1、Java 中的对象都以引用来进行操作，与 C++不同，声明引用时，不一定需要初始化（绑定到一个对象）。但是，在绑定到对象之前，该引用不能被操作。

2、C++中，内层名字会隐藏外层的名字，Java 中不允许这样

3、用 New 创建的对象可以存活在作用域之外，但会被 Java 的垃圾回收器监视。（例如：System.out.println(new Data())；创建 Data 的对象后，除了这句再也没有地方用到过这个对象，那么垃圾回收器就会发现他，在适当的时候将其回收）

4、C++中类定义的数据成员以及成员函数在 Java 中被称为字段和方法

5、与 C++不同，字段若是基本类型，那么即使没有初始化，Java 也会确保它获得一个默认值。但是若是局部变量，那么未初始化的基本类型的值可能是任意的，因此 C++将会给予警告，而 Java 直接报错

6、返回类型为 void，用 return；来退出方法

7、命名体系：C++引入了命名空间来避免名字的混淆，Java 通过使用 Internet 域名来避免混淆

8、Java 中所有代码必须写在类里，用 import 指示编译器导入一个包

import java.util.ArrayList;

import java.util.*;　一次导入 util 目录下的所有类

9、static 关键字：意味着这个域或方法不会与包含它的那个类的任何对象实例关联在一起。代表只是作为整个类而不是某个特定对象而存在的。可以通过类名或者对象名加点号来使用，使用类名是首选方式，因为这与他的意义相符合。静态方法也是如此（static 说明符加在方法返回类型之前）

9.1 static 方法：在不创建任何对象的前提下就可以调用它。main()方法是运行一个应用时的入口点

9.2 静态方法中不能使用非静态字段（理由很简单，静态方法不依赖于类的对象而存在，那么静态方法自然不能调用某个依赖特定对象的非静态字段）,但是静态字段能在非静态方法中使用

9.3 当前类的静态方法中不能直接调用当前类的非静态方法，必须创建当前类的对象再调用非静态方法（main 函数中不能直接使用该类的非静态方法，必须创建对象）

10、Java 中没有指针

11、一个 public 类中的 main 函数，实际上就是该类的一个 public 静态方法，因此在该方法中，该类的私有成员以及私有方法都是可见的！！！

12、对基本数据类型的赋值与对对象进行赋值的差异：对一个对象进行赋值操作时，作用的是对对象的引用

13、别名现象：用两个对象的引用进行赋值操作时，赋值左边的引用会转为指向赋值右边的对象，从而左边引用的原对象将会被垃圾回收器监控。

14、== ！ = 对于基本类型，若值相同，则相同；若是比较两个拥有相同值的对象的引用，那么情况就会不同，因为即使对象的值相同，引用也是不同的，因此是不同的。若要比较两个对象的值是否相同，用方法：equals　v1.equals(v2) 返回 boolean 类型的值（equals 的默认行为是比较引用）

15、与 C++中不同，Java 中不可将一个非布尔值当做布尔值在逻辑表达式中使用；在应该使用 String 的地方使用了布尔值，那么布尔值会自动转换成适当的文本形式。

16、短路，逻辑与中：左边 false 就不计算右边　　逻辑或中：左边 true 就不计算右边

17、直接常量：

前缀：0X(0x):十六进制

1　　：八进制

后缀：L　:Long

D　:double

F　:float

18、如果表达式以一个字符串起头，那么后续所有操作数都必须是字符串型

Int x=0,y=1,z=2;

String s=”x,y,z”;

那么 s+x+y+z=”x,y,z012”;

s+(x+y+z)=”x,y,z3”

19、类型转换

自动类型转换，如 int i=3.14；编译器会自动将浮点数 3.14 转换为整数
强制类型转换　　long a=(long) 200;

窄化转换：将能容纳更多信息的数据类型转换成无法容纳这么多信息的类型
扩展转换：新的数据类型必然包含元数据类型的所有信息，不会造成信息的丢失

19.1 布尔类型不允许进行任何类型的转换处理
19.2 类数据类型不允许进行类型转换，除非采用特殊的方法

20、Java 中数据类型在所有机器上的大小都是相同的

21、**Java 不允许将一个数字作为布尔值使用**（C++中，非零为真，0 为假）
也就是说 if (a) 不能使用，但可以用 if (a!=0) 来代替

22、逗号操作符：Java 中唯一用到的地方就是 for 循环的循环控制表达式（逗号分隔符与逗号操作符是两个概念，逗号分隔符用来分隔函数的不同参数）**此外，for 循环中的初始化部分可以有任意数量的同一类型的定义**

23、**与 C++不同，Java 中前置递增（减）后置递增（减）返回的都是右值**

24、Java 中也有范围 for 语句，但是不能用 auto 来为声明子元素的类型

25、**System.out.println()会换行　　System.out.print()不会换行**

26、break Label1:跳出 Label1 所指的循环，Label1 与循环语句之内不可以有任何语句
Continue Label1:终止当前 Label1 所指循环的迭代，并继续下一次迭代

27、switch(integral_selector) 必须对整数因子进行选择，也就是整数或字符

28、**方法命名风格：首字母小写**

29、String
遍历 String：用下标[]即可
通过 s.charAt(i)将第 i 个元素转化为 char 类型
s.split("")分解成单个字符（还是 String 的类型）的集合，可以作为 *Arrays*.<String>asList 的输入
s.toCharArray()分解成单个字符（char 类型）构成的数组

30、**Java 中参数传递的问题**
30.1 若是基本类型，参数会得到原值得一个拷贝，改变参数不会导致原值的改变
30.2 **若是类类型，传递的参数实际上是一个引用，参数引用会和原引用指向同一个对象，改变引用会导致对象的改变，有一个例外 String，它是 final 的，任何改变它值得操作都会生成一个新的 String 来代替原来的 String，从而导致参数引用在改变后不再指向原来的对象**

31、char 的四则运算
首先会将 char 字符转型为 int，其大小为所代表的 ASCII 码
‘1’,’2’,...’9’转为 1,2,3...9-->**’i’-‘0’**

32、Java 传入的参数若为引用，则不能改变该引用所引用的对象？int[] 和 ArrayList<Integer>不可以

33、计算程序运行时间

```
long start=System.nanoTime();
    //do something
    double duration=System.nanoTime()-start;
    System.out.format("%.2f\n",duration/1.0e9);
```

34、让系统停止 n 毫秒
Thread.sleep(n);

35、异步调用是通过使用单独的线程执行的。原始线程启动异步调用，异步调用使用另一个线程执行请求，而与此同时原始的线程继续处理。

同步调用则在继续之前必须等待响应或返回值。如果不允许调用继续即无响应或返回值，就说调用被阻塞了,不能继续执行。

Chapter 1.　对象导论

Chapter 2. 一切都是对象

Chapter 3.　操作符

Chapter 4.　控制执行流程

Chapter 5.　初始化与清理

C++引入了构造器的概念，这是一个在创建对象时被自动调用的特殊方法。Java 也采用了构造器，并额外提供了"垃圾回收器"：对于不再利用的内存资源，垃圾回收期能自动将其释放。

5. 1. 用构造器确保初始化

1、 构造器：采用 C++中的解决方法，构造器采用与类相同的名称，没有返回类型；调用构造器是编译器的责任；

2、构造器类型

- 默认构造器：不接受任何参数（无参构造器）
  - **当类中没有定义构造器时，编译器会创建一个默认构造器**
  - **当类中定义了任何构造器时，编译器不会创建默认构造器**
- 接受参数的构造器：

5. 2. 方法重载

1.2 初始化与创建：从概念上讲，初始化与创建是独立的；但是在代码中确没有显式调用构造器的语句，在创建时由编译器调用构造器来进行初始化

2、this

C++中，this 是指向该类型对象的常量指针（可以指向该类型的常量和非常量）

Java 中，this 是该类型对象的引用

2.1 构造函数中调用构造函数

this 后跟参数列表：对符合此参数列表的构造器明确调用

必须将被调用的构造器的定义置于调用它的构造器之前（C++好像类内的编译没有明确顺序）

3、清理：终结处理和垃圾回收

3.1 C++中，对象一定会被销毁（如果程序中没有缺陷的话）

Java 中，对象确并非总是被垃圾回收

也　　就　　是　　说　　：

1.对象可能不被垃圾回收

2.垃圾回收并不等于“析构”

3.2 Java 并没有提供与“析构函数”相似的概念（**即使存储空间没有消耗殆尽，一旦调用了析构函数，那么该对象所占的资源就会被释放**），要做类似的清理工作，必须手动创建一个执行清理工作的普通方法。**也就是说，只要程序没有濒临存储空间用完的那一刻，对象占用的空间就总也得不到释放，随着程序退出，资源会全部交还给操作系统（回收垃圾本身也有开销，因此这么做是恰当的）**

3.3 finalize()的用途:Java 允许在类中定义一个名为 finalize()的方法。**工作原理：一旦垃圾回收器准备好释放对象占用的存储空间，将首先调用其 finalize()方法，并且在下一次垃圾回收动作发生时，才会真正回收对象占用的内存**

与 3.1 中引出的两点相呼应

3.垃圾回收只与内存有关（**也就是说，垃圾回收器的唯一原因是为了回收程序不再使用的内存**）

4、垃圾回收器

4.1 垃圾回收器对于提高对象的创建速度有明显的提高，也就是说，**Java 从堆分配空间的速度，可以和其他语言从堆栈上分配空间的速度相媲美**

4.1 Java 的“堆指针”只是简单地移动到尚未分配的区域（类比于传送带），效率比得上 C++在堆栈上分配空间的效率

4.2 垃圾回收器对对象重新排列，实现了一种高速的，有无限空间可供分配的堆模型。

5、成员初始化

5.1 局部变量：如果未初始化，则会编译错误

5.2 数据成员：**若数据成员是基本类型，编译器会保证有一个初始值，若数据成员是对象的引用，该引用的值就是 null**（**C++中类内的内置类型若没有初始化，那么值是未定义的**）
5.3 指定初始化：**在定义类成员变量的地方为其赋值**（**C++中若成员是类类型，那么只能用列表初始化，或者拷贝初始化，而不能用直接初始化，即圆括号**）

6、构造器初始化
6.1 在运行时刻，可以调用方法或执行某些动作来确定初值。但是，无法阻止自动初始化的进行，也就是说在构造器被调用之前，自动初始化已经发生（基本类型为该类型的标准初始值，如 int 为 0，类类型的引用的初值则是 null）
6.2 静态数据的初始化：static 关键字不能应用于局部变量，因此它只能作用于域。**如果一个域是静态的基本类型域，而且没有对它进行初始化，那么它就会获得基本类型的标准初始值；如果它是一个对象的引用，那么它的默认初始化值就是 null**
6.2.1 **静态字段初始化只在必要时刻进行**，如果不创建对象也不引用该字段，那么静态的字段将不会被创建，**只有在第一个对象创建或第一次访问静态字段时，静态字段才会被初始化**。此后，静态字段不会再次初始化
6.2.2 静态字段初始化的顺序先于非静态字段
6.2.3 **构造器实际上也是静态方法**
6.2.4 对象创建的过程：（假设类为 Dog）
1、首次创建 Dog 的对象时，或者 Dog 类的静态方法/静态域首次被访问时，Java 解释器必须查找类路径，以定位 Dog.class 文件
2、载入 Dog.class 有关静态初始化的所有动作都会执行。因此，静态初始化只在 Class 对象首次加载的时候进行一次。
3、当用 new Dog()创建对象时，首先在堆上为 Dog 对象分配足够的存储空间
4、这块存储空间会清零，这就自动将 Dog 对象中的所有基本类型数据都设成了默认值，而引用则被设置成了 null
5、执行所有出现于字段定义处的初始化动作
6、执行构造器
6.2.5 显式的静态初始化：静态子句，或静态块

```
class Cups{
    static Cup cup1;
    static Cup cup2;
    static{           //静态子句，或静态块
            cup1=new Cup(1);
            cup2=new Cup(2);
    }

    Mug mug1;
    Mug mug2;
    {                 //非静态实例初始化
        mug1=new Mug(1);
        mug2=new Mug(2);
        print("Mug1&Mug2 initialized");
    }
}
```

6.3 非静态实例初始化（如上述代码）
无论显式调用哪个构造器，这些操作都会发生（参见 6.2.4 中的 5，优先于构造器调用）

6.4 数组初始化
6.4.1 数组定义　**下面 a1 是引用，引用，引用**
int[] a1;　类名后加空的方括号（Java 中常用的格式）
int a1[];　标识符后加空的方括号（C++中常用的格式）

6.4.2 与 C++不同（C++中要实现动态大小的数组，格式为：T* a=new T[n];new 返回的是指向类型 T 的指针），Java 中编译器不允许指定数组大小

6.4.3 如 6.4.1 定义的 a1 是对数组的一个引用，已经为该引用分配了存储空间，但是没有给数组对象本身分配任何空间。为了给数组创建相应的存储空间，必须写初始化表达式。

初始化表达式的形式有两种：

A、对于数组，初始化动作可以出现在代码的任何地方（使用 new）

```
int[] a1,a2;
a1=new int[5];
a2=new int[]{1,2,3,4,5};
```

B、也可以用特殊的初始化表达式：必须在创建数组的地方出现，由一对花括号括起来的值组成（这种情况，存储空间的分配（等价于使用 new）由编译器控制）

格式如：

```
int[] a1={1,2,3,4,5};
```

6.4.4 数组的赋值

```
a1=a2;  //结果是：a1、a2 引用相同的对象，a1 之前引用的对象将被垃圾回收器监视，改变 a1 或 a2 的值，另一个也随之改变，因为他们指向相同的对象
```

6.4.5 数组的固有成员 length

可以获知数组内包含了多少个元素，能使用的最大下标是 length-1。（Java 数组计数也是从 0 开始）若访问的索引超过 length-1，Java 会出现异常（C++或 C 会产生错误）

6.4.6 实现动态大小的数组定义：（编写程序时，并不知道数组的大小，实际运行时才知道）

在 new 在数组里创建元素（即使元素是基本类型）

```
int[] a2;
a2=new int[n];  //基本数据类型会自动初始化成空值
```

6.4.7 如果创建了一个非基本类型的数组，那么相当于创建一个引用数组（C++中不允许创建引用的数组），那么直到创建对象并赋值给引用，初始化进程才算结束！！！（若没有创建对象，就直接使用数组，将会产生异常）

Java 中数组常用的初始化语句:

```
Integer[] a={
        new Integer(1),
        new Integer(2),
        3,

};
```

or

```
Integer[] b=new Integer[]{
        new Integer(1),
        new Integer(2),
         3,

};
```

6.5 可变参数列表

6.5.1 所有的类都直接或间接继承于 Object 类，因此创建以 Object 数组为参数的方法，就可以实现可变参数

```
 static void printArray(Object[] args){
for(Object obj:args)
    System.out.print(obj+" ");
System.out.println();
}
```

```
//Java SE5 中添加的特性，可以这样定义，类似于 C++中的模板参数包和函数参数包
 static void printArray(Object...args){
    for(Object obj:args)
        System.out.print(obj+" ");
    System.out.println();
}
    printArray(new Integer(47),new Float(3.4),new Double(11.11));
    printArray(47,3.14F,11.11);
    printArray((Object[])new Integer[]{1,2,3,4});//转成 Object 数组后调用
```

若不加(Object[])，编译器会发出警告，因为 Integer[]本身是一个数组，而可变参数列表也是一个数组，通过强制转换告诉编译器，这是一个可变参数的数组

6.5.2 任意类型的参数列表，输入只能是该类型的对象

void f(T…args){}

6.5.3 可变参数列表与自动包装机制可以和谐共处

void f(Integer…args){}

f(10,new Ineger(11),12)；//自动包装机制会将 10 和 12 提升为 Integer

6.6 枚举类型：关键字 enum

enum a {, , ,…} 花括号内是常量集

enum 的方法：

ordinal() 返回该常量在 enum 中定义的顺序(从 0 开始)

values() 返回由所有常量组成的数组，用于范围 for 语句

for(a a1:a.values());

6.6.1 enum 可以在 switch 中使用

Chapter 6. 访问权限控制

1、名词：
1.1 构件：
1.2 类库单元（library：类库）
1.3 包：库单元
1.4 编译单元（转移单元）：.java 文件（每个编译单元最多只能有一个标记为 public 的类，且该类的类名应该与文件名相同）
1.5 输出文件：.class 文件（编译少量.java 文件会得到大量的.class 文件
1.6 Java 可运行程序：一组可以打包并压缩为一个 Java 文档文件（JAR）的.class 文件

2、package
2.1 package 语句必须是文件中除注释意外的第一句程序代码
2.2 package 相当于 C++的 namespace，本质：利用操作系统的文件独立性来避免名字冲突
2.3 package 名称：（**全部小写**）
第一部分：创建者的反顺序的 Internet 域名（如域名 MindView.net →net.mindview）
第二部分：机器上的一个目录（如 mypackage）
包名为：net.mindview.mypackage

3、方法、字段访问说明符（**在哪里可以使用该类的方法或字段**）
访问权限的理解：访问权限指的是一个范围，比如 private，指的是在类内才能访问，friendly，指的是在包内才能访问，public，指的是在任何地方都能访问。而并非指谁的访问权限（之前 C++的理解是，类的对象可以访问 public，不能访问 private，这并不对，由于类的对象在类外创建，自然可以访问 public 的字段或者方法，但是不能访问 private 的字段或方法）
3.1 public：任何情况下都可以被访问
3.2 protected：只要是继承关系就可以访问（可以跨包）（同样含有包访问权限）
3.3 friendly：（包访问说明符）在包内都能访问
3.4 private：**只有该类的内部能够访问（该类的方法）**

4、类的访问权限（**在哪里可以创建该类的对象**）
4.1 public：这样的类可以为某个客户端程序员所用，**客户端程序员可以创建该类的对象**
4.2 default：（什么都不加），这样的类，可以被在**同一个包**中的其他类利用它来创建对象（**包访问权限，同一个包中，可以创建该类的对象，即使该包访问权限的类内有 public 的静态方法，在其他包中也不能调用，也就是说，对于其他包来说，该类的名字是隐藏的，因此也不能调用该类的 public static 方法**）
4.3 abstract：**声明成抽象类**
4.4 final：**声明无法被继承的类**
4.5 **只有被声明为 public 的类，才能在其他包中可见（与是否用 import 无关，import 只是将类导入到当前包中，使用该导入类的时候不需要加包的名字，也就是 packagename.classname->classname）（相当于 C++中的 using 指示，将特定命名空间中的名字导入到包含命名空间和 using 指示所在域的域中）**
4.6 每个编译单元（.java 文件）只能有一个 public 类（可以没有）public 类的名称必须与文件名完全一致。

Chapter 7. 复用类

1、 组合语法

toString()方法：每一个非基本类型的对象都有一个 toString()方法，而且当编译器需要一个 String 而你只有一个对象时，该方法便会被调用，并且必须是 public 修饰的

2、 继承语法

2.1 继承：一般规则是将所有的数据成员都指定成 private，所有的方法都指定成 public

2.2 初始化基类：**在导出类的构造器中调用基类的构造器（Java 会自动在导出类的构造器中插入对基类默认构造器（如果不存在就不行）的调用，并且在执行导出类构造器函数体之前调用基类的构造器）**

若没有默认的基类构造器，或者想调用一个带参数的基类构造器，必须用关键字 super 显式的编写调用基类构造器的语句，并配以适当的参数列表,并且该语句必须是导出类构造函数体中的第一个语句。

2.3 **super 关键字**：

1、方法的覆盖中，调用其**直接基类**的方法，假设方法名为 f 则 super.f(args);

2、**在导出类中初始化基类时，super 关键字配上参数列表即可 super(args);**

3、代理

4、使用复合语句

4.1 垃圾清理：

```
try{
}finally{
}
```

无论 try 块（被称为保护区）如何退出，finally 子句中的代码总要被执行

4.2 **名称屏蔽：如果基类中有某个已被多次重载的方法名称，那么在导出类中重新定义该方法名称并不会屏蔽基类中的任何版本（与 C++相同），也就引入一个新的重载方法；若形参列表也相同，若不是 static 或 final 方法，那么会覆盖掉基类的方法，即动态（与 C++不同，C++中只有指明了 virtual 才会发生覆盖，否则就是替代）**

4.3 组合和继承之间的选择：

4.3.1 组合（包含关系）：想在新类中使用现有类的功能而非它的接口，一般情况下，想让新类的用户看到新类所定义的接口，因此将现有类设置成 private

4.3.2 继承（是-关系）：使用一个通用类，并为了某种特殊需要而将其特殊化

4.3.3 **生成和使用程序代码最有可能采用的方法就是直接将数据和方法包装进一个类中（即组合），并使用该对象，因此要慎用继承，如果需要从新类向基类进行转型，那么继承时必须的**

4.4 protected 关键字：

对于**继承于此类的导出类或其他任何位于同一个包内的类来说**，它是可以访问的

4.5 @Override 注解:不是关键字，但是可以当关键字使用

方法返回类型前添加这个注解：表明该导出类中的方法覆盖基类中的方法而非重载，当不小心写成重载后，编译器会提示错误（**含义与 C++中一致，C++中书写的位置在形参列表后**）

4.5 向上转型

4.5.1 继承最重要的一点：用来表现新类和基类之间的关系，可以用“新类是现有类的一种类型”来概括

4.5.2 **实现多态：在基类中定义 static 的方法,并使其接受基类类型的参数，在调用该函数时候，使用基类的类名，但可以向其传递导出类的类型作为形参（C++中利用指向基类的指针或基类的引用，并将其绑定到派生类上实现多态）**

```
ExtendClass a=new ExtendClass();
BaseClass.f(a);
or
BaseClass a=new ExtendClass();
```

4.5.3 向上转型是从一个较专用的类型向较通用的类型的类型转换，因此总是安全的

4.6 final 关键字

4.6.1 与 C++中的 const 类似，但与 C++中的 final 不同，C++中的 final 说明符指的是该类不可被继承，或者该成员函数不能被覆盖。

4.6.1 final 数据：

1、一个永远不改变的编译时常量

2、一个在运行时被初始化的值，而你不希望它被改变

这类常量必须是基本数据类型，定义该常量时必须对其赋值

3、**对于对象而言，运用 fianl 关键字，由于实际定义的是对象的引用，因此 final 关键字表示该引用的引用对象不能改变，但是该对象自身确是可以改变的（类似于常量引用的意思）**

4、一般书写形式：public static fianl int VALUE=99;

public:表示可以被用于包外

static:表示只有一份

final:表示是一个常量

4.6.2 空白 final：

定义：指被声明成 final 但又未给定初始值的域，但必须保证空白 final 在使用前必须被初始化（**在域的定义处或者每个构造器中用表达式对 final 进行赋值**）

用处：一个类中的 final 域可以做到根据对象而有所不同，但又保持其恒定不变的特性

4.6.3 final 参数

Java 允许在参数列表中以声明的方式将参数指明为 final，**这意味着你无法在方法中更改参数引用所指向的对象，即引用对象不能改变，该对象自身值可以改变**

4.6.4 final 方法

1、将方法锁定，以防任何继承类修改它的含义（确保在继承中使方法行为保持不变，并且不会被覆盖）

2、**早期的 Java 编译**器建议使用 final 方法：可以提高效率，当编译器发现一个 final 方法调用命令时，会将参数压入栈，跳至方法代码处执行，然后跳回并清理栈中的参数，处理返回值，这样可以消除方法调用的开销。（**适用于小型的方法**）

3、**最近的 Java 版本不需要使用 final 方法来进行优化**

4.6.5 **final 和 private 关键字：类中所有的 private 方法都隐式的指定为 final。由于 private 对导出类不可见，也就无法覆盖。**

