第 2 章
C++语言的特性和原理

从上一章中，我们知道 C++语言在保留 C 语言的高效性和底层控制能力的同时，引入了面向对象编程和其他高级特性，提供了更强大和灵活的编程工具。C 语言语法在前面章节已有介绍，这里不再赘述。

本章要点

- ☑ 从面向对象概念出发，探讨 C++语言的一些底层原理。
- ☑ 类比 CPU 异常处理，C++语言异常处理和 Java 异常处理，展示学习底层知识的魅力。
- ☑ 介绍 C++语言的特性。
- ☑ 结合汇编学习成本，C 语言和 C++语言存在的最常见问题，推导其他高级语言的底层逻辑。

2.1 对象和类原理

根据上一章的内容进行推理，对象交互如图 2.1 所示。

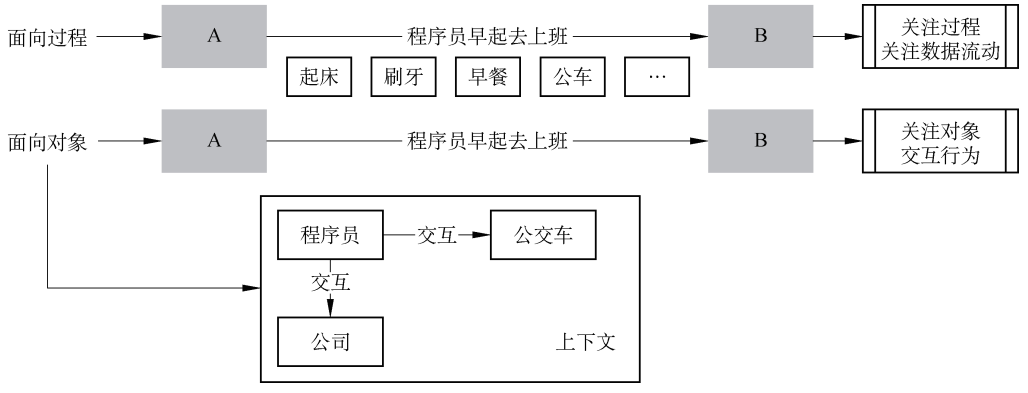


图 2.1 对象交互

面向过程关注从 A 到 B 都做了什么事情，关注数据流动。

面向对象关注从 A 到 B 有哪些对象，以及这些对象在上下文环境中都发生了什么交互。在这个过程中，面向对象引入了两个概念。

类（class）定义了一组属性（数据成员）和方法（成员函数或交互行为），描述了对象的特征和行为。

对象（object）是类的实例化，它是内存中的一个具体实体，具有类定义的属性和方法。通过创建对象，可以使用类定义的功能。

由于各个对象之间存在关系，同时需要隐藏对象的内部实现细节，只暴露必要的接口供外部访问。因此又引入了以下概念。

（1）封装（encapsulation）是将数据和相关操作封装在一起，形成一个类的特性。它隐藏了对象的内部实现细节，只暴露必要的接口供外部访问。这样可以实现数据的安全性和模块化，减少对外部的依赖。

（2）继承（inheritance）支持创建一个新类（称为子类或派生类），它继承了另一个已存在的类（称为父类或基类）的属性和方法。子类可以重用父类的代码，并可以添加新功能或修改继承的行为。

（3）多态（polymorphism）支持以不同的方式对同一个类进行操作。通过多态，可以根据对象的实际类型，在运行时选择相应的方法实现，提高了代码的灵活性和可扩展性。

面向对象基本特征如图 2.2 所示，这些概念的出现根本上是在编程语言中融入了人类的思维模式，或者更贴近人类的思维模式。

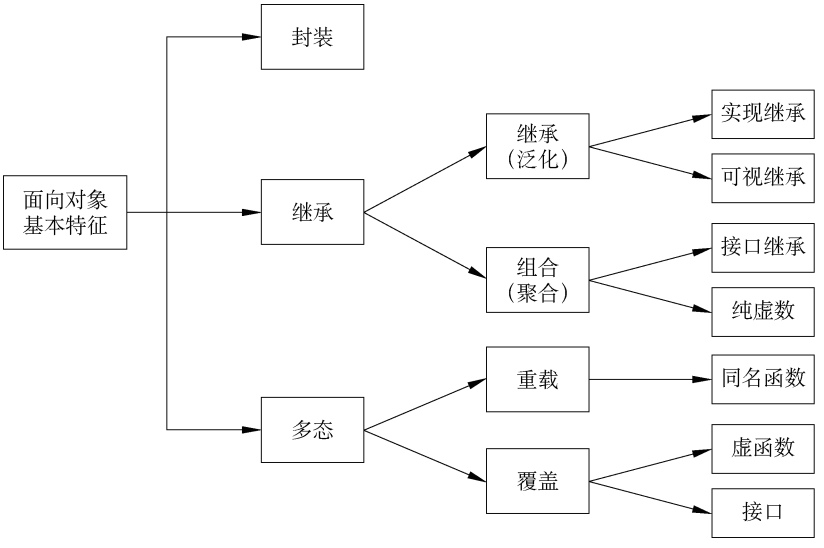


图 2.2　面向对象基本特征

2.1.1　C++语言的 class 关键字

C 语言没有关键字和编译器特性来支持识别对象和对象交互这两个功能。因此，C++语言规范引入了 class，new，extends 等关键字来支撑面向对象特性。但是这些特性必然有 C 语言的实现方法，其实现方法的推导如图 2.3 所示。

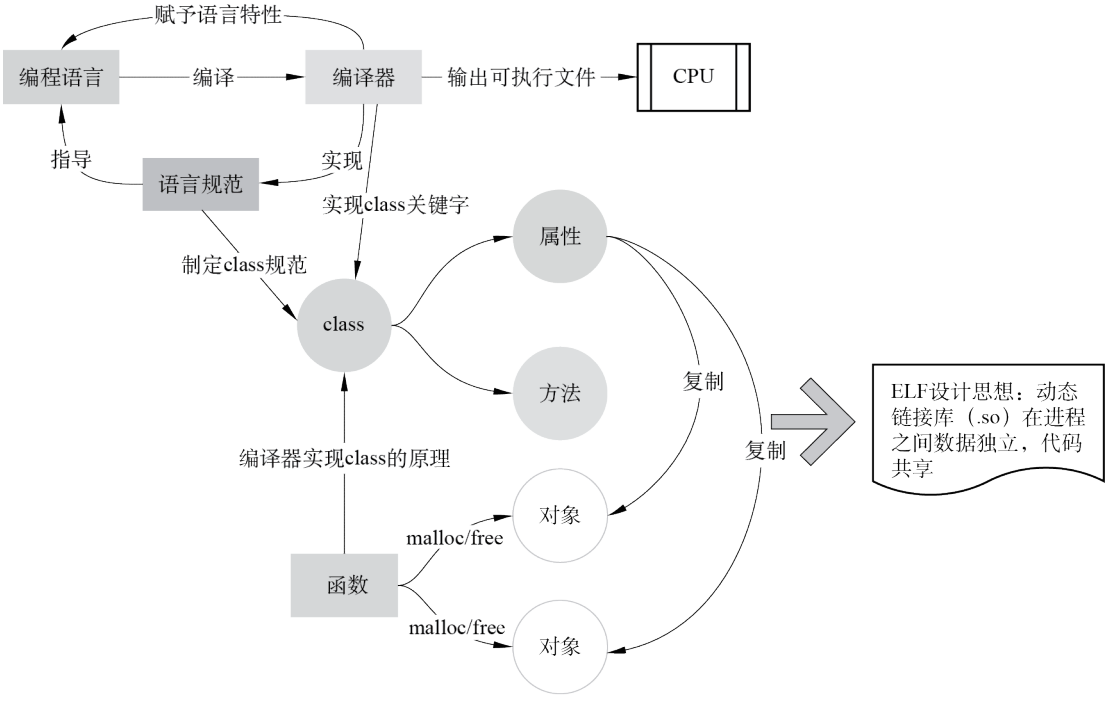


图 2.3　类 class 原理

如图 2.3 所示，通过函数读取 class 的静态数据（元数据）信息。然后使用 malloc 分配堆内存，将属性信息复制到对象中。此时就将对象的属性隔离了，而方法行为则共享。这就是 class 关键字的原理。

概括一下，class 封装了对象的静态属性和行为数据，然后通过分配堆内存和复制属性信息，将各个对象的属性数据隔离，共享行为数据。这就是方法共用、属性（数据）私有（独立）。

2.1.2　C++语言的 new/delete 运算符

前面提到，通过 malloc/free 分配堆内存和释放，对于有 C++语言编程经验的开发者，

可能不接受此做法。在 C++语言中，new 和 delete 是用于动态分配和释放内存的运算符。它们的工作原理如下。

1. new 运算符工作原理

当使用 new 运算符创建一个对象时，编译器首先检查需要分配的内存空间大小，然后调用 operator new 函数分配足够大小的内存。operator new 函数在堆上分配一块足够大小的内存，并返回指向该内存的指针。编译器接着调用对象的构造函数来初始化这块内存，将其转换为一个有效的对象。然后 new 运算符返回指向新创建对象的指针。

2. delete 运算符工作原理

当使用 delete 运算符释放对象时，编译器调用对象的析构函数，以便正确地销毁对象并释放它占用的资源。然后，编译器调用 operator delete 函数，将对象占用的内存空间释放回堆。最后，operator delete 函数将内存标记为可用，并在需要的情况下将其返给操作系统或内存管理器。

