# 计算机网络

## 概述

### 主机之间的通信方式

* 客户-服务器（C/S）：客户是服务的请求方，服务器是服务的提供方。
* 对等（P2P）：不区分客户和服务器

### 电路交换和分组交换

#### 电路交换

电路交换用于电话通信系统，两个用户要通信之前建立一条专用的物理链路，并且在整个通信过程之中始终占用该链路，由于通信的过程中不可能一直在使用传输线路，因此电路交换对线路的利用率很低，往往不到10%。

#### 分组交换

每个分组都有首部和尾部，包含了源地址和目的地址等控制信息，在同一个传输线路上同时传输多个分组互相不会影响，因此在同一条传输线路上允许同时传输多个分组，也就是说分组交换不需要占用传输线路。

在一个邮局通信系统中，邮局收到一份邮件之后，先存储下来，然后把相同目的地的邮件一起转发到下一个目的地，这个过程就是存储转发过程，分组交换也使用了存储转发过程。

### 时延

总时延 = 排队时延 + 处理时延 + 传输时延 + 传播时延



#### 排队时延

分组在路由器的输入队列和输出队列中排队等待的时间，取决于网络当前的通信量。

#### 处理时延

主机或路由器收到分组时进行处理所需要的时间，例如分析首部、从分组中提取数据、进行差错检验或查找适当的路由等。

#### 传输时延

主机或路由器传输数据帧所需要的时间。

其中l表示数据帧的长度，v表示传输速率。

#### 传播时延

电磁波在信道中传播所需要花费的时间，电磁波传播的速度接近光速。

其中l表示信道长度，v表示电磁波在信道上的传播速度。

### 计算机网络体系结构



#### 五层协议

* 应用层：为特定应用程序提供数据传输服务，例如 HTTP、DNS 等协议。数据单位为报文。
* 传输层：为进程提供通用数据传输服务。由于应用层协议很多，定义通用的传输层协议就可以支持不断增多的应用层协议。运输层包括两种协议：传输控制协议 TCP，提供面向连接、可靠的数据传输服务，数据单位为报文段；用户数据报协议 UDP，提供无连接、尽最大努力的数据传输服务，数据单位为用户数据报。TCP 主要提供完整性服务，UDP 主要提供及时性服务。
* 网络层：为主机提供数据传输服务。而传输层协议是为主机中的进程提供数据传输服务。网络层把传输层传递下来的报文段或者用户数据报封装成分组。
* 数据链路层：网络层针对的还是主机之间的数据传输服务，而主机之间可以有很多链路，链路层协议就是为同一链路的主机提供数据传输服务。数据链路层把网络层传下来的分组封装成帧。
* 物理层：考虑的是怎样在传输媒体上传输数据比特流，而不是指具体的传输媒体。物理层的作用是尽可能屏蔽传输媒体和通信手段的差异，使数据链路层感觉不到这些差异。

#### OSI

其中表示层和会话层用途如下：

* 表示层：数据压缩、加密以及数据描述，这使得应用程序不必关心在各台主机中数据内部格式不同的问题。
* 会话层：建立及管理会话。

五层协议没有表示层和会话层，而是将这些功能留给应用程序开发者处理。

#### TCP/IP

它只有4层，相当于5层协议中的数据链路层和物理层合并为网络接口层。

TCP/IP 体系结构不严格遵循 OSI 分层概念，应用层可能会直接使用 IP 层或者网络接口层。



#### 数据在各层之间的传递过程

在向下的过程中，需要添加下层协议所需要的首部或者尾部，而在向上的过程中不断拆开首部和尾部。

路由器只有下面三层协议，因为路由器位于网络核心中，不需要为进程或者应用程序提供服务，因此也就不需要传输层和应用层。

## 物理层

### 概览

String被声明为final，因此它不可被继承。

在Java8中，String内部使用char数组储存数据。

|  |
| --- |
| public final class String  implements java.io.Serializable, Comparable<String>, CharSequence {  /\*\* The value is used for character storage. \*/  private final char value[];  } |

