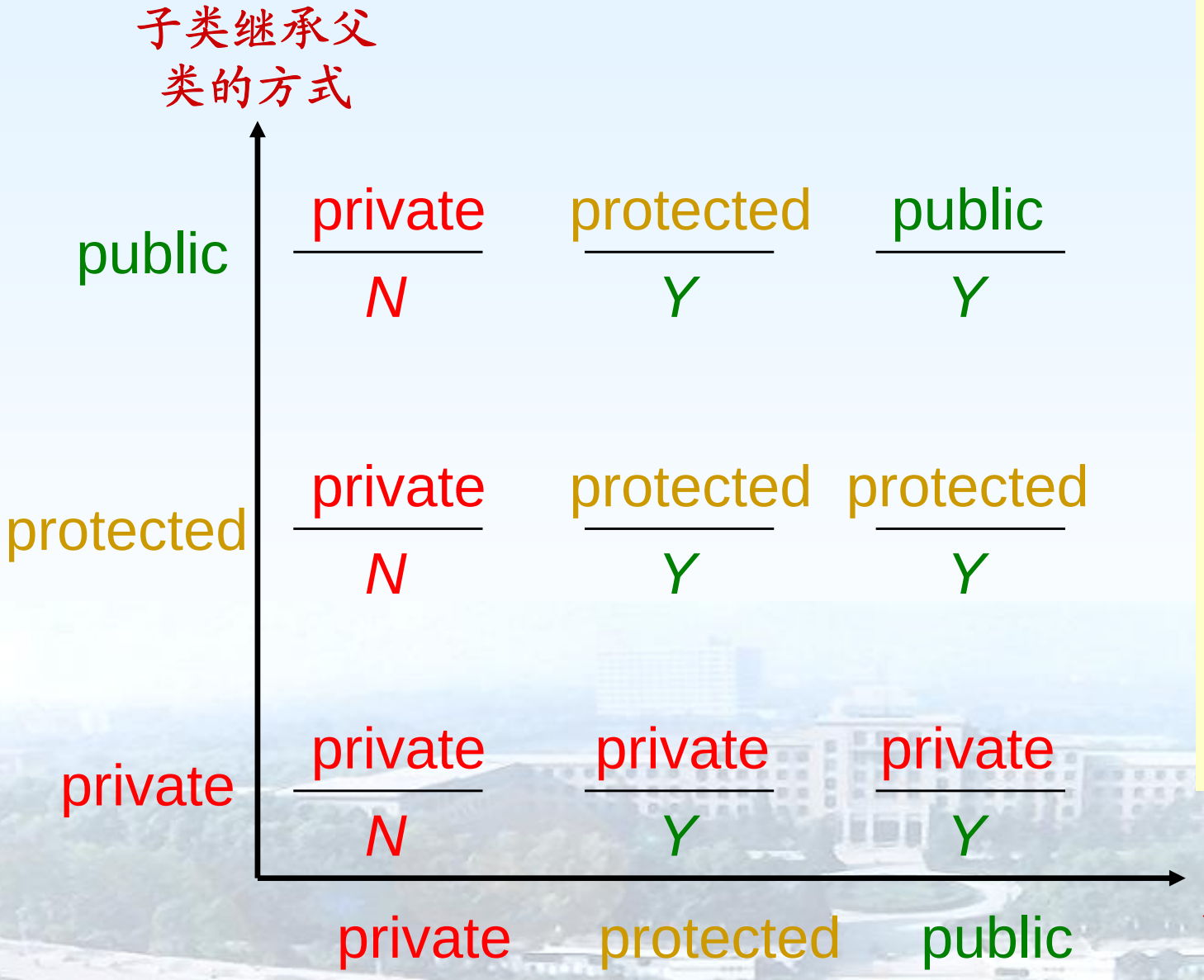

```
class B {
  void pvB();
protected:
  void ptB();
public:
  void pbB();
};
class D : public B {
  void f()
  { pvB();✗ptB();✓
    pbB();✓
  // ...
};
void g( D& d)
{
  d.pvB();✗d.ptB();✗
  d.pbB();✓
}
```

