


钱进培训是哈法地区资深工程师组成的培训机构，通过各位老师的现身说法，帮助各位学员迅速掌握实战知识，为求职打下坚实的基础。电子邮件：jin.qian.canada@gmail.com 钱老师报名、答疑微信号：qianjincanada，或扫描以下二维码添加：



钱老师 
加拿大



扫一扫上面的二维码图案，加我微信

 钱进老师



 钱进老师

多态的基本概念

多态的深入理解

多态的本质

多态（Polymorphism）是面向对象（Object-Oriented，OO）思想"三大特征"之一，其余两个分别是封装（Encapsulation）和继承（Inheritance）。从一定角度来看，封装和继承几乎都是为多态而准备的。多态是一种机制、一种能力，而非某个关键字。它在类的继承中得以实现，在类的方法调用中得以体现。

多态(Polymorphism)的定义：同一消息可以根据发送对象的不同而采用多种不同的行为方式。具体的说，多态是指程序中定义的引用变量所指向的具体类型和通过该引用变量发出的方法调用在编程时并不确定，而是在程序运行期间才确定，即一个引用变量到底会指向哪个类的实例对象，该引用变量发出的方法调用到底是哪个类中实现的方法，必须在由程序运行期间才能决定。

因为在程序运行时才确定具体的类，这样，不用修改源程序代码，就可以让引用变量绑定到各种不同的类实现上，从而导致该引用调用的具体方法随之改变，即不修改程序代码就可以改变程序运行时所绑定的具体代码，让程序可以选择多个运行状态，这就是多态性。

定义了一个子类**Cat**，它继承了**Animal**类，那么后者就是前者是父类。我可以通过

`Cat c = new Cat();` 实例化一个**Cat**的对象

`Animal a = new Cat();` 这代表什么意思呢？

表示定义了一个**Animal**类型的引用，指向新建的**Cat**类型的对象。

由于**Cat**是继承自它的父类**Animal**，所以**Animal**类型的引用是可以指向**Cat**类型的对象的。

定义一个父类类型的引用指向一个子类的对象既可以使用子类强大的功能，又可以抽取父类的共性。

多态存在的三个必要条件

一、要有继承；

二、要有重写；

三、父类引用指向子类对象。

```
class Human {
public void showName() {
System.out.println("I am Human");
}
} // 继承关系
class Doctor extends Human {
// 方法重写
public void showName() {
System.out.println("I am Doctor");
}
}
class Programmer extends Human {
public void showName() {
System.out.println("I am Programmer");
}
}
public class Test {
// 向上转型
public Human humanFactory(String humanType) {
if ("doctor".equals(humanType)) {
return new Doctor();
}
if ("programmer".equals(humanType)) {
return new Programmer();
}
return new Human();
}
public static void main(String args[]) {
Test test = new Test();
Human human = test.humanFactory("doctor");
human.showName();// Output:I am Doctor
human = test.humanFactory("programmer");
human.showName();// Output:I am Programmer
// 一个接口的方法，表现出不同的形态，意即为多态也
}
}
```

对于多态，可以总结它为：

一、使用父类类型的引用指向子类的对象；

二、该引用只能调用父类中定义的方法和变量；

三、如果子类中重写了父类中的一个方法，那么在调用这个方法的时候，将会调用子类中的这个方法；（动态连接、动态调用）

四、变量不能被重写(覆盖),"重写"的概念只针对方法，如果在子类中”重写“了父类中的变量，那么在编译时会报错。

多态出现的缘由：Java的引用变量有两种类型：一个是编译时的类型，一个是运行时类型。

编译时类项：声明该变量时使用的类型决定。

运行时类项：实际赋给该变量的对象决定

如果编译时和运行时类项不一样就会出现所谓的多态（polymorphism）。

多态的深入理解

方法的**signature**是指方法的组成结构，具体包括方法的名称和参数，涵盖参数的数量、类型以及出现的顺序，但是不包括方法的返回值类型，访问权限修饰符，以及**abstract**、**static**、**final**等修饰符。比如下面两个就是具有相同型构的方法：

```
public void method(int i, String s) {
```

```
}
```

```
public String method(int i, String s) {
```

```
}
```

而这两个就是具有不同型构的方法：

```
public void method(int i, String s) {
```

```
}
```

```
public void method(String s, int i) {
```

```
}
```

重载，英文名是**overloading**，是指在同一个类中定义了一个以上具有相同名称，但是型构不同的方法。在同一个类中，是不允许定义多于一个的具有相同型构的方法的。

一个方法名，参数不同

```
void foo(String str);
```

```
void foo(int number);
```

在某个类中的方法可以重载（**overload**）另一个方法，只要它们具有相同的名字和不同的签名。由调用所指定的重载方法是在编译期选定的。

Java代码

```
class CircuitBreaker{  
    public void f (int i){} //int overloading  
    public void f(String s){} //String overloading  
}
```

```
public class OverloadPriority {

    public static void print(Object arg) {

        System.out.println("parameter type = Object");

    }

    public static void print(int arg) {

        System.out.println("parameter type = int");

    }

    public static void print(long arg) {

        System.out.println("parameter type = long");

    }

    public static void print(double arg) {

        System.out.println("parameter type = double");

    }

    public static void print(float arg) {

        System.out.println("parameter type = float");

    }

    public static void print(char arg) {

        System.out.println("parameter type = char");

    }

    public static void print(Character arg) {

        System.out.println("parameter type = Character");

    }

    public static void print(char... arg) {

        System.out.println("parameter type = char...");

    }

    public static void print(Serializable arg) {

        System.out.println("parameter type = Serializable");

    }

    public static void print(Comparable<?> arg) {

        System.out.println("parameter type = Comparable");

    }

}
```

```
public static void main(String[] args) {  
    // int  
    print('g');  
}
```

重写，英文名是**Override**，是指在继承情况下，子类中定义了与其基类中方法具有相同型构的新方法，就叫做子类把基类的方法重写了。这是实现多态必须的步骤。

```
class Parent {  
    void foo() {  
        System.out.println("Parent foo()");  
    }  
}  
class Child extends Parent {  
    void foo() {  
        System.out.println("Child foo()");  
    }  
}
```

在子类中构建与父类相同的方法名、输入参数、输出参数、访问权限（权限可以扩大），并且父类、子类都是静态方法，此种行为叫做隐藏（**Hide**），它与覆写有两点不同：

表现形式不同。隐藏用于静态方法，覆写用于非静态方法。在代码上的表现是：**@Override**注解可以用于覆写，不能用于隐藏。

职责不同。隐藏的目的是为了抛弃父类静态方法，重现子类方法，是为了遮盖父类的方法，也就是期望父类的静态方法不要破坏子类的业务行为；而覆写则是将父类的行为增强或减弱，延续父类的职责。

Java代码

```
class Base{  
    public static void f(){}  
}  
  
class Derived extends Base {  
    private static void f(){} //hides Base. f()  
}
```

1.静态变量与静态方法说继承并不确切，静态方法与变量是属于类的方法与变量。而子类也属于超类，比如说Manage extends Employee，则Manage也是一个Employee，所以子类能够调用属于超类的静态变量和方法。注意，子类调用的其实就是超类的静态方法和变量，而不是继承自超类的静态方法与变量。但是如果子类中有同名的静态方法与变量，这时候调用的就是子类本身的，因为子类的静态变量与静态方法会隐藏父类的静态方法和变量。