在Java9之后，String类的实现改用byte数组存储字符串，同时使用coder来标识了使用哪种编码。

|  |
| --- |
| public final class String  implements java.io.Serializable, Comparable<String>, CharSequence {  /\*\* The value is used for character storage. \*/  private final byte[] value;  /\*\* The identifier of the encoding used to encode the bytes in {@code value}. \*/  private final byte coder;  } |

value数组被声明为final，这意味着value数组初始化之后就不能再引用其它数组。并且String内部没有改变value数组的方法，因此可以保证String不可变。

### 不可变的好处

#### 可以缓存hash值

因为String的hash值经常被用到，例如 String 用做 HashMap 的 key。不可变的特性可以使得 hash 值也不可变，因此只需要进行一次计算。

#### String Pool的需要

如果一个 String 对象已经被创建过了，那么就会从 String Pool 中取得引用。只有 String 是不可变的，才可能使用 String Pool。



#### 安全性

String 经常作为参数，String 不可变性可以保证参数不可变。例如在作为网络连接参数的情况下如果 String 是可变的，那么在网络连接过程中，String 被改变，改变 String 对象的那一方以为现在连接的是其它主机，而实际情况却不一定是。

#### 线程安全

String 不可变性天生具备线程安全，可以在多个线程中安全地使用。

### String、StringBuilder、StringBuffer

#### 可变性

* String不可变
* StringBuilder和StringBuffer可变

#### 线程安全

* String不可变，所以String是线程安全的
* StringBuilder不是线程安全的
* StringBuffer是线程安全的，内部使用synchronized进行同步。

### String Pool

字符串常量池（String Pool）保存着所有字符串字面量（literal strings），这些字面量在编译时期就确定。不仅如此，还可以使用 String 的 intern() 方法在运行过程中将字符串添加到 String Pool 中。

当一个字符串调用 intern() 方法时，如果 String Pool 中已经存在一个字符串和该字符串值相等（使用 equals() 方法进行确定），那么就会返回 String Pool 中字符串的引用；否则，就会在 String Pool 中添加一个新的字符串，并返回这个新字符串的引用。

下面示例中，s1 和 s2 采用 new String() 的方式新建了两个不同字符串，而 s3 和 s4 是通过 s1.intern() 方法取得一个字符串引用。intern() 首先把 s1 引用的字符串放到 String Pool 中，然后返回这个字符串引用。因此 s3 和 s4 引用的是同一个字符串。

|  |
| --- |
| String s1 = new String("aaa");  String s2 = new String("aaa");  System.out.println(s1 == s2); // false  String s3 = s1.intern();  String s4 = s1.intern();  System.out.println(s3 == s4); // true |

如果是采用 "bbb" 这种字面量的形式创建字符串，会自动地将字符串放入 String Pool 中。

|  |
| --- |
| String s5 = "bbb";  String s6 = "bbb";  System.out.println(s5 == s6); // true |

在 Java 7 之前，String Pool 被放在运行时常量池中，它属于永久代。而在 Java 7，String Pool 被移到堆中。这是因为永久代的空间有限，在大量使用字符串的场景下会导致 OutOfMemoryError 错误。

### new String(“abc”)

使用这种方式一共会创建两个字符串对象（前提是 String Pool 中还没有 "abc" 字符串对象）。

* "abc" 属于字符串字面量，因此编译时期会在 String Pool 中创建一个字符串对象，指向这个 "abc" 字符串字面量；
* 而使用 new 的方式会在堆中创建一个字符串对象。

创建一个测试类，其 main 方法中使用这种方式来创建字符串对象。

|  |
| --- |
| public class NewStringTest {  public static void main(String[] args) {  String s = new String("abc");  }  } |

使用javap –verbose进行反编译，得到以下内容：

|  |
| --- |
| // ...  Constant pool:  // ...  #2 = Class #18 // java/lang/String  #3 = String #19 // abc  // ...  #18 = Utf8 java/lang/String  #19 = Utf8 abc  // ...  public static void main(java.lang.String[]);  descriptor: ([Ljava/lang/String;)V  flags: ACC\_PUBLIC, ACC\_STATIC  Code:  stack=3, locals=2, args\_size=1  0: new #2 // class java/lang/String  3: dup  4: ldc #3 // String abc  6: invokespecial #4 // Method java/lang/String."<init>":(Ljava/lang/String;)V  9: astore\_1  // ... |