3.2 程序设计语言中的OOP机制




```
class B {
  void pvB();
protected:
  void ptB();
public:
  void pbB();
};
class D : protected B {
  void f()
  { pvB();✗ptB();✓
    pbB();✓
  // ...
};
void g( D& d)
{
  d.pvB();✗d.ptB();✗
  d.pbB();✗
}
```

3.2 程序设计语言中的OOP机制




```
class B {
  void pvB();
protected:
  void ptB();
public:
  void pbB();
};
class D : private B {
  void f()
  { pvB();✗ptB();✓
    pbB();✓
  // ...
};
void g( D& d)
{
  d.pvB();✗d.ptB();✗
  d.pbB();✗
}
```

3.2 程序设计语言中的OOP机制

在实际程序设计时需要更多的机制支持

- 类、对象、消息、方法和继承性只是提供了面向对象程序设计的“基础设施”。
- 在实际程序设计中，遇到的问题会很多，有些看上去不起眼，但解决得圆满并不容易，一般会涉及到语言的支持程度问题，语言的支持越强，程序的可用性水平越容易提高，但也可能由于灵活性的提高而带来一些副作用。

3.2 程序设计语言中的OOP机制

3.2.5 重置（Overriding）

问题：要求所继承的方法符合子类语义：

```
#include “Employee.h”

class Manager: public Employee {
  int   Level;
public:
  void changeLevel( int l );
  Manager( char *n, int a, int l );
  ~Manager();
};
```

```
class Employee {
  char *Name;
  int   Age;
public:
  void changeAge(int newAge);
  void retire();
  Employee( char *n, int a );
  ~Employee();
};
```

```
#include “Employee.h”
void Employee::retire()
{ if (Age > 55)
    delete this;
}
```

不符合子类的语义要求

3.2 程序设计语言中的OOP机制

3.2.5 重置

在子类中增加方法?

```
#include "Employee.h"

class Manager: public Employee {
   int   Level;
public:
   void changeLevel( int l );
   void Mretire();
   Manager( char *n, int a, int l );
   ~Manage
};
```

```
#include "Manager.h"
void Manager::Mretire()
{ if (Age > 60)
    delete this;
}
```

```
class Employee {
   char *Name;
   int   Age;
public:
   void changeAge(int newAge);
   void retire();
   Employee( char *n, int a );
   ~Employee();
};
```

```
#include "Employee.h"
void Employee::retire()
{ if (Age > 55)
    delete this;
}
```

3.2.5 重置

- 分析:
  - 要在子类使用一个新的方法名，仅仅是为了与继承而来的方法相区别。那么，如果在父类的其他子类以及该子类自己的子类中也出现了类似的情况，系统中就会存在许多个不同的方法名，使得类的定义令人（特别是用户）难以理解，从而损害了 OOP 的自然性。
  - 象这样的方法，不管是否真正需要子类都得继承。这就使得越是底层的子类，承受的这类负担越重，从而难以控制那些语法上正确、语义上不正确的消息传递。

3.2 程序设计语言中的OOP机制

3.2.5 重置

用数据成员加以控制?

```
#include "Employee.h"

class Manager: public Employee {
  int   Level;
public:
  void changeLevel( int l );
  Manager( char *n, int a, int l );
  ~Manager();
};
```

```
class Employee {
  char *Name;
  int   Age;
protected:
  int   RetireAge;
public:
  void changeAge(int newAge);
  void retire();
  Employee( char *n, int a );
  ~Employee();
};
```

```
#include "Employee.h"
void Employee::retire()
{ if (Age > RetireAge)
    delete this;
}
```

3.2 程序设计语言中的OOP机制

3.2.5 重置

- 分析:
  - 这意味着要修改父类的已有构造，而且仅仅是为了满足程序设计的需要而不是出于应用的要求。这实际是传统程序设计观念的一种反映。如果该父类的子类们都提出这种性质的修改要求，这个类在一段时间后将变得面目全非。
- 应当采用的原则:
  - 如果对已有的或别人写的程序不满意，应该先继承下来，再在此基础上写适合自己要求的程序，而不是先考虑如何修改要继承的程序。