4.6.6 final 类:表明该类无法被继承

final class a{}

4.7 初始化及类的加载：

4.7.1 C++等语言中：程序时作为启动过程中的一部分立刻被加载的，然后初始化，接着程序开始运行，初始化必须小心控制，确保定义为 static 的变量的初始化顺序不会造成麻烦。

4.7.2 Java：每个类的编译代码都存在于它的独立文件中。该文件只需要在使用程序代码时才会被加载。**一般来说，可以理解为：类的代码在初次使用时才被加载（创建类的第一个对象时，或者访问静态方法，静态字段时）**

4.7.3 继承与初始化

1、加载：在导出类第一次被使用时，首先会加载导出类的编译代码，然后加载该导出类的基类的编译代码，若该基类仍有基类，那么继续加载该基类，直到根基类。

2、初始化：首先初始化根基类的 static，因为导出类中的 static 初始化可能依赖于基类的 static

3、必要的类都已加载完毕后，对象中所有的基本类型都会被设为默认值，对象引用被设为 null-通过将对象内存设为二进制零值而一举生成。

4、**从根基类到导出类依次调用构造器**

Chapter 8. 多态

1、向上转型：由导出类向基类进行转型是安全的，因为导出类的接口包含基类的接口。

1.1 如何实现动态绑定（后期绑定，运行时绑定）：编写接受基类为参数的方法

2、绑定

2.1 定义：将一个方法调用同一个方法主体关联起来被称作绑定。

2.2 前期绑定：在程序执行前进行绑定（由编译器和连接程序实现）这是面向过程语言中的默认绑定方式，例如 C 语言

2.3 后期绑定：在运行时根据对象的类型进行绑定。也被称为动态绑定或运行时绑定

2.4 编译器一直不知道对象的类型，但会在对象中安置某种“类型信息”

2.5 Java 中除了 static 方法和 final 方法，*其他所有方法都是后期绑定*。故 final 有关闭动态绑定的含义

2.6 有了动态绑定，就可以之编写与基类打交道的方法(即接受基类形参的方法),也就是说发消息给某个对象，让该对象去断定应该做什么事

或者将基类的引用绑定到基类或者导出类上，对该引用调用方法也会进行动态绑定

```
Shape s=new Circle(); // Shape 是基类，Circle 是导出类
s.dosomething();//会执行 Circle 中的 dosomething
```

2.7 只有非 private 的方法才能实现多态，因此在导出类中对于基类的 private 的方法最好采用不同的名字以免混淆（在导出类中对基类的 private 方法进行“重载”，编译器不会报错（会认为这是两个不同的方法，并不会覆盖），但是也与我们的期望不同）。

2.8 域：如果直接访问某个域，那么访问就将在编译器进行解析，即静态（域无法实现多态）

在基类中定义了字段，在导出类中修改了该字段的值，那么在两个类中这个字段会处于不同的域中（存储空间）（因为在不同的类中定义，也就属于不同的{}，因此这两个字段的域是不同的），任何域访问操作都将由编译器解析。

```
class A{                class B extends A{
    int i=1;                int i=2;
    int f(){return i;}          int f(){return I;}
}                   }
A a=new B();  那么 a.i 指的是 A 中的 i，也就是 1
              a.f()返回的是 2
```

2.9 静态方法和 final 方法不具有多态性

3、构造器和多态

3.1 构造器不具有多态性（构造器是隐式的 static 方法）

3.2 基类的构造器总是在导出类的构造过程中被调用，而且按照继承层次逐渐向上链接，以使每个基类的构造器都能得到调用。

3.3 构造器检查对象是否被正确地构造，而导出类只能访问自己的成员，不能访问基类中的成员，只有基类的构造器才具有恰当的指示和权限来对自己的元素进行初始化，因此，编译器必须令所有构造器都得到调用

3.4 复杂继承对象的构造顺序：

1、调用基类构造器：首先构造根基类，依次向下到直接基类

2、按声明顺序调用成员的初始化方法

3、调用导出类构造器的主体

4、清理与继承

4.1 在基类中定义一个清理的函数，例如 dispose()，在导出类中对其进行覆盖，并调用基类的版本 super.dispose()。清理的顺序与初始化的顺序相反（与 C++中的析构函数相似），因为清理导出类可能会用到基类的方法，因此不能过早的清理基类。

4.2 如果多个类型 B 的对象共享一个类型 A 的对象，那么只有当最后一个 B 类型的调用 dispose 的时候才会调用 A 的 dispose，因此需要在类型 A 中设置计数器（C++中，类似于指针到智能指针的转变）

4.3 构造器内部的多态方法（应当尽量避免）

如果要调用构造器内部的一个动态方法，就会用到那个方法的被覆盖后的定义。会导致被覆盖的方法在对象被完全构造之前就会被调用，

4.4 初始化的实际过程：

1）在其他任何事物发生之前，将分配给对象的存储空间初始化成二进制的零
2）从基类到派生类的静态变量初始化
3）按照基类成员声明的顺序调基类成员的初始化方法
4）调用基类的构造器主体
5）按照声明的顺序调用导出类成员的初始化方法
6）调用导出类的构造器主体

4.5 协变返回类型（C++中，若虚函数的返回值是基类的类型，包括指针，引用等，那么在继承类中返回类型可以变为继承类的类型）

4.6 纯继承：导出类中没有任何额外的方法以及字段

Chapter 9. 接口

1、抽象类和抽象方法（C++中虚基类含有 virture 关键字的成员函数）：

1.1 抽象类的对象没有任何意义，**创建抽象类是希望通过这个通用接口操纵一些列类**

1.2 如果一个类包含一个或多个抽象方法**（如下，只有声明，没有定义，并且前面有关键字 abstract）**，那么该类被限定为抽象的（必须在类前添加 abstract 关键字）

```
abstract void f();
```

如果要将一个不包含任何抽象方法的类定义成抽象的，那么在类前添加关键字 abstract 即可

1.3 如果抽象类的导出类没有覆盖定义所有抽象类中的抽象方法，那么该导出类也是抽象类（与 C++类似）

类 A 含有一个方法 a，类 B 有一个方法 b，类 C 想要同时获得类 A 类 B 的方法 a 和方法 b，**C++的解决方案是，C 多重继承类 A 和类 B；Java 的解决方案是将类 AB 进行进一步抽象成接口，并且删去其余冗余部分（Java 不支持同时继承多个类，但是可以同时实现多个接口）。**

2、接口（关键字 interface，比抽象类更进一步，即更抽象）

2.1 interface 产生一个**完全抽象**的类，**没有任何具体实现，允许创建者确定方法名、参数列表和返回类型**

2.2 **接口被用来建立类与类之间的协议**

2.3 创建接口：用 interface 关键字代替 class 关键字（与抽象类不同），**在 interface 前面可以添加 public 关键字（但仅限于该接口在于其同名的文件中被定义），否则就是包访问权限的**

2.4 接口也能包含**域（字段？）（这些域是隐式的 static 和 final）**

2.5 **遵循某个特定的接口要用关键字 implements（继承用的关键字是 extends）**

一个类 Class 必须以 implements 的方式遵循接口，而不能用 extends，而且被遵循的方法必须被显示声明为 public，因为接口中的方法都是隐式 public 的

一个接口必须以 extends 的方式继承接口，而不能用 implements

2.6 **因为接口是与外部交互的，因此接口中所有的方法都是隐式 public 的，也可以显式注明**

2.7 **若一个类遵循某个接口，那么必须定义接口中的方法。**

2.8 **如果创建不带任何方法定义和成员变量的基类，那么就应该选择接口而不是抽象类。**

3、完全解耦

3.1 在此之前，方法都是与类紧密结合在一起的

3.2 **只要一个方法操作的是类而不是接口，那么你就只能使用这个类及其子类，如果想要将这个方法应用于不在此继承结构中的某个类，那么就会触霉头**

3.3 复用代码：

1、遵循接口来编写自己的类

2、当无法修改现有类时，创建适配器，建立现有类与接口的联系

3.4 **当将一个具体类和多个接口组合到一起时，这个具体类必须放在前面，后面跟着的才是接口**

4、接口适配器

4.1 **其实适配器只是一个类，它实现了某种接口，提供了方法体。这样，再用到这个接口时，可以直接继承适配器，这样就不需要把接口中的每一个方法再填充一遍了，只需要在这个类中复写一下需要用的方法。**

5、接口中的域

5.1 有了 enum 关键字就尽可能用 enum

5.2 **接口中的域都是隐式 static 和 final 的**

5.3 **接口中的域不能使‘空 final’即没有在定义处给定初始值的域，因为没有方法为该域进行赋值**

5.3 接口中的域可以被非常量表达式初始化

6、嵌套接口：即接口可以嵌套在类或其他接口中（C++中类可以嵌套在函数中）
6.1 嵌套接口不仅拥有非嵌套接口的’public’和’包访问’两种可视性，嵌套接口还可以拥有’private’的可视性
6.2 嵌套在一个接口中的接口是隐式 public 的

7、接口和工厂
7.1 两个类 A 和 B 的关系应该仅仅是 A 创建 B 或 A 使用 B，而不能两种关系都有
7.2 引入工厂类和客户类。工厂类只涉及对象的创建，客户类只涉及对象的使用
7.3 对于一个接口的几种实例化的类而言，这几种实例化的类彼此之间是有联系的，那么只需要一个工厂即可；对于一个接口的几组不同实例化的类而言，每组类之间是相互有联系的，那么用一个工厂是无法满足需求的，因此需要定义一个工厂接口来生产接口，每一组类别分别用一个工厂接口实例化的类来生产该组的类

Chapter 10. 内部类

1、内部类：可以将一个类的定义放在另一个类的定义内部，这就是内部类

1.1 内部类拥有对外围类(enclosing object)所有元素的访问权限

1.2 Java 的内部类与 C++的嵌套类非常不同，C++中只是单纯的名字隐藏机制，与外围对象没有联系，也没有隐含的访问权限

1.3 （内部类为非静态时）内部类的对象只能在与其外围类的对象相关联的时候才能被创建，因为内部类对象会暗暗地连接到创建它的外部类对象上

（内部类为静态时，即嵌套类）那么创建它就不需要对外部类对象的引用

1.4 如何创建内部类的对象

```
class B{
    String s;
    class BInner{
        String s;
        String f1(){return s;}//等同于 this.s  都是内部类的 s 域
        String f2(){return B.this.s;}//外部类的 s 域
    }
}
B b=new B();
B.BInner c=b.new BInner();
```

1.5 一个类的修饰词只能是：public default(什么也不加) abstract final；但是一个内部类的修饰词可以是 private，private 内部类只对外围类可见，只有在外围类中才能创建 private 内部类的对象，因此在外围类的方法中创建内部类是不需要利用外围类的对象。

1.6 .this 与.new，见上述例子

.this：需要生成外部类对象的引用，可以试用外部类名字后跟.this（仅限非静态内部类）

2、内部类与向上转型

2.1 从实现了某个接口的对象，得到对此接口的引用，与向上转型为这个对象的基类，实质一样

2.2 内部类—某个接口的实现—能够完全不可见，并且不可用。所得到的指向基类或接口的引用能够方便地隐藏细节

2.3 外部类不能访问内部类的域和方法，无论这个域或方法是 public、private、protected、friendly

2.4 一个 private 内部类若是一个接口的实现，那么方法是 public 的，但是整个类确是 private，无法通过内部类类型来调用方法，因为内部类类型在外围类之外是不可见的，因此只能通过动态方式来调用，即用接口的引用来调用这个 public 方法

3、在方法和作用域内的内部类

3.1 可以在一个方法里面或者在任意的作用域内定义内部类

A、实现了某类型的接口，于是可以创建并返回对其的引用（内部类是某接口的实现）

B、想要创建一个类来辅助解决方案，但又不希望这个类是公共可用的

3.2 若一个方法有内部类，且返回内部类类型的向上转型（如接口类型）是可以的！！！

4、匿名内部类（只能用于实例初始化）

4.1 new A(args){ 匿名类的定义 };的分号前插入匿名内部类定义，返回向上转型后的类型 A 的引用

4.2 匿名内部类会向上转型成返回类型的类型，无论这个返回类型是接口或是一个普通的具有具体实现的类。

4.3 如果一个匿名内部类需要使用一个在其外部定义的对象，那编译器会要求参数引用是 final 的？

4.4 匿名内部类中任何定义以外的语句都需要用花括号括起来。

定义的字段可以在定义处赋初值，可以不用加花括号

若对字段进行了赋值，那么必须用花括号括起来

4.5 匿名内部类与正规的继承相比有些受限：匿名内部类可以扩展类或实现接口，但不能两者兼备，

也就是不能即继承一个类，又实现一个接口，而且也不能实现多个接口
4.6 匿名内部类对于其“继承”的基类所特有的函数，在返回后是不可见的，因为发生了向上转型，因此这些函数需要在匿名内部类的定义时进行调用（用{}）如例 12

5、工厂与匿名类的结合 P200

1、定义 Service 接口

2、将该类的构造函数 Builder 设为私有

3、定义 Facotry 接口，含有一个返回 Service 类型的方法 f

4、为该类定义一个 public static 的域，其类型 Factory

5、定义一个匿名类作为工厂接口的实例化（赋值给该类的类型为 Factory 的静态域），其中方法 f 会调用该类型的构造函数，并返回该类型对象的引用，且向上转型为 Service 接口类型。

```
public static Factory f=new Factory(){
    public Serviec f(){return new Builder();}
};
```

5.2、优先使用类而不是接口，如果需要某个接口，必须先了解它。将第五点的接口均改为类即可

6、嵌套类（static）

6.1 static 内部类对象与外围类对象之间没有联系

6.2 普通的内部类对象隐式地保存一个指向创建它的外围类的引用

6.3 创建嵌套类的对象，不需要外围类的对象

6.4 不能从嵌套类的对象中访问非静态的外围类对象（因为嵌套类与外围类没有联系了）

6.5 普通内部类不能含有 static 数据 static 方法和 static 字段，也不能包含嵌套类，但是嵌套类可以

6.6 与 C++嵌套类大致相似，但是 C++中嵌套类不能访问私有成员，而 Java 可以

6.7 创建嵌套类的对象

```
class B{
    String s;
    static class BInner{
        String s;
        String f1(){return s;}//等同于 this.s  都是内部类的 s 域
        String f2(){return B.this.s;}//错误，只有非静态内部类才能回调外部类的字段或方法
    }
}
B.BInner c= new B.BInner();
```

7、接口内部的类

7.1 接口内部类：自动都是 public 和 static 的

7.2 可以在接口内部类中实现外围接口

7.3 可以将每一个放在 main 函数中对于该类的测试代码放置在 public static 内部类的代码中，会生成独立的类 EnclosingClass&InnerClass，发布产品时将它删除即可

7.4 多层嵌套类能够访问它所嵌入的外围类的所有成员

8、为什么需要内部类

8.1 一般而言，内部类继承自某个类或实现某个接口，内部类的代码操作‘创建它的外围类的对象’

可以认为内部类提供了某种进入其外围类的窗口。

8.2 如果只需一个对接口的引用，就直接通过外围类来实现这个接口

8.3 每个内部类都能独立地继承自一个（接口的）实现，所以无论外围类是否已经继承了某个（接口的）实现，对于内部类都没有影响◊内部类使得多重继承的解决方案变的完整

8.4 由于 Java 不支持 class A extends B,C{} 这样的语法，因此用内部类可以完美解决这个问题

8.5 对于接口：一个类要实现多个接口

A、采用单一类

B、使用内部类

对于抽象类：只能用内部类实现多重继承问题

8.6 内部类的特性：

①内部类可以有多个实例，每个实例都有自己的状态信息，并且预期外围类对象的信息相互独立

②在单个外围类中，可以让多个内部类以不同的方式实现同一个接口，或继承同一个类

③创建内部类对象的时刻并不依赖于外围类对象的创建

④内部类并没有令人迷惑的“is-a”的关系，它就是一个独立的实体

8.7 闭包(closure)与回调

闭包：closure 是一个可调用对象，它记录了一些信息，这些信息来自于创建它的作用域。

因此非静态内部类是面向对象的闭包，因为它不仅包含外围类对象（创建内部类的作用域）的信息，还自动拥有一个指向此外围类对象的引用，在此作用域内，内部类有权操作所有的成员，包括 private 成员。

回调：就是允许客户类通过内部类引用来调用其外部类的方法。通过回调，对象能够携带一些信息，这些信息允许它在稍后的某个时刻调用初始的对象。

8.8 内部类与控制框架

应用程序框架：被设计用以解决某类特定问题的一个类或一组类。通常要机场一个或多个类，并覆盖某些方法（用内部类来实现多重继承，或者可以在单一的类里面产生对同一个基类的多种导出版本）

时间驱动系统：主要用来响应事件的系统

9、内部类的继承

由于内部类的构造器必须连接到指向其外围类对象的引用。（导出类也要满足）问题在于，那个指向外围类对象的“秘密的”引用必须被初始化，而在导出类中不再存在可连接的默认对象。

构造函数必须传递外围类对象的引用，并且在内部类的导出类的构造函数中使用以下语法：

enclosingClasReference.super();//enclosingClasReferenc 外围类类型的对象（形参符号，不是类型名称）

10、内部类可以被覆盖吗？不可以

10.1 这两个同名内部类是完全独立的两个实体，各自在自己的命名空间内

11、局部内部类（在方法体力创建）

11.1 局部内部类不能有访问说明符，因为它不是外围类的一部分

11.2 局部内部类可以访问当前代码块内的常量以及此外围类的所有成员

11.2 使用局部内部类而不是匿名内部类的情况：需要不止一个该内部类的对象

Chapter 11. 持有对象

容器类的基本类型：List、Set、Queue、Map

1 泛型和类型安全的容器

1.1 ArrayList：可以自动扩充自身尺寸的数组。add()插入对象 get(index)访问对象

1.2 如果一个类没有显式声明继承自哪个类，那么默认继承自 Object

1.3 ArrayList 保存的是 Object，因此可以存放不同种类的对象，但是用 get 提取对象时得到的是 object 的引用（一般不这么用，都是会指定容器保存类型的）

1.4 指定容器保存的类型 ArrayList<E>,此时 get 会获取 E 类型的对象，因为 get 方法中包含了从 Object 到 E 的强制类型转换；还可以保存向上转型为 E 的类型

1.5 可以用 foreach 来遍历容器元素

2、基本概念：Java 容器库的用途是保存对象，可以分为两个不同概念

2.1 Collection（其余类型都是 Collection 的导出类）：一个独立元素的序列，这些元素都服从一条或多条规则：如 List 必须按插入的顺序保存元素；Set 不能有重复元素；Queue 按照排队规则来确定对象产生的顺序（通常与插入顺序相同）

2.2 Map（其余类型都是 Map 的导出类）：一组成对的“键值对”对象，允许使用键来查找值

2.3 初始化：

```
List<E> c1=new LinkedList<E>(Arrays.asList(args));
List<E> c2=new LinkedList<E>(); Collections.addAll(c2,args);
List<E> c3=new LinkedList<E>(c1); c3 和 c1 引用的是两个不同的 List，只是拷贝 c1 的元素而已
```

3、添加一组元素

3.1 Arrays.asList()方法接受一个数组或是一个用逗号分隔的元素列表（使用可变参数），并将其转换成 List 对象，一般用法是将其作为构造器的参数

Arrays.asList()将一个数组转化为一个 List 对象，这个方法会返回一个 ArrayList 类型的对象， 这个 ArrayList 类并非 java.util.ArrayList 类，而是 Arrays 类的静态内部类！用这个对象对列表进行添加删除更新操作，就会报 UnsupportedOperationException 异常。

Arrays.asList()的局限：它对所产生的 List 的类型做出了最理想的假设，而并没有注意你会对它赋予什么样的类型（好像是可以编译的？并没有出现书上的错误），可以插入一个‘线索’Arrays.<E>asList()

3.2 Collections.addAll()方法接受一个 Collection 对象，以及一个数组或是一个用逗号分隔的列表，将所有元素添加到 Collection 中

3.3 Map 除了用另一个 Map 之外，Java 标准库没有提供任何其他自动初始化的方式

4、容器的打印

4.1 对于数组，必须使用 Arrays.toString()来产生数组的可打印表示

4.2 对于容器无需任何帮助

Collection 的内容用方括号括住，每个人元素有逗号分隔

Map 的内容用大括号括住，键与值用等号联系，键在等号左边，值在等号右边

5、List（接口）：将元素维护在特定序列中，在 Collection 的基础上添加大量方法，有两种基本类型

5.1 ArrayList：它长于随机访问元素，但是在 List 的中间插入和移除元素时较慢（类似于 Vector）

LinkedList：在 List 中间插入和删除操作较快，提供了优化的顺序访问，但是随机访问较慢

5.2 方法：List<E> L; List 只是个接口！！！不能 new List<E>();

L.get(n)：取出索引为 n 的元素

L.add(e)：向尾部添加元素 e

L.add(n,e)：在索引为 n 处插入元素 e,插入后 e 的索引为 n（从 0 开始）

L.add(sub)：在尾部插入 sub 中的所有元素
L.addAll(n,sub)：在索引为 n 处插入 sub 中的所有元素，sub 中第一个元素的索引为 n
L.contains(e)：判断对象 e 是否存在列表中,返回 boolean 类型
L.containsAll(sub)：判断片段 sub 是否在列表 L 中（顺序任意），返回 boolean 类型
L.remove(e)：移除该对象 e,返回 boolean 类型
L.remove()：移除并返回列表的头元素
L.removeAll(sub)：移除片段 sub 中的所有元素
L.indexOf(e)：查询对象 e 在列表 L 中的索引号（从 0 开始）若查询不到返回-1
List<E> L1=L.subList(i,j)：L 中 i 到 j 的片段作为 List 的引用初始化 L1（改变 L1 会导致 L 的改变），包括 i 不包括 j（i，j 从 0 开始）原因很简单，没有用 new
Collections.sort(L)：排序 L
Collections.shuffle(L)：打乱 L
Collections.shuffle(L,rand)：以 rand 为种子打乱 L
L.retainAll(L1)：保留同时在 L 与 L1 中的元素
L.set(n,e)：在 n 位置将元素替换为 e
L.isEmpty()：判断 L 是否为空，返回 boolean 类型
L.clear()：清空 L

6、迭代器（接口）
6.1 迭代器时一个对象，它的工作是遍历并选择序列中的对象
6.2 迭代器被称为轻量级对象：即创建它的代价小
6.3 Iterator 迭代器：
方法 L.iterator()要求容器返回一个 Iterator，Iterator 将准备好返回序列的第一个元素（下一个 next()将会返回第一个元素）A a=new A(); Iterator<A> it=a.iterator();
1）Iterator.next()获取序列中的下一个元素
2）Iterator.hasNext()检查序列中是否还有元素
3）Iterator.remove()将迭代器新近返回的元素删除（即将最近一次调用 next()返回的元素删除）
6.4 通过迭代器可以将遍历序列的操作与序列底层的结构分离。迭代器统一了对容器的访问方式（还是需要指定迭代器实例化的类型：Iterator<E>，但也可以不指定，会有 warning）
（C++中，也有迭代器，但与 Java 不同，每种类型的容器有每种类型容器实例化后所特有的迭代器类型，例如 vector<int>::iterator 并没有使得遍历序列的操作与序列底层的结构分离）
6.5 ListIterator：Iterator 的子类型，只能用于各种 List 类的访问
6.5.1 ListIterator 可以双向移动
6.5.2 ListIterator.set()：替换它访问过的最后一个元素（it.previous() or it.next()）
6.5.3 ListIterator.listIterator()：产生一个指向 List 开始出的 ListIterator，指向表前元素
6.5.4 ListIterator.listIterator(n)：创建一个位于第 n 个和第 n+1 个元素之间的迭代器，调用 next()指向第 n+1 个元素，调用 previous()指向第 n 个元素（元素从第一个开始算起）
若元素从第 0 个算起，那么该迭代器位于索引为 n-1 和 n 之间
6.5.5 ListIterator.nextIndex()：返回最后一次调用 it.next()返回的元素的下一个元素的索引
6.5.5 ListIterator.previousIndex()：返回最后一次调用 it.next()返回元素的索引
6.6.6 it.add(E e)：在当前位置添加元素，并做好访问下一个元素的准备
7、LinkedList（非接口）
7.1 LinkedList 添加了可以使其用作栈，队列或双端队列的方法：
栈：LinkedList.push()添加在 List 的表头，即栈顶元素为表头,用 L.getFirst()和 L.element()访问栈顶元素。
7.2 L.getFirst()和 L.element()：完全一样，都返回列表的头（第一个元素）若 List 为空，则抛出 NoSuchElementException。
7.3 L.remove() 和 L.removeFirst()：完全一样，都移除并返回列表的头，在列表为空时，则抛出 NoSuchElementException。