在Constant Pool中，#19存储着字符串字面量“abc”，#3是String Pool的字符串对象，它指向#19这个字符串字面量，在main方法中，0行使用new #2在堆中创建一个字符串对象，并且使用ldc #3将String Pool中的字符串对象作为String构造函数的参数。

以下是String构造函数的源码，可以看到，在将一个字符串对象作为另一个字符串对象的构造函数参数时，并不会完全复制value数组的内容，而是都会指向同一个value数组。

|  |
| --- |
| public String(String original) {  this.value = original.value;  this.hash = original.hash;  } |

## 链路层

### 参数传递

Java的参数传递是以值传递的形式传入方法中，而不是引用传递。

以下的代码Dog dog的dog是一个指针，存储的是对象的地址。在将一个参数传入一个方法中时，本质上是将对象的地址以值的方式传递到形参中。因此在方法中使指针引用其他对象，那么两个指针此时指向的是完全不同的两个对象，在一方改变其所指向的对象的内容时对另一方没有影响。

|  |
| --- |
| public class Dog {  String name;  Dog(String name) {  this.name = name;  }  String getName() {  return this.name;  }  void setName(String name) {  this.name = name;  }  String getObjectAddress() {  return super.toString();  }  } |

|  |
| --- |
| public class PassByValueExample {  public static void main(String[] args) {  Dog dog = new Dog("A");  System.out.println(dog.getObjectAddress()); // Dog@4554617c  func(dog);  System.out.println(dog.getObjectAddress()); // Dog@4554617c  System.out.println(dog.getName()); // A  }  private static void func(Dog dog) {  System.out.println(dog.getObjectAddress()); // Dog@4554617c  dog = new Dog("B");  System.out.println(dog.getObjectAddress()); // Dog@74a14482  System.out.println(dog.getName()); // B  }  } |

如果在方法中改变对象的字段值会改变原对象该字段值，因为改变的是同一个地址指向的内容。

|  |
| --- |
| class PassByValueExample {  public static void main(String[] args) {  Dog dog = new Dog("A");  func(dog);  System.out.println(dog.getName()); // B  }  private static void func(Dog dog) {  dog.setName("B");  }  } |

### float与double

Java不能隐式地执行向下转型，因为这会使精度降低。

1.1字面量属于 double 类型，不能直接将 1.1 直接赋值给 float 变量，因为这是向下转型。

|  |
| --- |
| // float f = 1.1; |

1.1f 字面量才是 float类型。

|  |
| --- |
| float f = 1.1f; |

### 隐式类型转换

因为字面量 1 是 int 类型，它比 short 类型精度要高，因此不能隐式地将 int 类型下转型为 short 类型。

|  |
| --- |
| short s1 = 1;  // s1 = s1 + 1; |

但是使用 += 或者 ++ 运算符可以执行隐式类型转换。

|  |
| --- |
| s1 += 1;  // s1++; |

上面的语句相当于将 s1 + 1 的计算结果进行了向下转型：

|  |
| --- |
| s1 = (short) (s1 + 1); |

### switch

从Java7开始，可以在switch条件判断语句中使用String对象。

|  |
| --- |
| String s = "a";  switch (s) {  case "a":  System.out.println("aaa");  break;  case "b":  System.out.println("bbb");  break;  } |

switch 不支持 long，是因为 switch 的设计初衷是对那些只有少数的几个值进行等值判断，如果值过于复杂，那么还是用 if 比较合适。

switch判断语句中支持的类型：char, byte, short, int, Character, Byte, Short, Integer, String, or an enum。