3.2 程序设计语言中的OOP机制

3.2.5 重置

- 这个例子提出的问题:
  - 如果子类对父类的全盘继承不能适应应用的特点，能不能有选择地继承，而且不用改变父类的基本构造?
- 重置的基本思想是:
  - 通过一种动态绑定机制的支持，使得子类在继承父类接口定义的前提下，用适合于自己要求的实现去置换父类中的相应实现（子类保持父类关于特定方法的语法，同时在自己及自己的子类中改变这样的方法的语义）。

3.2 程序设计语言中的OOP机制

3.2.5 重置

- C++中的重置机制：虚拟函数（*virtual functions*）

3.2 程序设计语言中的OOP机制

3.2.5 重置

- C++中的重置机制：虚拟函数（

```
#include "Employee.h"

class Manager: public Employee {
   int   Level;
public:
   void changeLevel( int l );
   void retire();
   Manager( char *n, int a, int l );
   ~Manager();
};
```

重置方法

```
#include "Manager.h"
void Manager::retire()
{ if (Age > 60)
    delete this;
}
```

```
class Employee {
protected:
   char *Name;
   int   Age;
public:
   void changeAge(int newAge);
   virtual void retire();
   Employee( char *n, int a );
   ~Employee();
};
```

允许重置

```
#include "Employee.h"
void Employee::retire()
{ if (Age > 55)
    delete this;
}
```

3.2 程序设计语言中的OOP机制

3.2.5 重置

- 重置机制的特点:
  - 子类不改变父类中的已有接口定义（尽管它可能有不同语义的方法体），这就保持了OOP的自然性。
  - 即使在用父类定义的方法对子类的实例进行操作时，方法体中如果引用了被子类重置了的方法，也将与子类中重新定义的方法体绑定。
  - 重置机制并不是强加于程序员的，他可以根据应用的要求，灵活地决定在哪些子类中需要对哪些方法进行重置，哪些则需要继承父类定义的方法语义，包括继承那些被父类重置后的方法语义。

3.2 程序设计语言中的OOP机制

3.2.5 重置

- 重置机制的特点:
  - 子类不改变父类中的已有接口定义（尽管它可能有不同语义的方法体），这就保持了OOP的自然性。
  - 即使在用父类定义的方法对子类的实例进行操作时，方法体中如果引用了被子类重置了的方法，也将与子类中重新定义的方法体绑定。
  - 重置机制并不是强加于

```
#include "Employee.h"
void Employee::someMethod()
{ // ...
  retire();
}
```

```
#include "Employee.h"
#include "Manager.h"
void userFunc()
{ Employee e1("张三",24);
  Manager  m1("王五", 35, 2);
  // ...
  e1.someMethod(); // 调用的是 e1.retire()
  m1.someMethod(); // 调用的是 m1.retire()
  // ...
}
```

3.2.5 重置

- 重置机制的特点：
  - 子类不改变父类中的已有
    义的方法体），这就保持
  - 即使在用父类定义的方法
    体中如果引用了被子类重
    定义的方法体绑定。

```
#include “Manager.h”
void Manager::anotherMethod()
{ // …
  if ( … )
    Employee::retire(); // 执行父类方法
  else
    retire(); // 执行子类方法
}
```

  - 重置机制并不是强加于程序员的，他可以根据应用的要求，灵活地决定在哪些子类中需要对哪些方法进行重置，哪些则需要继承父类定义的方法语义，包括继承那些被父类重置后的方法语义。

3.2 程序设计语言中的OOP机制

3.2.5 重置

- 重置机制是怎么实现的？
  - 在C++语言中，用虚拟函数跳转表（*virtual functions jump tables*，*vtbl*）及其指针（*vptr*）的协同工作，来实现重置机制的核心：动态绑定机制。
  - C++程序中含有至少一个虚拟函数的每一个类，编译程序都为之生成对应的一个*vtbl*。*vtbl* 是由若干个虚拟函数体入口地址组成的一个线性表。
  - 子类的 *vtbl* 的前半部分由父类的 *vtbl* 得出（但内容不一定相同），后半部分则对应着在子类中新定义的虚拟函数。