其中 operator new 与 operator delete 函数是系统提供的全局函数。

```
   /*
   operator new:该函数实际通过 malloc 申请空间，当 malloc 申请空间成功时直接返回;申请空间失败，尝试
执行空间不足应对措施，如果更改应对措施用户设置，则继续申请，否则抛出异常
   */
   void*   CRTDECL operator new(size t size)  THROW1( STD bad alloc)
   {
      // try to allocate size bytes
      void* p;
      while ((p = malloc(size)) == 0)
         if (_callnewh(size) == 0)
         {
            // report no memory
            // 如果申请内存失败，这里会抛出 bad_alloc 类型异常
            static const std::bad alloc nomem;
            _RAISE(nomem);
         }

      return (p);
   }

   /*
   operator delete: 该函数最终是通过 free 释放空间的
   */
   void operator delete(void* pUserData)
   {
       CrtMemBlockHeader* pHead;

      RTCCALLBACK(_RTC_Free_hook, (pUserData, 0));
```

```
    if (pUserData == NULL)
        return;
     mlock( HEAP LOCK);  /* block other threads */
    __TRY

        /* get a pointer to memory block header */
        pHead = pHdr(pUserData);

        /* verify block type */
        _ASSERTE(_BLOCK_TYPE_IS_VALID(pHead->nBlockUse));

         free dbg(pUserData, pHead->nBlockUse);

      FINALLY
        munlock( HEAP LOCK);  /* release other threads */
    __END_TRY_FINALLY

    return;
}

/*
free 的实现
*/
#define free(p) _free_dbg(p, _NORMAL_BLOCK)
```

通过上述两个全局函数的实现，可以知道 operator new 通过 malloc 申请空间。如果 malloc 申请空间成功就直接返回，否则执行用户提供的空间不足应对措施，如果用户提供了该措施就继续申请，否则就抛出异常。operator delete 最终是通过 free 释放空间的。需要注意的是，C++语言还提供了 new[]和 delete[]运算符，用于动态分配和释放数组。其原理与 new 和 delete 类似，但在分配和释放内存时考虑数组元素的个数。

推论如下。

- ☑ C++语言是基于 C 语言添加了面向对象的特性。
- ☑ 在 C 语言中，分配和释放堆内存分别使用 malloc 和 free 函数。
- ☑ C++语言则使用 new 和 delete 运算符进行内存分配和释放，但底层调用的仍是 malloc 和 free。

new 和 delete 运算符实际工作内容如下。

（1）new 使用 malloc 分配内存，然后编译器调用对象的构造函数初始化这块内存，将其转换为一个有效的对象。

（2）delete 使用 free 释放内存，在释放前编译器调用对象的析构函数，以便正确地销毁对象并释放它占用的资源。

2.1.3 C++语言的 this 指针

在底层原理中，C++语言的 this 指针实际上是通过函数的参数传递实现的，编译器将 this 指针作为隐含的首个参数传递给成员函数。这意味着在成员函数内部，可以通过访问第一个参数获取当前对象的地址。

编译器在编译成员函数时，将成员函数的定义进行转换。例如，对于如下的成员函数定义

```
void MyClass::memberFunction(int arg) {
    // ...
}
```

编译器将其转换为类似以下形式的函数定义。

```
void memberFunction(MyClass* this, int arg) {
    // ...
}
```

在函数体内部，通过 this 指针即可访问成员变量和其他成员函数。例如，使用 this->member 或(*this).member 可以访问当前对象的成员变量。this 指针原理如图 2.4 所示。

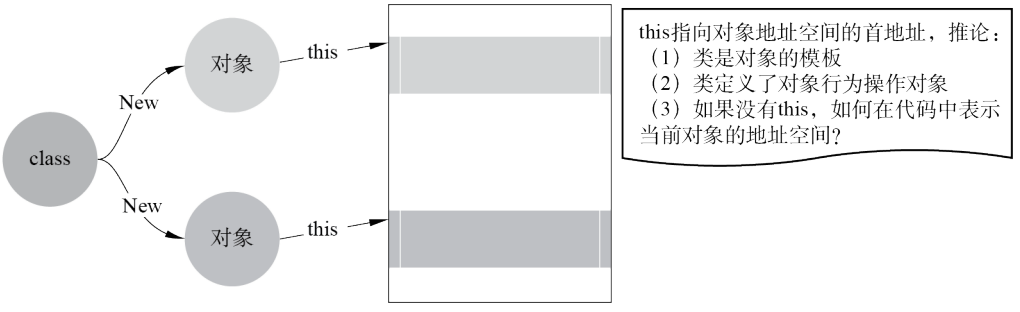


图 2.4 this 指针原理

需要注意的是，底层实现中的 this 指针是一个常量指针（const 指针），它指向当前对象，并且不能修改指向的对象。这是因为成员函数在定义时可以声明为 const，表示该函数不会修改对象的状态，因此 this 指针被视为指向常量对象的指针。

此外，底层实现中的 this 指针通常使用机器的寄存器来传递，以提高访问效率。编译器根据具体的硬件架构和调用约定进行优化。无论具体实现如何，this 指针的功能和作用在高层面上都是一致的。

总之，C++语言的 this 指针通过函数参数传递来实现，在底层原理中通过编译器的转换和机器寄存器访问当前对象，并提供了对成员变量和其他成员函数的访问。

2.2　异 常 处 理

异常是指在程序执行期间出现的意外或错误情况，可能导致程序无法继续正常执行。例如，程序可能试图打开一个不可用的文件，请求过多的内存，或者进行除以 0 的操作。通常，程序员都试图预防这些意外情况。

回顾之前学习的知识。CPU 异常处理原理：CPU 处理函数查找中断描述符表（interrupt descriptor table，IDT），IDT 表指明段描述符所在信息，段描述执行代码信息。CPU 实现故障异常处理，当触发指令触发了一个异常，CPU 查找 IDT 对应的处理异常程序，然后找到段描述符，压入错误代码，就可以表述因为什么方式导致异常，然后查表，通过错误代码分派执行。

2.2.1　C++语言异常处理

C++语言异常处理的基本语法是使用 try-catch 块。以下是异常处理的一般用法。

```
try {
    // 可能引发异常的代码
} catch (ExceptionType1& ex1) {
    // 处理 ExceptionType1 类型的异常
} catch (ExceptionType2& ex2) {
    // 处理 ExceptionType2 类型的异常
} catch (...) {
    // 处理其他类型的异常
}
```

在 try 块中放置可能引发异常的代码。如果在 try 块内发生了异常，程序的控制流将跳转到与异常类型匹配的 catch 块。catch 块用于捕获和处理特定类型的异常。

```
#include <iostream>

int divide(int dividend, int divisor) {
    if (divisor == 0) {
        throw std::runtime error("Divide by zero exception");
    }
    return dividend / divisor;
}

int main() {
    int dividend = 10;
    int divisor = 0;
    try {
```

```
        int result = divide(dividend, divisor);
        std::cout << "Result: " << result << std::endl;
    } catch (std::exception& e) {
        std::cout << "Exception caught: " << e.what() << std::endl;
    }
    return 0;
}
```

在上述代码中，我们定义了一个名为 divide 的函数，用于执行除法运算。如果除数为 0，则通过 throw 关键字抛出一个 std::runtime_error 类型的异常，并传递异常信息 "Divide by zero exception"。

在 main 函数中调用 divide 函数，并使用 try-catch 块捕获异常。在 try 块中调用 divide 函数，并将结果存储在 result 变量中。如果发生了除数为 0 的异常，控制流跳转到 catch 块。在 catch 块中捕获异常并打印异常信息。

通过这种方式，当除数为 0 时可以捕获并处理异常，避免程序崩溃或产生未定义的行为。

需要注意的是，C++语言中的异常处理是一种运行时机制，它涉及异常的抛出、捕获和处理。在异常被抛出后，控制流根据异常处理程序的规则进行跳转，直到找到合适的异常处理块。如果没有找到合适的异常处理块，程序将终止，并显示未处理的异常信息。

2.2.2 Java 异常处理

Java 异常处理涉及以下几个关键概念。

（1）异常类：在 Java 中，异常被表示为特定类型的对象。Java 提供了一系列预定义的异常类，如 ArithmeticException、NullPointerException 等，同时也支持自定义异常类。异常类用于封装异常的相关信息，如异常类型、错误消息等。

（2）try-catch 块：用于标识可能引发异常的代码块，并提供异常处理机制。try 块内放置可能引发异常的代码，而 catch 块用于捕获和处理特定类型的异常。

（3）throw 关键字：用于在代码块中抛出异常。当遇到某个异常情况时，可以使用 throw 关键字手动抛出一个异常对象。

（4）throws 关键字：用于在方法签名中声明该方法可能抛出的异常类型。当方法可能引发某种类型的异常时，使用 throws 关键字在方法声明中指定该异常。

```
public class ExceptionHandlingExample {
    public static void main(String[] args) {
        int dividend = 10;
        int divisor = 0;
        try {
            int result = divide(dividend, divisor);
            System.out.println("Result: " + result);
        } catch (ArithmeticException e) {
```

```
            System.out.println("Exception caught: " + e.getMessage());
        }
    }

    public static int divide(int dividend, int divisor) {
        if (divisor == 0) {
            throw new ArithmeticException("Divide by zero exception");
        }
        return dividend / divisor;
    }
}
```

在上述代码中定义了一个名为 divide 的静态方法，用于执行除法运算。如果除数为 0，则使用 throw 关键字抛出一个 ArithmeticException 类型的异常，并传递异常信息 "Divide by zero exception"。