## 网络层

### 访问权限

Java中有3个访问权限修饰符，private、protected以及public。如果不加访问修饰符，表示包级可见。

可以对类或类中的成员（字段以及方法）加上访问修饰符。

* 类可见表示其它类可以用这个类创建实例对象。
* 成员可见表示其它类可以用这个类的实例对象访问到该成员；

protected 用于修饰成员，表示在继承体系中成员对于子类可见，但是这个访问修饰符对于类没有意义。

设计良好的模块会隐藏所有的实现细节，把它的 API 与它的实现清晰地隔离开来。模块之间只通过它们的 API 进行通信，一个模块不需要知道其他模块的内部工作情况，这个概念被称为信息隐藏或封装。因此访问权限应当尽可能地使每个类或者成员不被外界访问。

如果子类的方法重写了父类的方法，那么子类中该方法的访问级别不允许低于父类的访问级别。这是为了确保可以使用父类实例的地方都可以使用子类实例，也就是确保满足里氏替换原则。

字段决不能是公有的，因为这么做的话就失去了对这个字段修改行为的控制，客户端可以对其随意修改。例如下面的例子中，AccessExample 拥有 id 公有字段，如果在某个时刻，我们想要使用 int 存储 id 字段，那么就需要修改所有的客户端代码。

|  |
| --- |
| public class AccessExample {  public String id;  } |

可以使用公有的 getter 和 setter 方法来替换公有字段，这样的话就可以控制对字段的修改行为。

|  |
| --- |
| public class AccessExample {  private int id;  public String getId() {  return id + "";  }  public void setId(String id) {  this.id = Integer.valueOf(id);  }  } |

但是也有例外，如果是包级私有的类或者私有的嵌套类，那么直接暴露成员不会有特别大的影响。

|  |
| --- |
| public class AccessWithInnerClassExample {  private class InnerClass {  int x;  }  private InnerClass innerClass;  public AccessWithInnerClassExample() {  innerClass = new InnerClass();  }  public int getValue() {  return innerClass.x; // 直接访问  }  } |

### 抽象类与接口

#### 抽象类

抽象类和抽象方法都使用 abstract 关键字进行声明。抽象类一般会包含抽象方法，抽象方法一定位于抽象类中。

抽象类和普通类最大的区别是，抽象类不能被实例化，需要继承抽象类才能实例化其子类。

|  |
| --- |
| public abstract class AbstractClassExample {  protected int x;  private int y;  public abstract void func1();  public void func2() {  System.out.println("func2");  }  } |

|  |
| --- |
| public class AbstractExtendClassExample extends AbstractClassExample {  @Override  public void func1() {  System.out.println("func1");  }  } |

|  |
| --- |
| // AbstractClassExample ac1 = new AbstractClassExample(); // 'AbstractClassExample' is abstract; cannot be instantiated  AbstractClassExample ac2 = new AbstractExtendClassExample();  ac2.func1(); |

#### 接口

接口是抽象类的延伸，在 Java 8 之前，它可以看成是一个完全抽象的类，也就是说它不能有任何的方法实现。

从 Java 8 开始，接口也可以拥有默认的方法实现，这是因为不支持默认方法的接口的维护成本太高了。在 Java 8 之前，如果一个接口想要添加新的方法，那么要修改所有实现了该接口的类。

接口的成员（字段 + 方法）默认都是 public 的，并且不允许定义为 private 或者 protected。

接口的字段默认都是 static 和 final 的。

|  |
| --- |
| public interface InterfaceExample {  void func1();  default void func2(){  System.out.println("func2");  }  int x = 123;  // int y; // Variable 'y' might not have been initialized  public int z = 0; // Modifier 'public' is redundant for interface fields  // private int k = 0; // Modifier 'private' not allowed here  // protected int l = 0; // Modifier 'protected' not allowed here  // private void fun3(); // Modifier 'private' not allowed here  } |

|  |
| --- |
| public class InterfaceImplementExample implements InterfaceExample {  @Override  public void func1() {  System.out.println("func1");  }  } |

|  |
| --- |
| // InterfaceExample ie1 = new InterfaceExample(); // 'InterfaceExample' is abstract; cannot be instantiated  InterfaceExample ie2 = new InterfaceImplementExample();  ie2.func1();  System.out.println(InterfaceExample.x); |