2.如果子类中没有定义同名的变量和方法，那么调用 "子类名.静态方法/变量"调用的是父类的方法及变量

3,.如果子类中只定义了同名静态变量，而没有定义与父类同名静态方法，则调用”子类名.静态方法"时，调用的是父类的静态方法，静态方法中的静态变量也是父类的

4.如果子类中既定义了与父类同名的静态变量，也定义了与父类同名的静态方法，这时候调用”子类名.静态方法"时，完全与父类无关，里面的静态变量也是子类的

遮蔽（shadow）

一个变量、方法或类型可以分别遮蔽（**shadow**）在一个闭合的文本范围内的具有相同名字的所有变量、方法或类型。如果一个实体被遮蔽了，那么你用它的简单名是无法引用到它的;根据实体的不同，有时你根本就无法引用到它。

Java代码

```
class WhoKnows{  
    static String sentence="I don't know.";  
    public static void main(String[] args){  
        String sentence="I don't know."; //shadows static field  
        System.out. println (sentence); // prints local variable  
    }  
}
```

遮掩（obscure）

一个变量可以遮掩具有相同名字的一个类型，只要它们都在同一个范围内:如果这个名字被用于变量与类型都被许可的范围，那么它将引用到变量上。相似地，一个变量或一个类型可以遮掩一个包。遮掩是唯一一种两个名字位于不同的名字空间的名字重用形式，这些名字空间包括:变量、包、方法或类型。如果一个类型或一个包被遮掩了，那么你不能通过其简单名引用到它，除非是在这样一个上下文环境中，即语法只允许在其名字空间中出现一种名字。遵守命名习惯就可以极大地消除产生遮掩的可能性:

Java代码

```
public class Obscure{
    static String System;// Obscures type java.lang.System
    public static void main(String[] args)
        // Next line won't compile:System refers to static field
        System.out.println("hello, obscure world!");
    }
}
```

`instanceof`是Java的一个二元操作符，和`==`，`>`，`<`是同一类。由于它是由字母组成的，所以也是Java的保留关键字。它的作用是测试它左边的对象是否是它右边的类的实例，返回`boolean`类型的数据。举个例子：

```
String s = "I AM an Object!";  
boolean isObject = s instanceof Object;
```

我们声明了一个`String`对象引用，指向一个`String`对象，然后用`instanceof`来测试它所指向的对象是否是`Object`类的一个实例，显然，这是真的，所以返回`true`，也就是`isObject`的值为`True`。

```
// obj instanceof T
boolean result;
if (obj == null) {
    result = false;
} else {
    try {
        T temp = (T) obj; // checkcast
        result = true;
    } catch (ClassCastException e) {
        result = false;
    }
}
```

instanceof 是javac能识别的一个关键字，做词法分析的时候扫描到"instanceof"关键字就映射到了一个Token.INSTANCEOF token

编译器的抽象语法树节点有一个JCTree.JCInstanceOf类用于表示instanceof运算。做语法分析的时候解析到instanceof运算符就会生成这个JCTree.JCInstanceOf类型的节点

生成字节码的时候为JCTree.JCInstanceOf节点生成instanceof字节码指令

向上转型

子类引用的对象转换为父类类型称为向上转型。通俗地说就是将子类对象转为父类对象。此处父类对象可以是接口。

```
Animal animal = new Cat(); //向上转型
```

```
    animal.eat();
```

```
    animal = new Dog();
```

```
    animal.eat();
```

向上转型时，子类单独定义的方法会丢失。比如上面Dog类中定义的run方法，当animal引用指向Dog类实例时是访问不到run方法的，animal.run()会报错。

子类引用不能指向父类对象。Cat c = (Cat)new Animal()这样是不行的。

向下转型

与向上转型相对应的就是向下转型了。向下转型是把父类对象转为子类对象

向下转型注意事项

向下转型的前提是父类对象指向的是子类对象（也就是说，在向下转型之前，它得先向上转型）

```
public class Animal {  
    int num = 10;  static int age = 20;  
    public void eat() {  
        System.out.println("动物吃饭");}  
    public static void sleep() {  
        System.out.println("动物在睡觉");}  
    public void run() {  
        System.out.println("动物在奔跑");}  
    }  
    public class Cat extends Animal {  
        int num = 80;  static int age = 90;  
        String name = "tomCat";  
        public void eat() {  
            System.out.println("猫吃饭");}  
        public static void sleep() {  
            System.out.println("猫在睡觉");}  
        public void catchMouse() {  
            System.out.println("猫在抓老鼠");}}  
}
```



```
public class CatDemo {  
    public static void main(String[] args) {  
        Animal am = new Cat();  
        am.eat();  
        am.sleep();  
        am.run();  
        // am.catchMouse();  
        // System.out.println(am.name);  
        System.out.println(am.num);  
        System.out.println(am.age);  
    }  
}
```

子类Cat重写了父类Animal的非静态成员方法am.eat();的输出结果为：猫吃饭。

子类重写了父类(Animal)的静态成员方法am.sleep();的输出结果为：动物在睡觉

未被子类（Cat）重写的父类（Animal）方法am.run()输出结果为：动物在奔跑

成员变量

编译看左边(父类),运行看左边(父类)

成员方法

编译看左边(父类), 运行看右边(子类)。动态绑定

静态方法

编译看左边(父类), 运行看左边(父类)。

父类的私有属性和私有方法，子类是不能访问的，当然一些父类的私有属性可能可以通过相应的方法访问到，但是私有的方法似乎不能简单的访问，这里暂不考虑Java反射机制。

子类不能继承父类私有的属性及方法？

在一个子类被创建的时候，首先会在内存中创建一个父类对象，然后在父类对象外部放上子类独有的属性，两者合起来形成一个子类的对象。所以所谓的继承使子类拥有父类所有的属性和方法其实可以这样理解，子类对象确实拥有父类对象中所有的属性和方法，但是父类对象中的私有属性和方法，子类是无法访问到的，只是拥有，但不能使用。

就像有些东西你可能拥有，但是你并不能使用。所以子类对象是绝对大于父类对象的，所谓的子类对象只能继承父类非私有的属性及方法的说法是错误的。可以继承，只是无法访问到。

```
public class FieldDemo {  
    public static void main(String[] args){  
        Student t = new Student("Jack");  
        Person p = t;//父类创建的引用指向子类所创建的对象  
        System.out.println(t.name+","+p.name);  
        System.out.println(t.getName()+"-"+p.getName());  
    }  
  
}  
  
class Person{  
    String name;  
    int age;  
    public String getName(){  
        return this.name;  
    }  
}
```

```
class Student extends Person{
    String name; // 属性和父类属性名相同，但在做开发时一般不会和父类属性名相同！！
    public Student(String name){
        this.name = name;
        super.name = "Rose"; // 为父类中的属性赋值
    }
    public String getName(){
        return this.name;
    }
}
```

返回结果是：

Jack,Rose

Jack,Jack

在Java中，属性绑定到类型，方法绑定到对象

静态属性、静态方法和非静态的属性都可以被继承和隐藏而不能被重写，因此不能实现多态，不能实现父类的引用可以指向不同子类的对象。非静态方法可以被继承和重写，因此可以实现多态。