7.4 L.removeLast()：移除并返回列表的最后一个元素
7.5 L.add(),L.addLast()：相同，他们都将某个元素插入到列表的尾部
7.6 L.addFirst()：将元素插入到列表的首部

8、Set（接口）
8.1 拥有与 Collection 完全一样的接口，没有额外的功能
8.2 Set 是基于对象的值来确定归属性的
HashSet：提供了快速查找的计数，没有按明显的顺序来保存元素（哈希表）
TreeSet：升序存储
LinkedHashSet

9、Map（接口）
HashMap：提供了快速查找的计数，没有按明显的顺序来保存元素（哈希表）
TreeMap：按照比较结果的升序保存键
LinkedHashMap：按照插入顺序保存键，同时保留了 HashMap 的查询速度
9.1 Map 特有的方法
m.put(key,value)方法增加一个值，并将它与某个键关联起来
m.get(key)方法将产生与这个键相关联的值，若键不在容器中，则 get(key)返回 null
m.containsKey(key)：m 中是否包含关键字 key
m.containsValue(value)：m 中是否包含值 value
m.keySet()：获取关键字构成的 Set
m.values()：获取值构成的 Collection

10、Queue（接口）：先进先出（FIFO）的容器
10.1 LinkedList 提供了方法以支持队列的行为，并且实现了 Queue 的接口，因此 LinkedList 可以用作 Queue 的一种实现。通过将 LinkedList 向上转型为 Queue
10.2
q.offer(e)：将元素 e 插入到队尾，成功返回 true 失败返回 false
q.peek() q.element() 在不移除的情况下返回队头，但是 peek()方法在队列为空时返回 null;element()在队列为空时抛出 NoSuchElementException 异常
q.poll() q.remove()方法将移除并返回队头，但是 poll()方法在队列为空时返回 null;remove()在队列为空时抛出 NoSuchElem entException 异常

11、PriorityQueue（非接口）
11.1 先进先出描述了最典型的队列规则，队列规则是指在给定一组队列中元素的情况下，确定下一个弹出队列的元素的规则。先进先出声明的下一个元素应该是等待时间最长的元素。
11.2 优先级队列声明下一个弹出的元素是具有最高优先级的元素
11.3 在 PriorityQueue 上调用 offer(e)，这个对象 e 会在队列中进行排序（默认的排序是利用对象在队列中的自然顺序（不是插入顺序，比如 Integer 就是大小顺序，小的优先级高，String 就是字典序，排前面的优先级高），可以通过提供自己的 Comparator 来修改这个顺序
11.4 peek() element() poll() remove() 会保证获取的元素将是具有最高优先级的元素
11.5 QueueDemo.printQ(queue)：会利用 poll()对 queue 进行打印，打印结束后，队列为空
11.6 要在 PriorityQueue 中使用自己的类，必须包括额外的功能以产生自然排序，或者必须自己提供 Comparator

12、Collection 和 Iterator
12.1 C++类库中并没有其容器的公共基类—容器之间的所有关系都是通过迭代器达成的
12.2 Java 用 Collection 作为容器的公共基类（除了 Map）,并且将 Iterator 与 Collection 绑定到一起，

意味着实现 Collection 就必须实现 Iterator

13、Foreach 与迭代器

13.1 foreach 语句主要用于数组，也可以应用于任何 Collection 对象

13.2 Iterable 接口，该接口包含一个能够产生 Iterator 的 iterator()方法

13.2 foreach 语句可以应用于任何 Iterable

13.4 Collection 与 Iterable 的关系：由于 Collcetion 有能够产生 Iterator 的 iterator()方法，也就是 Collection 中包含 Iterable 接口（因为 Collection 本身是接口，不能说实现，只能说包含）

13.5 for(T t:E) E 必须是数组类型，或者是 Iterable 导出类型，或者是 Colleciont 导出类型

13.6 foreach 语句中如果有定义的语句，必须加大括号

```
for(String s:s1)
    String s2=s;//会编译失败
```

13.7 遍历 Map 的方法

```
Map<U,V> map=new HashMap<U,V>();
for(Map.Entry<U,V> a:map.entrySet())
```

List<? extends Pet>任何继承 Pet 的类型都能添加进 List（一种模糊的实例化）？？？

List<? super Pet>任何 Pet 的基类（直接或间接）都能添加进 List（一种模糊的实例化）

总结：

1、数组将数字域对象联系起来，数组一旦生成，其容量就不能改变

2、容器不能持有基本类型，但可以持有基本类型的包装类型

3、常用的容器 ArrayList、LinkedList、HashMap、HashSet

4、注意点：容器存的是引用，用 add(e)\addAll(e)添加元素后，并改变其值，会导致 e 所引用的对象发生改变;但是若 e 是基本类型，则不同，因为基本类型的参数传递机制不同

Chapter 12. 异常

1、概念

1.1 Java 的基本理念：结构不佳的代码不能运行

发现错误的理想时机是在编译阶段，其余问题必须在运行时解决◊需要通过某种方式，把适当的信息传递给某个接受者◊接受者将知道如何正确处理这个问题

1.2 异常处理是 Java 中唯一正式的错误报告机制，并且通过编译器强制执行

2、基本异常

2.1 **普通问题：在当前环境下能够得到足够的信息，总能处理这个错误**

2.2 **异常情形：是指阻止当前方法或作用域继续执行的问题。在当前环境下无法获得必要的信息来解决问题，因此只能从当前环境跳出，并且把问题提交给上一级环境—异常抛出**

2.3 **异常抛出：同其他对想创建一样，使用 new 在堆上创建对象，然后，当前执行路径被终止，并且从当前环境中弹出对异常对象的引用。此时异常处理机制接管程序，并开始寻找一个恰当的地方来继续执行程序，恰当的地方就是指异常处理程序，它的任务就是讲程序从错误状态中恢复，以使程序能换一种方式或继续运行下去**

2.4 异常最终要的一方面：如果发生问题，就不允许程序沿着其正常的路径继续走下去

2.5 标准异常类都有两个构造器：一个是默认构造器，另一个是接受字符串作为参数，以便能把相关信息放入异常对象的构造器；

2.6 用 new 创建异常对象后，此对象的引用将传给 throw

3、捕获异常

3.1 **监控区域：它是一段可能产生异常的代码，后面跟着处理这些异常的代码**

3.2 **如果在方法内部抛出了异常，这方法将在抛出异常的过程中结束。要是不希望方法就此结束，可以在方法内设置一个特殊的块来捕获异常，try 块**

3.3 异常处理程序：紧跟在 try 块之后，以关键字 catch()表示，接受一个且仅接受一个特殊类型的参数的方法。**可以在处理程序的内部使用标识符 catch(Type i)，也可以不用 catch(Type)？？？，因为异常的类型已经给了足够的信息来对异常进行处理**

3.4 只有匹配的 catch 子句才能得到执行，与 switch 不同，需要 break 来避免进行其他 case 子句

3.5 终止与恢复

A、**终止模型：程序无法返回到异常发生的地方继续执行。主流！（C++也是这个模式）但是，一旦被某一层的 catch 所捕获并处理，那么该层之外的程序会继续执行**

B、**恢复模型：异常处理程序的工作是修正错误，然后重新尝试调用出问题的方法，并认为第二次能成功（把 try 块放在 while 循环里）**

4、创建自定义异常

4.1 **必须从已有的异常类继承，最简单的方法就是让编译器产生默认构造器，大括号内什么都不写**

4.2 **finally 子句：执行完 try 后必须会执行的子句**

4.3 异常与记录日志？？？P253 没太看懂，看完 18 章再来

4.4 Throwable 有 getMessage()方法，类似于 Object 中的 toString(); **Throwable 是 Exception 的基类；Object 是 Throwable 的基类**

5、异常说明

5.1 使用了附加关键字 throws，后面接一个所有潜在异常类型的列表，以逗号分隔

5.2 **可以声明方法将抛出异常，但实际不抛出**；

5.3 **当函数体内部没有 try 块以及 catch 子句时，实际会抛出异常的方法必须含有异常说明**

5.3 **当函数体内部含有 try 块以及 catch 子句时，实际会抛出异常的方法可以没有异常说明**

6、捕获所有异常

6.1 Exception 是所有异常类的基类，Exception 的基类是 Throwable

6.2 Throwable 的方法：

String getMessage() 类似于 toString 的方法，**当调用 printStackTrace()时会将信息（返回的 String）显示在异常名加冒号之后**

String getLocalizedMessage()

String toString() 从 Object 继承的方法

void printStackTrace() 打印调用栈轨迹

void printStackTrace(e)（e 为输出流）

6.3 栈轨迹

6.3.1 printStackTrace()方法所提供的信息可以通过 getStackTrace()来直接访问

6.3.2 getStackTrace()返回一个由栈轨迹中元素所构成的数组，每一个元素都表示栈中的一帧，**元素 0 是栈顶元素，是调用序列中的最后一个方法调用。最后一个元素是栈底，是调用序列中的第一个方法调用。该数组的元素类型是 StackTraceElement，含有方法 getMethodNmae()**

6.4 重新抛出异常

6.4.1 **重抛异常会把异常抛给上一级环境中的异常处理程序，同一个 try 块的后续 catch 子句将被忽略**

6.4.2 **如果只是把当前异常对象重新抛出，那么 printStackTrace()方法显示的将是原来异常抛出点的调用栈信息，而非重新抛出点的信息**

6.4.3 **利用 fillInStackTrace()方法，返回一个 Throwable 对象，会把当前调用栈信息填入原来那个异常对象（之前的信息会被清除，调用 fillInStackTrace()的地方就成了新异常发生地），（需要调用强制类型转换，转换成 Exception 或其子类）**

6.4.4 利用 new 来重新抛出异常与利用 fillInStackTrace()来重新抛出异常的异同

同：有关原来异常发生点的信息会丢失，重新抛出的异常类型可能发生变换（调用 fillInStackTrace()需要强制类型转换）

异：抛出异常时附加的 String 信息，new 可以完全不同，fillInStackTrace()会相同

6.5 异常链：常常会想要在捕获一个异常后抛出一个异常

6.5.1 **Throwable 的子类在构造器中都可以接受一个 cause 对象作为参数，这个 cause 就用来表示原始异常，这样通过把原始异常传递给新的异常，使得即使在当前位置创建并抛出了新的异常，也可能通过这个异常链追踪到异常发生的最初位置**

6.5.2 只有以下三种基本的异常类提供了带 cause 参数的构造器：Error Exception RuntimeException

如果要把其他类型的异常链连接起来，应该使用 initCause()方法而不是构造器

7、Java 标准异常

7.1 Throwable：**所有异常类的基类，可分为两种类型**：

Error 用来表示编译时和系统错误（除特殊情况外，一般不用你关心）

Exception 是可以被抛出的基本类型（程序员关心的基类型）

7.2 RuntimeException

属于运行时的异常类型由很多，它们会自动被 Java 虚拟机抛出，所以不必在异常说明中把他们列出来，这些异常都是从 RuntimeException 继承而来

7.3 **只能在代码中忽略（可以没有异常说明）RuntimeException（及其子类）类型的异常**，其他类型异常的处理都是由编译器强制实施的

8、finally

8.1 **无论 try 块中的异常是否抛出，它们都能得到执行。首先执行 try 块中的语句，若 try 块抛出异常且被该 try 块的 catch 捕获到，那么执行 catch 的语句，最后执行 finally 语句，否则执行完 try 块后直接执行 finally 语句**

8.2 由于 Java 中的异常不允许回到异常抛出的地点，可以使用 finally 来保证一些操作的执行

8.3 finnaly 可以用于：把除内存之外的资源恢复到它们的初始状态

8.4 在 return 中使用 finally，会在执行 return 后执行 finally 中的语句（return 在 try 语句中）
8.5 异常丢失：嵌套 try 语句中，内层抛出异常，不进行捕获，但是含有 finally 语句，若 finally 语句也抛出一个异常，那么会丢失第一个异常。
在 try 语句中抛出异常，在该 try 块语句中不进行捕获，在 finally 进行 return，也会造成异常丢失，而且无法被外层的 catch 捕获。

9、异常的限制
9.1 覆盖方法时，只能抛出基类方法的异常说明里列出的那些异常（保证基类使用的代码应用到派生类对象的时候，一样能够工作）
9.2 在继承和覆盖过程中，某个特定方法的“异常说明的接口”不是变大了而是变小了—这恰好和类接口在继承时的情况相反
9.3 异常限制对构造器不起作用，派生类构造器的异常说明必须包含基类构造器的异常说明，即异常说明接口可以变大

10、构造器
10.1 对于在构造阶段可能会抛出异常，并且要求清理的类，最安全的方法是使用嵌套的 try 子句：（构造成功则进入内层 try，一定会执行 finally 中的清理；若构造失败，则会被外部 try 的 catch 或更上一级的 catch 捕获，此时对象还没有构造成功，不需要执行清理）在创建需要清理的对象之后，立即进入一个 try-finally 语句块，（无论构造器是否声明抛出异常，都能保证在成功创建对象后会清理对象）

11、异常匹配
11.1 异常处理系统会按照代码的书写顺序找出最近的处理程序
11.2 查找的时候并不要求抛出的异常同处理程序所声明的异常完全匹配，捕获基类异常的处理程序也能捕获派生类异常
11.3 多个 catch 都能捕获的时候，只看书写顺序，匹配的精确程度并没有影响（抛出派生类的异常，第一个 catch 捕获基类异常，第二个 catch 捕获该派生类的异常，那么会被捕获基类异常的 catch 捕获成功，调换书写顺序，编译就不会通过）

12、其他可选方式
12.1 当异常发生时，正常的执行已经变得不可能或不需要了
12.2 异常处理的原则：只有在你知道如何处理的情况下才捕获异常
12.3 异常处理的目标：把错误处理的代码同错误发生的地点相分离
12.4 吞食则有害问题：被检查的异常（C++没有）：编译器强迫立刻写下代码来处理异常（此时并不知道如何处理异常），导致异常会被吞食
12.5 解决吞食则有害：将被检查的异常转化为不检查的异常：可以不写 try-catch 子句或异常说明，直接忽略异常，利用 RuntimeException 封装（P280），让它沿着调用栈往上冒泡，并有 getCause 捕获并处理特定异常

Chapter 13. 字符串

1、不可变 String

1.1 String 类中每一个看起来会修改 String 值的方法，实际上都是创建了一个全新的 String，以包含修改后的字符。

2、重载“+”与 StringBuilder

2.1 用于 String 的”+”和”+=”是 Java 中仅有的两个重载过的操作符

2.2 Java 不允许程序员重载任何操作符

2.3 编译器会自动引入 java.lang.StringBuilder 类，该类更高效

2.4 当为一个类编写 toString 时，如果使用循环，最好自己创建一个 StringBuilder 对象来构造结果，可以避免多次重新分配缓冲（如果不使用 StringBuilder 而直接用 String 进行操作，那么将在每个循环都会构建一次 StringBuilder）

2.5 StringBuilder 的方法：insert(),repleace(),substring(),reverse(),delete(),append(),toString

3、无意识的递归

3.1 Java 中所有类都继承自 Object，因此都有 toString()方法，Object 的 toString 会打印内存地址

3.2 对于没有明确表示继承自哪个基类的类，可以看做继承自 Objcet，因此调用 super.toString()会得到内存地址

4、String 上的操作(只列出常用的)

4.1 charAt(int)：取得 String 中该索引位置上的 char

4.2 length()：String 中字符的个数

4.2 toCharArray()：生成一个 char[]，包含 String 的所有字符

4.3 toLowerCase()：变为小写

4.4 toUpperCase()：变为大写

4.3 需要改变字符串的内容时，会返回一个新的 String 对象的引用，如果内容没有改变，那么返回原对象的引用

5、格式化输出

5.1 System.out.format()或 System.out.printf()模仿自 C 语言中的 printf

5.2 Formatter 类：可以看做翻译器，将格式化字符串与数据翻译成需要的结果

5.3 格式化说明符

%[argument_index$][flags][width][.precision]conversion

%[argument_index$][flags][width]conversion

%[flags][width]conversion

width 和 precision：用来指明尺寸大小，不同数据类型 precision 意义不同，浮点数的 precision 表示小数部分的位数（默认 6 位），整数没有该意义，会出发异常，浮点数长度当 precision 与 width 不匹配时，主要由 precision 决定

argument_index$：argument_index 是一个十进制整数，用于表明参数在参数列表中的位置，第一个参数用 1$引用

flags：修改输出格式的字符集，有效标识集取决于转换类型（如“-”将右对齐改为左对齐）

conversion：表明应该如何格式化参数的字符

5.4 类型转换

d:整数型　　c:Unicode 字符　　b:Boolean 值　　s:String　f:浮点数　　e:浮点数（科学计数）

x:整数（16 进制）　　h:散列码（十六进制）　　%:字符”%”

5.4.1 其中 b 转换，它对各种类型的转换都是合法的。对于 boolean 基本类型和 Boolean 对象，其转换结果是对应的 true 和 false。但是对于其他类型的参数，只要参数不为 null，那转换结果就是 true，即使是数字 0（与 C 语言 C++不同）

5.5 String.format：接受与 Formatter.format()方法一样的参数，但是返回一个 String 对象
String.format 内部：创建一个 Formatter 对象，将参数传递给 Formatter，然后返回 String 对象的引用

6、正则表达式
6.1 正则表达式提供了一种完全通用的方式，能够解决各种字符串处理相关的问题：匹配、选择、编辑以及验证
6.2 正则表达式就是以某种方式来描述字符串：如果一个字符串含有这些东西，那么它就是正在找的字符串
6.3 在其他语言中\\表示“我想要在正则表达式中插入一个普通的反斜线，请不要给他任何特殊意义”
Java 中\\表示“我要插入一个正则表达式的反斜线，所以其后的字符具有特殊意义”
6.4 反斜线\ 有两个含义：引用和转义
6.4.1 引用（转义字符）：反斜线用于引用其他将被解释为非转义构造的转义字符。
比如”“在 Java 中是转义字符，因此要引用”“需要变为\”\”
6.4.2 转义：反斜线可以用于引用转义构造
比如 n 原本没有任何其他意义，也就是非转义字符，前面加上\就代表换行符
在不表示转义构造的任何字符前使用反斜线都是错误的，如\v 等等
6.5 基础语法（左边都是正则表达式）
\\d：匹配一位数字
\\\\：匹配反斜线\　　\\\\在 String 作为\\
-?\\d+：可能有一个负号，后面跟着一位或多位数字
\\W：匹配一个非单词字符
\\w：匹配一个单词字符
+：匹配一个或多个之前的表达式　　\\+：普通加号
*：匹配 0 或多个之前表达式
？：匹配 0 或 1 个之前表达式
单斜线合法的仅有：\b \t \n \f \r \" \' \\
String 表示的路径中的\要写成\\
String[] args 赋值不需要加”“以空格分隔，且需要\\的地方可以用\（正则表达式：String 中编译错误的\b\w 等正确，写成\\b\\w 反而错误，路径：\和\\都正确）
Windows 下路径：F:\\Workspace\\JavaLearn\\src\\chapter18
Mac 下路径：/Users/HCF/Documents/workspace/JavaProgram/src/chapter18
6.6 String 类的方法 split(String regex)：将字符串从正则表达式匹配的地方切开
String 类的方法 replaceFirst(String regex)：替换与正则表达式匹配的第一个字符串
String 类的方法 replaceAll(String regex)：替换与正则表达式匹配的所有字符串
6.7 细节介绍 P288-299
6.8 Pattern 和 Matcher

```
interface CharSequence{
    charAt(int i);
    length();
    subsequence(int start,int end);
    toString();
}
```

CharBuffer、String、StringBuffer、StringBuilder 都实现了该接口

```
Pattern p=Pattern.complie(String regex); //regex：正则表达式
Matcher m=p.matcher(s);      //s：目标字符串
```

Pattern:
static Pattern compile(String regex)：将给定的正则表达式编译并赋予给 Pattern 类
Matcher matcher(CharSequence input)：生成一个给定命名的 Matcher 对象

String[] split(CharSequence input)： Pattern 对象将 CharSequence 对象按 regex（利用 Pattern.compile(regex)生成 Pattern 对象 p）分割成多个部分

String[] split(CharSequence input,int limit)：限制分割数量最多为 limit，达到 limit 就不再分割

String 也有 split 方法，功能与 Pattern 的 split 方法类似

String.split(String regex)

String.split(String regex,int limit)

Matcher:（目标字符串就是指创建 Matcher 对象时所绑定的字符串 Matcher m=p.matcher(s);）

boolean matches()：对整个目标字符串展开匹配检测，也就是只有目标字符串与模式字符串完全匹配时才返回真值

boolean find()：尝试在目标字符串里查找下一个匹配子串（能够像迭代器那样前向遍历输入字符串）

boolean find(int start)：接受整数作为参数，该整数表示目标字符串中字符的位置，并以其作为搜索的起点，可以不断重新设定搜索的起始位置

boolean lookingAt()：检测目标字符串（不必是整个字符串）的始部分是否能够匹配模式

组：A(B(C))D：组 0：ABCD 组 1：BC 组 2：C (a)(b)(c) 组 0：abc 组 1：a 组 2：b 组 3：c

public int groupCount()：返回该匹配器的模式中的分组数目，不包括第 0 组

public String group()：返回前一次 find 所匹配的子串

public String group(int i)：返回在前一次 find 所匹配的子串中匹配组 i 的子串

public int start(int group)：返回在前一次匹配操作中寻找到的组的起始索引

public int end(int group)：返回在前一次匹配操作中寻找到的组的最后一个字符索引加一

replaceFirst(String replacement)：用参数字符串替换掉第一个匹配的地方

replaceAll(String replacement)：用参数字符串替换掉所有匹配的地方

以上这两种替换方式，参数 replacement 只能是固定字符串，不如直接利用 String.replaceFirst(String regex,String replacement)和 String.replaceAll(String regex,String replacement)方便

appendReplacement(StringBuffer sbuf,String replacement)：参数 replacement 可以是表达式

渐进式的替换：

调用 find()会记录上一次调用 find()或 find(int i)的记录

调用 find(int i)会清除之前的 find()或 find(int i)的调用，本次调用作为第一次调用

1、首先，会将本次调用 find 所匹配的子串与上一次 find 调用所匹配的子串之间的子串存入 sbuf(若本次是第一次调用，那么会将本次调用 find 所匹配的子串之前的子串存入 sbuf)

2、然后，将本次调用 find 所匹配的子串用 replacement 替换后存入 sbuf

appendTail(StringBuffer sbuf)：会将最近一次调用 appendReplacement(StringBuffer sbuf,String replacement)之后的子串存入 sbuf。调用 find(int i)会重置目标子串，以 i 作为起始位置，并且清除之前 appendReplacement(StringBuffer sbuf,String replacement)调用的记录

reset()：可以将现有的 Matcher 应用于新的目标字符串

6.9 Pattern 标记

Pattern 类的 compile()方法还有另一个版本，接受一个标记参数，以调整匹配的行为

Pattern.CASE_INSENSITIVE：该模式对大小写不敏感

Pattern.MULTILINE：该模式（多行模式）中^和$分别匹配一行的开始和结束

可以通过（|）操作符组合多个标记功能

```
Pattern p=Pattern.compile(regex,Pattern. Pattern.CASE_INSENSITIVE| Pattern.MULTILINE);
```

7、扫描输入

7.1 StringReader：**将 String 转化为可读的流对象**

7.2 BufferReader：可以用 StringReader 初始化

7.2.1 BufferReader 的方法 readLine()将一行输入转化为 String 对象

7.3 Scanner 类：

7.3.1 **Scanner 的构造器可以接受任何类型的输入对象**

7.3.2 **所有的输入、分词以及翻译的操作都隐藏在不同类型的 next 方法中，所有基本类型（除了 char）都有 next 方法（所有的 next 方法，只有在找到一个完整的分词之后才会返回）**