#### 比较

* 从设计层面上看，抽象类提供了一种 IS-A 关系，那么就必须满足里式替换原则，即子类对象必须能够替换掉所有父类对象。而接口更像是一种 LIKE-A 关系，它只是提供一种方法实现契约，并不要求接口和实现接口的类具有 IS-A 关系。
* 从使用上来看，一个类可以实现多个接口，但是不能继承多个抽象类。
* 接口的字段只能是 static 和 final 类型的，而抽象类的字段没有这种限制。
* 接口的成员只能是 public 的，而抽象类的成员可以有多种访问权限。

#### 使用选择

**使用接口：**

* 需要让不相关的类都实现一个方法，例如不相关的类都可以实现 Compareable 接口中的 compareTo() 方法；
* 需要使用多重继承。

**使用抽象类：**

* 需要在几个相关的类中共享代码。
* 需要能控制继承来的成员的访问权限，而不是都为 public。
* 需要继承非静态和非常量字段。

在很多情况下，接口优先于抽象类。因为接口没有抽象类严格的类层次结构要求，可以灵活地为一个类添加行为。并且从 Java 8 开始，接口也可以有默认的方法实现，使得修改接口的成本也变的很低。

### super

* 访问父类的构造函数：可以使用 super() 函数访问父类的构造函数，从而委托父类完成一些初始化的工作。
* 访问父类的成员：如果子类重写了父类的某个方法，可以通过使用 super 关键字来引用父类的方法实现。

|  |
| --- |
| public class SuperExample {  protected int x;  protected int y;  public SuperExample(int x, int y) {  this.x = x;  this.y = y;  }  public void func() {  System.out.println("SuperExample.func()");  }  } |

|  |
| --- |
| public class SuperExtendExample extends SuperExample {  private int z;  public SuperExtendExample(int x, int y, int z) {  super(x, y);  this.z = z;  }  @Override  public void func() {  super.func();  System.out.println("SuperExtendExample.func()");  }  } |

|  |
| --- |
| SuperExample e = new SuperExtendExample(1, 2, 3);  e.func(); |

|  |
| --- |
| SuperExample.func()  SuperExtendExample.func() |

### 重写与重载

#### 重写（Override）

存在于继承体系中，指子类实现了一个与父类在方法声明上完全相同的一个方法。

为了满足里式替换原则，重写有有以下两个限制：

使用 @Override 注解，可以让编译器帮忙检查是否满足下面的三个限制条件。

* 子类方法访问权限为 public，大于父类的 protected。
* 子类的返回类型为 ArrayList，是父类返回类型 List 的子类。
* 子类抛出的异常类型为 Exception，是父类抛出异常 Throwable 的子类。
* 子类重写方法使用 @Override 注解，从而让编译器自动检查是否满足限制条件。

|  |
| --- |
| class SuperClass {  protected List<Integer> func() throws Throwable {  return new ArrayList<>();  }  }  class SubClass extends SuperClass {  @Override  public ArrayList<Integer> func() throws Exception {  return new ArrayList<>();  }  } |

#### 重载（Overload）

存在于同一个类中，指一个方法与已经存在的方法名称上相同，但是参数类型、个数、顺序至少有一个不同。

应该注意的是，返回值不同，其它都相同不算是重载。

#### 实例

|  |
| --- |
| 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 Test {  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(a1.show(b)); // A and A  System.out.println(a1.show(c)); // A and A  System.out.println(a1.show(d)); // A and D  System.out.println(a2.show(b)); // B and A  System.out.println(a2.show(c)); // B and A  System.out.println(a2.show(d)); // A and D  System.out.println(b.show(b)); // B and B  System.out.println(b.show(c)); // B and B  System.out.println(b.show(d)); // A and D  }  } |

涉及到重写时，方法调用的优先级为：

* this.show(O)
* super.show(O)
* this.show((super)O)
* super.show((super)O)