多态的本质

// 定义一个类Foo

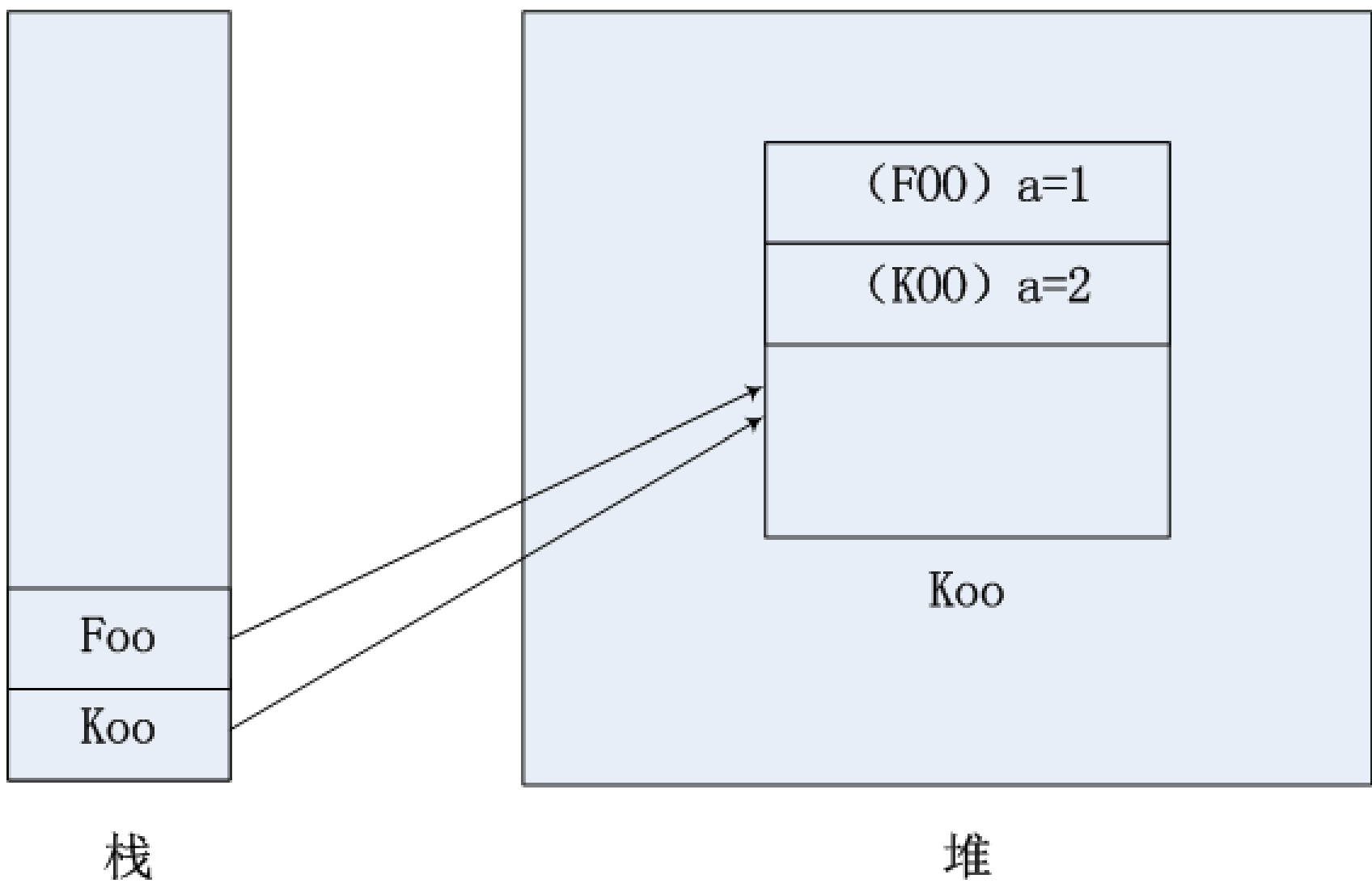
```
class Foo {  
    int a = 1;  
    public int getA() {  
        return a;  
    }  
}
```

// 定义一个类Koo继承于Foo

```
class Koo extends Foo {  
    int a = 2;  
    public int getA() {  
        return a;// 这里相当于是this.a , 如果换成super.a , 则表示访问父类a  
    }  
}
```

/ 方法是动态绑定的，属性是静态绑定的

```
public class TestDemo {  
    public static void main(String[] args) {  
        Koo koo = new Koo();  
        Foo foo = koo;  
        System.out.println(koo.a + ", " + koo.getA()); // 输出: 2, 2  
        System.out.println(foo.a + ", " + foo.getA()); // 输出: 1, 2  
        boolean isFoo = koo instanceof Foo;  
        boolean isKoo = koo instanceof Koo;  
        System.out.println(isFoo + ", " + isKoo); // 输出: true, true  
    }  
}
```

- `Koo koo = new Koo();` // 堆中开辟一个KOO内存区，由于KOO继承于FOO，子类实例化的时候，先父类再子类，而且属性是静态绑定，所以，内存区中先有一个FOO的a，再有一个KOO的a，此时，a都没有赋值，父类和子类的属性都有内存区后，再一个个赋值，先给Foo的a赋值1，再Koo的a赋值2。栈中的koo指向堆中koo的位置，可以理解为栈中的koo存储的值是堆中koo的地址标识。
- `FOO foo = koo;` // 把koo的值赋给foo，那么foo也存储着堆中koo的地址标识，如上图1所示。
- `System.out.println(koo.a + ", " + koo.getA());` // koo.a的值为2，由于属性是静态绑定，koo.a的值仍然取得是Koo类中的；`koo.getA()`的值是2，调用自身的方法，取得koo中的a，为2。
- `System.out.println(foo.a + ", " + foo.getA());` // foo.a的值为1，由于属性是静态绑定，foo.a的值仍然是Foo类中的；`foo.getA()`的值是2，方法是动态绑定的，`foo.getA()`调用的是子类的`getA()`方法，取得的是子类的a的值。如果这里想要取得父类的a的值，那么可以用`super.a`。
- 总之，一句话，属性是静态绑定，方法是动态绑定

绑定指的是一个方法的调用与方法所在的类(方法主体)关联起来。对Java来说，绑定分为静态绑定和动态绑定；或者叫做前期绑定和后期绑定。

(1)静态绑定：

在程序执行前方法已经被绑定，此时由编译器或其它连接程序实现。例如：C。

针对java简单的可以理解为程序编译期的绑定；这里特别说明一java当中的方法只有final，static， private和构造方法是前期绑定。

(2)动态绑定：

后期绑定：在运行时根据具体对象的类型进行绑定。

若一种语言实现了后期绑定，同时必须提供一些机制，可在运行期间判断对象的类型，并分别调用适当的方法。

也就是说，编译器此时依然不知道对象的类型，但方法调用机制能自己去调查，找到正确的方法主体。

不同的语言对后期绑定的实现方法是有所区别的。但我们至少可以这样认为：它们都要在对象中安插某些特殊类型的信息。

- **java**当中的方法只有**final**、**static**、**private**和构造方法是静态绑定（前期绑定、非运行时绑定）：
- （1）**final**方法虽然可以被继承，但是不能被重写，虽然子类对象可以调用，但是调用的都是父类的**final**方法。所以被定义为**final**的方法，可以有效的防止被重写，并有效的关闭动态绑定。
- （2）**private**方法是私有的，无法被继承，只能通过该类自身的对象来调用，所以**private**方法无法动态绑定。
- （3）**static**方法可以被继承，也可以被重写，但是它可以看成是类的方法，与具体类绑定在一起，**static**方法无法动态绑定

构造方法是不能被继承的，子类的构造方法会默认先调用父类的无参构造（如果有指定**super(...)**的参数，那么会先调用父类的指定构造函数），构造函数无法动态绑定