异常处理流程如下。

（1）当程序执行到 throw 语句时，创建一个异常对象。

（2）异常对象被抛出，并沿着调用堆栈向上传递，直到找到匹配的 catch 块。

（3）当异常被捕获时，相关 catch 块中的代码被执行，以处理异常情况。

（4）如果没有找到匹配的 catch 块，则程序将终止，并显示未处理的异常信息。

使用命令 javap -v ExceptionHandlingExample 查看.class 文件的相关内容，可以看到异常处理的实现细节。

```
public class ExceptionHandlingExample
 ...
public static void main(java.lang.String[]);
   descriptor: ([Ljava/lang/String;)V
   flags: (0x0009) ACC_PUBLIC, ACC_STATIC
   Code:
     stack=3, locals=4, args size=1
        0: bipush       10
        2: istore 1
        3: iconst 0
        4: istore_2
        5: iload 1
        6: iload 2
        7: invokestatic  #2              // Method divide:(II)I
       10: istore 3
  ...
  ...
     Exception table:
        from    to  target type
           5    36    39   Class java/lang/ArithmeticException
  ...
```

在上述代码中，0～10 代表类对象的属性和行为信息。这段行为信息可能导致异常，

当发生异常时，程序会提供一张表（exception table）。表中的 from 和 to 表明它所处的区域。任何在 5～36 之间发生的异常都可到该表中查询匹配。根据异常类型（type）匹配，若匹配到一个表项，则跳转到该表项的 target 行，执行相应的异常处理代码。

Java 处理异常的底层过程和 CPU 异常处理过程非常相似。

exception table 类比中断表 IDT，IDT 的表项指明了段描述符所在的信息（即处理异常的代码段位置）。

exception table 的 target 则类比于段描述符的段选择子，指向全局描述符表（global descriptor talle，GDT）的表项。

这就是底层的魅力，学 1 得 10。如果大家对 CPU 异常处理过程不熟悉或者有所遗忘，可以回顾之前的内容。

2.3 C++语言的特性

接下来，我们简要介绍 C++语言的语法规则和语言特性。

2.3.1 C++语言的 hello world

C++语言的 hello world 程序示例代码如下。

```
#include <iostream>

int main() {
    std::cout << "Hello, World!" << std::endl;
    return 0;
}
```

让我们逐行解析这个程序的语法规则。

#include <iostream>是一个预处理指令，用于将<iostream>头文件包含到程序中。该文件定义了 cin、cout、cerr 和 clog 对象，分别对应于标准输入流、标准输出流、非缓冲标准错误流和缓冲标准错误流。

int main()是程序的入口点。main 函数是程序开始执行的地方。int 是返回类型，表示 main 函数将返回一个整数值给操作系统。()表示函数参数列表，main 函数在这里不接受任何参数。

{ }是代码块的开始和结束大括号，表示 main 函数的范围。

std::cout 是标准输出流对象，用于将数据输出到控制台。std::前缀表示这个对象是 std 命名空间中的成员。

<<是输出流插入运算符，用于将数据插入到输出流中。

"Hello, World!"是要输出的文本字符串。

std::endl 是输出流控制符，用于在输出流中插入换行符，并刷新输出流。

;是语句结束符号，用于表示一行代码的结束。

return 0;是 main 函数的返回语句，将整数 0 返给操作系统，表示程序正常退出。

2.3.2　C++语言的数据类型

1. C++数组

C++数组的声明和初始化方式如下。

```
// 声明一个整数数组
int numbers[5];

// 声明并初始化数组
int scores[] = {90, 85, 95, 80, 88};
```

访问和修改数组元素的代码如下。

```
int value = numbers[2];                    // 访问数组中索引为 2 的元素
numbers[3] = 42;                           // 修改数组中索引为 3 的元素的值
```

循环遍历数组的代码如下。

```
for (int i = 0; i < 5; i++) {
   cout << numbers[i] << " ";
}
```

多维数组的代码如下。

```
int matrix[3][3] = {
   {1, 2, 3},
   {4, 5, 6},
   {7, 8, 9}
};
```

注意，数组的大小必须在声明时确定，并且在后续操作中不能改变。此外，数组索引必须在有效的范围内，否则会导致访问越界错误。

2. C++字符串

在 C++语言中，字符串是用于存储和操作文本数据的数据类型。C++语言提供了两种主要的字符串表示形式：C 风格字符串和 C++标准库字符串。

C 风格字符串示例如下。

```
#include <iostream>

int main() {
```

```
    char greeting[] = "Hello, World!";        // C 风格字符串的声明和初始化

    std::cout << greeting << std::endl;       // 输出字符串

    return 0;
}
```

C++语言标准库字符串示例如下。

```
#include <iostream>
#include <string>

int main() {
    std::string greeting = "Hello, World!";     // C++语言标准库字符串的声明和初始化

    std::cout << greeting << std::endl;       // 输出字符串

    return 0;
}
```

C++语言标准库字符串提供了许多有用的成员函数和操作符，如字符串连接、截取、比较等，使字符串处理更加方便。

```
std::string str1 = "Hello";
std::string str2 = "World";

std::string result = str1 + ", " + str2;     // 字符串连接
std::cout << result << std::endl;

if (str1 == "Hello") {                        // 字符串比较
    std::cout << "str1 is equal to \"Hello\"" << std::endl;
}

std::string substring = str2.substr(0, 3);   // 字符串截取
std::cout << substring << std::endl;
```

C++语言标准库字符串类不仅提供了更多的功能和灵活性，而且在处理字符串时也更加安全，因为它自动管理内存和长度。所以推荐在 C++语言中使用 C++语言标准库字符串来处理和操作字符串。

2.3.3 C++语言的指针和引用

C++语言中的指针是一种变量类型，用于存储内存地址。指针可以指向其他变量或对象，并通过地址访问它们或进行操作。指针提供了对内存的直接访问和操作，是 C++语言中非常重要和强大的特性。

1. 声明和初始化指针

```
int* ptr;                                     // 声明一个整型指针
```

```
double* pDouble = nullptr;                    // 声明并初始化一个双精度浮点型指针为空指针
```

2. 获取指针的地址和访问指针所指向的值

```
int number = 42;
int* ptr = &number;                           // 获取变量 number 的地址并赋给指针

std::cout << "值: " << *ptr << std::endl;     // 输出指针所指向的值
```

3. 动态内存分配和释放

```
int* ptr = new int;                           // 动态分配一个整型变量的内存
*ptr = 10;                                    // 对动态分配的内存进行赋值

delete ptr;                                   // 释放动态分配的内存
```

在 C++语言中，引用（reference）是一种别名，它支持我们使用一个变量名引用另一个已存在的对象。引用提供了对对象的直接访问，而不需要通过指针解引用或使用成员访问运算符。

4. 声明和初始化引用并使用引用进行操作

```
int number = 42;
int& ref = number;                            // 声明一个整型引用，并初始化为变量 number

ref = 10;                                     // 修改引用所指向的值
cout << number;                               // 输出变量 number 的值，结果为 10

number = 20;                                  // 修改变量 number 的值
cout << ref;                                  // 输出引用 ref 的值，结果为 20
```

5. 引用作为函数参数

```
void increment(int& ref) {
   ref++;
}

int number = 42;
increment(number);                            // 传递变量 number 的引用给函数
cout << number;                               // 输出变量 number 的值，结果为 43
```

引用在 C++语言中的主要用途是作为函数参数，特别是用于传递参数并对其进行修改。它提供了一种简洁和直接的方式操作已存在的对象，避免了指针的复杂性。需要注意的是，引用必须在声明时进行初始化，并且不能重新赋值为其他对象。

2.3.4　C++语言的类与对象

在 C++语言中，类是一种用户自定义的数据类型，用于封装数据和操作（方法）的集

合。对象是类的实例，通过实例化类来创建。

1. 声明和定义类

```
class MyClass {
    // 成员变量
    int myInt;
    double myDouble;

    // 成员函数
    void myFunction();
};
```

2. 定义成员函数

```
void MyClass::myFunction() {
    // 函数实现
    // 可以访问和操作成员变量
    myInt = 42;
    myDouble = 3.14;
    // ...
}
```

3. 创建对象并访问成员

```
MyClass obj;                              // 创建一个对象
obj.myInt = 10;                           // 访问和修改成员变量
obj.myDouble = 2.5;
obj.myFunction();                         // 调用成员函数
```

4. 构造函数和析构函数

```
class MyClass {
public:
    // 构造函数
    MyClass() {
        // 构造函数的实现
    }

    // 析构函数
    ~MyClass() {
        // 析构函数的实现
    }
};
```

在前面的学习中，我们了解了 C 语言编译后的可执行文件，C++语言与之类比如图 2.5 所示。

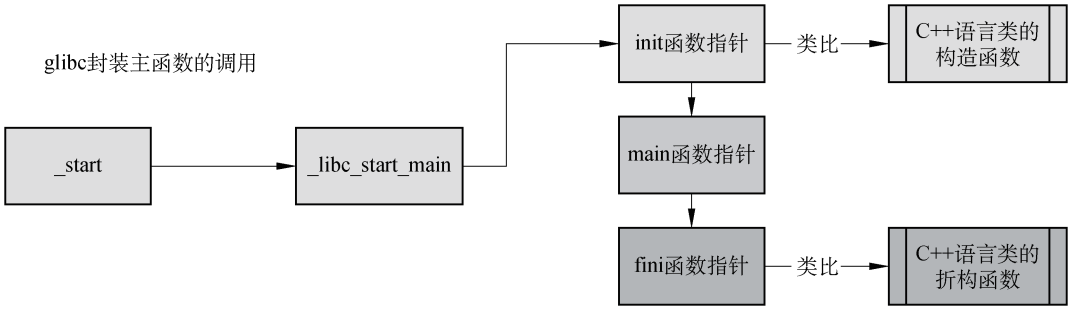


图 2.5　C 语言编译后的可执行文件与 C++语言的类比

面向过程的语言存在调用 main 函数前进行初始化，调用 main 函数后进行清理的行为。因此，通过类比的方法，可以加深我们对面向对象的理解。

（1）构造函数在对象创建时自动调用，用于初始化对象的状态。

（2）析构函数在对象销毁前自动调用，用于清理资源和执行必要的清理操作。

2.3.5　C++语言的多态

在 C++语言中，多态（polymorphism）是面向对象编程的一个重要概念，它支持通过统一的接口处理不同类型的对象，从而提高代码的灵活性和可扩展性。多态通过使用基类指针或引用以引用派生类对象，实现了动态绑定和运行时多态性。