7.4 Scanner 定界符

7.4.1 **默认情况下，Scanner 根据空白字符对输入进行分词，但是可以用正则表达式指定自己所需的定界符**

7.5 **next 可以配合正则表达式使用 next(regex)，调用 Scanner 的 match()方法就能获得匹配结果。注意：若正则正则表达式包含定界符，那么永远不可能匹配成功，因为，它仅仅针对下一个输入分词进行匹配，而包含正则表达式的子串会在两个分子中**

8、String StringBuilder、StringBuffer

Chapter 14. 类型信息

1、RTTI（Run-Time Type Identification）：在运行时，识别一个对象的类型

1.1 **Java 中所有的类型转换都是在运行时进行正确性检查的**

1.2 **运行时类型信息使得你可以在程序运行时发现和使用类型信息**

2、Class 对象

Class 对象的生成方式：

A、*Class*.forName("类名字符串") 包名+类名

B、类名.class

C、实例对象.getClass()

2.1 **Class 对象包含了与类有关的信息，Class 对象就是用来创建类的所有"常规"对象的**

2.2 Java 使用 Class 对象来执行 RTTI

2.3 **每个类都有一个 Class 对象，换言之，每当编写并编译了一个新类，就会产生一个 Class 对象（更恰当地说，是被保存在一个同名的.class 文件中）**

2.4 为了生成 Class 对象，运行这个程序的 Java 虚拟机（JVM）将使用被称为"类加载器"的子系统

2.5 类加载器子系统实际上可包含一条类加载器链，但只有一个原生类加载器（所谓的可信类，包括 Java API 类，他们通常是从本地盘加载的）**Class 对象仅在需要的时候才被加载**

2.6 **所有的类都是在对其第一次使用时，动态加载到 JVM 中的。当程序创建第一个对类的静态成员的引用时，就会加载这个类。（构造器也是静态方法，因此使用 new 操作符创建类的新对象也会被当做对类的静态成员的引用）**

2.7 **Java 程序在它开始运行之前并非被完全加载，各个部分是在必须时才加载的。这一点（动态加载）与传统语言（静态加载，如 C++）不同。**

2.8 **无论如何，想在运行时使用类型信息，就必须首先获得对恰当的 Class 对象的引用（不需要为了获得 Class 引用而持有该类型的对象，该类型的对象公有一个对应的 Class 对象）**

2.9 常用方法

2.9.1 *Class*.forName(String name)：**获取 Class 对象的引用，静态方法，其中 name 必须包含包名**

其副作用：如果类 name 还没有被加载就加载它。

当找不到类 name 时，抛出 ClassNotFoundException 异常（动态的，而?.class 是静态的，不必用 try）

2.9.2 Class.getSimpleName()：获取不含包名的类名

2.9.3 Class.getCanoicalName()：获取包含包名的类名(全限定的类名)

2.9.4 Class.getName()：获取包含包名的类名(全限定的类名)

2.9.5 Class.getSuperclass()：**查询其直接基类，返回的是基类的 Class 对象（即使该 Class 对象为泛型：某类型的 Class 对象，该函数返回的对象也不能用泛型，只能使用普通 Class 或超类 Class<?super A>）**

2.9.6 Class.newInstance()：创建类的对象**（该类必须含有默认构造器）**返回的是 Object 类型的对象

泛型 Class 对象的 newInstance()将返回该确切类型的对象

若泛型为 Class<? extends Pet>那么返回对象会自动转型为 Pet

会抛出两种异常 InstantiationException 和 IllegalAccessException

2.9.7 Class.isAssignableFrom(Class a)：该函数检查传入的参数 a 是否为该类型或其导出类

2.9.8 Class.isInstance(obj)：检查实例 obj 是否属于该类型

2.10 instanceof 关键字

result=object instanceof class：检查引用 object 的类型是否为 class，返回 boolean 值

2.11 为了使用类而做的准备工作实际包含三个步骤：

1、**加载：这是有类加载器执行的。该步骤将查找字节码（通常在 classpath 所指定的路径中查找，但这并非是必须的），并从这些字节码中创建一个 Class 对象。**

2、链接：在链接阶段将验证类中的字节码，为静态域分配存储空间，并且如果必须的话，将解析这个类的创建对其他类的所有引用。

3、初始化：如果该类具有超类(父类)，则对其初始化，执行静态初始化器和静态初始化块。初始化被延迟到了对静态方法或非常数静态域进行首次引用时才执行。

2.12 类字面常量(classname.class;)

Java 提供了另一种方法来生成对 Class 对象的引用，即使用类字面常量。在编译时就会受到检查（不需要置于 try 语句中，但是 forName 需要），根除了对 forName()方法的调用。仅适用.class 语法来获取对类的 Class 引用不会引发初始化，但是 forName()会引发初始化

2.13 如果一个 static final 值是“编译期常量”，那么这个值不需要被初始化就能被读取，否则对其的访问会首先强制进行整个类的初始化（并非仅仅只是该 static final 域）

2.14 如果一个 static 域不是 final 的，那么在对它访问时，总是要求在它被读取之前，要先进行链接（为这个域分配存储空间）和初始化（初始化该存储空间）

2.15 泛化的 Class 引用

1、Class 引用总是指向某个 Class 对象，它可以制造类的实例，并包含可作用于这些事例的所有方法代码

2、Class 引用还包含类的静态成员

3、Class 引用表示的就是它所指向的对象的确切类型，而该对象便是 Class 类的一个对象

2.16 泛型类引用只能赋值为指向其声明的类型；但是普通的类引用可以被重新赋值为指向任何其他的 Class 对象。通过使用泛型语法，可以让编译器强制执行额外的类型检查

泛型：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

2.17 为了使用泛化的 Class 引用时放松限制，可以使用通配符“？”：表示任何事物，且由于平凡的 Class，不会产生编译时的警告，表明并非碰巧或疏忽而是明确使用一个非具体的类引用

被限定为某种类型或该类型的任何子类型，可以将通配符与 extends 关键字结合

Class<? extends BaseClass>

2.18 超类（父类）：假定 A 继承 B。用指向 A 的 Class 类的引用的 getSuperClass()方法返回，但是只能用 Class<? super A>来接受返回值，不能用 Class<B>来接受返回值

2.19 新型转型语法 cast()方法：接受参数对象，并将其转型为 Class 引用的类型

2.20 总结：

1、Class 类在 Java 语言中定义一个特定类的实现。

2、Class 类的对象用于表示当前运行的 Java 应用程序中的类和接口。例如：每个数组均属于一个 Class 类对象，所有具有相同元素类型和维数的数组共享一个 Class 对象

3、我们自己无法生成一个 Class 对象（构造函数为 private），而这个 Class 类的对象是在各类被调入时，由 Java 虚拟机自动创建 Class 对象，或通过类装载器中的 defineClass 方法生成

4、我们可以把每个 Class 类的对象当做一个类在内存中的代理。而且在每个 Class 类对象中会有字段记录他引用的这个类的类加载器。

5、一个类代表要执行的代码，而数据则表示其相关状态。状态时常改变，而代码则不会。当我们将一个特定的状态与一个类相对应起来，也就意味着将一个类实例化。尽管相同的类对应的实例其状态千差万别，但其本质都对应着同一段代码。

3、类型转换前先做检查

3.1 RTTI 的形式：传统的类型转换（如”shape”）；代表对象的类型的 Class 对象

3.2 在 C++中，经典的类型转换（”shape”）并不是 RTTI，而是简单地告诉编译器将这个对象作为新的类型对待；而 Java 要执行类型检查，这通常被称为“类型安全的向下转型”，编译器允许自由地做向上转型的赋值操作（不需要显示的转型操作），但是编译器不允许非显式的向下转型操作。编译器将检查向下转型是否合理，即不允许向下转型为不是待转型类型的子类的类型）

3.3 RTTI 的第三种形式：instanceof。只能将其与命名类型比较而不能与 Class 对象比较

3.4 动态的 instanceof:ClassObject.isInstance(object)方法提供了一种动态的测试对象的途径:判断实例 object 是否属于 Class 对象 ClassObject 所代表的类型（换言之，可以与 Class 对象进行比较）

4、注册工厂

4.1 问题：为了实现 Pet 的随机生成的方法，会在某个静态域中存放生成 Class 对象的信息。但是如果添加了一个新类，想要用这个方法生成新类的对象，就必须把新类的 Class 对象信息存入这个静态域中。但是在类加载之前，这个静态域是不会运行的。而我们的目的就是为了获取对象，也就是需要在对象生成之前调用这个静态域（但在对象生成之前，该类米有加载，该静态域也不会运行）-->鸡和鸡蛋的问题

4.2 解决方法：将这个包含 Class 对象信息的列表置于一个中心的，位置明显的地方，继承体系的基类就是这样一个地方

4.3 instaneof 和 Class 的等价性

4.3.1 instanceof：你是这个类吗？或者你是这个类的派生类吗？

4.3.2 用==来比较 Class 对象，不会考虑继承结构

4.3.3 用 equals 来比较 Class 对象，会考虑继承结构

5、反射：运行时的类信息

5.1 RTTI 的限制：一个类型在编译时必须是已知的，这样才能使用 RTTI 识别它。也就是说，在编译时，编译器必须知道所有要通过 RTTI 来处理的类

5.2 场景介绍：从磁盘网络中获取了一串字节，并且被告知这些字节代表了一个类，那么怎么使用这样的类呢？（在编译时无法获知的类型该如何使用）

5.3 反射提供了一种机制—用来检查可用的方法，并返回方法名

5.3 在运行时获取类的信息的动机：希望在提供跨网络的远程平台上创建和运行对象的能力

5.4 Class 类与 java.lang.reflect 类库一起对反射的概念进行了支持，该类库包含了 Field、Method 以及 Constructor 类（每个类都实现了 Member 接口）

5.4.1 可以用 Constructor 创建新的对象；

5.4.2 用 Class.get() 和 Class.set()方法读取和修改与 Field 对象关联的字段；

5.4.3 用 Class.invoke()方法调用与 Class.Method 对象关联的方法

5.4.4 用 Class.getFiled()、Class.getMethods()、Class.getConstructors()返回字段、方法以及构造器的对象的数组

5.5 通过反射与一个未知类型的对象打交道时，JVM 知识简单地检查这个对象，看它属于哪个特定的类（就像 RTTI 那样）在用它做其他事情之前必须先加载那个类的 Class 对象。因此这个类的.class 文件对于 JVM 来说必须是可获取的：在本地机器上或者通过网络获取

5.6 RTTI 和反射机制的区别:

RTTI：编译器在编译时打开和检查.class 文件。（通过普通方式调用对象的所有方法）

反射机制：运行时打开和检查.class 文件。

5.7 类方法提取器：展示完整接口（这个类的定义中只有定义和被覆盖的方法，继承自基类的方法找不到）

5.7.1 Class. getMethods()、Class.getConstructors()方法返回 Method 对象数组和 Constructor 对象数组

5.8 如何使用反射机制，通过非默认构造器创建对象

```
Class<?>c = Class.forname("?.?")
Constructor<?>[] ctors=c.getConstructors();
Object obj=ctors.newInstance(Objcet[] initargs);//其中，只有声明为 public 的构造器才会被 Class. getConstructors()获取，编译器自动创建的默认构造器是 public 的
```

6、动态代理：P338

静态代理，一个代理类只能代理一个类，而动态代理可以动态绑定不同的的委托类

6.1 代理模式

抽象角色：声明真实对象和代理对象的共同接口

代理角色：代理对象角色内部含有对真实对象的引用（也许是接口的引用？），从而可以操作真实对象，同时代理对象提供了真实对象相同接口以便任何时候都能替代真实对象。同时,代理对象可以对象可以在执行真实对象操作时，附加其他的操作，相当于对真实对象进行封装。

真实角色：代理角色所代表的真实对象，是我们最终要引用的对象。

客户端通过代理来实现对真实角色的操作

6.2 代理是基本的设计模式之一，它是你为了提供额外的或不同的操作，而插入的用来代替“实际”对象的对象。

6.3 通过静态调用(interfacename)Proxy.newProxyInstance(loader, interfaces, h)可以创建动态代理，**其中 loader 是类加载器（代理类与真实类所实现的接口？），通过 Class.getClassLoader()获取；interfaces 是你希望该代理实现的接口列表（不能是类或抽象类），指的是委托类实现了其中一个或多个接口，因此代理类也必须实现，否则无法代理；h 是 InvocationHandler 接口的一个实现（代理类）**。

6.4 InvocationHandler 接口：只含有一个方法**（该方法会被自动调用）**

```
public Object invoke(Object proxy, Method method, Object[] args)
        throws Throwable;
```

proxy：代理类的对象

method：会传入该代理类正在尝试调用的方法

args：调用该方法的参数列表

对接口的调用会被定向为对代理类的调用

例子：

```
interface Business{
    void trade();
}
class Client implements Business{
    public void trade(){}
}
class Handler implements InvocationHandler{
    Object object;
    Handler(Object object){this.object=object;}
    public Object invoke(Object proxy, Method method,Object[] args){
        method.invoke(object,args) return null
    }
}
public static void main(String[] args){
    Business obj=(Business) Proxy.newProxyInstance(
                        Business.class.getClassLoader(),
                        new Class[]{Business.class},
                        new Handler(new Client))
    obj.trade();
}
```

7、空对象

7.1 内置的 null 表示缺少对象时，每次引用时都必须测试其是否为 null，这很枯燥，产生乏味的代码

7.2 空对象：可以接受传递给它的所代表的对象的消息，但是将返回表示为实际上并不存在任何“真实”对象的值。通过这种方式，你可以假设所有的对象都是有效的，而不必浪费编程精力去检查 null

7.3 通常，空对象都是单例

7.4 **某些地方仍然需要测试空对象，这与 null 没有差异（这些地方不允许存在没有意义的对象）**

7.5 **某些地方可以不用测试空对象，这与 null 有差异（如果允许对没有意义的对象进行操作，那么就不必费力进行检查了）**

7.6 P345 程序

8、接口与类型信息

8.1 **接口实现多态，若该接口绑定的对象的类型在当前是可见的，那么就可以通过强制转换成该对象的类型，从而调用接口所不具有的方法**

8.2 若想避免这种情况，可以利用包访问权限来抑制。P347

8.3 **利用反射机制（通过将 Method\Field\Constructor.setAccessible(ture);）可以忽略任何访问权限（即使是 private 的方法、匿名内部类、私有域等等）但是，只有 final 域遭遇修改时时安全的**

Chapter 15. 泛型

Java 中的泛型并不是纯粹的泛型，创建泛型的实例会很困难

1、与 C++比较

1.1 Java 的设计灵感来源于 C++

1.2 **有助于了解 Java 泛型的边界，只有知道某个技术不能做到什么，才能更好地做到所能做的**

2、简单泛型

2.1 促成泛型的原因：创造容器类

2.2 **泛型的主要目的：指定容器要持有什么类型的元素，而且由编译器来保证类型的正确性**

2.3 Java 泛型的核心概念：告诉编译器想使用什么类型，然后编译器帮你处理一切细节

2.4 一个元组类库：

2.4.1 return 只能返回单个对象

2.4.2 **元组：它是将一组对象直接打包存储于其中的一个单一对象（类似于 C++的 pair，tuple）**

3、泛型接口

3.1 **泛型也可以应用于接口，例如生成器，一种专门负责创建对象的类，这是工厂方法设计模式的一种应用。不过当生成器创建新的对象时，不需要任何参数，而工厂方法需要参数，也就是说生成器无需额外的信息就知道如何创建新对象**

3.2 Java 泛型的一个局限性：基本类型无法作为参数

4、泛型方法：

4.1 **基本原则：无论何时，只要你能做到，就应该尽量使用泛型方法。也就是说，如果使用泛型方法可以取代整个类泛型化，那么久应该只是用泛型方法**

4.2 **对于 static 方法，无法访问泛型类的类型参数，因此，如果 static 方法需要使用泛型能力，就必须使其成为泛型方法**

4.3 定义泛型方法：只需将泛型参数列表置于返回值之前

```
public <T> void f(T x){}
```

4.4 **在使用泛型类时，必须在创建对象的时候指定类型参数的值，而使用泛型方法的时候，通常不必指明参数类型（与 C++类似）。因为编译器会为我们找出具体的类型，这成为类型参数推断**

4.5 **使用泛型会导致更多的代码，如 List<String> lst=new LinkedList<String>(); 在 new 时又注明了泛型参数，可以利用泛型方法，让类型参数推断帮助简化（通过对返回类型推断泛型参数，不提倡！！！）**

4.6 类型推断只对赋值操作有效，其他时候并不起作用

4.7 **在泛型方法中可以显示指明类型，在点操作符与方法名之间插入尖括号**

如果在定义该方法的类的内部，必须在点操作符之前使用 this:this.<T>f()

如果使用 static 的方法，必须在点操作符之前加上类名，如：如 *Arrays*.<String>asList()

4.8 可变参数与泛型方法，需要输入同一类型的多个参数，void f(T...args)

4.9 通过泛型函数简化元组的使用，不必指定类型了，靠编译器对输入参数进行的参数推断

4.10 Set 实用工具(net.mindview.util，起始没啥用，要用自己随便写)

5、匿名内部类可以与泛型很好地结合

6、构建复杂模型（多层容器）

7、擦除的神秘之处

7.1 泛型公有一个 Class 类的对象

7.2 在泛型代码内部，无法获得任何有关泛型参数类型的信息（可以暂时理解为类型 T 的 Class 对象）
7.3 Java 泛型是使用擦除来实现的，这意味着当你使用泛型时，任何具体的类型信息都被擦除了，唯一知道的就是你在使用一个对象，因此 List<String> 和 List<Integer>在运行时事实上是相同的类型；这两种形式都被擦除成它们的“原生”类型，即 List
7.4 泛型类型参数将擦除到它的第一个边界,如<T extends A>，T 类型必须是 A 类型及其导出类
7.4.1 只有当你希望使用的类型参数比某个具体类型（以及它的所有子类型）更加“泛化”时，也就是说，当你希望代码能够跨多个类工作时，使用泛型才有所帮助
7.5 Java 中的擦除并非是一个语言特性，而是 Java 实现中的一种折中（泛型并非 Java 一开始就有的），擦除减少了泛型的泛化性，但是泛型在 Java 中仍然是有用的
7.6 在基于擦除的实现中，泛型类型被当做第二类型处理，即不能再某些重要的上下文环境中使用的类型。泛型类型只有在静态类型检查时期才出现，在此之后，程序中所有泛型类型都将被擦除，替换为它们的非泛型上界，未指定边界的情况下擦除为 Object
7.7 擦除的核心动机：它使得泛化的客户端可以用非泛化的类库来使用。
7.8 问题：如 T get();这样的函数返回的对象的 Class 的对象不是 Object 而是确定的泛化时的类型
回答：因为这样的函数并没有用到类型信息，所以是可以的
如果函数体中包含 instanceof,new T()等等需要知道具体类型信息就不行
7.9 边界处的动作：即使擦除在方法或类内部移除了有关实际类型的信息，编译器仍旧可以确保在方法或类中使用的类型的内部一致性
7.9.1 边界：即对象进入和离开方法的地点，这些地点正是编译器在编译期执行类型检查并插入转型代码的地点。
7.9.2 泛型中的所有动作都发生在边界处---对传递进来的值进行额外的编译期检查，并插入对传递出去的值得转型

8、擦除的补偿
8.1 擦除丢失了泛型代码中执行某些操作的能力，任何运行时需要知道确切信息的操作（比如 arg instanceof T, new T (), 都无法工作。
8.2 有时必须通过引入类型标签（传递 Class 对象）来对擦除进行补偿
8.3 创建类型实例：new T()的尝试是无法实现的，部分原因是因为擦除，而另一部分原因是由于编译器不能验证 T 具有默认构造器
8.3.1 Java 解决方案是传递一个工厂对象，并用它来创建新实例，最便利的工厂对象就是 Class 对象
8.3.2 传递显示的工厂，避免传入 Class 对象作为工厂时，但该类却没有默认构造器，newInstance()会产生异常
8.3.3 模板方法设计模式？？？P382
8.4 泛型数组
8.4.1 Java 不能创建泛型数组，一般的解决方案是在任何想要创建泛型数组的地方使用 ArrayList
8.4.2 问题在于数组将跟踪它们的实际类型，而这个类型是在数组被创建时确定的，也就是说，即使在泛型内部创建数组，并且进行类型转换，如：(T[]) new Object[sz],在运行时其实际类型仍然是 Object
8.4.3 没有任何方式可以推翻底层的数组类型，它只能是 Object[]，除非传递类型标记

9、边界
9.1 因为擦除移除了类型信息，所有，可以用无边界泛型参数调用的方法只是那些可以用 Objcet 调用的方法。但是如果能够将这个参数限制为某个类型子集，那么就可以用这些类型子集来调用这些方法，为了执行这种限制，Java 泛型重用了 extends 关键字
即为了调用不属于 Object 的方法 f：T.f() 若不加边界，则会编译失败

10、通配符

10.1 数组的特殊行为：可以向导出类型的数组赋予基类型的数组引用（**注意这里指的是这种情况：BasedClass[] ary=new ExportClass[N];即包含一个向上转型**），编译器允许，但是在运行时会抛出类型为 ArrayStoreException 的异常。
10.2 泛型的主要目标之一就是将 10.1 这种错误已入到编译期
10.3 有时想要在两个类型之间建立某种类型的向上转型关系，这正是通配符所允许的
10.4 **List<? extends BasedClass>并不意味着可以持有任何类型的 BasedClass。*不能向其添加对象*，即使是声明持有的对象也不行，因为编译器将直接拒绝对参数列表中涉及通配符的方法（如 add()）的调用，像 contains()和 indexOf(),参数类型是 Object，不涉及通配符，编译器允许调用**
10.5 超类型通配符<? super MyClass> <? super T>，可以安全地传递一个类型对象到泛型类型中，即，有了超类通配符就能向 Collecation 写入了。
10.6 超类型边界放松了在可以向方法传递的参数上所作的限制
10.7 写入时：超类型边界 **void write(List<? super T> list)**是安全的，即可以向其写入该类 T 以及该类 T 的**子类**，因为**该泛型参数类型是类型 T 的某个基类**，因此向其写入 T 以及 T 的子类是安全的
10.8 读取时：子类型边界 **T read(List<? extends T> list)**是安全的，即可以向其读取该类 T 以及该类 T 的**子类**，因为**该泛型参数类型是类型 T 的某个子类**，因此读取该参数类型并向上转型为 T 是安全的
10.9 **无界通配符<?>，可以被认为是一种装饰。如 List<?>表示具有某种特定类型的非原生 List**
10.10 **捕获转换：如果向一个使用<?>的方法传递原生类型，那么对编译器来说，可能会推断出实际的类型参数，使得这个方法可以回转并调用另一个使用这个确切类型的方法。这项技术也能用于写入，但是需要同时传递一个具体类型，才能使得捕获转换能够工作**

11、问题
11.1 不能将基本类型用作类型参数
11.2 **一个类不能同时实现一个泛型接口的两种变体，因为由于擦除的原因，这两个变体会成为相同的接口，因此不能实现同一个接口两次。**
11.3 **使用带有泛型类型参数的转型(T)a 或 instanceof 不会有任何效果，由于擦除 T 被擦除到第一个边界，默认情况下是 Object，因此将 Object 转型为 Object 没有任何效果**
11.3.1 通过泛型来转型：List.class.cast() 注意，不能用具体类型(如 List<String>.class.cast())
11.4 重载，由于擦除的原因，重载方法将产生相同的类型签名，**因此，当被擦除的参数不能产生唯一的参数列表时，必须提供明显有区别的方法名**
如 void f(List<T> v){} 与 void f(List<W> v){}

12、自限定的类型（古怪的循环 CRG）
12.1 **简单版本：** class A extends Genericity<A>**创建一个类，它继承自一个泛型类型，这个泛型类型接受了我的类的名字作为其泛型参数，Java 中的泛型关于参数和返回类型，因此它能够产生使用导出类作为其参数和返回类型的基类**
12.2 **CRG 的本质：基类用导出类代替其参数，这意味着泛型基类变成了一种其所有导出类的公共功能模板，但这些功能对于其所有参数和返回值，将使用导出类型**
- **对于导出类覆盖其基类的函数**
  - 当基类中返回类型是基类本身是，导出类型可以直接将其改为导出类型
  - 当基类中参数类型是基类类型时，导出类型不能将其改为导出类类型，必须采用古怪的循环，才能将其参数改为导出类类型