而动态绑定的典型发生在父类和子类的转换声明之下：

比如： `Parentp=newChildren();`

其具体过程细节如下：

1： 编译器检查对象的声明类型和方法名。

假设我们调用`x.f(args)`方法，并且`x`已经被声明为`C`类的对象，那么编译器会列举出`C`类中所有的名称为`f`的方法和从`C`类的超类继承过来的`f`方法

2： 接下来编译器检查方法调用中提供的参数类型。

如果在所有名称为`f`的方法中有一个参数类型和调用提供的参数类型最为匹配，那么就调用这个方法，这个过程叫做“重载解析”

3： 当程序运行并且使用动态绑定调用方法时，虚拟机必须调用同`x`所指向的对象的实际类型相匹配的方法版本。假设实际类型为`D`(`C`的子类)，如果`D`类定义了`f(String)`那么该方法被调用，否则就在`D`的超类中搜寻方法`f(String)`,依次类推。

Java的重载解析过程是以两阶段运行的。第一阶段 选取所有可获得并且可应用的方法或构造器。第二阶段在第一阶段选取的方法或构造器中选取最精确的一个。如果一个方法或构造器可以接受传递给另一个方法或构造器的任何参数，那么我们就说第一个方法比第二个方法缺乏精确性

```
public class TestNull {  
    public void show(String a){  
        System.out.println("String");  
    }  
    public void show(Object o){  
        System.out.println("Object");  
    }  
    public static void main(String args[]){  
        TestMain t = new TestMain();  
        t.show(null);  
    }  
}
```

结果是: String

(1) 所有私有方法、静态方法、构造器及初始化方法<clinit>都是采用静态绑定机制。在编译器阶段就已经指明了调用方法在常量池中的符号引用，JVM运行的时候只需要进行一次常量池解析即可。

(2) 类对象方法的调用必须在运行过程中采用动态绑定机制。

首先，根据对象的声明类型(对象引用的类型)找到“合适”的方法。具体步骤如下：

- ① 如果能在声明类型中匹配到方法签名完全一样(参数类型一致)的方法，那么这个方法是最合适的。
- ② 在第①条不能满足的情况下，寻找可以“凑合”的方法。标准就是通过将参数类型进行自动转型之后再行匹配。如果匹配到多个自动转型后的方法签名f(A)和f(B)，则用下面的标准来确定合适的方法：传递给f(A)方法的参数都可以传递给f(B)，则f(A)最合适。反之f(B)最合适。
- ③ 如果仍然在声明类型中找不到“合适”的方法，则编译阶段就无法通过。

然后，根据在堆中创建对象的实际类型找到对应的方法表，从中确定具体的方法在内存中的位置。

总结

```
class A {  
    public String show(D obj) {  
        return ("A and D");    }  
    public String show(A obj) {  
        return ("A and A");}  
}
```

```
class B extends A{  
    public String show(B obj){  
        return ("B and B");    }  
    public String show(A obj){  
        return ("B and A");    }  
}
```

```
class C extends B{}
```

```
class D extends B{}
```

```
public class Demo {  
    public static void main(String[] args) {  
        A a1 = new A();  
        A a2 = new B();  
        B b = new B();  
        C c = new C();  
        D d = new D();  
        System.out.println("1--" + a1.show(b));  
        System.out.println("2--" + a1.show(c));  
        System.out.println("3--" + a1.show(d));  
        System.out.println("4--" + a2.show(b));  
        System.out.println("5--" + a2.show(c));  
        System.out.println("6--" + a2.show(d));  
        System.out.println("7--" + b.show(b));  
        System.out.println("8--" + b.show(c));  
        System.out.println("9--" + b.show(d));  
    }  
}
```

//结果:

//1--A and A

//2--A and A

//3--A and D

//4--B and A

//5--B and A

//6--A and D

//7--B and B

//8--B and B

//9--A and D

比如④，`a2.show(b)`，`a2`是一个引用变量，类型为A，则`this`为`a2`，`b`是B的一个实例，于是它到类A里面找`show(B obj)`方法，没有找到，于是到A的`super`(超类)找，而A没有超类，因此转到第三优先级`this.show((super)O)`，`this`仍然是`a2`，这里O为B，`(super)O`即`(super)B`即A，因此它到类A里面找`show(A obj)`的方法，类A有这个方法，但是由于`a2`引用的是类B的一个对象，B覆盖了A的`show(A obj)`方法，因此最终锁定到类B的`show(A obj)`，输出为"B and A"。

变量看左边，
方法看右边，
静态看左边。

再比如⑧，`b.show(c)`，`b`是一个引用变量，类型为`B`，则`this`为`b`，`c`是`C`的一个实例，于是它到类`B`找`show(C obj)`方法，没有找到，转而在`B`的超类`A`里面找，`A`里面也没有，因此也转到第三优先级`this.show((super)O)`，`this`为`b`，`O`为`C`，`(super)O`即`(super)C`即`B`，因此它到`B`里面找`show(B obj)`方法，找到了，由于`b`引用的是类`B`的一个对象，因此直接锁定到类`B`的`show(B obj)`，输出为"`B and B`"。

静态分派 意思是 所有依赖静态类型来定位方法执行版本的分派过程就叫做静态分派，静态分派最典型的应用就是方法重载。

动态单分派 意思是 根据运行期实际类型确定方法执行版本的分派过程叫做动态分派，动态分派最典型的应用就是方法重写。

单分派： 就是说 依据只有一个

多分派： 就是说 依据可以有多个

在程序设计语言中，许多时候同一个概念的操作或运算可能需要针对不同数量、不同类型的数据而做不同的处理。既然是“同一概念”，如果能用同样的名字来命名这个操作或运算的函数，会有助于程序代码清晰的表达出语义。但是函数的名字一样了，程序该如何判断应该选用同名函数的哪个版本就成了个问题，这里就需要在编译时由编译器来选择，或在运行时进行方法分派。

编译器选择的过程：在类加载的解析阶段完成，这个解析阶段解析的部分“符号引用”必须满足下面的条件：

方法在程序真正运行前就有一个确定的版本。

这个方法的调用版本在运行期间是不可改变的。

方法调用过程是指 确定被调用方法的版本（即调用哪一个方法），Class 文件的编译过程中并不包括传统编译中的连接步骤，一切方法调用在 Class 文件,调用里面存储的都只是符号引用，而不是方法在实际运行时的内存布局入口地址，也就是说 符号引用解析成直接引用的过程。这个特性使得Java 具有强大的动态扩展能力，但也使得 Java方法调用过程变得复杂起来，需要在类加载器件，甚至是运行期间才确定目标方法的直接引用。

在类加载的解析阶段，会将其中一部分符号引用直接转化为直接引用，前提是：方法在程序真正运行之前就有一个可确定的版本，并且这个方法的调用版本在运行期是不可改变的。换句话说，调用目标在程序代码写好，编译器进行编译时必须确定下来。这类方法的调用称为解析（Resolution）。