## 传输层

### 概览

|  |
| --- |
| public native int hashCode()  public boolean equals(Object obj)  protected native Object clone() throws CloneNotSupportedException  public String toString()  public final native Class<?> getClass()  protected void finalize() throws Throwable {}  public final native void notify()  public final native void notifyAll()  public final native void wait(long timeout) throws InterruptedException  public final void wait(long timeout, int nanos) throws InterruptedException  public final void wait() throws InterruptedException |

### equals()

#### 等价关系

* 自反性

|  |
| --- |
| x.equals(x); // true |

* 对称性

|  |
| --- |
| x.equals(y) == y.equals(x); // true |

* 传递性

|  |
| --- |
| if (x.equals(y) && y.equals(z))  x.equals(z); // true; |

* 一致性

多次调用equals()方法结果不变

|  |
| --- |
| x.equals(y) == x.equals(y); // true |

* 与null比较

对任何不是 null 的对象 x 调用 x.equals(null) 结果都为 false

|  |
| --- |
| x.equals(null); // false; |

#### 等价与相等

* 对于基本类型，==判断两个值是否相等，基本类型没有equals()方法。
* 对于引用类型，==判断两个变量是否引用同一个对象，而equals()判断引用的对象是否相等价。

|  |
| --- |
| Integer x = new Integer(1);  Integer y = new Integer(1);  System.out.println(x.equals(y)); // true  System.out.println(x == y); // false |

#### 实现

* 检查是否为同一个对象的引用，如果是直接返回 true；
* 检查是否是同一个类型，如果不是，直接返回 false；
* 将 Object 对象进行转型；
* 判断每个关键域是否相等。

|  |
| --- |
| public class EqualExample {  private int x;  private int y;  private int z;  public EqualExample(int x, int y, int z) {  this.x = x;  this.y = y;  this.z = z;  }  @Override  public boolean equals(Object o) {  if (this == o) return true;  if (o == null || getClass() != o.getClass()) return false;  EqualExample that = (EqualExample) o;  if (x != that.x) return false;  if (y != that.y) return false;  return z == that.z;  }  } |

### hashCode()

hashCode() 返回散列值，而 equals() 是用来判断两个对象是否等价。等价的两个对象散列值一定相同，但是散列值相同的两个对象不一定等价。

在覆盖 equals() 方法时应当总是覆盖 hashCode() 方法，保证等价的两个对象散列值也相等。

下面的代码中，新建了两个等价的对象，并将它们添加到 HashSet 中。我们希望将这两个对象当成一样的，只在集合中添加一个对象，但是因为 EqualExample 没有实现 hasCode() 方法，因此这两个对象的散列值是不同的，最终导致集合添加了两个等价的对象。

|  |
| --- |
| EqualExample e1 = new EqualExample(1, 1, 1);  EqualExample e2 = new EqualExample(1, 1, 1);  System.out.println(e1.equals(e2)); // true  HashSet<EqualExample> set = new HashSet<>();  set.add(e1);  set.add(e2);  System.out.println(set.size()); // 2 |

理想的散列函数应当具有均匀性，即不相等的对象应当均匀分不到所有可能的散列值上。这就要求了散列函数要把所有域的值都考虑进来。可以将每个域都当成R进制的某一位，然后组成一个R进制的整数。R 一般取 31，因为它是一个奇素数，如果是偶数的话，当出现乘法溢出，信息就会丢失，因为与 2 相乘相当于向左移一位。

一个数与31相乘可以转换成移位和减法：31\*x == (x<<5)-x，编译器会自动进行这个优化。

|  |
| --- |
| @Override  public int hashCode() {  int result = 17;  result = 31 \* result + x;  result = 31 \* result + y;  result = 31 \* result + z;  return result;  } |