C++语言中的多态性可以通过以下两种方式实现。

1. 虚函数

在基类中声明虚函数（virtual function），并在派生类中进行重写。通过使用 virtual 关键字，使得派生类的同名函数能够在运行时动态绑定正确的函数实现。

基类指针或引用可以引用派生类对象，并在运行时调用正确的派生类函数。

```
class Shape {
public:
    virtual void draw() {
        // 基类虚函数的默认实现
    }
};

class Circle : public Shape {
public:
    void draw() override {
        // 派生类重写的虚函数实现
    }
};
```

```
int main() {
    Shape* shape = new Circle();
    shape->draw();                              // 在运行时调用派生类的函数实现
    delete shape;
    return 0;
}
```

2. 纯虚函数

可以在基类中声明纯虚函数（pure virtual function），通过在函数声明的末尾使用“= 0"告诉编译器该函数没有实现。

包含纯虚函数的类称为抽象类，抽象类不能被实例化，只能作为基类使用。

派生类必须实现基类的纯虚函数，否则派生类也会成为抽象类。

```
class Shape {
public:
    virtual void draw() = 0;                    // 声明纯虚函数
};

class Circle : public Shape {
public:
    void draw() override {
        // 派生类实现纯虚函数
    }
};

int main() {
    Shape* shape = new Circle();
    shape->draw();                              // 在运行时调用派生类的函数实现
    delete shape;
    return 0;
}
```

多态性可以使用通用接口来处理不同的对象，而无须了解对象的具体类型。这增加了代码的灵活性、可维护性和可扩展性。通过基类的指针或引用，程序可以在运行时动态地选择正确的函数实现，实现了多态行为。

需要注意的是，在使用多态时，通常需要将基类的析构函数声明为虚函数，以确保在删除基类指针时正确调用派生类的析构函数，避免内存泄漏。

2.3.6 C++语言的泛型编程

在 C++语言中，泛型编程是一种编程范式，它支持编写通用的、独立于特定数据类型的代码。C++语言通过模板（template）支持泛型编程，提供了一种在编译时生成特定类型

代码的机制。泛型编程的主要优点是可以实现代码的重用和类型安全，同时提高代码的可读性和可维护性。通过使用模板，可以编写通用的代码，以适应不同的数据类型，并在编译时根据实际的类型生成特定的代码。

C++语言的模板是一种用于创建通用代码的机制。模板支持程序员编写通用的函数或类，可以用于处理不同类型的数据，而无须为每种类型编写重复的代码。C++语言的模板主要分为函数模板（function template）和类模板（class template）。函数模板支持定义一个通用的函数，可以处理不同类型的参数。通过使用模板参数（template parameters），可以在函数中使用占位符表示参数的类型。

```
template <typename T>
T max(T a, T b) {
    return (a > b) ? a : b;
}

int main() {
    int a = 10, b = 20;
    cout << max(a, b) << endl;        // 使用函数模板处理整型参数

    double x = 3.14, y = 2.71;
    cout << max(x, y) << endl;        // 使用函数模板处理浮点型参数

    return 0;
}
```

类模板支持定义一个通用的类，可以处理不同类型的成员和操作。类模板使用模板参数定义类中的数据成员、成员函数和类型。

```
template <typename T>
class Stack {
private:
    T* data;
    int top;
    int size;

public:
    Stack(int maxSize) {
        data = new T[maxSize];
        top = -1;
        size = maxSize;
    }

    void push(T item) {
        if (top == size - 1)
            cout << "Stack is full." << endl;
        else
            data[++top] = item;
    }
```

```
    T pop() {
        if (top == -1) {
            cout << "Stack is empty." << endl;
            return T();
        } else {
            return data[top--];
        }
    }
};

int main() {
    Stack<int> intStack(5);                // 使用类模板创建整型栈对象
    intStack.push(10);
    intStack.push(20);
    cout << intStack.pop() << endl;

    Stack<double> doubleStack(3);          // 使用类模板创建浮点型栈对象
    doubleStack.push(3.14);
    doubleStack.push(2.71);
    cout << doubleStack.pop() << endl;

    return 0;
}
```

泛型系统的工作原理如图 2.6 所示。

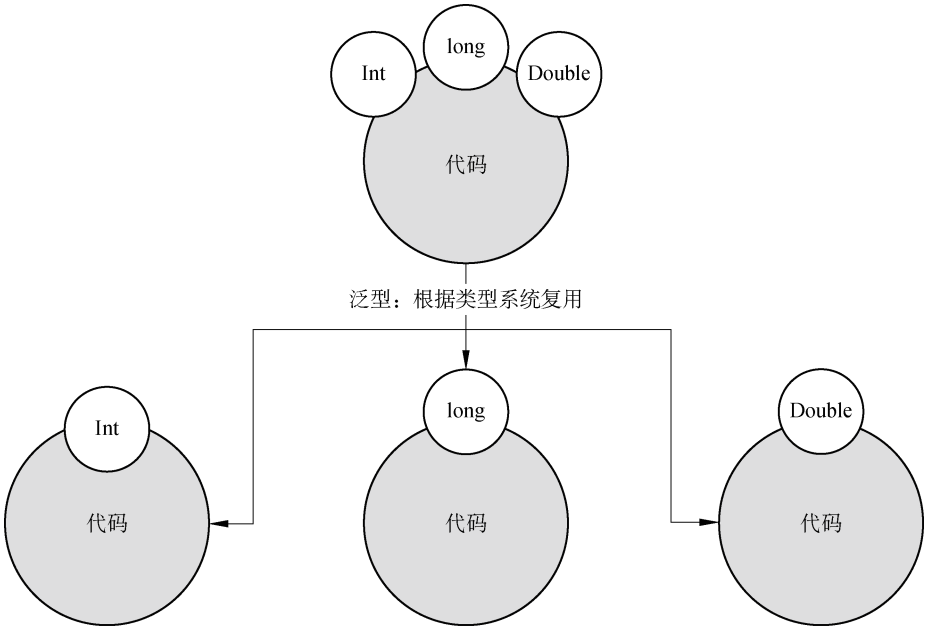


图 2.6　泛型系统工作原理——代码膨胀

图 2.6 是泛型实现方式之一：使用编译器根据调用时传入的类型，将源代码复制一份对应类型的代码，从而造成代码膨胀。这种方式牺牲更多的空间，换取更清晰的调用堆栈。

泛型实现方式之二：编译过程中，将泛型类型的具体信息擦除（即将泛型代码中的类型信息替换为占位符），通过使用虚函数和基类指针，在运行时根据实际类型调用正确的函数。这种方式不会造成代码膨胀，但调用的代码层级结构不够清晰。

泛型的本质一言以蔽之：屏蔽数据及其操作的细节，让算法更为通用，让程序员更多地关注算法的结构，而不是算法中处理的不同数据类型。

2.4　汇编、C 和 C++语言存在的问题

结合之前的学习，从计算机底层开始，我们已经接触了汇编、C 和 C++这三门编程语言，如图 2.7 所示。

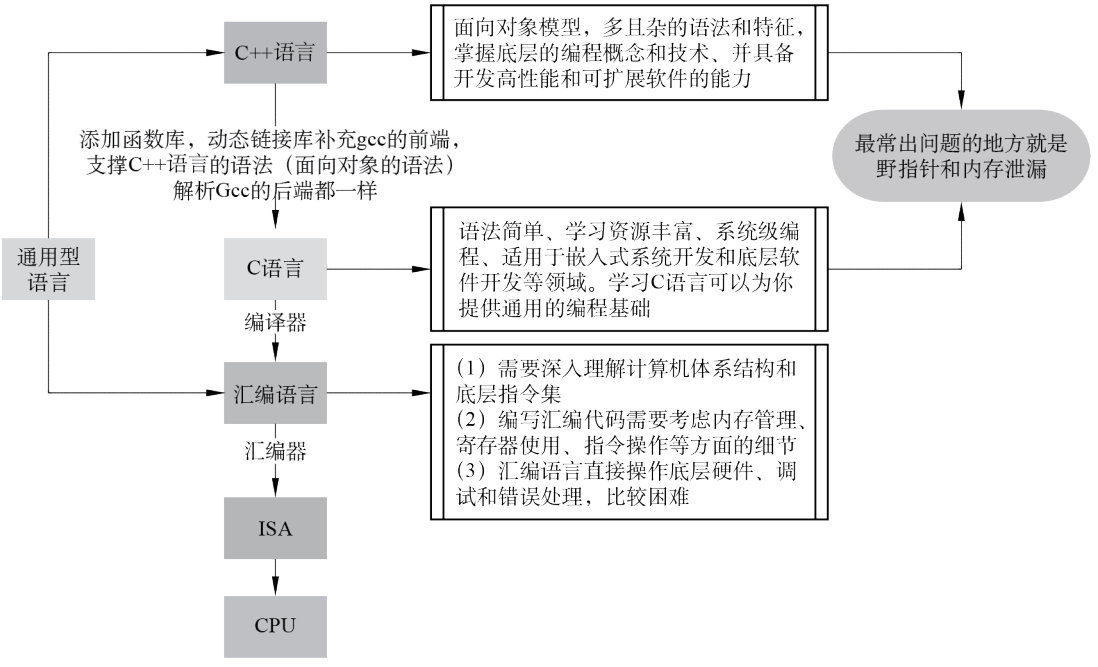


图 2.7　汇编、C 和 C++语言存在的问题

2.4.1　汇编语言

汇编语言是一种低级语言，它直接使用机器指令编写程序。与高级语言相比，汇编语

言更接近底层硬件，一直以来，人们对于汇编语言的认识和评价可以分为两种，一种是认为它非常简单，另一种是认为它学习起来非常困难。

学习汇编语言非常困难的根本原因在教科书上。我们经常看到汇编语言教科书一上来就介绍复杂的寻址方式，然后就被劝退了。

学习汇编语言非常简单的人，通常是因为他们先学习了对应的计算机系统结构和指令集。毕竟汇编语言与计算机硬件结构密切相关。计算机的基本组成和工作原理，包括处理器、内存、寄存器等，掌握计算机体系结构将为你理解汇编语言提供基础。很多人在掌握了若干的计算机指令后，通过汇编语言编写了一些加减乘除的程序，就认为自己掌握了汇编语言。