12.3 **自限定将采用额外的步骤，强制泛型当做其自己的边界参数来使用。**
class SelfBounded<T extends SelfBounded<T>>{}
自限定要做的，就是强制要求在继承关系中，像下面这样使用这个类
class A extends SelfBounded<A> 而不允许 class A extends SlefBounded<B>其中 B 不是自限定
在简单自限定中，class A extends SlefBounded<B>不会编译失败！！！
12.3.1 **自限定用于泛型方法：可以防止这个方法被应用于自限定参数之外的任何事物上**
12.4 **对于 T get():自限定泛型事实上将产生确切的导出类型作为其返回值**
12.5 **自限定的价值在于它们可以产生协变参数类型---方法参数类型会随子类而变化 P409**

对于基类中的方法，若其输入参数或输出为基类类型，那么在导出类中继承得到的方法将是输入参数或输出为导出类类型

13、动态类型安全

13.1 java.util.Collections 提供了一组类型检查的静态方法，来保证使用旧式代码的正确性

Collections.checkedCollection()、*Collections*.checkedList()、*Collections*.checkedSet()

Collections.checkedMap()、*Collections*.checkedSortedMap()、*Collections*.checkedSortedSet()

这些方法以容器作为第一个参数，并希望强制要求的类型(Class 对象）作为第二个参数

13.2 对于原生容器，当你将对象从容器中取出时，才会通知你出现问题，通过静态类型检查方法，可以发现试图插入的不良对象

14、异常

14.1 由于擦除的原因，将泛型应用于异常时是非常受限的，catch 语句不能捕获泛型类型的异常，因为在编译期和运行时都必须知道异常的确切类型。泛型类也不能直接或间接继承自 Throable

14.2 类型参数可能会在一个方法的 throws 子句中用到，这使得可以编写随检查型异常的类型而发生变化的泛型代码

15、混型

15.1 其最基本的概念是混合多个类的能力，以产生一个可以表示混型中所有类型的类

15.2 C++中，混型就是继承自其类型参数的类 template<typename T> Class A : public T{};

15.3 与接口混合：每个混入的类型都要求在混型类中有一个相应的域，在混型类中必须编写所有方法（实现接口），当混型变得复杂时，代码数量会急剧增加

15.4 使用装饰器模式：

15.4.1 装饰器经常用于满足各种可能的组合，而直接子类化会产生过多的类

15.4.2 装饰器模式使用分层对象来动态透明地向单个对象中添加责任，装饰器指定包装在最初的对象周围的所有对象都具有相同的基本接口

15.4.3 装饰器是通过组合和形式化结构（可装饰物/装饰器层次结构）来实现的，而混型是基于继承的，因此可以将基于参数化类型的混型当做是一种泛型装饰器机制，这种机制不需要装饰器设计模式的继承结构

15.4.4 使用装饰器所产生的对象类型是最后被装饰的类型，尽管可以添加多个层，但是最后一层才是实际的类型，因此只有最后一层的方法是可视的，因此对于装饰器来说，其明显的缺陷是它只能有效的工作于装饰中的一层（最后一层），所以，装饰器只是对有混型提出的问题的一种局限的解决方案

15.5 与动态代理混合

15.5.1 可以使用动态代理来创建比装饰器更贴近混型模型的机制

15.5.2 由于动态代理的限制，每个被混入的类都必须是某个接口的实现

16、潜在类型机制

16.1 指定泛型类型的边界，以安全地调用代码中的泛型对象上的具体方法。这是对“泛化”概念的一种明显限制，因为必须限制你的泛型类型，使它们继承自特定的类，或者实现特定的接口。

16.2 Phthon（动态类型语言）和 C++（静态类型语言）支持潜在类型机制，可以实现以下操作：

```
template<typename T> void f(T anything){
    anything.dosomething();
}
```

C++会静态检查 anything 调用的合法性，而 Phthon 会动态检查

16.3 Java 不支持上述特性，由于擦除，要想实现特定方法的调用，必须提供泛型边界，但是一旦提供了边界，实际上与用一个普通的类无区别（传入接口类型的对象），丧失了泛型，因为必须实现指定的接口

17、对缺乏潜在类型机制的补偿

17.1 Java 仍旧可以创建正真的泛型代码，但是需要付出一些额外的努力

17.2 反射提供一些有趣的可能性（通过传入 T 类型的对象，该对象可以用 getClass()方法获取其 Class 对象，这个不算是需要知道特定类型信息的操作），它将所有的类型检查都转移到了运行时。虽然反射的实现比非反射要慢一些，但这并不失为一个解决方案

17.3 当并未碰巧拥有正确的接口时，代码往往不够真正的泛化（因为需要设定边界）比如要对 T 类型的对象调用 add 函数，必须将类型边界限定到 Collection。P422-423

17.4 用适配器仿真潜在类型机制

17.4.1 矛盾：Java 没有潜在类型机制，但我们又需要像潜在类型机制这样的东西去编写能够跨类边界应用的代码（也就是更泛化的代码）

17.4.2 潜在类型机制：我不关心我再这里使用的类型，只要它具有这些方法即可。实际上，潜在类型机制创建了一个包含所需方法的隐式接口，它遵循这样的规则：如果我们手工编写了必须的接口（Java 没有帮我们完成这些）那么它就应该能够解决问题

17.4.3 为了实现：static <T>void f(T t,Object...args){t.add(args);} 首先传入的参数必须实现了某个特定的接口（其他潜在类型机制中的隐式接口在 Java 中显式化）。比如 interface Addable<T>{void add();}。用一个适配器去适配任意的（具有 add 方法或者类似 add 方法）的类型 T，使其实现了 Addable 接口。转为 static<T> void f(Addable<T> t,Object...args){t.add(args);}需要为每个类型 T 编写适配器以适配接口 Addable，这就是额外的工作

与传入实现了该接口的类的对象不同，这里要求的是 T 并不一定实现了接口 Addable，添加了适配器后，更加地泛化。

Chapter 16. 数组

1、数组的特殊性

1.1 树组与其他种类的容器之间的区别有三方面：效率、类型、和保存基本类型的能力。

1.2 数组就是一个简单的线性序列，这使得元素访问非常快速，代价就是数组对象的大小被固定，并且在其生命周期不可改变

1.3 数组优于泛型之前的容器是在于可以创建持有具体类型的数组，通过编译器检查来防止插入错误的类型或抽取不正当的类型

2、数组是第一级对象

2.1 无论使用哪种类型，数组标识符其实只是一个引用，指向在堆中创建的一个真实对象，这个数组对象可以保存指向其他对象的引用。

2.2 可以作为数组初始化语法的一部分隐式地创建对象，或者用 new 表达式显式创建对象

2.3 只读成员 length 是数组对象的一部分（唯一可以访问的字段或方法）

2.4 对象数组保存引用，基本类型数组直接保存值

2.5 聚集初始化操作必须在数组定义的位置，如 int[] a={1,2,3};

3、返回一个数组

3.1 对于 C 和 C++这样的语言来说，返回一个数组比较困难。首先只能返回数组指针，并且如果返回数组是新建数组的话，必须使用 new 创建的指针（局部变量在函数调用结束后将被析构），因此使得控制数组的声明周期变得困难，而且容易造成内存泄漏，必须在弃用之后 delete 掉

4、多维数组

4.1 可以通过花括号将每个向量分隔开，每对花括号括起来的集合都会把你带到下一级数组

4.2 *Arrays*.deepToString()方法可以将多维数组转换为多个 String

4.3 基本类型数组的值在不进行显示初始化的情况下，会被自动初始化，对象数组会被初始化为 null

4.4 粗糙数组：int[][][] a=new int[1][][];（不允许 int[] a=new int[];）

5、数组与泛型

5.1 通常，数组与泛型不能很好地结合，不能实例化具有参数化类型的数组

```
ArrayList<Integer>[] ayr=new ArrayList<Integer>[5];//Illegal
```

因为擦除会移除参数类型信息，而数组必须知道它们所持有的确切类型

5.2 可以参数化数组本身的类型（T[] a)

5.3 编译器不允许实例化泛型数组，但是允许创建对这种数组的引用（如 List<String>[] ls;）

5.4 尽管不能创建实际的持有泛型的数组对象，但是可以创建非泛型数组然后将其转型

5.5 数组是协变类型的,任何除基本类型之外的类型的数组都是 Object 的数组

```
String[] a=new String[2];  Object[] obj=a;  obj[0]=new LinkedList<Integer>();
//编译器不会有错误，但是运行时抛出异常
```

5.6 如果创建一个 A[]，Java 在编译器和运行时都会要求只能将 A 类型对象置于该数组中，但是如果创建 Object[]，那么可以将除基本类型之外的任何对象置于数组中

5.7 不能用 foreach 语句来初始化泛型数组

6、创建测试数据

6.1 Java 标准类库 Arrays 有一个 fill()方法，用同一个值填充各个位置或者指定的区域，对于对象而言，复制同一个引用进行填充（指向同一个对象）

6.2

7、Arrays 实用功能

7.1.1 equals(a,b)：比较两个数组是否相等

7.1.2 deepEquals(a,b)：比较两个多维数组是否相等
7.1.3 fill(a,formindex,toindex,val)：填充数组
7.1.4 sort(a)：用于数组排序
7.1.5 binarySearch()用于在已经排序的数组中查找元素（二分法）
7.1.6 toString()：产生数组的 String 表示
7.1.7 hashCode()：产生数组的散列码
7.1.8 Arrays.<T>asList()

7.2 复制数组
System.arraycopy(a,astart,b,bstart,length)：比 for 循环复制要快很多，越界会抛出异常
astart 和 bstart 表示起始索引，从 0 开始计
System.arraycopy 不会执行自动包装盒自动拆包，两个数组必须具有相同的确切类型
7.3 **数组比较，对象数组的相等是基于内容的（通过 Object.equals()比较）**
7.4 **数组元素的比较，Java 有两种方式来提供比较功能**

其一：实现 java.lang.Comparable 接口（这个接口很简单，只有一个 compareTo()一个方法。此方法接受另一个 Object 为参数，如果当前对象*小于（优先于）*参数返回负值；如果*相等*返回零；如果当前对象*大于（滞后于）*参数则返回正值。小于等于大于由返回结果控制，也就是可以自己定义怎么样算小于等于大于

其二：Comparator 接口，有两个方法，其一是 compare(),另一个是 equals()，不一定要实现 equals()

```
class T implements Comparator<T>{
    public int compare(T t1,T t2){
        return t1<t2? -1:(t1==t2?0:1)//t1 小于 t2 时，t1 排在 t2 前面
        return t1>t2? -1:(t1==t2?0:1)//t1 大于 t2 时，t1 排在 t2 前面
                                                                    }
}
```

Chapter 17. 容器深入研究

1、完整的容器分类法 P459

2、填充容器，与 java.util.Arrays 一样，fill 也只是复制同一个对象引用来填充整个容器

2.1 所有的 Collection 子类型都有一个接受另一个 Collection 对象的构造器

2.2 Map 生成器：net.mindview.util 提供的方法

2.3 Abstract 类：每个容器都有其子集的 Abstract 类，提供了容器的部分实现。

2.4 享元：使得对象的一部分可以被具体化，因此与对象中所有事物都包含在对象内部不同，我们可以在更加高效的外部表中查找对象的一部分或整体

类：*Map*.Entry:每个 Map.Entry 对象不存储实际的键和值，而只是存储索引

public Set<Map.Entry<U,V>> Map.entrySet(){}

3 Collection 的功能方法

boolean add(T)

boolean addAll(Collection<? extends T>)

void clear()

boolean contains(T)

Boolean containsAll(Collection<?>)

boolean isEmpty()

Iterator<T> iterator()

Boolean remove(Object)

boolean removeAll(Collection<?>)

Boolean retainAll(Collection<?>)

int size()

Object[] toArray()

<T> T[] toArray(T[] a)//返回类型会与 a 的类型相同

4、可选操作

4.1 执行各种不同的添加和移除的方法在 Collection 接口中都是可选操作（UnsupportedOperationException)

4.2 未获支持的操作：最常见的未获支持的操作都来源于背后由固定尺寸的数据结构支持的容器

如这样初始化一个 List： List<String> L=Arrays.asList("A B C D E").split(" ")

4.3 **Arrays.asList()会生成一个 List，它基于一个固定大小的数组，仅支持那些不会改变数组大小的操作，对它而言是有道理的。任何会引起底层数据结构的尺寸进行修改的方法都会产生一个 UnsupportedOperationException 异常，以表示对未获支持操作的调用**

4.4 应该把 Arrays.asList()的结果作为构造器的参数传递给任何 Collection(或者使用 addAll()方法）

6、Set 和存储顺序

6.1 Set 每个元素必须是唯一的，Set 不保存重复元素。加入 Set 的元素必须定义 equals()方法以确保对象的唯一性。Set 与 Collection 有完全一样的接口

6.2 SortedSet(接口):保证元素处于排序状态（并非指插入顺序）

Object firtst():返回容器第一个元素

Object last():返回容器最后一个元素

SortedSet subSet(fromElement,toElement)生成 Set 的子集，范围左闭右开

SortedSet headSet(toElement) 生成 Set 的子集，由小于 toElement 的元素组成（右开）

SortedSet tailSet(fromElement) 生成 Set 的子集，右大于或等于 fromElement 的元素组成（左闭）

fromElement,toElement 都是关键字类型，而非索引

SortedSet<T> ss=new TreeSet<T>();

6.3 若要元素保持插入时的顺序，则应该用 LinkedHashSet 来实现

7、队列

7.1 Java 中仅有的两个实现是 LinkedList 和 PriorityQueue(用最小堆可实现）

7.2 优先队列通过实现 Comparable<T>(public int compareTo(T t))接口而进行排序

7.2 双向队列（不常用）：Java 中没有显示的双向队列的接口，LinkedList 中包含支持双向队列的方法

8、Map

8.1 Map 的实现有 HashMap、LinkedHashMap、TreeMap、WeakHashMap、ConcurrentHashMap、IdentityHashMap。首选 HashMap

8.2 SortedMap 接口：TreeMap 是现阶段的唯一实现（基于红黑树）有一下额外的功能

Comparator comparator()：返回当前 Map 使用的 Comparator，或者返回 null 表示自然方式排序

T firstKey()：返回 Map 中第一个键

T lastKey()：返回 Map 中最末一个键

SortedMap subMap(fromKey,toKey)：生成此 Map 的子集，范围由 fromKey（包含）到 toKey（不包含）

SortedMap headMap(toKey)：生成此 Map 的子集，由键小于 toKey 的所有键值对组成

SortedMap tailMap(fromKey)：生成此 Map 的子集，由键大于或等于 fromKey 的所有键值对组成

8.3 LinkedHashMap：为了提高速度 LinkedHashMap 散列化所有元素，但在遍历键值对时，却又以元素的插入顺序返回键值对，采用基于访问的最近最少使用（LRC）算法（没有被访问过的（可被看作需要删除的）元素就会出现在队列的前面

9、散列与散列码

9.1 Object 的 hashCode()方法是默认采用对象的地址计算散列码的，而且默认的 Object 的 equals()方法也是比较对象的地址

9.2 覆盖 hashCode()方法的同时必须覆盖 equals()方法，否则可能产生非预期的结果

9.3 equals()方法必须满足 5 个条件

1）自反性：x.equals(x)==true

2）对称性：x.equals(y)==y.equals(x)

3）传递性：x.equals(y)==true and y.equals(z)==ture 则 x.equals(z)==true

4）一致性：只要信息没有改变，无论多少次，结果必须一致

5）对于非 null 的 x：x.equals(null)必须返回 false

9.4 Map.entrySet()方法产生一个 Map.Entry 对象集，而 *Map*.Entry 是一个接口，用于描述依赖于实现的结构，如果想要创建自己的 Map 类型，必须同时定义 Map.Entry 的实现。entrySet()的恰当实现应该在 Map 中提供视图

9.5 对于现代处理器来说，除法和求余是最慢的操作，使用 2 的整数次方长度的散列表，可以用掩码代替除法，由于 get 是使用最多的操作，求余数的操作是其开销最大的部分，而使用 2 的整数次方可以消除此开销

9.6 覆盖 hashCode

9.6.1 首先无法控制 bucket 数组的下标值得产生，这个值依赖于具体 HashMap 对象的容量，而容量的改变与容器的充满程度和负载因子有关

9.6.2 hashCode()生成的结果，经过处理后成为桶位的下标

9.6.3 设计 hashCode()最重要的因素：对同一个对象调用 hashCode()都应该生成同样的值（可以基于对象的内容，或者对象的地址（默认））

9.6.3 另一个因素：好的 hashCode()应该产生分布均匀的散列码

9.6.4 基本步骤

1）给 int 变量 result 赋予某个非零值常量

2）为对象内每个有意义的域 f（即每个可以做 equals()操作的域）计算出一个 int 散列码 c

3）合并计算得到的散列码
4）返回 result
5）检查 hashCode()最后生成的结果，确保相同的对象有相同的散列码

10、选择接口的不同实现
10.1 四中类型容器接口：Map,List,Set,Queue
10.2 **Hashtable、Vector、Stack 是过去遗留下来的类，在新版本中最好别用！**

Chapter 18. Java I/O 系统

1、File 类

1.1 "FilePath"（文件路径）对这个类来说是个更好的名字

1.2 FilenameFilter 接口：boolean accept(File dir, String me)

这个接口的目的在于把 accept()方法提供给 File.list 使用，使 File.list()可以回调 accept(),进而决定哪些文件包含在列表中。

File.list()方法会为此目录对象下的每个文件名调用 accept()，来判断该文件是否包含在内

FilenameFilter 实现的的一般模式如下：

```
private Pattern pattern = Pattern.compile(regex);
    public boolean accept(File dir, String name) {//此处 dir
的是定义 File 对象时指定的路径，而不是包含 name 的文件的 File 对象，因此这里无法用于判断是否是文件夹(isDirectory())
        return pattern.matcher(
          new File(name).getName()).matches();
    }
```

若要获取到在指定路径下的所有文件文件夹以及遍历其子文件夹，需要通过 File.listFiles()获取当前层的 File 对象数组，然后遍历数组判断是否是文件，递归调用将所有文件都搜索出来

1.3 File path=new File("") ：路径为当前工程的路径（eclipse 平台下）

当前工程路径：/Users/HCF/Documents/workspace/JavaProgram

假设在该路径下有一个文件 test.txt

File path=new File("test.txt")//可以定位到该文件

File path=new File("JavaProgram/test.txt")//不能定位到该文件

File path=new File("/Users/HCF/Documents/workspace/JavaProgram/test.txt")//可以定位到该文件

以上所说的能定位到文件指的是，将 File 对象作为 FileInputStream 的构造器的输入参数，能够正确地打开文件

开头加上"/"(mac 系统下) 表示是一个绝对路径，而不是工程路径下的子路径

开头没有"/"表示是工程路径下的子路径

直接传入 String 作为 FileInputStream 构造器的参数也满足上述规则

1.4 File

仅仅只代表

1.5 常用方法:

File.canWrite()：返回文件是否可写。

File.canRead()：返回文件是否可读。

File.getName()：返回文件名称

File.getParent()：返回父目录路径

File.getPath()：返回文件路径

File.length()：返回文件长度？？？

File.list()：返回文件或目录清单(String[])//不会继续访问子文件夹

File.listFiles()：返回文件或目录清单(File[])//不会继续访问子文件夹

File.exists()： 返回文件是否存在(boolean)

File.isDirectory()：返回该路径指示的是否是文件夹

File.isFile()：判断该路径指示的是否是文件（*.*）

File.renameTo()：文件更名

File.mkdirs()：生成指定的目录（当文件不存在时）

2、输入和输出

2.1 “流“屏蔽了实际的 I/O 设备中处理数据的细节

2.2 我们很少使用单一的类来创建流对象，而是通过叠合多个对象来提供所期望的功能（装饰器设

计模式）

2.3 流的迷惑之处：创建单一的结果流，却需要创建多个对象

2.4 I/O 分类成输入和输出，任何自 InputStream 或 Reader 派生而来的类都含有 read()的基本方法；同理，任何自 OutputStream 或 Write 派生而来的类都含有名为 write()的基本方法。但我们很少使用，之所以存在是因为别的类可以使用它来创建更有用的接口

2.5 InputStream

2.5.1 InputStream 的作用是用来表示那些从不同数据源产生输入的类

2.5.2 数据源包括：1、字节数组；2、String 对象；3、文件；4、“管道”，工作方式与实际的管道相似，从一端输入，从另一端输出；5、一个有其他种类的流组成的序列，以便我们可以将它们收集合并到一个流内；6、其他数据源

2.6 InputStream 类型

表格 181

类	功能	构造器参数/如何使用
ByteArrayInputStream	允许将内存缓冲区当做 InputStream	构造器参数：缓冲区，字节将从中取出 使用：作为一种数据源，将其与 FilterInputStream 对象相连已提供有用接口
StringBufferInputStream	将 String 转化成 InputStream	构造器参数：字符串（底层实现实际使用 StringBuffer） 使用：作为一种数据源，将其与 FilterInputStream 对象相连已提供有用接口
FileInputStream	用于从文件中读取信息	构造器参数：字符串，表示文件名、文件或 FileDescriptor 对象 使用：使用：作为一种数据源，将其与 FilterInputStream 对象相连已提供有用接口
PipedInputStream	产生用于写入相关 PipedOutputStream 的数据，实现管道化概念	构造器参数：PipedOutputStream 使用：作为多线程中数据源，将其与 FilterInputStream 对象相连已提供有用接口
SequenceInputStream	将两个或多个 InputStream 对象转换成单一 InputStream	构造器参数：两个 InputStream 对象或一个容纳 InputStream 对象的容器 Enumeration 使用：使用：使用：作为一种数据源，将其与 FilterInputStream 对象相连已提供有用接口
FilterInputStream	抽象类，作为“装饰器”的接口，装饰器为其他 InputStream 提供有用接口	见表格 18 3 见表格 18 3

2.7 OutputStream 类型

表格 182

类	功能	构造器参数/如何使用
ByteArrayOutputStream	在内存中创建缓冲区。所有送往“流”的数据都要放置在此缓冲区	构造器参数：缓冲区初始化尺寸（可选） 使用：用于指定数据的目的地，将其与 FilterOutputStream 对象相连已提供有用的接口
FileOutputStream	用于将信息写至文件	构造器参数：字符串，表示文件名、文件或 FileDescriptor 对象 使用：用于指定数据的目的地，将其与 FilterOutputStream 对象相连已提供有用的接口
PipedOutputStream	任何写入其中的信息都会自动作为相关 PipedInputStream 的输出。实现管道化概念	PipedInputStream 使用：指定用于多线程的数据的目的地，将其与 FilterOutputStream 对象相连已提供有用的接口
FilterOutputStream	抽象类，作为“装饰器”的接口。其中“装饰器”为其他 Output 提供有用功能	见表格 18 4 见表格 18 4

3、添加属性和有用的接口

3.1 **装饰器：抽象类 filter 是所有装饰器类的基类，装饰器必具有和它所装饰的对象相同的接口**

3.1.1 **装饰器的缺点：在编写程序时，它给我们提供了相当多的灵活性，我们可以很容易地混合匹配属性，但是它同时也增加了代码的复杂性。**

3.1.2 **Java I/O 类库操作不便的原因在于：我们必须创建许多类---“核心”I/O 类型加上所有的装饰器，才能得到我们所希望的单个 I/O 对象**

表格 183

类	功能	构造器参数/如何使用
DataInputStream	与 DataOutputStream 搭配使用，因此可以按照可移植方式从流读取基本数据类型(int ,char,long)等，其对应的方法就是 DataInputStream.readInt(...)/.readBoolean(...)/readLong(...)等等	构造器参数：InputStream 使用：包含用于读取基本类型数据的全部接口
BufferedInputStream	使用它可以防止每次读取时都得实行实际写操作。代表使用缓冲区	构造器参数：InputStream 使用：本质上不同恭接口，只不过是向进程中添加缓冲区所必须的。与接口对象搭配