### toString()

默认返回 ToStringExample@4554617c 这种形式，其中 @ 后面的数值为散列码的无符号十六进制表示。

|  |
| --- |
| public class ToStringExample {  private int number;  public ToStringExample(int number) {  this.number = number;  }  } |

|  |
| --- |
| ToStringExample example = new ToStringExample(123);  System.out.println(example.toString()); |

|  |
| --- |
| ToStringExample@4554617c |

### clone()

#### cloneable

clone()是Object的protected方法，它不是public，一个类不显示去重写clone()，其他类就不能直接去调用该类实例的clone()方法。

|  |
| --- |
| public class CloneExample {  private int a;  private int b;  } |

|  |
| --- |
| CloneExample e1 = new CloneExample();  // CloneExample e2 = e1.clone(); // 'clone()' has protected access in 'java.lang.Object' |

重写clone()得到以下的实现：

|  |
| --- |
| public class CloneExample {  private int a;  private int b;  @Override  public CloneExample clone() throws CloneNotSupportedException {  return (CloneExample)super.clone();  }  } |

|  |
| --- |
| CloneExample e1 = new CloneExample();  try {  CloneExample e2 = e1.clone();  } catch (CloneNotSupportedException e) {  e.printStackTrace();  } |

|  |
| --- |
| java.lang.CloneNotSupportedException: CloneExample |

以上抛出了 CloneNotSupportedException，这是因为 CloneExample 没有实现 Cloneable 接口。

应该注意的是，clone() 方法并不是 Cloneable 接口的方法，而是 Object 的一个 protected 方法。Cloneable 接口只是规定，如果一个类没有实现 Cloneable 接口又调用了 clone() 方法，就会抛出 CloneNotSupportedException。

|  |
| --- |
| public class CloneExample implements Cloneable {  private int a;  private int b;  @Override  public Object clone() throws CloneNotSupportedException {  return super.clone();  }  } |

#### 浅拷贝

拷贝对象和原始对象的引用类型引用同一个对象。

|  |
| --- |
| public class ShallowCloneExample implements Cloneable {  private int[] arr;  public ShallowCloneExample() {  arr = new int[10];  for (int i = 0; i < arr.length; i++) {  arr[i] = i;  }  }  public void set(int index, int value) {  arr[index] = value;  }  public int get(int index) {  return arr[index];  }  @Override  protected ShallowCloneExample clone() throws CloneNotSupportedException {  return (ShallowCloneExample) super.clone();  }  } |

|  |
| --- |
| ShallowCloneExample e1 = new ShallowCloneExample();  ShallowCloneExample e2 = null;  try {  e2 = e1.clone();  } catch (CloneNotSupportedException e) {  e.printStackTrace();  }  e1.set(2, 222);  System.out.println(e2.get(2)); // 222 |

#### 深拷贝

拷贝对象和原始对象的引用类型引用不同对象。

|  |
| --- |
| public class DeepCloneExample implements Cloneable {  private int[] arr;  public DeepCloneExample() {  arr = new int[10];  for (int i = 0; i < arr.length; i++) {  arr[i] = i;  }  }  public void set(int index, int value) {  arr[index] = value;  }  public int get(int index) {  return arr[index];  }  @Override  protected DeepCloneExample clone() throws CloneNotSupportedException {  DeepCloneExample result = (DeepCloneExample) super.clone();  result.arr = new int[arr.length];  for (int i = 0; i < arr.length; i++) {  result.arr[i] = arr[i];  }  return result;  }  } |

|  |
| --- |
| DeepCloneExample e1 = new DeepCloneExample();  DeepCloneExample e2 = null;  try {  e2 = e1.clone();  } catch (CloneNotSupportedException e) {  e.printStackTrace();  }  e1.set(2, 222);  System.out.println(e2.get(2)); // 2 |

#### clone()的替代方案

使用 clone() 方法来拷贝一个对象即复杂又有风险，它会抛出异常，并且还需要类型转换。Effective Java 书上讲到，最好不要去使用 clone()，可以使用拷贝构造函数或者拷贝工厂来拷贝一个对象。

|  |
| --- |
| public class CloneConstructorExample {  private int[] arr;  public CloneConstructorExample() {  arr = new int[10];  for (int i = 0; i < arr.length; i++) {  arr[i] = i;  }  }  public CloneConstructorExample(CloneConstructorExample original) {  arr = new int[original.arr.length];  for (int i = 0; i < original.arr.length; i++) {  arr[i] = original.arr[i];  }  }  public void set(int index, int value) {  arr[index] = value;  }  public int get(int index) {  return arr[index];  }  } |

|  |
| --- |
| CloneConstructorExample e1 = new CloneConstructorExample();  CloneConstructorExample e2 = new CloneConstructorExample(e1);  e1.set(2, 222);  System.out.println(e2.get(2)); // 2 |