汇编语言对于学习和理解高级语言如 C 语言，有极大的帮助。例如，C 语言中的指针概念，如果理解了汇编语言，就会明白指针本质上是存放地址的变量。

虽然汇编语言不适合编写大型程序，但它对于理解计算机原理很有帮助，特别是 CPU 的工作原理和运行机制，汇编语言的本职工作就是访问和控制 CPU。编写汇编代码需要考虑内存管理、寄存器使用、指令操作等方面的细节，理解和掌握这些细节需要更多的时间和精力，但这能帮助程序员编写更高效和精确的代码。

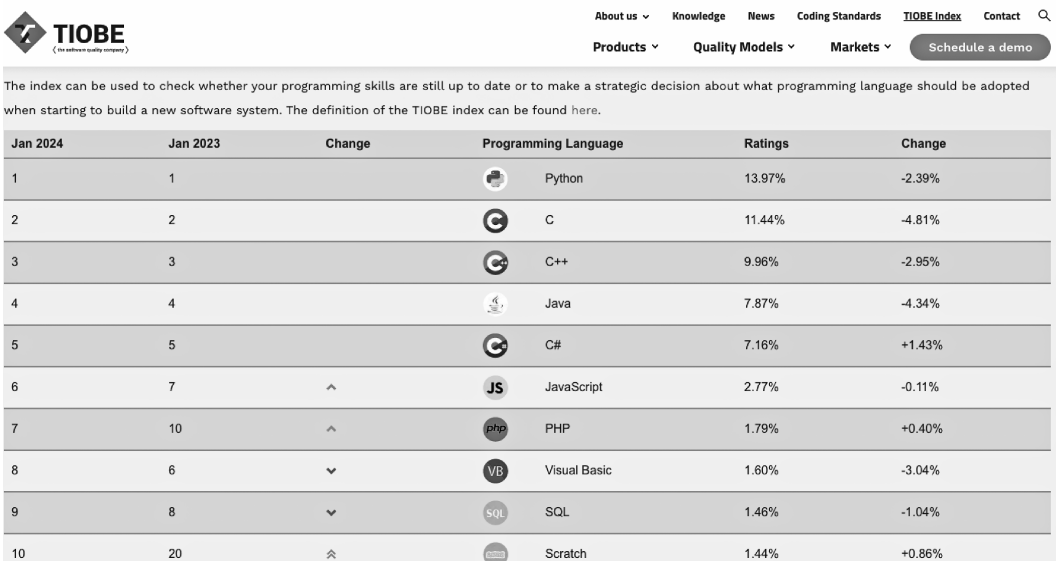
2.4.2 C 语言

C 语言于 1972 年创建，到 2024 年已接近半个世纪。随着计算机技术的不断发展，网络上讨论的大多是流行的语言，如 Java、Python、Go 和 C#等。大家自然会产生疑问，C 语言是否已经过时？答案是否定的。我们认为 C 语言是程序员最需要掌握的一门语言。

TIOBE 编程社区指数（TIOBE programming community index）是一种衡量编程语言流行度的标准，该指数涵盖了网民在 Google、MSN、雅虎、百度、维基百科和 YouTube 的搜索结果。该指数于 2001 年推出，每月更新一次。如图 2.8 所示，2024 年 1 月 C 语言占据了 11.44%的搜索比例，仅次于 Python 排名第 2。这表明 C 语言在编程社区仍然具有较高的地位。

从学习的角度看，C 语言作为一种高级语言，具有相对简单的语法和语义。它采用了直观的语法结构和基本的编程概念，使得初学者可以较快地上手。而且它也具有底层语言的一些特性，如汇编语言的控制能力，这使得学习者在学习过程中能够了解计算机的底层工作原理。C 语言可以提供通用的编程基础，并为更高级的学习和应用打下坚实的基础。

C 语言被广泛应用于系统级编程、嵌入式系统开发和底层软件开发等领域。对于电子、图像处理、音视频处理、通信等方向的工程师来说，掌握 C 语言是必不可少的。因为 C 语言可以避免其他编程语言带来的额外性能开销，能最有效地使用内存，获得 CPU 最大的运行速度，从而最大化地利用计算机的性能。



TIOBE

the software quality company

About us　Knowledge　News　Coding Standards　TIOBE Index　Contact

Products　Quality Models　Markets　Schedule a demo

The index can be used to check whether your programming skills are still up to date or to make a strategic decision about what programming language should be adopted when starting to build a new software system. The definition of the TIOBE index can be found here.

Jan 2024	Jan 2023	Change	Programming Language	Ratings	Change
1	1		Python	13.97%	-2.39%
2	2		C	11.44%	-4.81%
3	3		C++	9.96%	-2.95%
4	4		Java	7.87%	-4.34%
5	5		C#	7.16%	+1.43%
6	7	↑	JavaScript	2.77%	-0.11%
7	10	↑	PHP	1.79%	+0.40%
8	6	↓	Visual Basic	1.60%	-3.04%
9	8	↓	SQL	1.46%	-1.04%
10	20	⇈	Scratch	1.44%	+0.86%

图 2.8　TIOBE 编程社区指数

学习 C 语言无疑是一项值得的投资，它有助于提升你的技能，增强你的竞争力，并为你的职业生涯开辟更多机会。

2.4.3　C++语言

C++语言作为一门历史悠久且功能强大的编程语言，不仅在软件开发领域扮演着重要角色，同时也是令许多程序员爱恨交加的源泉。

C++语言的名字本身就带有程序员式的幽默。它在 C 语言的基础上增加了面向对象等特性，"++"是 C 语言中的自增运算符，意味着 C++语言是 C 语言的改进版，也象征着语言的进步。因此，对于已经熟悉 C 语言的人而言，学习 C++语言将更容易。C++语言的语法和语义中包含了 C 语言的大部分内容，这使得具备 C 语言基础的人在学习 C++语言时成本相对较低。

在编程语言的讨论中，经常会提到用不同的语言 Shooting yourself in the foot。在这个比喻中，C++语言被描绘为一种提供了足够绳索来吊死自己的语言，而且在实际射击自己的脚之前，你还能用 C++语言制造装备。这个比喻反映了 C++语言提供的强大功能和灵活性，同时也指出了其复杂性和潜在的风险。

C++语言还提供了编译器的魔法与挑战。C++语言的模板是一种强大的特性，支持编写通用和高效的代码。然而，模板元编程（TMP）也让 C++编译器的错误信息变得异常复杂，有时一行简单的代码可能产生几页的编译错误，让程序员摸不着头脑。这种情况常常被戏

称为“与编译器的斗争”。

随着标准的不断更新，C++语言已经有了多个版本，如 C++98、C++03、C++11、C++14、C++17、C++20 等。每个新版本都会引入新的特性和改进，但同时也会让人感到语言的复杂度在增加。社区中有个玩笑说，每推出一个新标准，C++语言就会变成一门全新的语言，这既是挑战也是机遇。

C++语言因其复杂性和强大的功能而著称，但这也使得它对新手来说是一个巨大的挑战。社区里经常有关于最佳学习路径的讨论，以及如何在不被 C++语言那众多陷阱和复杂特性吓退的情况下，有效地掌握这门语言。

C++语言以其出色的性能和对底层的直接控制能力而受到系统程序员、游戏开发者和需要高性能计算的应用程序开发者的青睐。社区中常有关于如何进一步优化 C++语言代码以提高性能的讨论，有时甚至钻研汇编语言级别的优化，展示了程序员对性能追求的极致。尽管 C++语言有其复杂性和挑战，但它仍然是一门非常受欢迎和强大的编程语言。它的灵活性和性能优势使得它在众多领域中继续保持其独特的地位。对于程序员而言，无论是爱它还是恨它，学习和掌握 C++语言无疑都是一次宝贵的经历。

2.4.4 最常见的问题

C 和 C++语言最常出现野指针和内存泄漏的问题，它们经常困扰程序员并导致程序出现各种不可预测的行为和严重的安全漏洞。以下是它们困扰程序员的一些主要原因。

（1）指针概念的复杂性：指针是 C 和 C++语言的核心特性之一，但同时也是导致问题的主要原因之一。程序员需要理解指针的概念、使用和生命周期管理，包括正确初始化、解引用、释放和避免悬空指针等。

（2）内存动态分配和释放：C 和 C++语言支持程序员手动进行内存分配和释放，这提供了更大的灵活性，但也需要程序员自行管理内存。如果程序员忘记释放动态分配的内存，就会导致内存泄漏。

（3）对象生命周期管理：在 C++语言中，对象的构造函数和析构函数对于正确管理资源和内存非常重要。如果程序员没有正确地实现析构函数或忘记在析构函数中释放资源，就会导致内存泄漏或资源泄漏。

（4）复杂的程序流程和错误处理：大型的 C 和 C++语言程序通常有复杂的程序流程和错误处理机制。在这种情况下，如果程序员没有正确管理指针和内存，就容易出现野指针和内存泄漏问题，影响程序的可靠性和稳定性。