表格 184

类	功能	构造器参数/如何使用
DataOutputStream	与 DataInputStream 搭配使用因此可以按照可移植方式向流中写入基本类型数据(int char,long)等，其对应的方法就是 DataOutputStream.writeInt(...)/.writeBoolean(...)/writeLong(...)等等	构造器参数：OutputStream 使用：包含用于写入基本类型数据的全部接口
PrintStream	用于产生格式化输出。其中 DataOutputStream 处理数据的存储，PrintStream 处理数据的显示	构造器参数：OutputStream，可以使用 boolean（选用）指示是否在每次换行时清空缓冲区 使用：对 OutputStream 对象的 final 封装
BufferedOutputStream	使用它以避免每次发送数据时都要进行实际的写操作。代表使用缓冲区，可调用 flush()清空缓冲区	构造器参数：OutputStream，可以指定缓冲区大小（可选） 使用：本质上并不提供接口，只不过向进程添加缓冲区所必须

4、Reader 和 Write

4.1 并不是用于替代 InputStream 和 OutputStream 的类

4.2 InputStream 和 OutputStream 在以面向字节形式的 I/O 中仍可以提供极有价值的功能

4.3 Reader 和 Write 提供兼容 Unicode 与面向字符的 I/O 功能

4.4 **为了实现来自于“字节”层次结构中的类和“字符”层次结构中的类结合起来使用，必须要用到“适配器”类：InputStreamReader 可以把 InputStream 转换为 Reader；OutputStreamWrite 可以把 OutputStream 转为 Write**

4.5 如何选择：尽量尝试使用 Reader 和 Write

表格 185

来源	去处
InputStream	Reader(适配器：InputStreamReader）
OutputStream	Writer(适配器：OutputStreamWriter)
FileInputStream	FileReader
FileOutputStream	FileWriter
StringBufferInputStream(已弃用)	StringReader
\	StringWriter
ByteArrayInputStream	CharArrayReader
ByteArrayOutputStream	CharArrayWriter
PipedInputStream	PipedReader
PipedOutputStream	PipedWriter

表格 186

来源	去处
FilterInputStream	FilterReader
FilterOutputStream	FilterWriter(抽象类，没有子类)
BufferedInputStream	BufferedReader(也有 readLine())
BufferedOutStream	BufferedWriter
DataInputStream	DataInputStream(当需要使用 readLine()时,应该使用 BufferedReader)
PrintStream	PrintWriter
LineNumberInputStream(已弃用)	LineNumberReader
StreamTokenizer	StreamTokenizer
PushbackInputStream	PushbackReader

5、自我独立的类：RandomAccessFile

5.1 **RandomAccessFile 不是 InputStream 或 OutputStream 继承层次结构中的一部分。它实现了 DataInput 和 DataOutput 两个接口（InputStream 和 OutStream 也实现了这两个接口），除此之外，RandomAccessFile 与 InputStream 或 OutputStream 基层结构没有任何联系**

5.2 从本质上来说，**RandomAccessFile 的工作方式类似于把 DataInputStream 和 DataOutStream 结合起来使用**，另外添加了一些方法

getFilePointer()：用于返回当前所处的文件位置

seet()：用于在文件内移至新位置

length()：用于判断文件的最大尺寸

5.3 **其构造器含有第二个参数："r"：随机读　"rw"：随机读写。打开文件后，"指针"指向 0，但是文件的内容仍是上次保存下的内容，如果要追加内容，要使用 seek 定位到文件尾**

5.4 方法

RandomAccessFile.seek(long pos);//移动到指定位置（每个位置代表一个字节）

RandomAccessFile.readInt(readDouble...);//读取当前位置的对应数据，并将“指针”移动该数据类型对应的字节数

RandomAccessFile.writeInt(readDouble...);//在当前位置写入数据，并将"指针"移动该数据类型对应的字节数

RandomAccessFile.setLength(long length);//更改文件大小

RandomAccessFile.length();//返回文件大小

6、I/O 流经典使用方式（**包装成缓冲是为了提高读写效率！，通常包装成缓冲**）

6.1 **缓冲输入（读取）文件**

6.1.1 想要打开一个文件用于字符输入（读取文件的内容）

6.1.2 **首先创建一个与指定文件连接的 FileReader，实际上，我们会用 BufferedReader 将其包装起来用以缓冲输入**

BufferedReader：（构造器接受 Reader 对象（**Reader 是抽象类**），文件的话就是 FileReader）

提供 readLine()方法读取一行（会将换行符删掉），当返回 null 意味着到达了文件尾

6.2 **从内存输入（读取）**

6.2.1 StringReader：（构造器接受一个 String）

提供 read()方法，每次读取一个字符，以 int 的形式返回

6.3 **格式化的内存输入（读取）**

6.3.1 要读取格式化数据，可以使用 DataInputStream 而不是 Reader 类

6.3.2 DataInputStream 是一个**面向字节**的 I/O 类

6.3.3 **创建一个 ByteArrayInputStream，将其装饰成 DataInputStream**

DataInputStream in=new DataInputStream(new ByteArrayInputStream(s.getBytes()));

必须为 ByteArrayInputStream 提供字节数组，String 包含了 getBytes()方法可以提供字节数组，

所产生的 ByteArrayInputStream 是一个合适传递给 DataInputStream 的 InputStream
由于读取的是字节，不能用读取行那样用读取到的 String 是否为空来判断是否到了文件尾，因为 byte 是基本类型，因此提供了 avaliable()方法返回尚未读取的字节数量，来作为循环条件（等价得作为是否到了文件尾的判断）

6.4 基本的文件输出

6.4.1 FileWrite 对象可以向文件写入数据。首先创建一个与指定文件连接的 FileWriter。实际上我们会用 BufferedWriter 将其包装起来用以缓冲输出，再次，为了提供格式化机制，BufferedWriter 被装饰成 PrintWriter。

```
PrintWriter out=new PrintWriter(new BufferedWriter(new FileWriter(filePath)));
```

6.5 存储和恢复数据

6.5.1 PrintWriter 可以对数据进行格式化，以便人们的阅读。但是为了输出可供另一个流恢复的数据，需要用 DataOutputStream 写入数据，并用 DataInputStream 恢复数据(这些指定类型的读写方法是这两个装饰类所特有的方法，而 InputStream 和 OutputStream 不具有这些方法)

6.5.2 首先创建一个与指定文件链接的 FileOutputStream，然后将其包装成 BufferedOutputStream，最后用 DataOutputStream 对其进行装饰

```
DataOutputStream out=new DataOutputStream(new BufferedOutputStream(
                        new FileOutputStream(filename)));
```

6.5.3 如果用 DataOutputStream 写入数据，Java 保证我们可以用 DataInputStream 准确地读取数据，无论平台多么不同。

6.5.4 当使用 DataOutputStream 时，写字符串并且让 DataInputStream 能够恢复它的唯一可靠做法就是使用 UTF-8 编码。

6.5.5 为了保证所有读方法都能正常工作，必须知道流中数据的确切位置，因为读取时的类型必须保证和写入时的类型一致才不会出错：1、为文件中的数据采用固定的格式；2、将额外的信息保存到文件中

6.6 判断是否到达了文件尾

对于面向字节的 I 文件流：FileInputStream

```
FileInputStream.available()!=0
```

对于面向字符的 I 文件流：FileReader

```
FileReader.readLine()!=null
FileReader.read()!=-1
```

对于 RandomAccessFile：

```
FileReader.readLine()!=null
FileReader.read()!=-1
```

6.7 文件的打开

FileOutputStream 或者 FileWriter：路径存在文件，则打开，不存在，先创建后打开
FileInputStream 或者 FileReader：路径存在文件，则打开，不存在抛出 FileNotFoundException

7、文件读写的实用工具：net.mindview.util.TextFile

8、标准 I/O

8.1 从标准输入中读取：Java 提供了 System.in、System.out、System.err。其中 System.out、System.err 都被包装成了 PrintStream，但 System.in 是一个未经包装的 InputStream。

8.2 通常我们用 readLine()一次一行地读取输入，为此，将 System.in 包装成 BufferedReader 来使用需要我们首先用 InputStreamReader 将 System.in 转换成 Reader

```
BufferedReader stdin=new BufferedReader(new InputStreamReader(System.in));
```

8.3 System.out 是一个 PrintStream，而 PrintStream 是一个 OutputStream。PrintWriter 有一个可以接受 OutputStream 作为参数的构造器。因此，只要需要，就可以使用该构造器把 System.out 转化为

PrintWriter。（这里使用具有两个参数的构造器，第二个参数设置为 true 以便开启自动清空缓存功能，否则，你将看不到输出

PrintWriter out=new PrintWriter(System.out,true);

8.4 标准 I/O 重定句：**如果我们突然开始在显示器上创建大量输出，而这些输出滚动的太快以至于无法阅读时，重定向输出就显得极为重要**；**对于想重复测试某个特定用户的输入序列的命令程序来说，重定向输入就很有价值。**

System.setIn(InputStream)

System.setOut(PrintStream)

System.setErr(PrintStream)

I/O 重定向操纵的是字节流而不是字符流，因此使用的是 InputStream 和 OutputStream 而不是 Reader 和 Writer

10、新 I/O (nio)

10.1 **通道和缓冲器：我们并没有和通道交互，而是与缓冲器交互，并把缓冲器派送到通道。通道要么从缓冲器获得数据，要么向缓冲器发送数据**

10.2 **FileInputStream、FileOutputStream、RandomAccessFile 被重写用以产生 FileChannel，这些是字节操纵流，与底层的 nio 性质一致。**

10.3 Reader、Writer 这类字符模式类不能用于产生通道，但是 java.nio.channels.Channels 提供了方法可以在通道中产生 Reader 和 Writer

10.4 **唯一直接与通道交互的缓冲器是 ByteBuffer，可以存储未加工字节的缓冲器。通过告知分配多少存储空间来创建一个 ByteBuffer 对象(对于只读访问，必须显示使用 allocate 方法来分配 ByteBuffer)。**

ByteBuffer.rewind();**//返回到数据的开始部分**

ByteBuffer.flip();**//让缓冲器做好让别人读取字节的准备**

ByteBuffer.clear();**//让缓冲器做好让别人写字节的准备**

ByteBuffer.allocate(int size);**//创建指定大小的 ByteBuffer 对象**

ByteBuffer.wrap(byte[]);**//将字节数组包装进 ByteBuffer**

capacity()	返回缓冲区容量
clear()	position=0;limit=capacity;
flip()	limit=position;position=0
limit()	返回 limit 值
limit(int limit)	设置 limit 值
mark()	mark=position
reset()	position=mark
position()	返回 position 值
position(int pos)	设置 position 值
remainning()	返回 limit-position
hasRemaining()	若有介于 position 和 limit 的元素，返回 true
get()	返回 position 指向的字节值，并将 position+1
put()	向 position 指向的字节写入值，并将 position+1
get(int pos)	返回指定 pos 指向的字节的值，position 与 pos 无关，且不做变化
put(int pos)	向指定 pos 指向的字节写入值，position 与 pos 无关，且不做变化

10.5 FileChannel fc; ByteBuffer buff=*ByteBuffer*.allocate(1024);

fc.write(*ByteBuffer*.wrap("Some text".getBytes()));//wrap 将字节数组包装到 ByteBuffer 中

fc.read(buff);

FileChannel 的 read 和 write 方法只能作用于 ByteBuffer

10.6 **当需要从缓冲器中读取数据时，必须调用 ByteBuffer.filp()让缓冲器做好让别人读取字节的准备;**

当需要向缓冲器中写入数据时，必须调用 ByteBuffer.clear()让缓冲器做好让别人写字节的准备

10.7 FileChannel.transferTo(long position,long count WritableByteChannel target)

FileChannel.transferFrom(ReadableByteChannel src,long position,long count)

可以将两个通道相连完成复制的操作

```
in.transferTo(0,in.size(),out);
out.transferFrom(in,0,in.size());
```

10.8 获取基本类型，向 ByteBuffer 插入基本类型数据的最简单方法是：利用 asCharBuffer、asShortBuffer、asLongBuffer、asFloatBuffer、asDoubleBuffer 等获得该缓冲器上的视图，然后使用视图的 put 方法添加对应的数据（只有 Short 需要加类型转换）；再用 ByteBuffer 中相应的 getShort()、getDouble()等方法获取数据即可。

```
ByteBuffer bb=ByteBuffer.allocate(1024);
bb.asCharBuffer().put("Hello");
bb.asShortBuffer().put((short)1);
bb.asLongBuffer().put(222);
bb.asIntBuffer().put(111);
bb.asDoubleBuffer().put(123.33);
...
```

10.9 视图缓冲器：可以让我们通过某个特定的基本数据类型的视窗查看其底层的 ByteBuffer，ByteBuffer 依然是实际存储数据的地方，“支持”着前面的视图，对视图的任何修改都会映射成为对 ByteBuffer 中数据的修改

视图缓冲器还允许我们一次一个地或成批的（数组）读写基本类型值

10.10 字节存放次序

big endian（高位优先）

little endian（低位优先）

如一个 short 型的数据

00000000 01100001 以高位优先形式读取为 97，以低位优先形式读取为 24832

用 ByteBuffer.order(ByteOrder.BIG_ENDIAN)

ByteBuffer.order(ByteOrder.LITTLE_ENDIAN)来设置优先形式

10.11 内存映射文件 P563

内存映射文件允许我们创建和修改那些因为太大而不能放入内存的文件。于是，我们可以假定整个文件都在内存中，而且完全可以把它当做非常大的数组来访问，这极大地简化了用于修改文件的代码。

为了既能读又能写，先由 RandomAccessFile 开始，获得该文件上的通道，然后调用 map 产生 MappedByteBuffer（继承自 ByteBuffer），这是一种特殊的直接缓冲器，必须指定映射文件的初始位置和映射区域的长度，这意味着我们可以映射某个文件的较小部分

```
MappedByteBuffer out=new RandomAccessFile("...","rw").getChannel().
                    map(FileChannel.MapMode_READ_WRITE,position,length);
```

因此对于缓冲器 out 的读写，会映射到文件的读写，相当于建立了缓冲器到文件的桥梁。

之前的读写必须借助 FileChannel.read(buffer)和 FileChannle.write(buffer) buffer 是 ByteBuffer

映射文件的所有输出（向文件写入）必须用 RandomAccessFile，而不能用 FileOutputStream。

11、压缩

11.1 压缩类库是按字节方式而不是字符方式处理的，是属于 InputStream 和 OutputStream 继承层次结构中的一部分。

11.2 用 GZIP 进行压缩，适合于只向对单个数据流（而不是一系列互异数据）进行压缩。

11.3 压缩类的使用十分直观，直接将输出流封装成 GZIPOutputStream 或 ZipOutputStream，并将输入流封装成 GZIPInputStream 或 ZipInputStream 即可

```
BufferedOutputStream out=new BufferedOutputStream(
                new GZIPOutputStream(
                        new FileOutputStream(path)));
```

压缩类只接受面向字节的 I/O 类型，然后将其再包装成缓冲类型

读取时,当然也可以再转成 Reader 类型

```
BufferedReader in2=new BufferedReader(
                new InputStreamReader(
                    new GZIPInputStream(
                        new FileInputStream(path))));
```

12、对象序列化

概念

1、如果对象能够在程序不运行的情况下仍能存在并保存信息，将会非常有用，

2、**Java 的对象序列化将那些实现了 Serializable 接口的对象转换成一个字节序列，并能够在以后将这个字节序列完全恢复为原来的对象。**这一过程甚至可通过网络进行，这意味着序列化机制能自动弥补不同操作系统之间的差异。不必担心数据的任何细节

3、对象的序列化实现轻量级持久性，持久性意味着一个对象的生存周期并不取决于程序是否在执行。它可以生存与程序的调用之间

4、对象序列化是为了支持两种主要特性：Java 远程方法调用(Remote Method Invocation RMI)

5、**对象序列化特别"聪明"的一个地方在于：它不仅保存了对象的全景图，而且能追踪对象内所包含的所有引用，并保存那些对象，接着又对对象内包含的每个这样的引用进行追踪，依次类推(蠕虫)**

6、**对象序列化是基于字节的，因此要使用 InputStream 和 OutputStream 继承层次结构**

7、使用方法

```
ObjectOutputStream out=new ObjectOutputStream(new FileOutputStream(@path));
```

// 将 OutputStream 包装成 ObjectOutputStream，这里是 FileOutputStream，还可以是其他 OutputStream 类型，

```
out.writeObject(<Object:something>);//something 必须是实现了 Serializable 的类，才能被序列化
out.close();
ObjectInputStream in=new ObjectInputStream(new FileInputStream(path));
String s=(String)in.readObject();//这里是将读取得到的对象调用了 toString 转为 String
```

12.1 **寻找类：将一个对象序列化，通过网络作文文件传送给另一台计算机，那么另一台计算机上无法只通过该文件内容来还原这个对象，必须保证 Java 虚拟机能够找到相关的.class 文件**

12.2 序列化的控制：

1、不希望对象中的某一部分被序列化。或者，一个对象被还原后，某子对象需要重新创建，从而不必将该子对象序列化

2、**可以通过实现 Externalizable 接口代替实现 Serializable 接口来对程序化过程进行控制（这个接口继承了 Serializable 接口，同时添加了两个方法，writeExternal()和 readExternal()，这两个方法会在序列化和反序列化还原过程中被自动调用，以便执行一些特殊的操作。**

3、对于 Serializable 对象：**对象完全以它存储的二进制位为基础来构造，不会调用构造器。**

对于 Externalizable 对象：**所有普通的默认构造器都会被调用（包括在字段定义时的初始化），然后调用 readExternal()，必须注意：所有默认的构造器都会被调用**

4、由于 Externalizable 对象的构造只会调用默认构造器，如果该对象内部含有字段，那么字段需要另外处理，例子如下：

字段：int i;String s;

```
public void writeExternal(ObjectOutput out) throws IOException{
        out.writeObject(s);
```

```
        out.writeInt(i);
    }

    public void readExternal(ObjectInput in) throws IOException,ClassNotFoundException{
        s=(String)in.readObject();
        i=in.readInt();
    }
```

5、transient（瞬时）关键字：**当我们对序列化进行控制时，某个特定子对象不想让 Java 序列化机制自动保存与恢复，比如敏感信息（密码）。即使这些信息是 private 的，一经序列化处理，人们就可以通过读取文件或者拦截网络传输的方式来访问到它。**

一种方法是：将类实现为 Externalizable，没有任何东西可以自动序列化，可以在 writeExternal()内部只对所需部分进行显示序列化

另一种方法：类仍然只实现 Serializable，但是给特定不想被序列化的字段加上 transient 标识

6、Externalizable 的替代方法

实现 Serializable 接口，并添加（而非覆盖或实现）名为 writeObject(),readObject()的方法，一旦对象被序列化或者反序列化还原，就会自动地调用这两个方法。

注意：这两个方法的书写必须具有准确的方法特征签名：

```
private void writeObject(ObjectOuputStream stream) throws IOException;
private void readObject(ObjectInputStream stream) throws IOxception;
```

这两个方法被定义成 private，只有这个类的其他成员才能调用，然而并没有调用，而是传入的参数 ObjectOutputStream 或 ObjectInputStream 的对象对其进行调用

另一方面，由于这两个方法是 private 的，因此不会是接口的一部分，也不会是覆盖（private 不可以被覆盖），因为接口自动是 public 的，因此必须要完全尊姓其方法特征签名，效果和实现了接口一样

12.3 使用持久性

1、**序列化：存储程序的一些状态，以便随后可以很容易地将程序恢复到当前状态。**

2、问题抛出：**如果两个对象中包含一个指向第三个对象的引用，那么将这两个对象序列化再反序列化还原，那么这第三个对象只会出现一次吗？如果将这两个对象序列化成独立的文件，然后在代码不同地方对他们进行反序列化还原，会怎么样呢**

3、问题解答：**只要将任何对象序列化到单一流中，就可以恢复出与我们写出时一样的对象网（比如，两个对象包含第三个对象的引用，反序列化还原时，第三个对象只有一个），并且无论对象在被序列化时处于什么状态，都可以被写出（比如在写出第一个对象和写出最后一个对象期间改变这些对象的状态）**

4、Class 对象时 Serializable 的，因此只需直接对 Class 对象序列化，可以很容易地保存 static 字段

13、XML

13.1 对象序列化的重要限制是它只是 Java 的解决方案，只有 Java 程序才能反序列化这种对象，一种更具互操作性的解决方案是将数据转化成 XML 格式，使得可以被各种各样的平台语言使用

14、Preferences

14.1 Preferences API 与对象序列化相比，前者与对象持久性更密切，因为它可以自动存储和读取信息。不过它只能用于小的，受限的数据集合---我们只能存储基本类型和字符串，并且每个字符串的存储长度不能超过 8K，用于存储和读取用户的偏好以及程序配置项的设置

Chapter 19. 枚举类型

混淆点：
enum 是一个关键字，而不是一个类型
enum 实例：Enum 的对象
enum 中的构造器用于构造 enum 实例即 Enum 对象
enum 中定义的方法是 enum 中定义的实例所具有的方法（每个实例内部还能各自定义方法）
enum 可以拥有多个实例

1、基本 enum 特性
1.1 创建 enum 时，编译器会为你生成一个相关的类，继承自 java.lang.Enum，并提供一系列方法
1.2 Enum.values()方法：可以遍历 enum 实例，该方法返回 enum 实例的数组，该数组中的元素严格保持其在 enum 中声明时的顺序，因此你可以在循环中使用 values()返回的数组
1.3 Enum.ordinal()方法：返回每个 enum 实例在声明时的次序，从 0 开始
1.4 Enum.name()方法：返回 enum 实例声明时的名字，与使用 toString()方法效果相同
1.5 *Enum*.valueOf()方法：根据给定名字返回相应 enum 实例，若不存在给定名字的实例，抛出异常
Integer.valueOf(int i)()：返回 Integer 的对象
String.valueOf(T t)：返回值为 t 的字符串（T 可以是 int,char.....）
1.6 Enum 类实现了 Comparable 接口，所以它具有 compareTo 方法，同时实现了 Serializable 接口
1.7 **import static <packagename>.<eumnname>.*; 可以将 enum 实例的标识符带入当前的命名空间，否则需要 enum 类型名来修饰其实例如 SomeThing.A;**

2、向 enum 中添加新方法
2.1 **除了不能继承自一个 enum 之外，基本上可以将 enum 看做一个常规类**
2.2 一般来说，我们希望每个枚举类型实例能够返回对自身的描述，而不仅仅只是默认的 toString()实现，toString()只能返回枚举实例的名字。为此可以提供一个构造器，专门处理这个额外的信息，然后添加一个方法，返回这个描述信息。
2.3 **如果打算定义自己的方法，必须在 enum 实例序列的最后添加一个分号，同时 Java 要求必须先定义 enum 实例。**若在定义 enum 实例之前定义了任何方法或属性，那编译时就会得到错误信息
2.4 case 后面可以直接跟上 enum 实例的名字

3、switch 中的 enum

4、values()的神秘之处
4.1 Enum 中并没有 values 方法,values()方法是由编译器插入到 enum 定义中的 static 方法，所以，如果你将 enum 实例向上转型为 Enum，那么 values 方法就不可访问了，但是可以通过反射 getEnumConstatcts()方法取得 enum 的所有实例

5、**所有的 enum 都继承自 java.lang.Enum 类，因此 enum 不能再继承其他类**
6、**随机选取 enum 实例：利用泛型，是的这个工作更一般化**
public static <T extends Enum<T>> T random(Class<T> ec){...}
其中<T extends Enum<T>>表示 T 是一个 enum 实例，利用 Class<T>作为参数，可以利用 Class.getEnumConstants()来获取 enum 实例数组

7、使用接口组织枚举
7.1 我们希望使用子类将一个 enum 中的元素进行分组
7.2 **在一个接口内部，创建实现该接口的枚举。以此将元素分组，已达到将枚举元素分类组织的目的。**
7.3 **接口的作用是讲 enum 组合成一个公共类型**

8、使用 EnumSet 代替标志

8.1 Java SE5 引入 EnumSet，通过 enum 创建一种替代品，以替代传统的基于 int 的"标志位"，这种标志是用来表示某种"开/关"信息。不过，使用这种标志，最总操作的只是一些 bit，而不是这些 bit 想要表达的概念，因此很容易写出令人难以理解的代码