符合这个方法要求的方法，主要包括：静态方法和私有方法。因为这两个方法的特点就决定了他们都不可能通过继承或别的方式重写其他版本。

解析调用是一个静态的过程，在编译器就完全确定了，在类载入时就会将涉及到的符号引用全部转变为可确定的直接引用，不会延迟到运行期再去执行。

由于解析是一个静态的过程，编译期间即可确定，在解析的时候就能把符号引用直接转化成直接引用。

与之相对应的，Java 虚拟机里面提供了5条方法调用字节码指令，分别如下：

`invokestatic`: 调用静态方法

`invokespecial`: 调用<init>方法、私有方法和父类方法

`invokevirtual`: 调用所有的虚方法

`invokeinterface`: 调用接口方法，会在运行时在确定一个实现此接口的对象

`invokedynamic`: 会在运行时动态解析出调用限定符所引用的方法，然后再执行该方法。

只能被 `invokestatic` 和 `invokespecial` 调用的方法，都可以在解析阶段中确定唯一的调用版本，符合这个条件的有 静态方法、私有方法、实例构造器、父类方法 4类，这些方法称为 非虚方法，由于 `final` 修饰的方法不能被覆盖，也属于非虚方法。与之相反，其他的方法称为虚方法。

Java中的成员方法（非静态方法）都是虚方法。这里的A.foo(int)与B.foo(int)就是同一继承链上signature相同的两个虚方法，B.foo(int)覆盖（override）A.foo(int)。从语义上说，在编译时无法判断一个虚方法调用到底应该采用继承链上signature相同的哪个版本，所以要留待运行时进行分派（dispatch）。

```
class A {  
    public void foo( int i ) {  
        System.out.println( "A.foo( int )" );  
    }  
}
```

```
class B extends A {  
    @Override  
    public void foo( int i ) {  
        System.out.println( "B.foo( int )" );  
    }  
}
```

```
public class Program {  
    public static void main( String[ ] args ) {  
        A b = new B( );  
        b.foo( 0 ); // B.foo( int )  
    }  
}
```

java中，还存在一种调用（分派调用），它既可以静态也可以动态。其实在这里，java多态的性质实现会得到体现。

重载(overload)与静态分派

```
public class Main {  
    static abstract class Father { }  
    static class Son extends Father { }  
    static class Daughter extends Father { }  
    public void getSex(Daughter daughter) {      System.out.println("i am a girl"); }  
    public void getSex(Son son) {      System.out.println("i am a boy"); }  
    public void getSex(Father son) {      System.out.println("i am a father"); }  
    public static void main(String[] args) {  
        Father son = new Son();  
        Father daughter = new Daughter();  
        Main main = new Main();  
        main.getSex(son);  
        main.getSex(daughter);  
    }  
}
```

main对象已经确认了，那么main在运行main.getSex(son);时选择方法的时候，到底是选择getSex(Son son)还是getSex(Father son)呢？

我们在代码中son的引用类型是Father,但是它的实际类型却是Son。

java编译器在重载的时候是通过参数的静态类型而不是实际类型来确定使用哪个重载的版本的。

依赖静态类型来定位方法执行的版本的分派动作成为静态分派。静态分派的典型应用是方法重载，而且静态分派发生在编译期间，因此，静态分派的动作是由编译器发出的。

编译器能确定出方法的重载版本，但在很多的时候，这个版本并不一定是唯一的


动态分派是选择的最合适的一个方法版本来重载，如果找不到精确匹配的方法，只有作出适当的妥协，向上转型，直到Object。

```
public class Test {  
    public static void main(String[] args) {  
        Inf inf = new Sub();  
        inf.f();  
    }  
}  
  
class Super {  
    public void f() {  
        System.out.println("Super f()");  
    }  
}  
  
interface Inf {  
    void f();  
}  
  
class Sub extends Super implements Inf {  
}
```


当集成父类并且同时实现接口的时候，可以不重写接口方法，此时测试会执行父类的**f()**方法。

钱进培训是哈法地区资深工程师组成的培训机构，通过各位老师的现身说法，帮助各位学员迅速掌握实战知识，为求职打下坚实的基础。电子邮件：jin.qian.canada@gmail.com 钱老师报名、答疑微信号：qianjincanada，或扫描以下二维码添加：



钱老师 
加拿大



扫一扫上面的二维码图案，加我微信

 钱进老师



 钱进老师

泛型

语法糖（Syntactic Sugar），也称糖衣语法，是由英国计算机学家 Peter.J.Landin 发明的一个术语，指在计算机语言中添加的某种语法，这种语法对语言的功能并没有影响，但是更方便程序员使用。**Java** 中最常用的语法糖主要有泛型、变长参数、条件编译、自动拆装箱、内部类等。虚拟机并不支持这些语法，它们在编译阶段就被还原回了简单的基础语法结构，这个过程成为解语法糖。

Java 泛型（generics）是 JDK 5 中引入的一个新特性, 泛型提供了编译时类型安全检测机制，该机制允许程序员在编译时检测到非法的类型。泛型的本质是参数化类型，也就是说所操作的数据类型被指定为一个参数。类型参数的意义是告诉编译器这个集合中要存放实例的类型，从而在添加其他类型时做出提示，在编译时就为类型安全做了保证。

这种参数类型可以用在类、接口和方法的创建中，分别称为

泛型类

泛型接口

泛型方法

假定我们有这样一个需求：写一个排序方法，能够对整型数组、字符串数组甚至其他任何类型的数组进行排序，该如何实现？

答案是可以使用 **Java 泛型**。

使用 **Java 泛型**的概念，我们可以写一个泛型方法来对一个对象数组排序。然后，调用该泛型方法来对整型数组、浮点数数组、字符串数组等进行排序。

术语

- `ArrayList<E>` -- 泛型类型
- `ArrayList` -- 原始类型
- `E` -- 类型参数
- `<>` -- 读作"typeof"
- `ArrayList<Integer>` -- 参数化的类型
- `Integer` -- 实际类型参数

Java 语言中的泛型基本上完全在编译器中实现，由编译器执行类型检查和类型推断，然后生成普通的非泛型的字节码。这种实现技术称为擦除（**erasure**）（编译器使用泛型类型信息保证类型安全，然后在生成字节码之前将其清除）

正确理解泛型概念的首要前提是理解类型擦除（**type erasure**）。Java中的泛型基本上都是在编译器这个层次来实现的。在生成的Java字节代码中是不包含泛型中的类型信息的。使用泛型的时候加上的类型参数，会被编译器在编译的时候去掉。这个过程就称为类型擦除。如在代码中定义的List<Object>和List<String>等类型，在编译之后都会变成List。JVM看到的只是List，而由泛型附加的类型信息对JVM来说是不可见的。

其实对于初学者对这一点可以将Java泛型简单理解为一种将因类型错误而引发的异常提前到编码阶段(避免在运行时出现ClassCastException)。当开发者在错误的调用泛型类和方法时IDE就会提示错误，而不用等到程序真正运行时再报错。

理解Java泛型最简单的方法是把它看成一种便捷语法，能节省你某些Java类型转换(casting)上的操作：

```
List<Apple> box = new ArrayList<Apple>();
```

```
box.add(new Apple());
```

```
Apple apple =box.get(0);
```

上面的代码自身已表达的很清楚：**box**是一个装有**Apple**对象的**List**。**get**方法返回一个**Apple**对象实例，这个过程不需要进行类型转换。没有泛型，上面的代码需要写成这样：

```
Apple apple = (Apple)box.get(0);
```

```
public class Box {  
    private Object object;  
  
    public void set(Object object) { this.object = object; }  
    public Object get() { return object; }  
}
```

因为类中的方法接受的参数类型是**Object**，所以可以接受任何类型的输入参数，。但是在编译期没有办法验证**Box**类是否被正确的使用了。如果期望从**Box**中获得一个**Integer**而却往**Box**中放入**String**类型的值，则会出现运行时错误。


```
public class Box<T> {  
    // T stands for "Type"  
    private T t;  
  
    public void set(T t) { this.t = t; }  
    public T get() { return t; }  
}
```

```
Box<Integer> integerBox = new Box<Integer>();
```