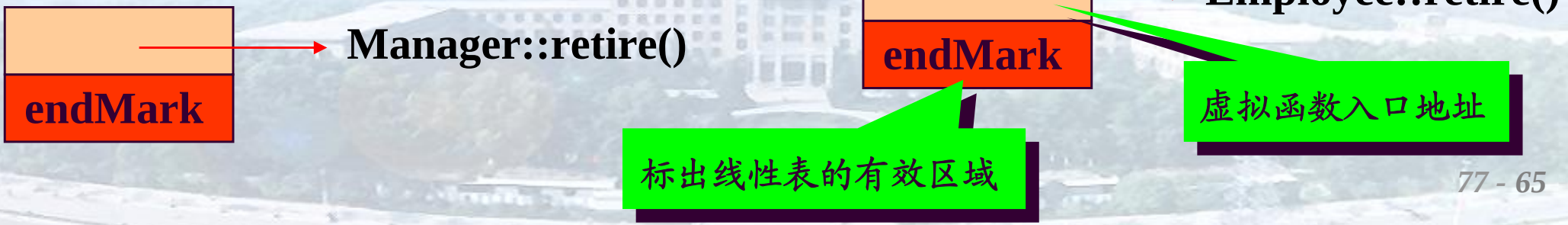
3.2 程序设计语言中的OOP机制

3.2.5 重置

```
#include "Employee.h"

class Manager: public Employee {
   int    Level;
public:
   void changeLevel( int l );
   void retire();
   Manager( char *n, int a, int l );
   ~Manager();
};
```

```
class Employee {
protected:
   char *Name;
   int    Age;
public:
   void changeAge(int newAge);
   virtual void retire();
   Employee( char *n, int a );
   ~Employee();
};
```



3.2 程序设计语言中的OOP机制

3.2.5 重置

```
#include "Employee.h"

class Manager: public Employee {
   int   Level;
public:
   void changeLevel( int l );
   // void retire();
   Manager( char *n, int a, int l );
   ~Manager();
};
```

```
class Employee {
protected:
   char *Name;
   int   Age;
public:
   void changeAge(int newAge);
   virtual void retire();
   Employee( char *n, int a );
   ~Employee();
};
```

Employee::retire()

Employee::retire()

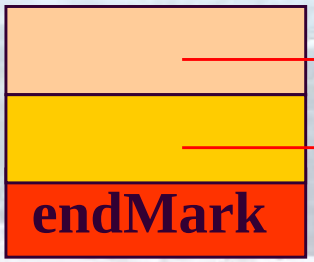
3.2 程序设计语言中的OOP机制

3.2.5 重置

```
#include "Employee.h"

class Manager: public Employee {
   int   Level;
public:
   virtual void changeLevel( int l );
   void retire();
   Manager( char *n, int a, int l );
   ~Manager();
};
```

```
class Employee {
protected:
   char *Name;
   int   Age;
public:
   void changeAge(int newAge);
   virtual void retire();
   Employee( char *n, int a );
   ~Employee();
};
```



Manager::retire()

Manager::changeLevel(int)

Employee::retire()