（5）缺乏工具和自动化支持：相比于其他高级语言，C 和 C++语言在静态分析和内存调试方面的工具和自动化支持相对较少。这增加了程序员发现和解决野指针和内存泄漏问题的难度。

（6）野指针问题：示例代码如下。

```
#include <stdio.h>

int* getPointer() {
    int value = 42;
    int* ptr = &value;
    return ptr;                                    // 返回局部变量的地址
}

int main() {
    int* danglingPtr = getPointer();
    printf("%d\n", *danglingPtr);                  // 未定义的行为，访问了已释放的内存
    return 0;
}
```

getPointer 函数返回一个指向局部变量 value 的指针。当函数执行完毕后，value 变量的生命周期随即结束，指针 danglingPtr 指向的内存变为无效，这就是野指针情况。

再来看看内存泄漏的问题。

```
class MyClass {
private:
    int* data;

public:
    MyClass() {
        data = new int[100];
    }

    ~MyClass() {
        // 忘记释放内存
    }
};

int main() {
    while (true) {
        MyClass* obj = new MyClass;
        // 对象没有被销毁，内存泄漏
    }

    return 0;
}
```

在这个示例中，MyClass 类在构造函数中使用 new 运算符动态分配一个包含 100 个整数的数组，但在析构函数中没有释放该内存。在 main 函数的无限循环中，对象 obj 被创建，但没有被销毁，导致内存泄漏。修改上面的代码避免内存泄漏。

```
class MyClass {
private:
```

```
    int* data;

public:
    MyClass() {
        data = new int[100];
    }

    ~MyClass() {
        delete[] data;                    // 释放内存
    }
};

int main() {
    while (true) {
        MyClass* obj = new MyClass;
        delete obj;                       // 销毁对象并释放内存
    }

    return 0;
}
```

在修改后的代码中，通过在析构函数中调用 delete[]释放动态分配的数组内存。在 main 函数中，通过 delete 关键字销毁对象并释放相关内存，避免了内存泄漏问题的发生。

C 类语言（C/C++语言）的内存泄漏和野指针问题，如同不定时炸弹，在软件的用户规模扩大的时候，就可能会爆发出来，导致软件崩溃，用户无法使用。

如果只是依赖程序员写代码时保证释放动态分配的内存，海量的业务需求就需要海量的程序员。海量的程序员都需要正确使用指针和正确释放动态分配的内存，这个问题实际无法从根本上解决。

2.5 Java 语言出现的推论

如果 C 类语言无法通过程序员编写代码来保证释放所有动态分配的内存，那么在 C 类语言中如何规避这个问题？是否可以设计一种语言，直接规避内存管理的问题？

2.5.1 内存泄漏和野指针规避

C 类语言（C/C++语言）如何解决内存泄漏和野指针问题呢？使用 C++语言中的智能指针（如 std::shared_ptr 和 std::unique_ptr）管理动态分配的内存，并自动释放内存，如图 2.9 所示。

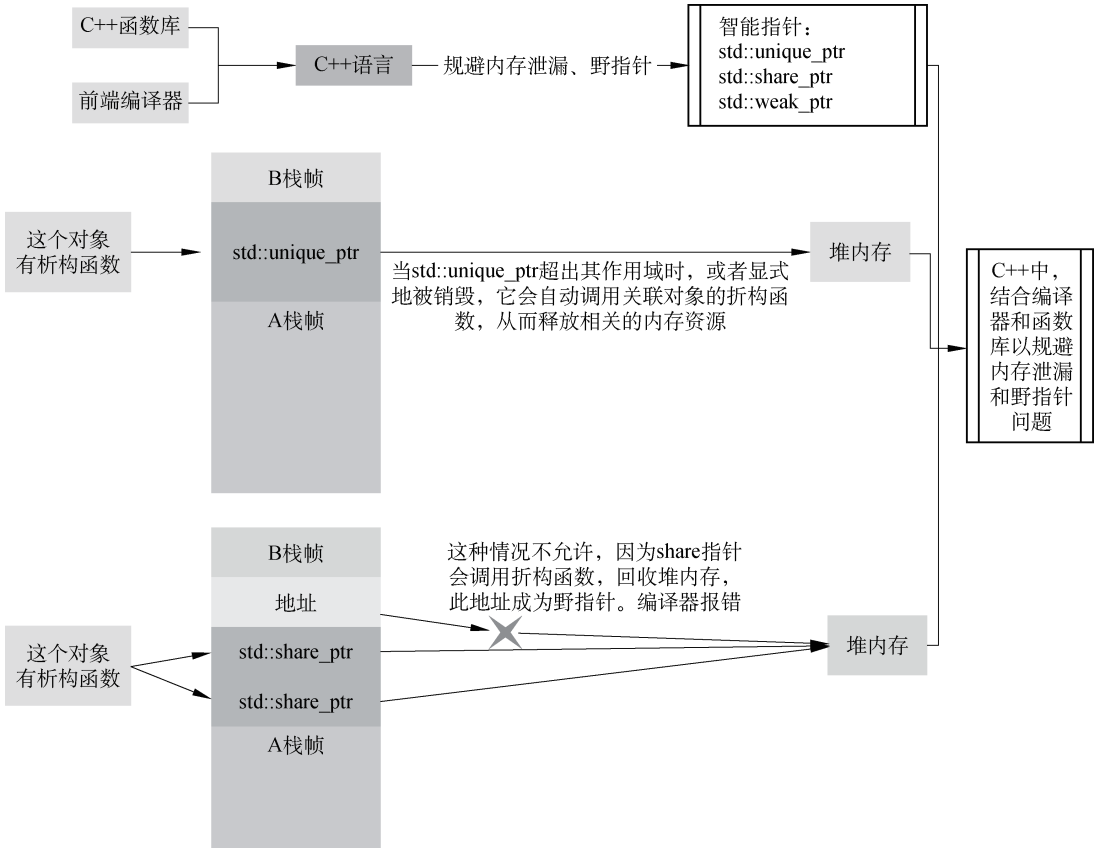


图 2.9　内存泄漏规避

编译器对 std::unique_ptr 的支持主要体现在以下几个方面。

（1）类型安全：std::unique_ptr 是类型安全的，编译器在编译时进行类型检查，确保只有相应类型的指针可以被分配给 std::unique_ptr。

（2）静态析构：当对象的 std::unique_ptr 超出其作用域时，编译器在编译时自动生成析构函数的调用，确保资源自动释放。

（3）移动语义优化：编译器对移动语义进行优化，尽可能避免不必要的拷贝操作，提高程序的性能和效率。

（4）错误检查：编译器可以检查一些潜在的错误，如使用已释放的资源或访问空指针等，提供静态的错误检查和警告。

编译器对 std::shared_ptr 的支持主要体现在以下几个方面。

（1）引用计数管理：编译器在编译时自动生成适当的引用计数代码，确保引用计数的正确维护和管理。

（2）循环引用检测：编译器无法在编译时检测循环引用问题，但它可以在运行时检测循环引用，并在引用计数变为零之前释放相关的内存资源。

（3）自动析构：编译器在适当的时机自动生成析构函数的调用，确保资源自动释放。

（4）多线程安全：std::shared_ptr 的引用计数是原子操作，编译器确保在多线程环境下对引用计数的操作是线程安全的。

std::weak_ptr 是 std::shared_ptr 的弱引用，允许观察对象但不拥有对象。它不增加引用计数，因此不影响对象的生命周期，可以用来避免 std::shared_ptr 的循环引用问题。

2.5.2 新语言的设计要求

根据我们之前的推论，去除枝叶，只保留最核心的内容，如图 2.10 所示。

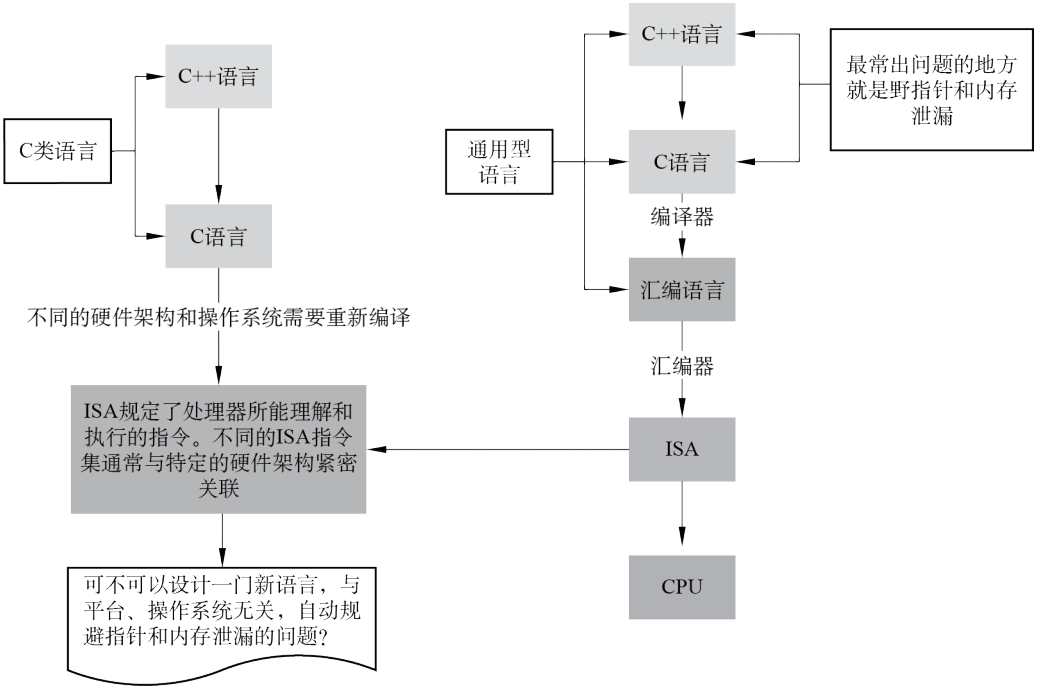


图 2.10　新语言设计关注的问题推论

新语言的设计要求如下。

（1）平台无关性：基于虚拟机（virtual machine，VM）的编程语言在编写代码时不直接针对底层操作系统或硬件进行编程，而是通过虚拟机提供一个统一的执行环境。这使得编写的代码可以在不同的平台和操作系统上运行，具有较高的可移植性。