类型形参（Type Parameter）和类型实参（Type Argument）：Box<T>中的T是类型形参，Box<Integer>中的Integer是类型实参。在能区分两者的语境中，经常不做区分的称为类型参数。

在引入范型之前，要让类中的方法支持多个数据类型，就需要对方法进行重载；在引入范型之后，可以更进一步定义多个参数以及返回值之间的关系。

例如，`public void write(Integer i, Integer[] ia);`及`public void write(Double d, Double[] da);`的范型版本为：`public <T> void write(T t, T[] ta);`

在Java 7以前的版本中使用泛型类型，需要在声明并赋值的时候，两侧都加上泛型类型。比方说这样：

```
Map<String,Integer> map = new HashMap<String,Integer>();
```

在Java SE 7中，这种方式得以改进，引入了类型推导，即type inference的出现，再写上面这样的代码的时候，可以省略掉对象实例化时的参数类型，也就变成了这个样子：

```
Map<String,Integer> map = new HashMap<>(); //注意后面的"<>"
```

在这条语句中，编译器会根据变量声明时的泛型类型自动推断出实例化HashMap时的泛型类型。再次提醒一定要注意new HashMap后面的“<>”，只有加上这个“<>”才表示是自动类型推断，否则就是非泛型类型的HashMap，并且在使用编译器编译源代码时会给出一个警告提示（unchecked conversion warning）。这一对尖括号"<>"官方文档中叫做"diamond"。

如果编译器可以根据上下文推断类型实参，则可以在调用构造方法时省略类型实参，例如：

```
Map<String, List<String>> myMap = new HashMap<String, List<String>>();
```

也可以写成这样：

```
Map<String, List<String>> myMap = new HashMap<>();
```

但是为了能够使编译器进行类型推断，调用构造方法的时候必须使用**the diamond**，否则编译器报编译警告：

```
Map<String, List<String>> myMap = new HashMap(); // unchecked conversion warning
```

原因是HashMap()构造方法是HashMap>的原始类型（raw types）

类型推断是Java编译器根据方法调用和相应的声明去推断类型实参的过程。类型推断算法往往取决于实参类型、赋值操作等号右边的类型或返回值类型。最后，类型推断算法会找到适合所有条件的最具体的类型作为推断结果。

例如，下面的例子，类型推断的结果是Serializable：

```
static <T> T pick(T a1, T a2) { return a2; }  
Serializable s = pick("d", new ArrayList<String>());
```

泛型类

泛型类和普通类的区别就是类名后有类型参数列表，既然叫“列表”了，当然这里的类型参数可以有多个

泛型类的定义格式

```
[访问权限] class 类名称<泛型标识符1,泛型标识符2,....,泛型标识符n>{  
[访问权限] 泛型类型标识 变量名称;  
[访问权限] 泛型类型标识 方法名称(){}  
[访问权限] 返回值类型声明 方法名称(泛型类型标识 变量名称){}  
}
```

泛型对象定义

类名称<具体类型> 对象名称 = new 类名称<具体类>();

规则：在定义带类型参数的类时，在紧跟类名之后的<>内，指定一个或多个类型参数的名字，同时也可以对类型参数的取值范围进行限定，多个类型参数之间用","进行分隔。

说明：定义完类型参数后，可以在类中定义位置之后的几乎任意地方使用类型参数（静态块，静态属性，静态方法除外），就像使用普通的类型一样。

注意：父类定义的类型参数不能被子类继承。

泛型接口

在JDK1.5之后泛型也可以应用到接口中，其可以利用如下的语法定义：

```
[访问权限] interface 接口名称<泛型标识>{  
    }
```

例如

```
Interface Info<T>{  
    public T getVar();  
}
```

在定义完泛型接口后，就要定义此接口的子类。定义泛型接口的子类有两种方式：

方式一

直接在子类后声明泛型

方式二

直接在子类实现的接口中明确地给出泛型类型

泛型方法

在类中可以定义泛型方法，泛型方法的定义与其所在的类是否是泛型类没有任何的关系。

泛型方法是自身引入类型参数的方法。与泛型类和接口引入的泛型参数不同，其作用范围只在方法声明范围内有效。静态方法、非静态方法以及泛型类的构造方法都可以声明为泛型方法

在泛型方法中可以定义泛型参数，此时参数的类型就是传入数据的类型，可以使用如下的格式定义泛型方法。

访问权限 <泛型标识> 泛型标识 方法名称([泛型标识 参数名称])


```
public class BoxDemo {  
    public static <U> void addBox(U u,    java.util.List<Box<U>> boxes) {  
        Box<U> box = new Box<>();  
        box.set(u);  
        boxes.add(box);  
    }  
    public static <U> void outputBoxes(java.util.List<Box<U>> boxes) {  
        int counter = 0;  
        for (Box<U> box: boxes) {  
            U boxContents = box.get();  
            System.out.println("Box #" + counter + " contains [" +    boxContents.toString() + "]);  
            counter++;  
        }  
    }  
}
```

```
public static void main(String[] args) {  
    java.util.ArrayList<Box<Integer>> listOfIntegerBoxes =  
        new java.util.ArrayList<>();  
    BoxDemo.<Integer>addBox(Integer.valueOf(10), listOfIntegerBoxes);  
    BoxDemo.addBox(Integer.valueOf(20), listOfIntegerBoxes);  
    BoxDemo.addBox(Integer.valueOf(30), listOfIntegerBoxes);  
    BoxDemo.outputBoxes(listOfIntegerBoxes);  
}  
}
```

泛型方法**addBox**定义了类型形参**U**，通常Java编译器可以通过方法调用推断类型实参，所以在多数情况下，不需要指定类型实参。例如调用方法**addBox**时，可以像这样指定类型实参：

```
BoxDemo.<Integer>addBox(Integer.valueOf(10), listOfIntegerBoxes);
```

也可以不指定，由编译器根据参数类型推断类型实参为**Integer**：

```
BoxDemo.addBox(Integer.valueOf(20), listOfIntegerBoxes);
```

```
public class GenericsFunction1 {  
  
    public <T> T fun(T t){  
        return t;  
    }  
  
    public static void main(String[] args){  
  
        GenericsFunction1 obj1 = new GenericsFunction1();  
  
        System.out.println(obj1.fun("this's String"));  
  
        System.out.println(obj1.fun(1));  
    }  
}
```

使用大写字母A,B,C,D.....X,Y,Z定义的，就都是泛型，把T换成A也一样，这里T只是名字上的意义而已

? 表示不确定的java类型

T (type) 表示具体的一个java类型

K V (key value) 分别代表java键值中的Key Value

E (element) 代表Element

<? extends T>上限通配，表示? 是T的一个未知子类。

<? super T>下限通配，表示? 是T的一个未知父类。

这里? 表示一个未知的类，而T是一个具体的类，在实际使用的时候T需要替换成一个具体的类，表示实例化的时候泛型参数要是T的子类。

```
public abstract class Fruit {  
    public abstract void eat();  
}  
  
public class Apple extends Fruit {  
    @Override  
    public void eat() {  
        System.out.println("我是苹果，我是酸酸甜甜的");  
    }  
}  
  
public class People<T extends Fruit> {  
    public void eatFruit(T t){  
        t.eat();  
    }  
}
```

```
People<Apple> p1 = new People<>();  
    p1.eatFruit(new Apple());
```