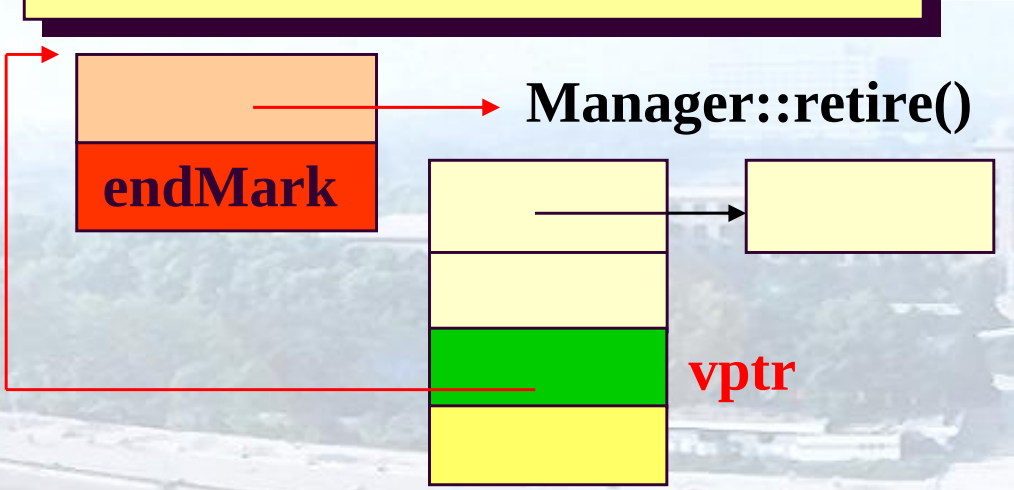
3.2 程序设计语言中的OOP机制

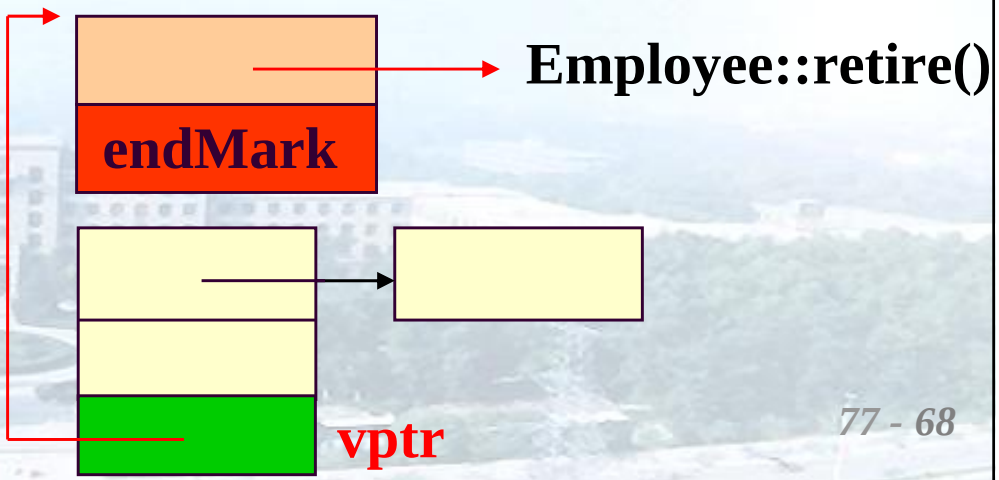
3.2.5 重置

```
#include "Employee.h"

class Manager: public Employee {
   int   Level;
public:
   void changeLevel( int l );
   void retire();
   Manager( char *n, int a, int l );
   ~Manager();
};
```

```
class Employee {
protected:
   char *Name;
   int   Age;
public:
   void changeAge(int newAge);
   virtual void retire();
   Employee( char *n, int a );
   ~Employee();
};
```





3.2 程序设计语言中的OOP机制

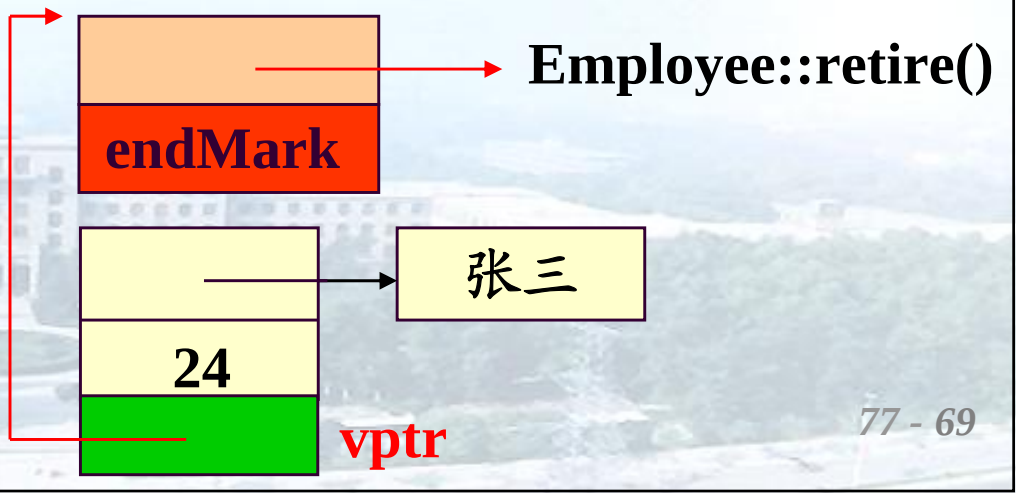
3.2.5 重置

```
#include "Employee.h"

class Manager: public Employee {
   int   Level;
public:
   void changeLevel( int l );
   void retire();
   Manager( char *n, int a, int l );
   ~Manager();
};
```

```
class Employee {
protected:
   char *Name;
   int   Age;
public:
   void changeAge(int newAge);
   virtual void retire();
   Employee( char *n, int a );
   ~Employee();
};
```

```
#include "Employee.h"
void userFunc()
{ Employee e1("张三",24);
   // ...
   e1.retire(); // (e1.vptr[0])();
   // ...
}
```