（2）中间代码：基于 VM 的编程语言通常将源代码编译为中间代码（intermediate code），

也称为字节码（bytecode）或类似的形式。中间代码是一种中间表示形式，可以在虚拟机上解释或编译成机器码执行。这种中间代码的存在使得编程语言能够兼顾可读性和执行效率。

（3）虚拟机执行：基于 VM 的编程语言通过虚拟机执行中间代码。虚拟机作为运行时环境，负责解释或编译中间代码，并提供对底层系统资源的访问。

（4）自动内存管理：使用垃圾回收（garbage collection）机制自动管理内存。程序员不需要手动分配和释放内存，而是由垃圾回收器负责在适当时机回收不再使用的对象，从而减少内存泄漏和悬挂指针等问题。

2.5.3　新语言的两种实现方法

基于编程语言的特性是编译器特供的，基于 VM 的编程语言有 2 种实现方法，如图 2.11 所示。

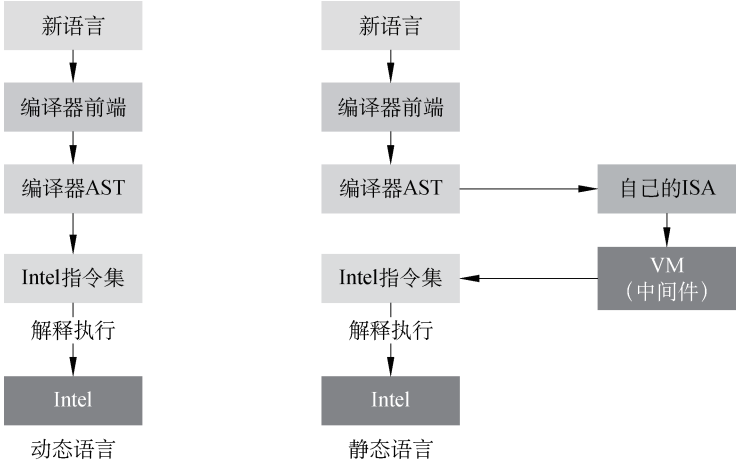


图 2.11　新语言的两种实现方法

1. 动态语言

动态语言是指在代码执行过程中进行类型检查和绑定的编程语言，具有以下特性。

（1）动态类型系统：动态语言支持变量在运行时绑定到不同的类型。这意味着在编写代码时无须显式声明变量的类型，而是根据赋值来确定变量的类型。这种灵活性使得动态语言更加适应变化。

（2）运行时类型检查：动态语言在运行时进行类型检查，而不是在编译时。这意味着变量的类型检查发生在代码执行阶段，可以动态处理类型转换和类型错误。这为开发者提供了更大的灵活性，但也增加了运行时错误的风险。

（3）动态内存管理：动态语言通常具有内置的垃圾回收机制，负责自动管理内存的分

配和释放。开发者无须手动分配和释放内存，减少了内存泄漏和悬挂指针等问题。这也使得动态语言更容易使用和编写。

（4）反射和元编程：动态语言通常提供反射和元编程的机制，使开发者可以在运行时动态检查、修改和生成代码。这些功能使动态语言能够实现更高级的编程技术，如动态加载类、修改对象结构和行为等。

（5）脚本化和解释执行：动态语言通常被用于编写脚本和解释执行的场景。它们可以直接在运行环境中执行，无须编译成机器码。这使得开发者可以更快地进行开发和调试，并且在运行时可以更容易地修改和调整代码。

（6）简洁而灵活的语法：动态语言通常具有简洁而灵活的语法，使开发者能够以更简洁的方式表达想法和实现功能。这样的语法特点使得代码更易读、易写，提高了开发效率。

2. 静态语言

静态语言在对代码进行编译时，会通过 AST 得到目标平台的机器码。静态语言具有以下特性。

（1）静态类型系统：静态语言在编译时进行类型检查，要求变量在声明时显式指定其类型，并在编译过程中进行类型匹配。这意味着变量的类型在编译时就确定了，不允许在运行时进行隐式的类型转换或类型错误。

（2）编译时类型检查：静态语言在编译阶段对代码进行类型检查，以捕获潜在类型错误。编译器会检查变量的使用方式是否与其声明的类型相符合，例如赋值操作、函数调用等，以确保类型的一致性和正确性。

（3）强类型系统：静态语言具有强类型系统，要求变量严格按照其声明的类型进行操作。类型的转换必须显式地进行，并且需要满足特定的规则。这有助于提高代码的安全性和可靠性。

（4）提前编译：静态语言通常需要在代码执行之前进行编译。在编译过程中，源代码被翻译成机器码或字节码，以便在运行时直接执行。这使得静态语言的执行速度较快。

（5）明确的接口和类型约束：静态语言通常要求明确定义接口和类型的约束。通过接口和类型定义，可以在编译时进行静态检查，确保代码在使用接口和类型时符合规定的约束条件。

（6）性能优化：由于在编译时进行类型检查和优化，静态语言通常具有更好的性能。编译器可以进行静态优化，如内联函数、死代码消除、循环展开等，以提高程序的执行效率。

基于 C 类语言最常见的问题，在应用程序开发时会面临巨大的不确定性。面对日益增长的应用程序开发需求，程序员需要一门新的编程语言。

由于指针规避等问题和新语言的设计要求，因此新语言的特性就显而易见了。

于是 Java 语言应运而生，如图 2.12 所示。

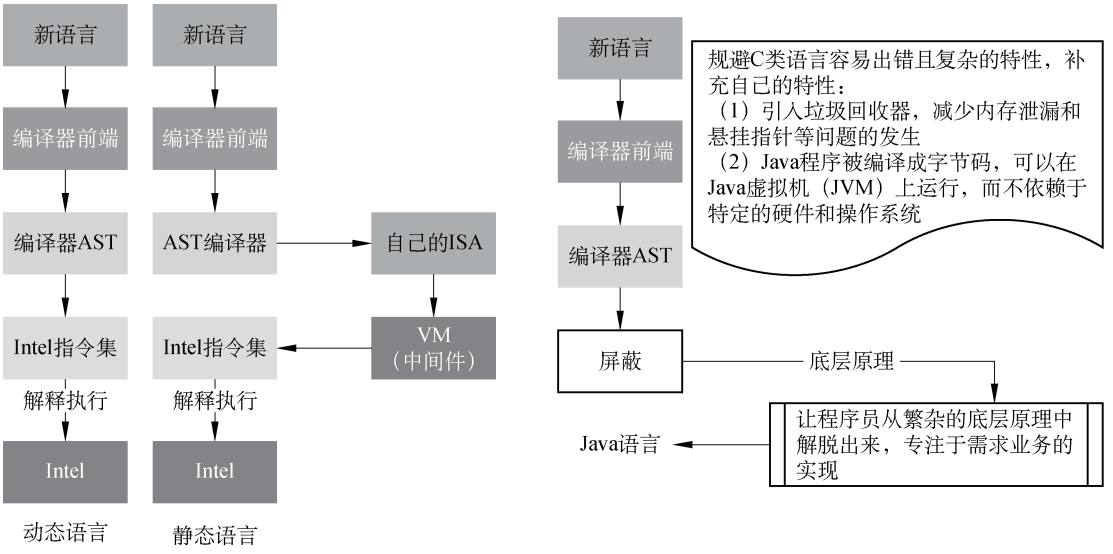


图 2.12　Java 语言的诞生

2.6　如何通过底层来学习不同的编程语言

回顾一下 C 语言在 Linux 下的编译执行过程。

```
#include <stdio.h>

int main() {
    printf("Hello, World!\n");
    return 0;
}
```

C 语言编译执行过程如图 2.13 所示。

类比 C 语言的编译执行过程，我们再来看一下 C++语言的编译执行过程。

```
#include <iostream>

#define MULTIPLY(x, y) (x * y)

int main() {
    int num1 = 5;
    int num2 = 10;
    int product = MULTIPLY(num1, num2);
    std::cout << "The product of " << num1 << " and " << num2 << " is: " << product
<< std::endl;
    return 0;
}
```