<? super T> 下限通配

这里? 表示一个未知的类，而T是一个具体的类，在实际使用的时候T需要替换成一个具体的类，表示实例化的时候泛型参数要是T的父类。

```
public static void addNumbers(List<? super Integer> list) {  
    for (int i = 1; i <= 10; i++) {  
        list.add(i);  
    }  
}
```

```
List<Integer> list1 = new ArrayList<>();  
    addNumbers(list1);
```

```
List<String> list4 = new ArrayList<>();  
    //编译错误，因为String不是Integer的父类  
    //addNumbers(list4);
```

钱进培训是哈法地区资深工程师组成的培训机构，通过各位老师的现身说法，帮助各位学员迅速掌握实战知识，为求职打下坚实的基础。电子邮件：jin.qian.canada@gmail.com 钱老师报名、答疑微信号：qianjincanada，或扫描以下二维码添加：



钱老师 
加拿大



扫一扫上面的二维码图案，加我微信

 钱进老师



 钱进老师

通常，对于一个给定的算法，我们要做两项分析。第一是从数学上证明算法的正确性，这一步主要用到形式化证明的方法及相关推理模式，如循环不变式、数学归纳法等。而在证明算法是正确的基础上，第二部就是分析算法的时间复杂度。算法的时间复杂度反映了程序执行时间随输入规模增长而增长的量级，在很大程度上能很好反映出算法的优劣与否。

为什么要进行算法复杂度分析？

预测算法所需的资源

计算时间（CPU 消耗）

内存空间（RAM 消耗）

通信时间（带宽消耗）

预测算法的运行时间

在给定输入规模时，所执行的基本操作数量。

或者称为算法复杂度（Algorithm Complexity）

算法的运行时间与什么相关？

取决于输入的数据。（例如：如果数据已经是排好序的，时间消耗可能会减少。）

取决于输入数据的规模。（例如：6 和 $6 * 10^9$ ）

算法分析的种类：

最坏情况（**Worst Case**）：任意输入规模的最大运行时间。（**Usually**）

平均情况（**Average Case**）：任意输入规模的期待运行时间。（**Sometimes**）

最佳情况（**Best Case**）：通常最佳情况不会出现。（**Bogus**）

例如，在一个长度为 n 的列表中顺序搜索指定的值，则

最坏情况： n 次比较

平均情况： $n/2$ 次比较

最佳情况：1 次比较

实际中，我们一般仅考量算法在最坏情况下的运行情况，也就是对于规模为 n 的任何输入，算法的最长运行时间。这样做的理由是：

一个算法的最坏情况运行时间是在任何输入下运行时间的一个上界（Upper Bound）。

对于某些算法，最坏情况出现的较为频繁。

大体上看，平均情况通常与最坏情况一样差。

渐近记号（Asymptotic Notation）通常有 O 、 Θ 和 Ω 记号法。 Θ 记号渐进地给出了一个函数的上界和下界，当只有渐近上界时使用 O 记号，当只有渐近下界时使用 Ω 记号。尽管技术上 Θ 记号较为准确，但通常仍然使用 O 记号表示。

$O(\text{big-O})$ $\Omega(\text{big-Omega})$ $\Theta(\text{big-theta})$

符号后面括号里的是他们相应的读法。

第一个符号的意义相当于“小于等于”， O 是一个算法最坏情况的度量

第二个符号的意义相当于“大于等于”， Ω 是最好情况的度量

第三个符号的意义相当于“等于”，表达了一个算法的区间，不会好于某某，不会坏于某

- $O(\text{big-O})$:

最普遍的符号。它是渐进上界，其作用是将我们得到的算法在最坏情况下（**worst case**）时间复杂度表达式简化成对应的多项式（比如 n^2 等）。所以在我们证明的过程中，目的是证明我们的式子要“小于等于”目标多项式。

- $\Omega(\text{big-Omega})$:

这个符号是渐进下界，其作用是将我们得到的算法在最好情况下（**best case**）时间复杂度表达式简化成对应的多项式（也比如 n^2 等）。所以在我们证明的过程中，目的是证明我们的式子要“大于等于”目标多项式。

- $\Theta(\text{big-theta})$:

如果 O 和 Ω 可以用同一个多项式表示，那么这个多项式就是我们所要求的渐进紧的界了。其作用是将我们可以较准确地得到算法的时间复杂度表达式对应的多项式（也比如 n^2 等）。所以在我们证明的过程中，目的是证明我们的式子要“等于”目标多项式。

算法执行时间需通过依据该算法编制的程序在计算机上运行时所消耗的时间来度量。而度量一个程序的执行时间通常有两种方法。

一、事后统计的方法

这种方法可行，但不是一个好的方法。该方法有两个缺陷：一是要想对设计的算法的运行性能进行评测，必须先依据算法编制相应的程序并实际运行；二是所得时间的统计量依赖于计算机的硬件、软件等环境因素，有时容易掩盖算法本身的优势。

二、事前分析估算的方法

因事后统计方法更多的依赖于计算机的硬件、软件等环境因素，有时容易掩盖算法本身的优劣。因此人们常常采用事前分析估算的方法。

一个算法的空间复杂度(Space Complexity) $S(n)$ 定义为该算法所耗费的存储空间，它也是问题规模 n 的函数。渐近空间复杂度也常常简称为空间复杂度。

空间复杂度(Space Complexity)是对一个算法在运行过程中临时占用存储空间大小的量度。一个算法在计算机存储器上所占用的存储空间，包括存储算法本身所占用的存储空间，算法的输入输出数据所占用的存储空间和算法在运行过程中临时占用的存储空间这三个方面。算法的输入输出数据所占用的存储空间是由要解决的问题决定的，是通过参数表由调用函数传递而来的，它不随本算法的不同而改变。存储算法本身所占用的存储空间与算法书写的长短成正比，要压缩这方面的存储空间，就必须编写出较短的算法。算法在运行过程中临时占用的存储空间随算法的不同而异，有的算法只需要占用少量的临时工作单元，而且不随问题规模的大小而改变，我们称这种算法是“就地”进行的，是节省存储的算法；有的算法需要占用的临时工作单元数与解决问题的规模 n 有关，它随着 n 的增大而增大，当 n 较大时，将占用较多的存储单元，例如快速排序和归并排序算法就属于这种情况。

如当一个算法的空间复杂度为一个常量，即不随被处理数据量 n 的大小而改变时，可表示为 $O(1)$ ；当一个算法的空间复杂度与以2为底的 n 的对数成正比时，可表示为 $O(\lg n)$ ；当一个算法的空间复杂度与 n 成线性比例关系时，可表示为 $O(n)$ 。若形参为数组，则只需要为它分配一个存储由实参传送来的一个地址指针的空间，即一个机器字长空间；若形参为引用方式，则也只需要为其分配存储一个地址的空间，用它来存储对应实参变量的地址，以便由系统自动引用实参变量。

渐进复杂度（Asymptotic Complexity）

计算代码块的渐进运行时间的方法有如下步骤：

确定决定算法运行时间的组成步骤。

找到执行该步骤的代码，标记为 **1**。

查看标记为 **1** 的代码的下一行代码。如果下一行代码是一个循环，则将标记 **1** 修改为 **1** 倍于循环的次数 $1 * n$ 。如果包含多个嵌套的循环，则将继续计算倍数，例如 $1 * n * m$ 。

找到标记到的最大的值，就是运行时间的最大值，即算法复杂度描述的上界。

(1) 找出算法中的基本语句;