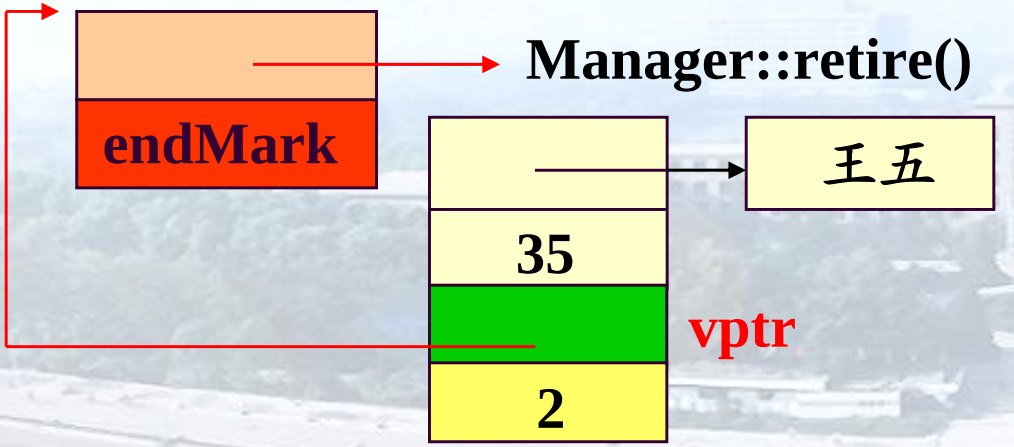
3.2 程序设计语言中的OOP机制

3.2.5 重置

```
#include "Employee.h"

class Manager: public Employee {
   int   Level;
public:
   void changeLevel( int l );
   void retire();
   Manager( char *n, int a, int l );
   ~Manager();
};
```

```
class Employee {
protected:
   char *Name;
   int   Age;
public:
   void changeAge(int newAge);
   virtual void retire();
   Employee( char *n, int a );
   ~Employee();
};
```




```
#include "Manager.h"
void userFunc()
{ Manager  m1("王五", 35, 2);
  // ...
  m1.retire(); // (m1.vptr[0])();
  // ...
}
```