## 应用层

### final

#### 数据

声明数据为常量，可以是编译时常量，也可以是在运行时被初始化后不能被改变的常量。

* 对于基本类型，final使数值不变；
* 对于引用类型，final使引用不变，也就是不能引用其他对象，但是被引用的对象本身是可以修改的。

|  |
| --- |
| final int x = 1;  // x = 2; // cannot assign value to final variable 'x'  final A y = new A();  y.a = 1; |

#### 方法

声明方法不能被子类重写。

private()方法隐式的被指定为final，如果在子类中定义的方法和基类中的一个private方法签名相同，此时子方类的方法不是重写基类方法，而是在子类中定义了一个新方法。

#### 类

声明类不允许被继承。

### static

#### 静态变量

* 静态变量：又称为类变量，也就是说这个变量属于类的，类所有的实例都共享静态变量，可以直接通过类名来访问它。静态变量在内存中只存在一份。
* 实例变量：每创建一个实例就会产生一个实例变量，它与该实例同生共死。

|  |
| --- |
| public class A {  private int x; // 实例变量  private static int y; // 静态变量  public static void main(String[] args) {  // int x = A.x; // Non-static field 'x' cannot be referenced from a static context  A a = new A();  int x = a.x;  int y = A.y;  }  } |

#### 静态方法

静态方法在类加载的时候就存在了，它不依赖于任何实例。所以静态方法必须有实现，也就是说它不能是抽象方法。

|  |
| --- |
| public abstract class A {  public static void func1(){  }  // public abstract static void func2(); // Illegal combination of modifiers: 'abstract' and 'static'  } |

只能访问所属类的静态字段和静态方法，方法中不能有 this 和 super 关键字。

|  |
| --- |
| public class A {  private static int x;  private int y;  public static void func1(){  int a = x;  // int b = y; // Non-static field 'y' cannot be referenced from a static context  // int b = this.y; // 'A.this' cannot be referenced from a static context  }  } |

#### 静态语句块

静态语句块在类初始化时运行一次。

|  |
| --- |
| public class A {  static {  System.out.println("123");  }  public static void main(String[] args) {  A a1 = new A();  A a2 = new A();  }  } |

|  |
| --- |
| 123 |

#### 静态内部类

非静态内部类依赖于外部类的实例，而静态内部类不需要。

|  |
| --- |
| public class OuterClass {  class InnerClass {  }  static class StaticInnerClass {  }  public static void main(String[] args) {  // InnerClass innerClass = new InnerClass(); // 'OuterClass.this' cannot be referenced from a static context  OuterClass outerClass = new OuterClass();  InnerClass innerClass = outerClass.new InnerClass();  StaticInnerClass staticInnerClass = new StaticInnerClass();  }  } |

静态内部类不能访问外部类的非静态的变量和方法。

#### 静态导包

在使用静态变量和方法时不用再指明 ClassName，从而简化代码，但可读性大大降低。

|  |
| --- |
| import static com.xxx.ClassName.\* |

#### 初始化顺序

静态变量和静态语句块优先于实例变量和普通语句块，静态变量和静态语句块的初始化顺序取决于它们在代码中的顺序。

|  |
| --- |
| public static String staticField = "静态变量";  static {  System.out.println("静态语句块");  }  public String field = "实例变量";  {  System.out.println("普通语句块");  }  public InitialOrderTest() {  System.out.println("构造函数");  } |

存在继承的情况下，初始化顺序为：

* 父类（静态变量、静态语句块）
* 子类（静态变量、静态语句块）
* 父类（实例变量、普通语句块）
* 父类（构造函数）
* 子类（实例变量、普通语句块）
* 子类（构造函数）