算法中执行次数最多的那条语句就是基本语句, 通常是最内层循环的循环体。

(2) 计算基本语句的执行次数的数量级;

只需计算基本语句执行次数的数量级, 这就意味着只要保证基本语句执行次数的函数中的最高次幂正确即可, 可以忽略所有低次幂和最高次幂的系数。这样能够简化算法分析, 并且使注意力集中在最重要的一点上: 增长率。

(3) 用大O记号表示算法的时间性能。

将基本语句执行次数的数量级放入大O记号中。

如果算法中包含嵌套的循环, 则基本语句通常是最内层的循环体, 如果算法中包含并列的循环, 则将并列循环的时间复杂度相加。例如:

```
for (i=1; i<=n; i++)
```

```
    x++;
```

```
for (i=1; i<=n; i++)
```

```
    for (j=1; j<=n; j++)
```

```
        x++;
```

第一个for循环的时间复杂度为 $O(n)$ ，第二个for循环的时间复杂度为 $O(n^2)$ ，则整个算法的时间复杂度为 $O(n+n^2)=O(n^2)$ 。

$O(1)$ 表示基本语句的执行次数是一个常数，一般来说，只要算法中不存在循环语句，其时间复杂度就是 $O(1)$ 。其中 $O(\log_2 n)$ 、 $O(n)$ 、 $O(n \log_2 n)$ 、 $O(n^2)$ 和 $O(n^3)$ 称为多项式时间，而 $O(2^n)$ 和 $O(n!)$ 称为指数时间。计算机科学家普遍认为前者（即多项式时间复杂度的算法）是有效算法，把这类问题称为P（Polynomial, 多项式）类问题，而把后者（即指数时间复杂度的算法）称为NP（Non-Deterministic Polynomial, 非确定多项式）问题。

常见的时间复杂度进行示例说明：

(1)、 $O(1)$

```
Temp=i; i=j; j=temp;
```

以上三条单个语句的频度均为1，该程序段的执行时间是一个与问题规模 n 无关的常数。算法的时间复杂度为常数阶，记作 $T(n)=O(1)$ 。注意：如果算法的执行时间不随着问题规模 n 的增加而增长，即使算法中有上千条语句，其执行时间也不过是一个较大的常数。此类算法的时间复杂度是 $O(1)$ 。

sum=0 ; (一次)

for(i=1;i<=n;i++) (n+1次)

 for(j=1;j<=n;j++) (n²次)

 sum++ ; (n²次)

解： 因为 $\Theta(2n^2+n+1)=n^2$ (Θ 即： 去低阶项， 去掉常数项， 去掉高阶项的常参得到) ， 所以
 $T(n)=O(n^2)$;

```
for (i=1;i<n;i++)  
{  
    y=y+1;    ①  
    for (j=0;j<=(2*n);j++)  
        x++;    ②  
}
```

解： 语句1的频度是 $n-1$

语句2的频度是 $(n-1)*(2n+1)=2n^2-n-1$

$f(n)=2n^2-n-1+(n-1)=2n^2-2$;

又 $\Theta(2n^2-2)=n^2$

该程序的时间复杂度 $T(n)=O(n^2)$.

一般情况下，对步进循环语句只需考虑循环体中语句的执行次数，忽略该语句中步长加1、终值判别、控制转移等成分，当有若干个循环语句时，算法的时间复杂度是由嵌套层数最多的循环语句中最内层语句的频度 $f(n)$ 决定的。

$O(n)$

[java] view plain copy

```
a=0;
```

```
    b=1;           ①
```

```
    for (i=1;i<=n;i++) ②
```

```
    {
```

```
        s=a+b;       ③
```

```
        b=a;         ④
```

```
        a=s;         ⑤
```

```
    }
```

解： 语句1的频度： 2,

语句2的频度： n,

语句3的频度： n-1,

语句4的频度： n-1,

语句5的频度： n-1,

$T(n)=2+n+3(n-1)=4n-1=O(n)$.

$O(\log_2 n)$

[java] view plain copy

i=1; ①

while (i<=n)

i=i*2; ②

解： 语句1的频度是1,

设语句2的频度是 $f(n)$, 则: $2^{f(n)} \leq n; f(n) \leq \log_2 n$

取最大值 $f(n) = \log_2 n$,

$T(n) = O(\log_2 n)$

$O(n^3)$

[java] view plain copy

```
for(i=0;i<n;i++)  
{  
    for(j=0;j<i;j++)  
    {  
        for(k=0;k<j;k++)  
            x=x+2;  
    }  
}
```

解：当 $i=m$, $j=k$ 的时候,内层循环的次数为 k 当 $i=m$ 时, j 可以取 $0,1,...,m-1$, 所以这里最内循环共进行了 $0+1+...+m-1=(m-1)m/2$ 次所以, i 从 0 取到 n , 则循环共进行了: $0+(1-1)*1/2+...+(n-1)n/2=n(n+1)(n-1)/6$ 所以时间复杂度为 $O(n^3)$.

| 复杂度 | 标记符号 | 描述 |
|--------------------|---------------------------------|---|
| 常量 (Constant) | $O(1)$ | <p>操作的数量为常数，与输入的数据的规模无关。</p> <p>$n = 1,000,000 \rightarrow 1\text{-}2$ operations</p> |
| 对数 (Logarithmic) | $O(\log_2 n)$ | <p>操作的数量与输入数据的规模 n 的比例是 $\log_2 (n)$。</p> <p>$n = 1,000,000 \rightarrow 30$ operations</p> |
| 线性 (Linear) | $O(n)$ | <p>操作的数量与输入数据的规模 n 成正比。</p> <p>$n = 10,000 \rightarrow 5000$ operations</p> |
| 平方 (Quadratic) | $O(n^2)$ | <p>操作的数量与输入数据的规模 n 的比例为二次平方。</p> <p>$n = 500 \rightarrow 250,000$ operations</p> |
| 立方 (Cubic) | $O(n^3)$ | <p>操作的数量与输入数据的规模 n 的比例为三次方。</p> <p>$n = 200 \rightarrow 8,000,000$ operations</p> |
| 指数 (Exponential) | $O(2^n)$
$O(k^n)$
$O(n!)$ | <p>指数级的操作，快速的增长。</p> <p>$n = 20 \rightarrow 1048576$ operations</p> |

注1：快速的数学回忆， $\log_a b = y$ 其实就是 $a^y = b$ 。所以， $\log_2 4 = 2$ ，因为 $2^2 = 4$ 。同样 $\log_2 8 = 3$ ，因为 $2^3 = 8$ 。我们说， $\log_2 n$ 的增长速度要慢于 n ，因为当 $n = 8$ 时， $\log_2 n = 3$ 。

注2：通常将以 10 为底的对数叫做常用对数。为了简便， N 的常用对数 $\log_{10} N$ 简写做 $\lg N$ ，例如 $\log_{10} 5$ 记做 $\lg 5$ 。

注3：通常将以无理数 e 为底的对数叫做自然对数。为了方便， N 的自然对数 $\log_e N$ 简写做 $\ln N$ ，例如 $\log_e 3$ 记做 $\ln 3$ 。

钱进培训是哈法地区资深工程师组成的培训机构，通过各位老师的现身说法，帮助各位学员迅速掌握实战知识，为求职打下坚实的基础。电子邮件：jin.qian.canada@gmail.com 钱老师报名、答疑微信号：qianjincanada，或扫描以下二维码添加：



钱老师 
加拿大



扫一扫上面的二维码图案，加我微信

 钱进老师



 钱进老师