3.2 程序设计语言中的OOP机制

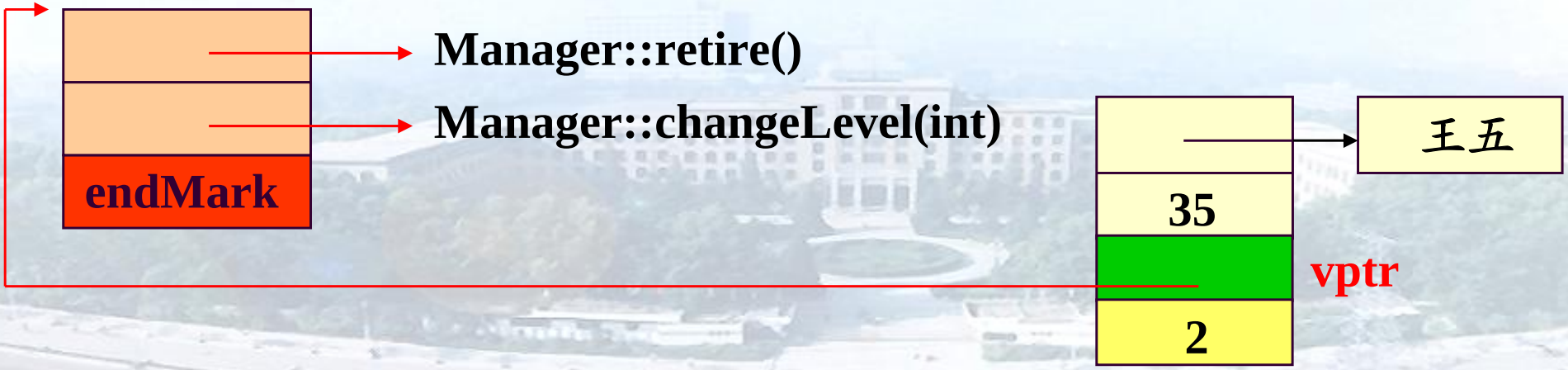
3.2.5 重置

```
#include "Employee.h"

class Manager: public Employee {
   int   Level;
public:
   virtual void changeLevel( int l );
   void retire();
   Manager( char *n, int a, int l );
   ~Manager();
};
```

```
class Employee {
protected:
   char *Name;
```

```
#include "Manager.h"
void userFunc()
{ Manager  m1("王五", 35, 2);
   // ...
   m1.changeLevel(1); // (m1.vptr[1])(1);
   m1.retire(); // (m1.vptr[0])();
   // ...
}
```



3.2 程序设计语言中的OOP机制

3.2.5 重置

- 从前面的例子看动态绑定:

```
#include “Employee.h”
void Employee::someMethod()
{ // ...
  retire(); // (this->vptr[0])();
}
```

```
#include “Employee.h”
#include “Manager.h”
void userFunc()
{ Employee e1(“张三”,24);
  Manager  m1(“王五”, 35, 2);
  // ...
  e1.someMethod(); // 调用的是 e1.retire()
  m1.someMethod(); // 调用的是 m1.retire()
  // ...
}
```

3.2 程序设计语言中的OOP机制

3.2.5 重置

- 用不同虚拟函数跳转表的表项实现动态绑定的前提:
  - 这些虚拟函数跳转表在同一位置上的表项所指向的方法，必须有相同的接口。
  - 为什么？自己思考。
- 虚拟函数跳转表的各表项中所存储的是对应方法的入口地址，而不是其方法体代码。
- 多重继承的虚拟函数跳转表与这里给出的不同，不妨自己试着用Debugger看一下。

3.2 程序设计语言中的OOP机制

3.2.5 重置

- 我们在前面曾经给出了子类型的概念:
  - A value of a subtype can be used everywhere a value of the supertype can be expected.
- 如果试图对应于子类与其父类的关系，就意味着要求:
  - An object of a subclass can be used everywhere an object of the superclass can be expected.
- 这里要求子类的对象在其父类的对象所出现的任何地方，都具有相同的行为。但是，如果在子类中重置了父类中的方法，就失去了这样的保证。
  - 因此，子类并不能保证具有子类型所要求的特性。

要点与引伸

- 要用并发、分布处理的观点来理解对象的特性，不要局限于串行、单进程处理的思路。
- 方法与过程的差别在于：既要动态地与具体对象相关，又要静态地与特定对象无关。核心机制是对象自身引用。
- 注意：**Employee** 的实例集合实际上并不包含 **Manager** 的实例集合，这两个实例集合的并集，才是所有的雇员对象。
- 分析一下：使得类 **Manager** 继承 **Employee**，与在这个类中直接定义一个类型为 **Employee**（或者 **Employee***）的数据成员，有那些相同点和不同点?

要点与引伸（续）

- 类仍然是关键：没有类，重置就意味着过载；没有类，过载必然失控，这是就不可能允许程序员来设计过载。
- 注意语言中重置机制的实现策略：把需要变化的部分（如函数指针）用数据（**vtbl** 和 **vptr**）来表示。

下一次课的内容

- 面向对象程序设计－程序设计语言中的OOP机制
  - 多态（方法名的过载）
  - 多态（操作符的过载）
  - 类属类
  - 类属函数
  - 数据成员值的共享
  - 无实例的类