8.2 **EnumSet 的设计充分考虑到了速度因素，就其内部而言，它就是讲一个 long 值作为比特向量，所以 EnumSet 非常高效，其优点是：它在说明一个二进制位是否存在时，具有更好的表达能力，并且无需担心性能**。

8.3 EnumSet 的基础是 long，一个 long 值有 64 位，而一个 enum 实例只需一位 bit 表示其是否存在，也就是说在不超过一个 long 的表达能力的情况下，你的 EnumSet 可以应用最多不超过 64 个元素的 enum。但是超过 64 个元素后，会增加下一个 long 值

8.4 **向 EnumSet 添加实例的顺序并不重要，因为其顺序取决于 enum 实例定义时的次序**

9、EnumMap

9.1 EnumMap 是一种特殊的 Map，他要求其中的键(key)必须来自一个 enum。由于 enum 本身的限制，EnumMap 在内部由数组实现，因此 EnumMap 的速度很快

9.2 只能用 enum 的实例作为键来调用 put

9.3 **enum 的每个实例作为一个键总是存在的，如果没有用 put 来存入相应值得花，其对应值就是 null**

9.4 优点：EnumMap 允许程序员改变值对象，而常量相关的方法在编译器就被固定了

10、常量相关方法

10.1 Java 允许程序员为 enum 实例编写方法，从而为每个 enum 实例赋予各自不同的行为。

10.2 需要为 enum 定义一个或多个 abstract 方法，为每个 enum 实现该抽象方法。

11、enum 状态机

11.1 枚举类型非常适合用来创建状态机。一个状态机可以具有有限个限定的状态，它通常根据输入，从一个状态转移到下一个状态，也可能存在瞬时状态，而一旦任务执行结束，状态机就会立刻离开瞬时状态。

Chapter 20. 注解

Chapter 21. 并发

- Java 是一种多线程语言，并且提出了并发问题

21.1. 并发的多面性

1、并发编程令人困惑的主要原因：使用并发时需要解决的问题有多个，而实现并发的方式也有多重，并且在这两者之间没有明显的映射关系。

2、用并发解决的问题大体上可以分为“速度”和“设计可管理性”两种

21.1.1. 更快的执行

1、**并发通常是提高运行在*单处理器*上的程序的性能**

2、阻塞：如果程序中某个任务因为该程序控制范围之外的某些条件（通常是 I/O）而导致不能继续执行，那么我们就说这个任务或线程阻塞了。

3、如果使用并发来编写程序，那么当一个任务阻塞时，其他任务还能继续执行。

4、从性能的角度看，如果没有任务会阻塞，那么在单处理器上使用并发就没有任何意义。

5、在单处理器系统中的性能提高的常见事例是**事件驱动的编程**

6、**使并发最吸引人的一个原因就是要产生具有可响应的用户界面。(事件驱动，用周期性地判断)**

7、**实现并发最直接的方式就是在操作系统级别使用进程。**

8、**进程是运行在它自己的地址空间内的自包容的程序**

9、**多任务操作系统可以通过周期性地将 CPU 从一个进程切换到另一个进程，来实现同时运行多个程序（进程），这使得每个进程“看起来”在其执行过程中停停歇歇，操作系统会将进程相互隔开，进程彼此不会产生干扰。但与此相反，Java 所使用的并发系统会共享诸如内存和 I/O 这样的资源，因此编写多线程（线程会共享资源）程序最基本的困难在于协调不同进程驱动的任务之间对这些资源的使用，以使得这些资源不会同时被多个任务访问**

10、**对于进程来说：他们之间没有任何通信的必要，他们都是完全独立的**

10、**Java 采取了更加传统的方式，在顺序型语言的基础上提供对*线程*的支持**

10、线程机制是在由执行程序表示的单一进程中创建任务

21.1.2. 改进代码设计

1、在单 CPU 机器上使用多任务的程序在任意时刻仍旧只在执行一项工作，从理论上讲，肯定可以不用任何任务而编写出相同的程序。但是并发可以使得程序设计极大地简化。

2、**Java 的线程机制是抢占式的，这表示调度机制会周期性的中断线程，将上下文切换到另一个线程，从而为每个线程都提供时间片，使得每个线程都会分配到数量合理的时间去驱动它的任务。**

21.2. 基本的线程机制

1、**并发编程使我们可以将程序划分为多个分离的，独立运行的任务。通过使用多线程机制，这些独立任务（也被称为子任务）中的每一个都将有执行线程来驱动。一个线程就是在进程中的一个单一的顺序控制流。**

21.2.1. 定义任务

1、线程可以驱动任务，由实现 Runnable 接口来提供（编写 run 方法）

2、通常 run()被写成无限循环的形式，这意味着，除非有某个条件使得 run()终止，否则它将永远运行下去。

3、*Thread*.yield()表示：我已经执行完生命周期中最重要的部分了，此刻正式切换给其他任务执行一段时间的大好时机。

4、**当从 Runnable 导出一个类时，它必须具有 run 方法，但是这个方法并无特殊之处---它不会产生任何内在的线程能力，要实现线程行为，你必须显式地将一个任务附着到线程上**

5、**main 函数也是一个线程**

21.2.2. Thread 类

1、将 Runnable 对象转变为工作任务的传统方式是把它提交给一个 Thread 构造器

2、**任何线程都可以启动另一个线程**

3、线程调度机制是非确定性的，每次运行顺序可能不完全一样

4、Thread 在创建时都“注册”了自己，有一个对它的引用，而且在它的任务退出其 run()并死亡之前，垃圾回收期无法清除它。（创建 Thread 时不捕获其引用，也不必担心被垃圾回收器回收）

21.2.3. Executor

1、我们可以用 Executor 来代替 Thread 对象
2、ExecutorService：具有服务生命周期的 Executor，（ExecutorSevice 继承了 Executor，两者都是接口），知道如何构建恰当的上下文来执行 Runnable 对象
3、ExecutorService exec=

- *Executors*.newCachedThreadPool();
- Executors.newFixedThreadPool(int);**//限制线程的数量**
- Executors.newSingleThreadExecutor();

4、Executor.execute(Runnable)：为任务创建一个线程
5、Executor.shutdown()：**防止新任务被提交给这个 Executor，当前线程将继续运行在 shutdown()被调用前提交的所有任务。**
6、Executor.shutdownNow()：**发送一个 interrupt()调用给它启动的所有线程。**
6、线程池
在任何线程池中，现有线程在可能的情况下都会被自动复用

- CachedThreadPool：**//为每个任务都创建一个线程。在程序执行过程中，通常会创建于所需数量相同的线程，然后在它回收旧线程时停止创建新线程，因此它是合理的 Executor 的首选**
- FixedThreadPool：**//限制线程的数量**
- SingleThreadExecutor：**//线程数量为 1，如果向 SingleThread 提交多个任务，那么这些任务将会排队，每个任务会在下一个任务开始之前运行结束。因此 SingleThreadExecute 会序列化所提交的任务，并维护它自己的悬挂任务队列。**

21.2.4. 从任务中产生返回值

1、Runable 是执行工作的独立任务，但它
返回任何值。
2、**若希望任务完成时返回值，可以实现 Callable<T>接口而不是 Runnable 接口**
3、Callable<T>是一种具有类型参数的泛型，类型参数 T 表示的是从方法 call()中返回的值，并且必须使用 ExecutorService.submit(Callable)方法调用它
4、**submit(...)方法会返回 Future<T>对象，他用 Callable 返回结果的特定类型进行了参数化。可以用 isDone()方法来查询 Future<T>是否已经完成，当任务完成时，它具有一个结果，可以用 Future.get()方法来获取该结果**

21.2.5. 休眠

1、影响任务行为的简单方法就是调用 sleep()，这将使任务中止执行给定的时间。
2、sleep 的调用会抛出 InterruptedException 异常
3、语法 TimeUnit.MILLISECONDS.sleep(100);//Java SE5/6-style
Thread.sleep(100);//之前的风格
4、任务进行睡眠（即阻塞），使得线程调度器可以切换到另一个线程，进而驱动另一个任务
5、**异常不能跨线程传播，必须本地处理所有在任务内产生的异常**

21.2.6. 优先级

1、线程的优先级将线程的重要性传递给了调度器，调度器倾向于让优先权高的线程先执行。
2、并不意味着优先权较低的线程将得不到执行（不会导致死锁）
3、**优先级较低的线程仅仅是执行的频率较低**
4、***Thread*.currentThread()：获取对驱动该任务的 Thread 对象的引用**
5、在任何时候都能用 setPriority()来修改优先级
Thread.currentThread().setPriority(Thread.MIN_PRIORITY);

6、JDK 有 10 个优先级

21.2.7. 让步

1、**如果知道已经完成了在 run()方法的循环一次迭代过程中所需的工作，就可以给线程调度机制一个暗示：你的工作已经做得差不多了，可以让别的线程使用 CPU 了，这个暗示通过调用 *Thread*.yield()方法来做出（这只是暗示，并不保证一定被采纳，并且只是建议相同优先级的其他线程可以运行）**

21.2.8. 后台线程

1、**后台线程是指程序运行的时候在后台提供一种通用服务的线程，并且这种线程并不属于程序中不可或缺的部分，因此当所有的非后台线程结束时，程序也就终止了，同时会杀死进程中所有的后台线程。反过来说，只要有任何非后台线程还在运行，程序就不会终止，比如 main 就是一个非后台线程**

2、Thread.setDaemon(true);必须在启动线程之前调用该方法。

3、通过编写定制的 ThreadFactory**（接口，含有方法 public Thread newThread(Runnable r)）**可以定制由 Executor 创建的线程的属性

4、Executors.newCachedThreadPool(ThreadFactory);**该方法接受 ThreadFactory 对象作为参数用于创建 Thread 对象。**

5、可以通过 Thread.isDaemon()方法来确定线程是否为后台线程

6、**后台线程创建的任何线程将被自动设置为后台线程**

7、**后台线程不执行 finnlay 子句的情况下就会终止其 run()方法。因为当最后一个非后台线程终止时，后台线程会突然终止，不会有任何你希望出现的确认形式，因此不能以优雅的方式来关闭后台线程。**

21.2.9. 编码的变体（别用！！！）

1、可以直接从 Thread 继承，代替继承 Runnable

2、利用适当的 Thread 构造器为 Thread 对象赋予具体的名称，这个名称可以通过 getName()获得。

3、start()是在构造器中调用的，这会产生问题：另一个任务可能会在构造器结束之前开始执行，这意味着该任务能够访问处于不稳定状态的对象。**因此优先选择 Executor 而不是显示创建 Thread 对象的另一个原因。**

21.2.10. 术语

1、执行的任务和驱动它的线程之间有一个差异。我们对 Thread 类实际没有任何控制权（执行器将替你处理线程的创建和管理）你创建任务，通过某种方式将一个线程附着到任务上，以使得这个线程可以驱动任务

2、Thread 类本身不执行任何操作，他只是驱动赋予它的任务。

3、**我们尝试在描述要执行的工作时使用术语"任务"，只有在引用到驱动任务的具体机制时才会使用术语"线程"**

21.2.11. 加入一个线程

1、一个线程可以在其他线程之上调用 join()方法，其效果是等待一段时间直到第二个线程结束才继续执行。

2、**如果某个线程 a 在另一个线程 t 上调用 t.join()，此线程 a 将被挂起，直到目标线程 t 结束才恢复**

3、**也可以在调用 join()时带上一个超时参数（单位可以是毫秒，或者纳秒），这样如果目标线程在这段时间到期时还没有结束的话，join()方法总能返回。**

4、对 join()方法的调用可以被终端，做法是在调用线程上调用 interrupt()方法，这时需要用到 try-catch 子句

21.2.12. 创建有响应的用户界面

1、使用线程的动机之一就是建立有相应的用户界面。

21.2.13. 线程组（忽略它即可）

1、最好把线程组看成试一次不成功的尝试，你只要忽略它就好了

21.2.14. 捕获异常??

1、**由于线程的本质特性，你不能捕获从线程中逃逸的异常。**

2、**为了解决这个问题，Java SE5 中引入了新接口，Thread.UncaughtExceptionHandler，它允许在每个 Thread 对象上都附着一个异常处理器，Thread.UncaughtExceptionHandler.uncaughtException()会在线程因未捕获异常而临近死亡时被调用。**

21.3. 共享受限资源

1、可以把单线程程序当做在问题域求解的单一实体，每次只能做一件事，永远不用担心诸如"两个实体视图同时使用同一个资源"的问题。

2、并发可以同时做许多事情了，同时也出现了两个或多个线程彼此互相干涉的问题。

3、***volatile 关键字*：确保本条指令不会因编译器的优化而省略，且要求每次直接读值。**

21.3.1. 不正确地访问资源

1、递增程序自身也需要多个步骤，递增不是原子性的操作，如果不保护任务，即使单一的递增也是不安全的。

21.3.2. 解决共享资源竞争

1、使用线程时的一个基本问题：你坐在桌边手拿着叉子，正要去叉盘子中的最后一片实物，当你的叉子就要够着它时，这片实物突然消失了，因为你的线程被挂起了，而另一个餐者进入并吃掉了它。

2、**防止冲突的方法就是当资源被一个任务使用时，在其上加锁，第一个访问某向资源的任务必须锁定这项资源，是其他任务在其被解锁前无法访问。**

3、**基本上所有并发模式在解决线程冲突问题的时候，都是采用序列化访问共享资源的方案，这意味着在给定时刻，只允许一个任务访问共享资源。**

4、**因为锁语句产生了一种互相排斥的效果，这种机制常常称为互斥量**

5、**当资源被锁住时，其他线程簇拥在门口，当当前线程打开锁准备离开时，"最接近"锁的线程将得到访问资源的权限。可以通过 yield()以及 setProirity()来给线程调度器提供建议，但建议未必有太多效果，这取决于具体平台和 JVM 实现**

6、***Java 以提供关键字 synchronized 的形式，为防止资源冲突提供了内置支持。当任务要执行被 synchronized 关键字保护的代码片段的时候，它将检查锁是否可用，然后获取锁，执行代码，解放锁。***

synchronized 不属于方法特征签名的组成部分，在覆盖方法的时候可以添加

7、共享资源一般是以对象形式存在的内存片段，但也可以是文件、输入/输出端口，或是打印机。

8、**要控制对共享资源的访问，得先把它包装进一个对象。然后把所有要访问这个资源的方法标记为 synchronized。如果某个任务处于一个对标记为 synchronized 的方法的调用中，那么在这个县城从该方法返回之前，其他所有要调用该类中任何标记为 synchronized 方法的线程都会被阻塞。**

9、声明 synchronized 方法的方式：

```
synchronized void f(){/*...*/}
```

10、**所有对象都自动含有单一的锁（也称为监视器），当在对象上调用任意 synchronized 方法的时候，此对象都被加锁，这是该对象上其他 synchronized 方法只有等到前一个方法调用完毕并释放了锁之后才能被调用。换言之，对于某个特定的对象来说，其所有的 synchronized 方法共享一个锁**

11、使用并发时，将域设置为 private 是非常重要的，否则 synchronized 关键字就不能防止其他任务直接访问域，这样就会产生冲突。

12、一个任务可以多次获得对象的锁：如果一个方法在同一个对象上调用第二个方法，后者又调用了同一对象上的另一个方法，就会发生这样的情况。

- **JVM 负责跟踪对象被加锁的次数，若一个对象被解锁（锁被完全释放），其计数变为 0。**
- 在任务第一次给对象加锁的时候，计数变为 1，每当这个相同的任务在这个对象上获得锁，计数都会递增。
- 显然，只有首先获得了锁的任务才能继续获得多个锁。
- 每当任务离开一个 synchronized 方法，计数递减。
- 当计数为 0 的时候，锁被完全释放，此时别的任务就能使用该资源

13、**针对每个类，也有一个锁（作为类的 Class 对象的一部分），所以 synchronized static 方法可以在类的范围内防止对 static 数据的并发访问。**

14、**如果在类中有超过一个方法在处理临界数据，那么必须同步所有相关的方法。如果只同步一个方法，那么其他方法将会随意的忽略这个对象锁，并可以在无任何惩罚的情况下被调用。每个访问临界共享资源的方法必须被同步，否则它们就不会正确地工作。**

15、显示使用 Lock 对象：

- Lock 对象必须被显式得创建、锁定和释放
- private Lock lock=new ReentrantLock();*//作为私有字段，惯用法内部化*
- 与内建的锁形式相比，缺乏有雅兴，但更具灵活性
- **如果在使用 synchronized 关键字时，某些事物失败了，那就会抛出一个异常，但是没有机会做任何清理工作，以维护系统使其处于良好状态；有了显式的 Lock 对象，就可以用 finally 子句将系统维护在正确的状态。**

```
lock.lock()
try{
    //do something
}finally{
    lock.unlock();
}
```

- **解决特殊问题才考虑使用显式的 Lock 对象。例如：synchronized 不能尝试着获取锁且最终获取锁会失败，或者尝试着获取锁一段时间，然后放弃他。**
  - boolean captured=lock.tryLock();**//可以失败，立即返回是否获取成功**
  - boolean captured=lock.tryLock(2,TimeUnit.SECONDS);**//该尝试可以在 2s 后失败：在 2 秒内若获取失败，继续获取直至成功，或者 2s 后返回获取失败**

21. 3. 3. 原子性与易变性

1、（不正确的？）有关 Java 线程的讨论中，一个不正确的知识是“原子操作不需要进行同步控制”。

2、（不正确的？）原子操作是不能被线程调度机制中断的操作；一旦操作开始，那么它一定可以在可能发生的“上下文切换”之前（切换到其他线程执行）执行完毕。

3、依赖于原子性是很棘手且很危险的事情

4、**原子性可以应用于除 long 和 double 之外的所有基本类型之上的“简单操作”。对于读取和写入除 long 和 double 之外的基本类型变量这样的操作，可以保证它们会被当做不可分（原子）的操作来操作内存。但是 JVM 可以将（64 位 long 和 double 变量）的读取和写入当做两个分离的 32 位操作来执行，这样就产生了一个在读取和写入操作中间发生的上下文切换，从而导致不同的任务可以看到不正确结果的可能性。**

5、**当定义 long 或 double 变量时，如果使用 volatile 关键字，就会获得（简单的赋值与返回操作的）原子性。**

6、**一个任务做出修改，即使在不中断的意义上讲是原子性的，对其他任务也可能是不可使的（例如，修改值是暂时性地存储在本地处理器的缓存中），因此不同的任务对应的状态有不同的视图。**

7、**volatile 关键字确保了应用中的可视性。如果将一个域声明为 volatile，那么只要对这个域产生了些操作，那么所有的读操作就都可以看到这个修改。volatile 域会立即被写入到主存中，而读取操**

作就发生在主存中。

8、在非 volatile 域上的原子操作不必刷新到主存中去，因此其他读取该域的任务也不必看到这个新值，如果多个任务在同时访问某个域，那么这个域就应该是 volatile 的，否则这个域就只能经由同步来访问，同步也会导致向主存中刷新。因此如果一个域完全由 synchronized 方法或语句块防护(synchronized 可以确保可视性)，那就不必将其设置为 volatile

9、一个任务所作的任何写入操作对这个任务来说都是可视的，因此如果它只需要在这个任务内部可视，就不需要将其设置为 volatile。

10、使用 volatile 而不是 synchronized 的唯一安全情况就是：类中只有一个可变的域。除此之外第一选择就应该使用 synchronized 关键字。

11、在 Java 中对域中的值做赋值（一个写操作）和返回（一个读操作）操作通常都是原子性的，很遗憾，在 Java 中，递增操作不是原子性的（涉及一个读操作和写操作）

12、基本上如果一个域可能会被多个任务同时访问，或者这些任务中至少有一个是写入任务，那么你就应该将这个域设置为 volatile，即告诉编译器：不要执行任何移出读取和写入操作的优化，这些操作的目的是用线程中的局部变量维护对这个域的精确同步。

21.3.4. 原子类

1、Java SE5 引入了 AtomicInteger、AtomicLong、AtomicReference 等特殊的原子性变量，他们提供下面形式的原子性条件更新操作：

```
boolean compareAndSet(expectedValue,updateValue)
```

2、这些类被调整为可以使用在某些现代处理器上的可获得的，并且是机器级别上的原子性。

3、对于常规编程来说，很少会派出用场。

21.3.5. 临界区

1、有时，你只是希望防止多个线程同时访问方法内部的部分代码而不是防止访问整个方法。通过这种方式分离出来的代码段被称为“临界区”，它也是用 synchronized 关键字建立。这里 synchronized 被用来指定某个对象，此对象的锁被用来对花括号内的代码进行同步控制。

```
synchronized(syncObject){
//这里的代码在一个时刻只能被一个任务访问，即带有锁
}
```

2、这也被称为同步控制块，在进入此段代码前，必须得到 syncObject 对象的锁，如果其他线程已经得到这个锁，那么就得等到锁被释放后才能进入临界区。

3、适用场景：某人交给你一个非线程安全的类(Pair)，需要在线程环境中使用它，通过创建该类的 Manager 类(PairManager)来实现，该 Manager 类持有该类的对象，并控制对它的一切访问。采用同步控制块（传入的参数就是 Manager 类的 this 本身）

4、采用同步控制块进行同步，对象不加锁的时间更长。使得其他线程能够更多地访问。

5、还可以显式使用 Lock 对象来创建临界区。

21.3.6. 在其他对象上同步

1、由于 synchronized 块必给定一个在其上进行同步的对象，最合理的方式是使用正在被调用的当前对象:synchronized(this)，在这种方式中，如果获得了 synchronized 块上的锁，那么该对象的其他 synchronized 方法和临界区就不能被调用了，因此如果在 this 上同步，临界区的效果就会直接缩小在同步的范围内。

2、有时必须在另一个对象上同步，但是如果你这么做，就必须确保所有相关的任务都是在同一个对象上同步的。

```
class DualSynch{
    private Object syncObject=new Object();
    public synchronized void f(){ //该方法在 this 同步
        for(int i=0;i<5;i++){
            print("f()");
```

```
                Thread.yield();
            }
        }
        public void g(){
            synchronized(syncObject){// g()在 syncObject 上同步
                for(int i=0;i<5;i++){
                    print("g()");
                    Thread.yield();
                }
            }
        }
    }
```

如果两个线程分别调用 g 和 f，不会导致阻塞；如果 g 方法中传入的是 this，会导致两个方法都在 this 上同步，因此当一个线程调用 f()或 g()后，另一个线程调用的剩下的方法将会被阻塞

21.3.7. 线程本地存储

1、防止任务在共享资源上产生冲突的第二种方式是根除对变量的共享。

2、线程本地存储是一种自动化机制，可以为使用相同变量的每个不同线程都创建不同的存储。因此，如果你有 5 个线程都要使用变量 x 所表示的对象，那么线程本地存储就会生成 5 个用于 x 的不同存储块，主要是，它们使得你可以讲状态与线程关联起来

3、java.lang.ThreadLocal 类创建和管理线程本地存储。ThreadLocal 对象通常当做静态域存储。

21.4. 终结任务

1、在之前的某些事例中，cancel()和 isCanceled()方法被防放置到了一个所有任务都可以看到的类中，通过检查 isCanceled()来确定何时终止它们自己(while 的判断条件，等等方式）

2、但在某些情况下，任务必须更加突然地终止

21.4.1. 在阻塞时终结

1、线程状态

- 新建(new)：当线程被创建时，它只会短暂地处于这种状态。此时它已经分配了必要的系统资源，并执行了初始化，此刻线程已经有资格获得 CPU 时间了，之后调度器将把这个线程转变为可运行状态或阻塞状态。
- 就绪(Runnable)：在这种状态下，只要调度器把时间片分配给线程，线程就可以运行。也就是说，在任意时刻，线程可以运行也可以不运行。只要调度器能分配时间片给线程，他就可以运行，这不同于死亡和阻塞状态。
- 阻塞(Blocked)：线程能够运行，但有某个条件阻止它运行。当线程处于阻塞状态时，调度器将忽略线程，不会分配给线程任何 CPU 时间，直到线程重新进入了就绪状态，它才有可能执行的操作。
- 死亡(Dead)：处于死亡或终止状态的线程将不再是可调度的，并且再也不会得到 CPU 时间，它的任务已结束，或不再是可运行的。任务死亡的通常方式是从 run()方法返回，但是任务的线程还可以被中断