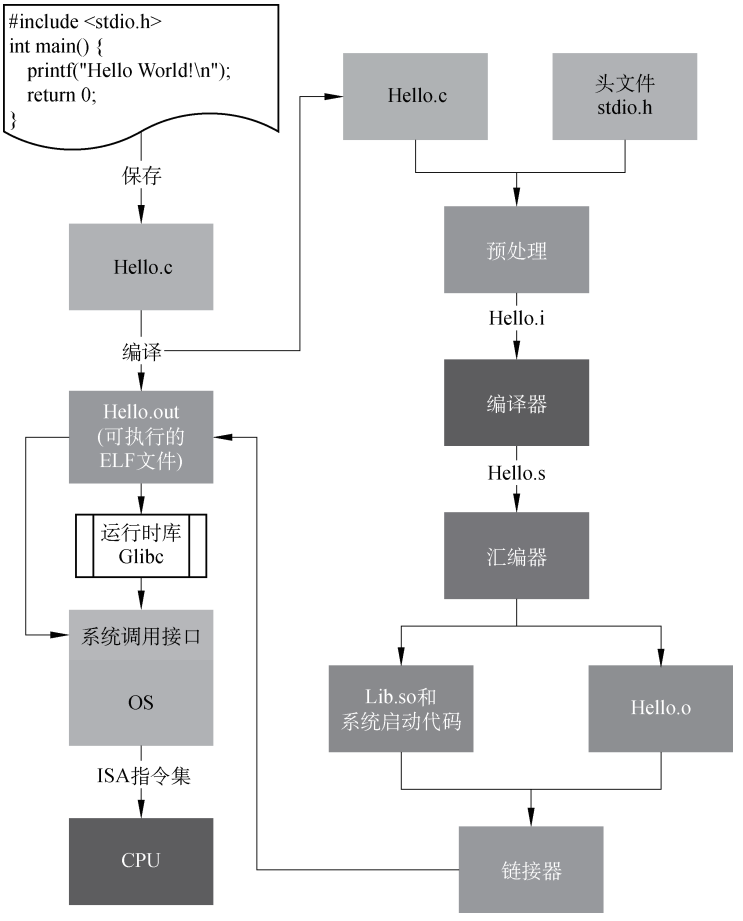


图 2.13　C 语言编译执行过程

将上述代码保存为 main.cpp，编译执行过程如下。

（1）预处理：在预处理阶段，预处理器将对代码进行处理，执行诸如宏展开和头文件包含等操作。在上述示例中，我们使用了一个宏定义#define 来定义一个乘法宏 MULTIPLY。预处理器将代码中的宏展开，生成预处理后的代码。

```
g++ -E main.cpp -o main.i
```

（2）编译器：在编译阶段，编译器将预处理后的代码翻译成汇编代码，即将 C++语言代码转换为汇编语言代码。

```
g++ -S main.i -o main.s
```

（3）汇编器：在汇编阶段，汇编器将汇编语言代码转换为机器码的目标文件。

```
g++ -c main.s -o main.o
```

（4）链接器：在链接阶段，链接器将目标文件与所需的库文件进行链接，生成可执行文件。

```
g++ main.o -o main
```

（5）加载：加载器将可执行文件加载到内存中，并分配所需的资源和空间。

- ☑ 为可执行文件分配足够的内存空间。
- ☑ 将可执行文件的代码段（text segment）和全局数据段（data segment）加载到分配的内存空间中。
- ☑ 处理可执行文件的重定位（relocation）信息，确保代码中的内存地址与实际的内存位置匹配。
- ☑ 加载器可能还会处理动态链接库（DLL）等外部依赖项的加载和链接。

（6）执行：执行器按照机器码的指令顺序执行加载到内存中的可执行文件。在这个例子中，程序将输出乘法的结果到标准输出流（终端或命令提示符）。

```
./main
The product of 5 and 10 is: 50
```

对比 C++和 C 语言在 Linux 中的编译和执行过程。我们会发现无非就是将 gcc 替换成 g++，基本一致。如果对 C 语言的预处理、编译器、汇编器、链接器、加载、执行有所了解，再学习 C++语言将会容易得多。

新高级语言的设计目的是屏蔽底层原理，解放程序员的生产力。让程序员专注于业务需求的实现。但程序员仍需提高对计算机底层原理的认识，提高解决实际问题的能力，具体内容可参考以下几点。

（1）学习计算机体系结构：了解计算机的基本组成和工作原理，包括处理器、内存、I/O 设备等。掌握计算机的底层知识可以为学习不同编程语言的底层实现提供基础。

（2）学习汇编语言：学习汇编语言可以帮助你理解不同编程语言的底层机器代码生成和执行过程。通过编写汇编代码，可以直接操作计算机的寄存器、内存和指令，深入了解底层计算机操作。

（3）探索编译器和解释器：学习编译器和解释器的原理和实现方式可以帮助你理解不同编程语言的编译和执行过程。了解编译器的词法分析、语法分析、语义分析和代码生成等阶段，可以更好地理解编程语言的工作原理。

（4）阅读源代码：深入研究开源编程语言的源代码可以帮助你理解其底层实现和设计思想。通过阅读编程语言的编译器、标准库或核心库的源代码，可以了解其内部机制和算法，从而更好地应用和理解该编程语言。

（5）实践项目：通过实践编写一些底层相关的项目，例如编写一个简单的编译器、解释器或虚拟机等，可以加深对编程语言底层机制的理解。通过手动实现一些底层功能，可

以更好地理解编程语言的运行原理。

（6）参考文档和书籍：查阅编程语言的官方文档和相关书籍，了解其底层实现细节和最佳实践。文档和书籍通常提供了深入的解释和示例代码，可以帮助理解编程语言的底层特性和机制。

注意，学习底层并不是为了在实际开发中始终使用底层技术，而是为了更好地理解和应用不同编程语言的特性，并进行性能优化。通过深入学习底层，可以更好地理解高级语言和框架的工作原理，提高编程能力和解决问题的能力。

2.7 小　　结

本章要点如图 2.14 所示，并总结如下。

- ☑ 面向对象出现的原因：让编程语言更贴近人类的思考方式。
- ☑ 面向对象引入的概念，基本都是在模仿人类的思考模式。而编译器基于模仿的角度，使用 ELF 文件格式的设计思想：各进程之间数据独立和代码共享来实现 C++语言的特性。
- ☑ 通过介绍 C++语言的 class，new/delete 的原理，进一步说明 C++语言特性的实现机制。
- ☑ 通过类比 CPU，C++语言，Java 的异常处理机制，表明计算机底层实现原理的思想同样适用于高级编程语言的实现方案。
- ☑ 基于汇编，C 和 C++语言存在的问题，推导出如果需要进一步解决软件开发的生产力问题，就需要新的编程语言。
- ☑ 从新语言的设计要求和编译器实现新语言的两种方法，得出 Java 应运而生的推论。
- ☑ 类比 C 语言编译执行和 C++语言编译执行，得出掌握计算机底层原理相当于修炼内功心法的结论。
- ☑ 通过深入学习底层，可以更好地理解高级语言和框架的工作原理，从而提高编程能力和解决问题的能力。

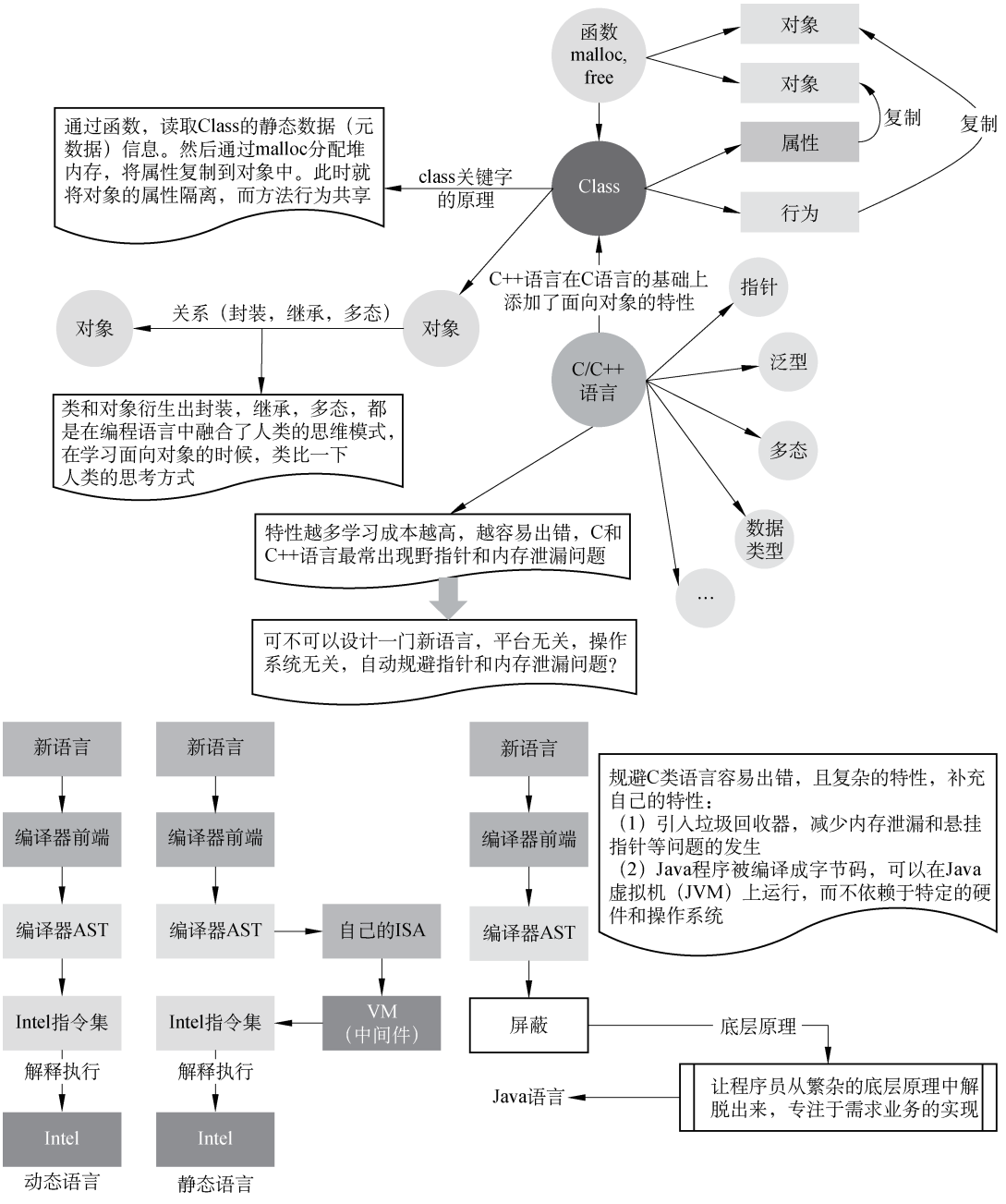


图 2.14　混沌知识树——C 类语言与新语言的出现