2、一个任务可以进入阻塞状态，原因有以下几种：

- 通过调用 sleep 使任务进入休眠状态，在这种情况下，任务在指定的时间内不会运行
- 通过调用 wait()使线程挂起，直到线程得到了 notify()或 notifyAll()消息（或者在 Java SE5 的 java.util.concurrent 类库中等价的 signal()或 signalAll()消息），线程才会进入就绪状态
- 任务在等待某个输入/输出完成
- 任务试图在某个对象上调用其同步控制方法，但是对象锁不可用，因为另一个任务已经获取了这个锁。

3、我们需要查看的问题：有时希望终止处于阻塞状态的任务，如果处于阻塞状态的任务，你不能等待其到达代码中能检查其状态值的某一点，因而决定让它主动地终止，那么你就必须强制这个任务跳出阻塞状态

21.4.2. 中断

1、在 Runnable.run()方法的中间打断它，与等待该方法到达对 cancel 标志的测试，或者到达程序员准备好离开该方法的其他一些地方相比，要棘手得多。

2、**当你打断被阻塞的任务时，可能需要清理资源，正因为这一点，在 run()方法中间打断，更像是抛出异常，因此在 Java 线程中的这种类型的异常中断中用到了异常**

3、**Thread 类包含 interrupt()方法，因此你可以终止被阻塞的任务，这个方法将设置线程的中断状态。**

4、**如果一个线程已经被阻塞，或者视图执行一个阻塞操作，那么设置这个线程的中断状态将抛出 InterruptedException。**

5、**当抛出该异常，或该任务调用 Thread.interrupted()时，中断状态将被复位。**

6、为了调用 interrupt，你必须持有 Thread 对象，但是新的 concurrent 类库避免对 Thread 对象的直接操作，转而尽量通过 Executor 来执行所有操作。**如果在 Executor 上调用 shutdownNow()，那么它将发送一个 interrupt()调用给它启动的所有线程。**

7、若希望至终端某个单一任务，可以通过使用 Executor.submit()来启动任务，就可以持有该任务的上下文，submit 将返回一个泛型 Future<?>，其中有一个未修饰的参数，因为你永远都不会在其上调用 get()。持有这个 Future，我们可以在其上调用 cancel()。并因此可以使用它来中断某个特定任务。

```
static void test(Runnable r) throws InterruptedException{
    Future<?> f=exec.submit(r);
    f.cancel(true);
}
```

8、**I/O 和在 synchronized 块上的等待是不可中断的（Tread.interrupt()或者向 Executor.submit()返回的 Future 对象调用 cancel()不会抛出 InterruptedException。）但是可以通过关闭任务在其发生阻塞的底层资源来等效中断（一旦底层资源关闭，任务将解除阻塞）**

9、nio 类提供了更人性化的 I/O 中断，被阻塞的 nio 通道会自动地响应中断

10、如果尝试着在一个对象上调用其 synchronized 方法，而这个对象的锁已经被其他任务获得，那么调用任务将被挂起（阻塞），直至这个锁可获得

11、*同一任务*能够调用在同一个对象中的其他 synchronized 方法，因为这个任务已经持有锁了。

12、无论在任何时刻，只要任务以不可中断的方式被阻塞，那么都有潜在的会锁住程序的可能。**因此 Java SE5 并发类库添加了一个特性，即在 ReentrantLock 上阻塞的任务具备可以被中断的，这与在 synchronized 方法或临界区上阻塞的任务完全不同。**

21.4.3. 检查中断

1、当在线程上调用 interrupt()时，中断发生的唯一时刻就是任务要进入到阻塞操作中，或者已经在阻塞操作内部时。

2、**如果只能通过在阻塞调用上抛出异常来退出，那么久无法总是可以离开 run()循环。因此，如果调用 interrupt()以停止某个任务，那么在 run()循环碰巧没有产生任何阻塞调用的情况下，你的任务将需要第二种方式来退出**

3、**Java 中断模型也是这么简单，每个线程对象里都有一个 boolean 类型的标识（不一定就要是 Thread 类的字段，实际上也的确不是，这几个方法最终都是通过 native 方法来完成的），代表着是否有中断请求（该请求可以来自所有线程，包括被中断的线程本身）。**

3、**这种机会是由中断状态来表示的，其状态可以通过 interrupt()来设置。**

4、**可以通过调用 interrupted()来检查中断状态，这不仅可以告诉你 interrupt()是否被调用过（返回**

true 则说明处于中断状态，即之前调用了 interrupt()），而且还可以清除中断状态，调用后将状态标记为非中断状态。
5、清除中断状态可以确保并发结构不会就某个人物被中断这个问题通知你两次，你可以经由单一的 InterruptedException 或单一的成功的 Thread.interrupted()测试来得到这种通知。
6、如果想要再次检查以了解是否被中断，则可以在调用 Thread.interrupted()时将结果存储起来
7、惯用法

```
public void run(){
    try{
        while(!Thread.interrupted()){
            NeedsCleanup n1=new NeedsCleanup(1);
            try{
                print("Sleeping");
                TimeUnit.SECONDS.sleep(1);
                NeedsCleanup n2=new NeedsCleanup(2);
                try{
                    print("Calculating");
                    for(int i=0;i<2500000;i++)
                        d=d+(Math.PI+Math.E)/d;
                    print("Finished time-consuming operation");
                }finally{
                    n2.cleanup();//记得清理资源
                }
            }finally{
                n1.cleanup();//记得清理资源
            }
        }
        print("Exiting via while() test");
    }catch(InterruptedException e){
        print("Exiting via InterruptedException");
    }
}
```

21.5. 线程之间的协作

1、当任务协作时，关键的问题是这些任务之间的握手，为了实现这种握手，我们使用了相同的基础特性：互斥。在这种情况下，互斥能够保证只有一个任务可以相应某个信号，这样就可以根除任何可能的竞争条件。
2、在互斥之上，我们为任务添加了一种新途径，可以将其自身挂起，直至某些外部条件发生变化。

21.5.1. wait()与notifyAll()

1、wait()使你可以等待某个条件发生变化，而改变这个条件超出了当前方法的控制能力。通常，这种条件将由另一个任务来改变。
2、忙等待（测试某个条件的同时，不断地进行空循环），通常是一种不良的 CPU 周期使用方式。
3、wait()会在等待外部事件产生变化的时候讲任务挂起，并且只有 notify()或 notifyAll()发生时，即表示发生了某些感兴趣的事务，这个任务才会被唤醒并去检查所产生的变化。
4、调用 sleep()的时候锁并没有释放，调用 yield()也是如此。但是当一个任务在方法里遇到了对 wait()的调用时，线程的执行被挂起，对象上的锁被释放，这意味着另一个任务可以获得这个锁，因此在该对象（现在是未锁定的）中的 *synchronized* 方法可以在 wait()期间被调用。这一点至关重要，因为这些其他的方法通常将会产生改变，而这种改变正是使被挂起的任务重新唤起所感兴趣的变化。

5、**当你在调用 wait()时，就是在声明："我已经刚刚做完能做的所有事情，因此我要在这里等待，但是我希望其他的 *synchronized* 操作在条件适合的情况下能够执行。**

6、有两种形式的 wait()

- 第一种版本接受毫秒数作为参数，与 sleep 接受参数的意思
- “在此期间暂停”，但是与 sleep 不同，对于 wait 而言：
  - 在 wait()期间对象锁是释放的
  - 可以通过 notify()、notifyAll()、或者命令时间到期，从 wait()中恢复执行。
- 第二种版本的不接受任何参数，这种 wait()会一直等待下去，直到线程接收到 notify()或者 notifyAll()消息。

7、**wait()、notify()以及 notifyAll()的特殊方面：这些方法是基于 Object 的一部分，而不是属于 Thread 的一部分。（为什么作为通用基类的实现？：因为这些方法操纵的锁也是对象的一部分）。因此，可以把 wait()放进任何同步控制方法里，而不用考虑这个类是继承自 Thread 还是实现了 Runnable 接口。（随便说说而已！！！）*实际上，只能在同步控制方法或同步控制块里调用 wait()、notify()和 notifyAll()*（因为不用操纵锁，所以 sleep 可以在非同步控制方法里调用），*如果在非同步控制方法里调用这些方法，程序能通过编译，但是运行时会得到 IllegalMonitorStateException 异常。并伴随着一些含糊的消息（比如当前线程不是拥有者：意思是，调用 wait()、notify()和 notifyAll()的任务在调用这些方法前必须拥有（获取）对象的锁）。***

8、**当调用 notify()或 notifyAll()，将唤醒在对 wait()调用中被挂起的任务，为了使该任务从 wait()中唤醒，它必须首先重新获得它进入 wait()时释放的锁，在这个锁变得可用之前，这个任务是不会被唤醒的。**

9、**必须用一个检查感兴趣的条件的 while 循环包围 wait()，这很重要：**

- 可能有多个任务出于相同原因在等待同一个锁，而第一个唤醒任务可能会改变这种状况（使得又该让这个任务进入 wait），如果属于这种情况，那么这个任务应该被再次挂起
- 在这个任务从 wait()被唤醒的时刻，可能会有其他的任务已经做出了改变，从而使得这个任务在此时不能执行，或者执行器操作已显得无关紧要。此时，应该通过再次调用 wait()将其挂起
- 可能某些任务出于不同的原因在等待你对象上的锁（在这种情况下，必须使用 notifyAll()），在这种情况下，你需要检查是否已经由正确的原因唤醒，如果不是，就再次调用 wait()将其挂起。

10、**错失的信号：如果不用检查感兴趣的条件包围 wait()会出现信号丢失的情况**

```
T1：
synchronized(sharedMonitor){
    <setup condition for T2>
    sharedMonitor.notify();
}
T2:
while(someCondition){
    //Point 1
    synchronized(sharedMonitor){
        sharedMonitor.wait();
    }
}
```

- T1 是通知 T2 的线程，T1 将条件作出某些改变，以通知 T2，但上述写法会出现信号的错失：如果此时 T1 在执行，且 T2 处于 Point 1 位置等待同步锁的释放，不久后，T1 更改了状态，并通知了 T2 且释放了锁，T2 获取锁后，直接进入了 wait()。问题出在 T2 并没有判断该状态是否是"感兴趣的状态"，因为 while 的条件在之前已经判断过了，T1 对条件的改变已经为时已晚。
- 可以将 T2 改为如下形式

```
synchronized(sharedMonitor){
```

```
        while(someCondition)
            sharedMonitor.wait();
    }
```

21.5.2. notify()与notifyAll()

1、在技术上，可能会有多个任务在单个对象上处于 wait()状态，因此调用 notifyAll()比调用 notify()更安全。

2、使用 notify():

- **使用 notify()而不是 notifyAll()是一种优化。使用 notify()时，在众多等待同一个锁的任务中只会有一个被唤醒，如果你希望使用 notify()，就必须保证被唤醒的是恰当的任务。**
- **另外，为了使用 notify()，所有任务必须等待相同条件，因为如果有多个任务在等待不同条件，那么你就不会知道是否唤醒了恰当的任务(???)。**
- **如果使用 notify()，当条件发生变化时，必须只有一个任务能够从中受益。**
- **最后，这些限制对所有可能存在的子类都必须总是起作用(???)**

3、对于 notifyAll()的描述：notifyAll()将唤醒所有正在等待的任务

- 事实上，notifyAll()因某个特定锁而被调用时，***只有等待这个锁的任务才会被唤醒***

21.5.3. 生产者与消费者

1、在典型的生产者-消费者实现中，应使用先进先出队列来存储被生产和消费的对象。

2、显式使用 Lock 和 Condition 对象

- 使用互斥并允许任务挂起的基本类是 Condition，可以通过在 Condition 上调用 await()来挂起一个任务。
- 当外部条件发生变化，意味着某个任务应该继续执行时，你可以通过调用 signal()来通知这个任务，从而唤醒一个任务，或者调用 signalAll()来唤醒所有在这个 Condition 上被其自身挂起的任务。（与 notifyAll()相比，signalAll()是更安全的方式）
- 惯用法
  - private Lock lock=new ReentrantLock();
  - private Condition condition=lock.newCondition();
- Condition 对象用于管理任务间的通信。
- Condition 对象本身不包含任何有关处理状态的信息，因此需要定义额外的表示处理状态的信息。

21.5.4. 生产者-消费者队列

1、wait()和 notifyAll()方法以一种非常低级的方式解决了任务互操作问题，即每次交互时都握手。

2、可以使用同步队列来解决任务协作的问题。同步队列在任何时刻都只**允许一个任务**插入或移除元素

3、java.util.concurrent.BlockingQueue 接口中提供了这个队列，这个接口有大量的标准实现：

- **ArrayBlockingQueue**：规定大小的 BlockingQueue,其构造函数必须带一个 int 参数来指明其大小.其所含的对象是以 FIFO(先入先出)顺序排序的
- **LinkedBlockQueue**：大小不定的 BlockingQueue,若其构造函数带一个规定大小的参数,生成的 BlockingQueue 有大小限制,若不带大小参数,所生成的 BlockingQueue 的大小由 Integer.MAX_VALUE 来决定.其所含的对象是以 FIFO(先入先出)顺序排序的
- **ProirityBlockingQueue**：类似于 LinkedBlockQueue,但其所含对象的排序不是 FIFO,而是依据对象的自然排序顺序或者是构造函数的 Comparator 决定的顺序
- **SynchronousQueue**：特殊的 BlockingQueue,对其的操作必须是放和取交替完成的

4、阻塞队列可以解决非常大量的问题，而其方式与 wait()和 notifyAll()相比，则简单并可靠得多。

5、**如果 BlockQueue 是空的，从 BlockingQueue 取出东西操作将会被阻断进入等待状态，直到 BlockingQueue 进了东西才会被唤醒。如果 BlockingQueue 是满的,任何试图往里存东西的操作也会被阻断进入等待状态,直到 BlockingQueue 里有空间才会被唤醒继续操作。**

6、方法：

- BlockingQueue.add(anObject)：将 anObject 加到 BlockingQueue 里，如果 BlockingQueue 可以容纳，则返回 true，否则返回 false
- BlockingQueue.offer(anObject)：表示如果可能的话，把 anObject 加到 BlockingQueue 里，如果 BlockingQueue 可容纳，返回 true，否则返回 false
- **BlockingQueue.put(anObject)：把 anObject 加到 BlockingQueue 里，如果 BlockingQueue 没有空间，则调用此方法的线程将被阻断直到 BlockingQueue 里面有空间再继续**
- BlockingQueue.poll(time)：取走 BlockingQueue 排在首位的对象，若不能立即取出，则可以等待 time 参数规定的时间，取不到时返回 null
- **BlockingQueue.take(anObject)：取走 BlockingQueue 里排在首位的对象，若 BlockingQueue 为空，调用此方法的线程将被阻断进入等待状态，直到 BlockingQueue 有新对象加入**

7、看 P715-P716 的例子，非常的赞

21.5.5. 任务间使用管道进行输入/输出

1、通过在输入/输出在线程间通信非常有用。提供线程功能的类库以"管道"的形式对线程的输入/输出提供了支持。它们在 Java 输入/输出类库中的对应物就是 PipedWriter 类（允许任务向管道写）和 PipedReader（允许任务向管道读取）。

2、**这个模型可以看做是"生产者-消费者"的变体，管道基本上是一个阻塞队列，存在于多个引入 BlockingQueue 之前的 Java 版本中。**

3、**调用 PipedReader.read()时，如果没有更多数据，管道将自动阻塞**

4、*BlockingQueue 用起来更加健壮和简易*

21.6. 死锁

1、一个对象可以有 synchronized 方法或其他形式的加锁机制来防止别的任务在互斥还没有释放的时候就访问这个对象。

2、任务可以变成阻塞状态，所有就可能出现以下情况：某个任务在等待另一个任务，而后者又在等待别的任务，直到这条链上的任务又在等待第一个任务，这样得到了一个任务间相互等待的死循环，没有哪个线程能继续，这被称为死锁。

3、死锁例子，哲学家进餐：

- n 个哲学家围城一圈坐好
- 每个哲学家都有一支筷子
- 每个哲学家都会去获取自己的这只以及右边哲学家的这支筷子作为其进食所用
- 当每个哲学家首先获取了右边哲学家的筷子时（在所有哲学家中没有一个获取了自身的筷子之前），此时将发生死锁，因为，哲学家只有进食完毕才会释放筷子，但是所有的筷子均被不同的哲学家所占有。

4、死锁条件：四个同时满足

- 互斥条件：任务使用的资源中至少有一个是不能被共享的。
- 至少有一个任务它必须持有一个资源且正在等待获取一个当前被别的任务持有的资源
- 资源不能被任务抢占，任务必须把资源释放当做普通的事情
- 必须有循环等待**（该条件最容易不满足）**

21.7. 新类库中的构件

Java SE5 的 java.util.concurrent 引入了大量涉及用来解决并发问题的新类

21.7.1. CountDownLatch

1、它被用来同步一个或多个任务，强制他们等待由其它任务执行的一组操作完成

2、你可以向 CountDownLatch 对象设置一个初始计数器，任何在这个对象上调用 wait()的方法都将阻塞，直到这个计数值到达 0。其他任务在结束工作时，可以在该对象上调用 countDown()来减少这个数值。CountDownLatch 被设计为支出法一次，计数值不能被重置，如果需要能够重置计数值的版本，可以使用 CyclicBarrier

3、调用 countDwon()的任务在产生这个调用时并没有阻塞，只有对 await()的调用会被阻塞，直到计数值到达 0

4、CountDownLatch 的典型用法是将一个程序分为 n 个相互独立的可解决任务，并创建值为 0 的 CountDownLatch，每当任务完成时，都会在这个锁存器上调用 countDown()，等待问题被解决的任务在这个锁存器上调用 await()，将它们自己拦住，直至锁存器技术结束

5、惯用的形式：见例子 P723

- 创建一个 CountDownLatch 对象 c
- 在所有相关的任务中定义一个私有的 CountDownLatch 的引用，构造器接受一个 CountDownLatch 对象，并将其绑定到私有引用上
- 这样所有任务在执行完各自的任务后调用 countDown()，并且调用 CountDownLatch.wait()进行挂起，直达计数为 0，被唤醒
- 这样与问题相关的并行任务就被同步到了这一个共享的 CountDownLatch 对象上

21.7.2. CyclicBarrier

1、适用情况：

- 你希望创建一组任务，它们并行地执行工作，然后在进行下一步之前，直至所有任务都完成。
- **它使得所有的并行任务都将在栅栏出列队，因此可以一致地向前移动，这非常像 CountDwonLatch，只是 CyclicBarrier 可以多次重用**

2、特征：

- CyclicBarrier 的 await()方法使当前线程进入等待状态，计数器自动-1（而 CountDwonLatch 需要手动调用 countDow()方法），当计数器为 0 时，当前线程被唤醒
- 可以向 CyclicBarrier 提供一个"栅栏"动作，它是一个 Runnable，当计数值到达 0 时自动执行，这是 CyclicBarrier 和 CountDownLatch 之间的另一个区别。
- **与 CountDownLatch 相同，所有并行的任务必须共享供一个 CyclicBarrier，并且利用该 CyclicBarrier 的对象调用 wait()才能使得计数以及栅栏动作正确运转。即将所有并行的任务同步到该唯一的 CyclicBarrier 对象上**

21.7.3. DelayQueue

1、介绍：

- 这是一个无界的 BlockingQueue，**用于放置实现了 Delayed 接口的对象**，其中的对象只能在其到期时才能从队列中取走
- 这种队列是有序的，即队头的对象延迟到期的时间最短/长（取决于 compareTo 的具体实现），如果没有任何延迟到期，那么就不会有头元素，调用 poll()会返回 null，调用 take()会自动调用 wait()直到对象满足超时条件
- **Delayed 接口中有一个 getDelay 的方法**，用来告知延迟到期的时间有多长，一般写法：
  - private final long trigger;
  - trigger=System.nanoTime()+NANOSECONDS.convert(delay,MILLISECONDS);
  - public long getDelay(TimeUnit unit){
  - return unit.convert(trigger-System.nanoTime(),NANOSECONDS);
  - }
- **Delayed 接口继承了 Comparable 接口，因此必须实现 compareTo()**，用于产生预期的排序
- 具体的方法参考生产者-消费者队列

2、惯用的实现：

- 定义任务实现 Runnable 与 Delayed
- 将任务的实例添加进 DelayQueue 的对象中
- 在一个线程中，在 while(!Thread.interrupted())中调用 DelayQueue.take().run()即可，（DelayQueue 会自动调用 getDelay()以检验是否超时，来选择执行或者 wait()）

21.7.4. PriorityBlockingQueue

1、用于线程之间的通信

2、具体的方法参考生产者-消费者队列

21.7.5. ScheduledExecutor

- ScheduledThreadPoolExecutor 提供了 schedule()以及 scheduleAtFixedRate()方法：
- public ScheduledFuture<?> schedule(Runnable command, long delay, TimeUnit unit)
  - command：要执行的任务
  - delay：时延，执行任务的时延
  - unit：提供时间单位以及转换方法的工具类
- public ScheduledFuture<?> scheduleAtFixedRate(Runnable command,long initialDelay, long period,TimeUnit unit)
  - initialDley：第一次执行任务的起始时延
  - period：执行任务的周期
-

21.7.6. Semaphore

- 正常的锁(来自 concurrent.locks 或内建的 synchronized 锁)，在任何时候都只允许一个任务访问一项资源，而计数信号量则允许 n 个任务同时访问这个资源，
- 可以将信号量看作是在向外分发使用资源的"许可证"，尽管事实上没有使用任何许可证对象
- 惯用的实现：
  - Semaphore available=new Semaphore(size,true);//放置 size 个许可证,true 什么意思？
  - **注意，这个 available 和被共享的资源没有任何的关联！**
  - 在访问资源开始的时候调用 available.acquire();**//当调用这个函数的时候，许可证已经发完，那么调用这个方法的线程将被阻塞。**
  - 在访问资源结束的时候调用 available.release();**//多余的签入会被忽略**

21.7.7. Exchanger

不懂

Extra 工厂模式

1、简单工厂

1.1 只能生产一种类型对象的工厂（与类的耦合度非常紧密）

1.2 可以生产属于生产列表中的任意对象（与类的耦合度变松）

1.3 可以生产任意对象，即可以生产新添加的类的对象而无需修改工厂，实现了工厂与类的分离（解耦）通过反射机制来完成：Class.newInstance()

2、工厂与匿名类的结合

易错点

1、StringBuilder.append('5'+'6') ≠"56" 而是"99"：因为'5'和'6'会作为 int 进行运算

2、对 int[]利用 Arrays.sort 进行降序排列的话 Comparator 怎么写

3、**空的 new StringBuilder().toString()≠""(废话因为字符串是类类型，要用 equals 来比较是否相等)，但是 length()==0**

4、条件语句必须作为 return 或者赋值语句的一部分？

```
1==1? i++;j++;//编译不通过
```

5、List.remove(i)：去掉第 i 个，Set.remove(i)：去掉值为 i 的数

6、a==b? 1:2+3; 会先计算 2+3！

7、Comparrator 中的 compare 返回-1，那么维持原顺序：

public int compare(T obj1,T obj2)：若想要升序，那么假定现在 obj1<obj2，那么利用 T 的属性返回一个负值即可；若想要降序，假定 obj1>obj2，利用 T 的属性返回一个负值即可

例如 T 为 Integer，若要降序，由于假定 obj1>obj2 了，因此 obj2-obj1 就是一个负值，因此返回 obj2-obj2 即可

8、统计整数的 1bit 数目

```
count=0;
for(int i=0;i<32;i++)
    if ((n&1<<i)!=0) count++//不要用等于 1 来判断
```

包装类型与基本类型

因为基本类型不能够作为泛型参数，泛型参数的最基本类型是 Object

包装类型没有任何默认构造器

基本类型	包装类型
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean