# 第3章 Java的基本程序设计结构

## 3.1 一个简单的java程序

Java是区分大小写的，java程序中所有内容都放在类中，源代码的名字与公共类的名字相同，java classname时，从这个类中找到main函数执行，main必须是public的。

## 3.2 数据类型



类型与机器无关，便于移植，从java7开始，可以写0b开头的二进制数字，同时10进制数字之间可以加下划线，并无实际意义，方便阅读。



Double的精度不能表示一些数据，在金融中使用bigdecimal表示。

Double有正无穷大、付无穷大、未知数3中表示，不能使用==判断。

Char是unicode-16编码，不能表示有意义的整数，boolean只是单纯的表示true或者false对象，不能与整数类型计算。

## 3.4 变量

变量定义，变量此时是未知值，需要显示初始化或者赋值后使用，不区分变量的声明与定义，都一样；final声明常量，定义时初始化，后面不能改，运算符与++，--等与C++一样，逻辑运算符与位运算符也差不多，移位运算符也差不多，>>>高位填0，>>高位填符号位；math类包含了大部分数学函数，直接调用；sqrt：方根，pow幂运算；三角函数、指数函数，math.PI以及math.E。



类型提升转换是不损失精度的，int转float时候除外，逆类型提升需要()式的强制类型转换，浮点数的四舍五入使用math.round方法得到；枚举类型enum，只能是给定范围的值。

## 3.6 字符串

Java没有内置的字符串类型，String是标准类库里面的””内的都是字符串，subString（startposition，endposition）；使用+拼接，拼接任意类型都将转化为String类型，String不能修改，字符串池保证字符串的共享引用，字符串的相等使用equals()判断，里面可以是字符串常量，equalsIgnoreCase不区分大小写判断相等，==判断字符串地址是否相等，只有字符串常量是共享的，临时变量等都不是共享的，null与空串不一样，想要知道所有的类方法需要阅读联机文档；假如需要多次拼接String，那么使用StringBuilder。

## 3.7 输入输出

要想输入，必须构造Scanner对象，里面含有很多next开头的函数可以读取相应的数据，使用console处理密码输入，readline函数每次读取一行，readpassword密码读到字符数组中去。格式说明符



Format创建格式化的字符串。





读文件使用Scanner，参数里不能是String，否则解析成字符数据，使用Paths。Get()获得路径，注意反斜杠要写2个，写文件使用printWriter，可以像out那样使用，相对路径默认为虚拟机启动路径，命令行下为当前访问路径，IDE下为用户路径，使用user.dir找到路径，要表明会发生的异常，在函数后面加入throwsFileNotFoundException。

## 3.8 控制流程

嵌套的块内不能声明同名变量，条件if，循环（注意浮点数的相等判断），switch（case标签只能是char，byte，short，int，枚举、字符串字面量类型）；带标签的break，调转到tag：statement后，判断是否是break跳出的。

## 3.9 大数值

BigInteger与Bigdecimal，一种类，使用专门的成员函数完成+，-等计算。

Valueof()函数完成普通数字到大数字的转换。

## 3.10 数组

数组使用new分配，个数可以是变量，数字数组初始化为0，length数组长度，不能改变大小，数组可以用初始化列表初始化，数组变量之间可以赋值，这是变量引用同一个数组，可以使用Arrays.copyOf(var,length)拷贝数组值，可以使用Arrays.sort()对数组排序。数组的toString函数可以生成[“”,””]等形式的输出。多维数组与C++相通，java可以创建不规则的数组，大致意思就是数组中可以存储指针（引用），在多维数组中，每一个内部值可以引用不同大小的数组。所以形成了不规则的数组，因为java数组是包含了大小含义的，但是C++的没有，只是一个指针名字，所以数组必须规则。

# 第4章 对象与类

## 4.1 面向对象程序概述

Java类关注数据，只在乎数据的状态；类有很多种，基本具有封装，就是隐藏底层数据，继承；对象-标识、状态、行为；类之间的关系：依赖（使用）、聚合（含有对象）、继承（继承）。

## 4.2 使用预定义类

首先构造对象，new；date表示了到1970年1.1午夜的毫秒数，由于日历各地都不同，所以使用gregorianCalender表示格林尼治标准时间。New之后可以表示当前时间，还有访问器方法与修改器方法。

## 4.3 用户自定义类

构造器与类同名，初始化对象，构造器只能与new搭配使用，java方法含有隐式参数this，表示当前对象；注意函数不应该直接返回private数据对象引用，破坏了封装性，使用对象的clone()方法返回拷贝的形式。方法可以访问所有对象的私有域数据，因为代码只有一份，与C++中是一样的，工具函数可以设置为private的，可以与数据尽管修改不会影响接口，final修饰的数据域在构造后就不会修改，修饰基本类型表示常量，假如修饰一般的类，只是表示类引用对象不能修改，对象本身还是可以修改。

## 4.4 静态域与静态方法

静态域属于类，所有对象共享，静态常量非常常用，Math类，还有System.out是system类的一个静态常量对象，静态方法不操作对象，可以操作静态数据，没有隐式this参数；每一个类都可以有一个main方法用于测试。

## 4.5 方法参数

Java中的传参总是传值调用，所以对于对象来说，传递的是引用，可以修改参数的状态（基本数据类型是值类型，对象理解为指针类型）。

## 4.6 对象构造

函数名与参数类型构成了函数签名，所以可以重载，如果构造函数没有初始化，执行默认初始化，基本类型为0，布尔值为false，对象为null，与方法中的局部变量是不同的，如果没有自定义构造器，系统自动提供无参构造器，如果有自定义则不提供，可以为实例域提供初始值即直接赋值，先于构造函数执行，适用于重载的构造函数都希望这个域是这个值的情况，初始值可以不是常量，也可以是静态方法。this除了作为参数的隐藏参数外，还可以应用于构造函数中调用其他的构造函数this(其他构造函数的参数)；类中也可以使用第三种初始化数据的方法就是初始化块，会先于构造函数执行。初始化块放在域定义之后，构造步骤：



静态数据可以直接赋值初始化或者使用静态初始化块初始化，类第一次加载时执行静态域的初始化，静态域也有默认值跟非静态域一样。

## 4.7 包

包是用来组织类的，也就是名字空间，使用类时可以使用完全的包+类名字，也可以使用import引入，java的import机制类似于

（这里是在公司看到的内容）

抽象类不能生成对象，但是可以引用对象；

private：仅对本类可见；public：对所有类可见；protected：对本包和所有子类可见；默认，对本包可见。

## 5.2 object：所有类的超类

object可以引用任何对象；超类中的equals函数将会比较2个对象的相等性，默认比较的是引用的相等性，需要改写，在最开始要调用比较超类equals的相等性；

# 第5章 初始化与清理

## 5.1 用构造器初始化

构造器是创建对象后执行的第一个方法，构造器方法与类名相同，实际是一种静态方法。所以说，new 构造器创建对象，所以java的初始化与创建对象的工作是绑定在一起的。

## 5.2 方法重载

类型提升。

## 5.3 默认构造器

如果没有构造器，编译器会自动生成一个无参的默认构造器。

## 5.4 this指针

this指的是当前对象自己，或者是在构造器中代表其他的构造方法调用，this会隐式的作为方法的第一个参数传入，static方法不会传入this对象。

## 5.5 清理：终结处理和垃圾回收

finalize()用于释放不是new分配的内存，这种内存无法被垃圾回收器回收。

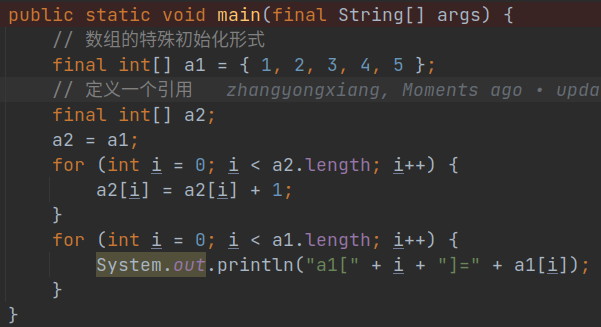
## 5.6 成员初始化

## 5.7 构造器初始化

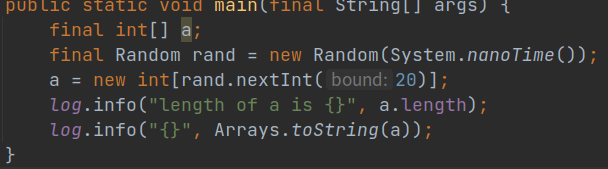
域初始化、构造方法初始化、静态数据初始化，静态块，普通块这些执行顺序。

## 5.8 数组初始化

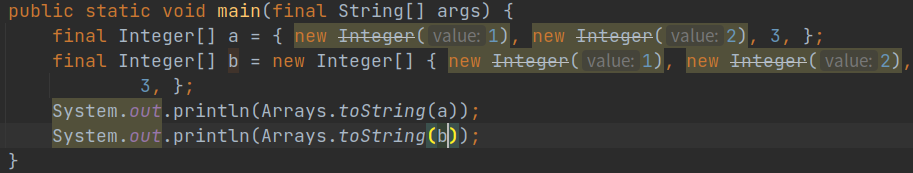
数组是类型相同的对象序列，通过[]定义，编译器不允许指定数组的大小，因为只是定义了一个数组引用，数组引用与数据的具体大小是无关的，因为还没有实际分配数据对象的空间，所以不需要定义数据的大小，可以使用特殊的初始化表达式来初始化数组，例子如下：



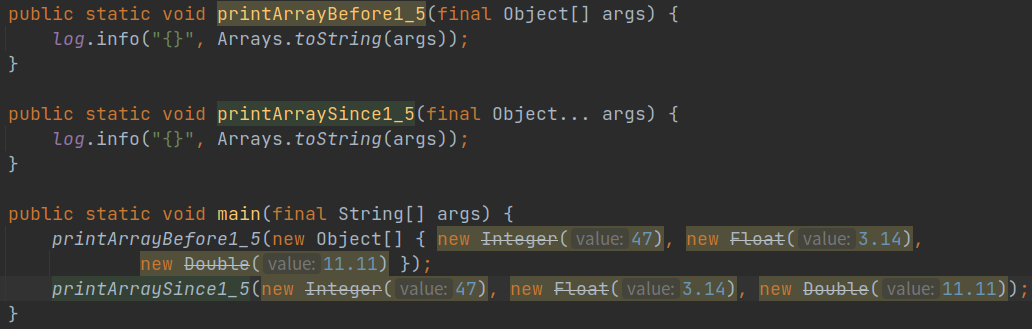
每个数组都有一个固有成员length，可以用new[size]的形式创建固定大小的数组，基本类型的数据会被初始化为空值，0或者false等，对象数据是null；



Arrays.toString()会产生数组的可打印的版本，2种初始化对象数组的形式，是一样的



因为所有的类都继承于Object类，可以通过Object[]对象传递任意类型与数量的参数到方法里面，java1.5版本之后，使用了可变参数列表...实际就是使用的数组，编译器会帮助填充数组，传递0个参数也是可以的，这适用于可选参数的传递场景。可变参数列表不依赖自动包装机制，并且可以与自动包装机制和谐共处，不冲突。需要注意可变参数函数的重载问题，如果什么也不传递，可能会发生编译错误，需要添加函数的标识参数以便重载没有问题。



## 5.9 枚举类型

枚举类型的值是常量，因此都用大些字母表示；

# 第9章 接口

接口与内部类为我们提供了一种将接口与实现分离的更加结构化的方法。抽象类是类与接口的一种中庸之道。

## 9.1 抽象类与抽象方法

通用接口：不同的子类可以用不同的方式表示此接口；建立起一种基本形式；也叫做抽象基类；为了编译时就可以判断抽象类对象的操作非法，使用abstract标识方法，达到了编译期就报错的效果；包含抽象方法的类是抽象类；abtract主要是为了阻止这个类创建对象；抽象类是很有用的重构工具，可以使得我们很容易的将公共方法沿着继承层次结构向上移动。

# 第10章 内部类

将一个类的定义放到另一个类的定义内部，就是内部类。逻辑相关的类组织到一起，控制内部类的可见性，可以与外部类通信。

## 10.1 创建内部类

在外部类中使用内部类跟其他类没什么区别，如果在静态或者外部类其他地方想要访问内部类代码，需要OuterClass.InnerClass这种形式方式。

## 10.2 链接到外部类

内部类对象拥有外部类对象内所有属性与方法的访问权，它们之间具有联系，内部类对象会捕获一个指向外部类对象的引用，所以，访问外部类成员时，就是根据这个引用访问的。

## 10.3 使用.this与.new

内部类中访问隐含的外部类引用，就是OuterClass.this，就可以访问捕获的外部类对象的引用；假如在其他代码中想创建一个内部类对象，要使用OuterClassObject.new InnerClass()的形式，因为创建内部类对象前，必须有外部类对象，内部类对象会暗暗链接到外部类对象上。

## 10.4 内部类与向上转型

内部类如果设置为private或者protect的，并且只向外返回内部类的接口的引用，就可以完全隐藏内部类的实现细节，并且外部代码也不能使用到内部类，只能通过接口使用对象。强制客户端程序员编码时不依赖具体类型而依赖接口或者抽象。

## 10.5 在方法和作用域内的内部类

内部类可以定义在任何地方，内部类的用途：返回接口时代码不依赖与具体类型；解决一个复杂的问题，不希望这个类是公共可用的。内部类定义在方法内部，就是方法作用域，方法外不能知晓这个类，{}也是一个作用域，一个类一个作用域，一个文件一个作用域，一个包一个作用域等等。继承就是作用域的延伸；类被定义在哪里就表明了它的作用域，它的名字可以被多大范围的类所知晓。

## 10.6 匿名内部类

new 构造器创建一个匿名的内部类，定义与创建类是在一起的；匿名内部类使用的外部对象时，这些外部对象必须是final的，匿名内部类中可以直接定义字段，字段会被直接初始化，{}块初始化可以达到构造函数的效果，匿名内部类用于工厂模式。

## 10.7 嵌套类

如果内部类不想与外部类对象发生联系，可以声明为static，这个内部类就是嵌套类，不含有外部类对象的引用，static意味着只能访问外部类的静态属性；普通的内部类不能含有static属性，嵌套类什么都可以含有。

接口内也可以定义内部类，因为接口不是类，所以接口内定义的类都是public与static的，这个嵌套类可以实现外部接口，一些接口的公共代码可以放到这种内部类中。

## 10.8 为什么需要内部类

内部类弥补了java的多重继承难题；内部类可以支持回调；控制框架中使用。

## 10.9 内部类的继承

普通继承，构造函数必选super。

## 10.10 内部类可以被覆盖吗？

内部类不会被覆盖，每个内部类都有自己的命名空间，即使继承也不会覆盖。

## 10.11 局部内部类

定义在方法内的，不能有说明符，但是也是可以访问外围类的属性，只是名字空间限定到了方法内。内部类也会生成单独的class文件，使用$符号表示class文件的名字嵌套；$1…$2等这种加序号的时匿名内部类。

# 第11章 持有对象

## 11.1 范型与类型安全的容器

容器类没有规定类型则都是Object，如果有范型，可以在编译期检查加入容器的对象类型，并在取出时不需要转型。

## 11.2 基本概念

Collection（独立元素的序列）与Map（字典）对象；所有的Collection都可以使用foreach语法遍历。

## 11.3 添加一组元素

11.4

# 第12章 通过异常处理错误

## 12.2 基本异常

throw new Exception();

## 12.3 捕获异常

## 12.4 异常说明

异常说明是方法声明的一部分，用throws表示，表示此方法可能会抛出的异常，要求调用者进行处理，调用者在编译阶段就需要处理异常，但是方法可以返回RuntimeException异常，这种异常不需要用throws进行声明，编译期强制检查的异常称为被检查的异常，可以在定义方法接口的阶段预先声明throws，方便后续添加。

## 12.5 捕获所有异常

使用Exception捕获所有的异常，因为Exception是编程相关的基类，所以不需要使用Throwable，使用getMessage类获得信息，使用继承的printStackTrace打印异常的调用栈信息，getStackTrace()用户获取调用栈信息，最顶是最后一个方法调用栈信息；可以在catch块中抛出重新抛出异常，异常的调用栈信息还是保留，如果想要改变异常的调用栈信息，调用fillInStackTrace()方法，将当前方法的调用栈信息填入异常对象，异常可以包裹在异常中形成异常链，Error（虚拟机错误）、Exception、RuntimeException都有接收另一个异常对象的构造方法，如果是不同的类型的异常，不能用构造方法的方式包裹，需要用initCause()方法。

## 12.7 java标准异常

Throwable用于表示所有可以被抛出类的父类，Error用来表示编译与系统错误，Exception表示被抛出的基本类型，有些异常不需要用户判断并抛出，java虚拟机可以自动在运行时判断并抛出，所以不需要在方法上声明，这些异常都是RuntimeException的子类，也被称为不受检查的异常，他们代表程序的错误而不是异常。

## 12.8 使用finally进行清理

Java有自动内存回收机制，所以finally主要用来在异常发生时恢复资源的初始状态。

## 12.9 异常的限制

## 12.10 构造器

# 第14章 类型信息

运行时类型信息使得你可以在程序的运行期间发现与使用类型信息。运行时识别对象与类的信息的2种方式：传统的RTTI与反射机制。

## 14.1 为什么需要RTTI

RTTI(Run-time type identification)，为什么需要RTTI呢？因为在使用到java的继承时，往往使用通用的类型来引用对象，或者在范型里面，也是用通用的父类型表示多个对象，那么就不知道具体的对象的类型，此时如果想要知道对象的具体类型，就需要RTTI；

## 14.2 Class对象

类型信息在运行时是通过Class对象表示的，它包含了类的类型信息，用来创建类的所有的常规对象，编写类后，JVM的类加载器将其加载为一个Class对象；JVM的类加载器是一条链，包含多种加载器，其中只有一个原生类加载器，原生类加载器加载的是可信类；在加载器链上可以挂载额外的加载器；所有的类都是第一次使用时动态加载到JVM的，用到类的静态引用时就会加载类，所以构造方法是类的静态方法；创建对象时，类加载器首先检查new后面的类是否加载，没有加载的话，就会去加载；static初始化块是在类加载时执行的；Class.forName()可以用来加载类；Class还有很多有用的方法；比如newInstance()，使用。class的方式创建Class对象的引用时，不会自动的初始化该Class对象，使用类经过3个步骤：1.加载，类加载器查找字节码并创建Class对象；2.链接，验证字节码，为静态域分配存储空间；3.初始化，初始化超类，执行静态初始化器和块；初始化是在对静态方法或者静态域的进行首次调用时，执行的；但是使用Class.forName()加载类时，是立即初始化的；一个static final是编译器期常量，那么没有初始化也可以读取值，如果这个常量需要计算得到，那么一定执行Class的初始化；如果一个static域不是final的，它被读取前都要进行初始化。Class对象的应用也有类字面常量的形式，class.class获得引用；这种方式可以用于任何类型，用于基本类型的包装器类时，还有一个TYPE字段，指向基本类型的class对象类型，因为Class本来就是指向的Class对象，因此Class引用表示的就是指向对象的确切类型，可以用范型确定具体的Class引用类型，如果想要放宽类型限制，使用Class<?>因为Class没有编译器警告信息，也可以使用Class</？extends class>来引用子类型的Class引用；但是没有？的直接引用是不允许的，因为Class对象之间没有继承关系；此时使用newInstance()方法将会返回具体的对象类型；Class里面又个cast()方法，可以将对象转型为Class的类型对象。

## 14.3 类型转换前先做检查

已知的RTTI形式包括：

1. 传统形式的类型转换，执行错误会抛出一个ClassCastException；
2. Class类型对象可以获得类型信息；

C++中类型转换只是单纯的类型转换，Java要做类型检查，任何无父子关系类之间的转型都叫做向下转型，都要进行类型检查。

Instanceof也能检查对象的类型，只能获得true/false的结果；Class对象的isInstance方法可以检测类型，等价于instanceof。

## 14.4 注册工厂

主要就是工厂方法设计模式，对象的创建工作交给类自己去完成。可以采用多态的模式，那么创建对象的过程将会延迟到子类中去执行。

## 14.5 instanceof与class的等价性

Instanceof与isIntsance方法在判断对象的类型的时候会考虑是否是继承关系，equal与==不会考虑继承关系，只会判断确切的类型。

## 14.6 反射：运行时的类信息

在java程序中，对象的类型可以通过RTTI来获得；在运行时获取类信息的一个动机就是提供在垮网络的远程平台上创建与运行对象的能力；称为远程方法调用；此时系统中没有加载Class信息，可以使用反射的机制获得与类型相关的信息；java.lang.reflect类库有Field、Method、Constructor类，每个类都实现了Member接口这些类型的对象都是JVM在运行时创建的，表示了未知类里的成员；此时可以使用Contructor构建对象，Field获取成员，invoke调用Method；RTTI与反射的区别是：RTTI在编译期打开并检查.class文件，并生成Class对象，而对于反射类说，.class文件在编译是不知道，是在运行时打开并检查class文件。

## 14.7 动态代理

代理是基本的设计模式之一；用来代替实际对象的对象以便提供额外的或者不同的操作；实际就是中人；代理模式就是装饰者模式差不多，封装修改，当需要使用额外的操作而又不想破坏现有的代码的时候，可以使用；代理对象与实际对象都含有同样的接口，代理对象内部有实际对象；java的动态代理更进一步；可以动态的创建代理并动态的处理对代理方法的调用；所有的调用都会被重定向到单一的调用处理器上。



Proxy.newProxyInstance()可以创建动态代理；需要的参数如下：



动态代理内需要有一个实现了InvocationHandler接口的对象，代理的接口的所有的方法访问都会被交给这个对象的invoke方法处理，同时代理对象也传递给了proxy参数，如果使用proxy调用，那么会陷入死循环，所以Invocationhandler对象内注入了一个被代理的对象。

## 14.8 空对象

在程序中使用null没有任何有益的地方，除了抛出NullPointerException，引入空对象来代替null是有很多的好处的，可以建立一个接口表示空对象然后使用instanceof测试对象的类型，如下代码所示：



## 14.9 接口与类型信息

Interface的目标就是允许程序员隔离构件，降低耦合性；但是通过instanceof还是可以检测到其他的类型信息，实际上并没有成功解耦；一种简单的方式是使用包访问权限。

# 第15章泛型

Java中多态是一种泛化机制，在使用类型的地方都使用基类代替，这样解耦了具体类的实现，也可以使用接口，但是这2种方式还是要具体的类型，泛型要求不依赖具体的类型，泛型就是参数化类型的概念，目的是解耦类或方法与使用类型之间的约束。

## 15.1 与C++比较

## 15.2 简单泛型

使用<T>放在类名后面，泛型最初的引入主要是为了解决容器类的问题；这样编译器就可以自动根据类型做一些判断，检查编译期的错误。

使用范型的元组返回任何组合的对象内容。

## 15.3 范型接口

范型也可以用到接口上，与用到类上没什么区别，比如工厂模式。

## 15.4 范型方法

只在方法上应用范型；类型推断只在赋值的时候起作用， 也能应用到可变参数上。

## 15.5 匿名内部类

范型还可以工作内部类与匿名内部类。

## 15.6 构建复杂模型

主要就是在集合中使用范型；一下时等价的：



Class.getTypeParameters()返回TypeVariable对象的数组，经过测试，只是返回了表示类型的占位符，并没有返回真实的类型信息；这是因为Java范型是使用擦出来实现的；任何具体的类型信息都被擦除了；只知道在使用一个对象，所以List<String>与List<Integer>在运行时时相同的类型，被擦除成为他们的原生类型即List。因为C++的范型是静态时替换，所以在编译期就可以检查范型类型的错误，并给出报警，但是java是运行时范型，编译器不能知道范型的类型，无法知道类型具有哪些方法，这就是因为擦除，在编译期假定都约束为Object对象；为了在编译期也可以知道范型的一些类型信息，可以使用extends指定假定的约束类型，编译器知道这个边界条件后，就把它擦除到extends之后的类型而不是Object；类中的范型类型被编译器替换为extends之后的类型，而不是原始的Object；

擦除减少了范型的泛化性；采用这种方式实现范型是为了兼容以前写的类库；在实际执行时，其实擦除就已经确定了类型，这也就是减少了泛化的原因；擦除的代价就是不能使用new、instanceof与转型等操作；擦除意味着在使用范型类时可以不指定类型，缺省就是Object；擦除可以让范型表示没有任何意义的事物；在范型中使用Array.newInstance是推荐的方式；边界是发生动作的地方，由于范型的擦除，类型返回前会执行转型，在参数传递时会执行类型检查。

## 15.8 擦除的补偿

擦除使得任何需要知道确切类型信息的操作都无法进行，为了能够让这样的操作进行，可以引入类型标签，就是传递类型的Class对象，new T()在Java中无法实现，解决方案是使用工厂对象创建对象，使用Class.newInstance可以创建对象，sun不推荐这种方式，推荐工厂模式实现范型的工厂接口，在实现类中规定范型的类型，这样可以具有编译期检查；不能创建范型数组，一般的办法是创建ArrayList对象；支持范型；

## 15.9 边界

因为类型擦除了，所以对象只能调用Object上的方法；可以使用extends关键字来确定类型边界；那么可以调用边界类型的方法；可以添加多个边界，但只能有一个是类，必须在最前面；那么可以调用更多的方法。

## 15.10 通配符

?问号就是通配符；<? Extends XXX>，设定了界限的通配符；超类型通配符<? Super XX>规定是某个类的任何基类；<?>无界通配符；

## 15.11 问题

任何基本类型不能作为类型参数；一个类不能实现同一个范型接口的2种变体；因为类型擦除会把这2个变成相同的对象；

## 15.12 自限定的类型



范型T的边界是自身。

## 15.13 动态类型安全

由于范型擦除的特性，当将容器传入到1.5以前的代码中去处理时，里面的代码可能会向容器里面加入不兼容的类型，这时不会爆出错误；Collections里面提供了很多的静态方法比如checkedList()等来处理这个问题。这个方法返回的是检查类型的容器类型，当向容器里面插入别的对象的时候，会爆出ClassCastException异常。



## 15.14 异常

## 15.16 潜在类型机制

编写尽可能广泛应用的代码，但是为了执行某些特定的不是Object的操作，需要指定范型边界，这就对代码的范化进行了限制，应用的类型必须继承某些类或者某些接口；提供的一种解决方案是潜在类型机制或者结构化类型机制；就是范型不关心是否继承了某个类或者实现某个接口，只要类型里面有特定的操作就可以，C++或者Python都有潜在类型机制，java没有，但是java可以使用反射的方式的得到需要运行的方法来弥补潜在类型机制的缺失。

# 第16章 数组

## 16.1 数组为什么特殊

数据区别与其他的方面：效率、类型与保存基本类型的能力。效率：数组的存取效率比List高；保存具体的类型可以在编译期进行检查；

## 16.2 数组是第一级对象

数组标示符只是一个引用；指向堆中创建的一个对象；这个对象保存指向其他对象的引用，对于基本类型来说，保存的是值。注意聚集初始化。

## 16.3 返回一个数组

直接返回，不像C++语言那样。需要返回指针。

## 16.4 多维数组

用{}括号圈定维度；数组中构成矩阵的每个子数组都可以具有任意的长度，所以是粗糙数组。

## 16.5 数组与范型

Arrays.fill()一下子创建很多的数据，Arrays.equals() 判断数组是否相等，Arrays.deepEquals() 用于多维数组；sort用于对数组排序，BinarySearch用于在排序的数组中查找元素；toString(),产生数组的字符表示；hashCode产生散列码；System.arrayCopy() 复制数组元素,执行的是浅复制；将保持不变的事务与会发生变化的事务相分离，java有2种方式实现比较功能，一种是实现java.lang.Comparable接口，一种是创建一个实现了Comparator接口的单独的类；

# 第17章 容器深入研究

主要深入讲解Java容器类库。

## 17.1 完整的容器分类法

容器相关的一些分类图去网上找找。单线程的多线程的等等。



Java SE5 增加了Queue接口于其实现PriorityQueue，BlockingQueue；ConcurrentMap接口与其实现ConcurrentHashMap；CopyOnWriteArrayList与CopyOnWriteArraySet，EnumSet，EnumMap；

## 17.2 填充容器

Cloolections.fill()，nCopies()等方法复制同一个对象填充容器；只对List有用；Collection子类型都有一个接收另一个Collection对象的构造器；addAll()方法的作用是这些构造函数一样。LinkedHashSet；

## 17.3 Collection的功能方法



Collection接口中的添加与移除方法都是可选操作，实现类可以不实现。

## 17.6 Set与存储顺序

Set需要一种方式来维护存储顺序，Set不保存重复元素，存入Set的元素必须定义equals方法来确保对象的唯一性；HashSet，存入hashSet的元素必须定义hashCode()方法；Treeset，有序set，底层的实现是树，需要元素实现Comparable接口；LinkedHashSet，链表实现，插入顺序；元素需要定义hashCode()方法；equal方法对于所有的set元素都是必须的，但是hashSet与LinkedHashSet额外必须HashCode()方法；

根据测试，HashSet只保持元素的唯一性，不对元素进行排序；LinkedHashSet在保证元素唯一性的情况保持插入顺序；TreeSet根据元素的Comparable接口对元素进行排序。如果不定义hashCode，那么hashSet与LinkedHashSet都将插入重复的数据，他们是根据hashCode判重的，因为此时生成的hashCode相当于对象的地址，但是TreeSet可以不需要自定义hashCode方法就可以排重，因为它是使用的euqals方法判重的，SortedSet接口的元素可以保证处于排序状态，



## 17.7 队列

Queue的实现是LinkedQueue与PriorityQueue；LinkedQueue是插入顺序，PriorityQueue是优先级顺序；还有ArrayBlockingQueue、ConcurrentLinkedQueue、LinkedBlockingQueue、PriorityBlockingQueue等；

优先级队列会根据Comparable接口来决定对象的顺序。还有双向队列Deque。

## 17.8 理解Map

HashMap，使用对象的散列码来完成对象的搜索；HashMap的存取速度最快，因为是基本的Map实现类，其他的Map加了其他的特性，所以慢了一些；HashMap（HashTable）,查询写入键值对是固定开销，LinkedHashMap，底层实现是链表，所有保存有插入顺序；TreeMap基于红黑树的实现，TreeMap会根据Comparable或者Comparator排序；WeakHashMap允许释放映射所指向的对象；ConcurrentHashMap，并发Map，IdentityHashMap使用==代替equals对键进行比较的映射，为特殊的问题而设计。

键必须具有equals方法，如果用于散列Map（比如HashMap）,要复写hashCode()方法，如果用于TreeMap，还要实现Comparable接口。

SortedMap是排序Map的接口其基本方法如下：

* comparator()：当前Map的Comparator，如果null就是自然方式排序，也就是按照hashCode的值排序。

LinkedHashMap可以设定采用LRU算法存储键值对的顺序。

## 17.9 散列码

自定义对象时，如果不定义hashCode()方法，则无法根据key找到value，但是hashCode只决定了值的地址，不定义euqals方法，也不行，因为hash算法计算得到的映射值相同时，就没办法取到值了。

Equals()方法满足的条件：自反性、对称性、传递性、一致性等。

散列的价值就在于速度。

## 17.10 选择接口的不同实现

ArrayList与LinkedList的性能比较问题。

可以通过手工调整HashMap来提高其性能，HashMap的术语：

* 容量：散列表的桶位数；
* 初始容量：创建时拥有的桶位数；
* 尺寸：表中当前占用的桶位数；
* 负载因子：尺寸/容量，负载因子小时，hash下标的计算重复率就会降低，提高了查找速度，但是有数据的复制开销与空间的浪费；默认的负载因子是0.75，这是一个综合的权衡，较大的负载开销查找会比较慢。

## 17.11 实用方法

unmodifiableCollection()等类似的方法提供了只读容器，可以保留容器的副本为private的数据，只有自己可以做修改，向外提供不可修改的容器；synchronizedCollection()等类似的方法返回线程安全的容器。

## 17.12 持有引用

# 第18章 java I/O系统

## 18.1 File类

File类既是指文件也是指目录。记住FilenameFilter这个辅助的过滤器。

## 18.2 输入与输出

流的概念：代表任何有能力产出数据的数据源对象或者是有能力接收数据的接收端对象，屏蔽了I/O设备中处理数据的细节。

InputStream与Reader都有read()方法用于读取字节，所有的输入类都继承于InputStream；

OutputStream与Writer都有write()方法用于写入字节；所有的输出类都继承于OutputStream；

不同的出数据源的输入流都实现了InputStream接口，主要有：

* ByteArrayInputStream：字节数组，将内存的缓冲区作为InputStream使用；
* StringBufferInputStream：底层实现用StringBuffer作为数据源；
* FileInputStream：文件作为数据源；
* PipedInputStream：产生用于写入管道的输入流；
* SequenceInputStream：将多个InputStream转换成单一的InputStream；
* FilterInputStream：抽象类，作为装饰器接口，为其他的InputStream类提供有用的功能。

不同的数据源的输出流都实现了outputStream接口，主要有：

* ByteArrayOutputStream：在内存中创建缓冲区，所有送往流的数据都放在此缓冲区；
* FileOutputStream：用于将信息写置文件；
* PipedOutputStream：
* FilterOutputStream；

## 18.3 添加属性和有用的接口

FilterInputStream与FilterOutputStream这2个类用来做java I/O系统的装饰器的基类。很多类都继承了这个类，并实现了一些扩展接口；还有一些特殊的实现了一些特定逻辑的InputStream，比如DataInputStream、BufferedInputStream、LineNumberInputStream、PushbackInputStream。

## 18.4 Reader与Writer

Reader与Writer主要是为了提供Unicode与字符的I/O功能，字节流与字符流需要组合起来使用，所以中间加了适配器，InputStreamReader把InputStream适配成字符输入流，OutputStreamWriter把OutputStream适配成字符输出流。

设计Reader与Writer主要就是为了国际化，因为老的流只能用于处理8位的字节，不能处理16位的unicode的字符。

Stream与Reader/Writer的对应关系可以参考书上的图。

|  |  |
| --- | --- |
| Java1.0类 | Java1.1类 |
| InputStream | Reader/InoutStreamReader |
| OutputStream | Writer/OutputStreamWriter |
| FileInputStream | FileReader |
| FileOutputStream | FileWriter |
| StringBufferInputStream | StringReader |
|  | StringWriter |
| ByteArrayInputStream | CharArrayReader |
| ByteArrayOutputStream | CharArrayWriter |
| PipedInputStream | PipedReader |
| PipedOutputStream | PipedWriter |

继承与FilterInputStream/FilterOutputStream的流在Reader/Writer下也是有对应关系的，也是用的装饰器模式。

|  |  |
| --- | --- |
| 装饰器类1.0 | 装饰器类1.1 |
| FilterInputStream | FilterReader |
| FilterOutputStream | FilterWriter（抽象类，没有子类） |
| BufferedInputStream | BufferedReader |
| BufferedOutputStream | BufferedWriter |
| PrintStream | PrintWriter |
| LineNumberInputStream | LineNumberReader |
| StreamTokenizer |  |
| PushbackInputStream | PushbackReader |

## 18.5 自我独立的类：RandomAccessFile

操作文件，里面有一些独特的方法，并不是java I/O体系的一部分。

## 18.6 I/O流的典型的使用方式



BufferedReader加快了读取速度，因为提供了缓存支持。



StringReader读取字符串。

读取格式化的数据使用DataInputStream，可以看到DataInputStream读取字节时，是无法识别到了文件末尾的，因为任何字节都是合法的数据，所以当到了文件末尾的读取抛出了EOF异常。



可以使用available()方法检测还有多少可供读取的字符。

FileWriter向文件写入数据，用BufferedWriter包装以提高性能，用PrintWriter包装以简化操作。



构造PrintWriter的过程时可以简化的，传入文件名，就可以，内部做了Buffered。

DataOutputStream与DataInputStream是对称的，可以用来写数据与恢复数据，比如：



可以看到结果。这种数据的交换方式已经被XML与其他的数据序列化方式替代。RandomAccessFile可以以任意的方式操作文件。

## 18.7 文件读写的实用工具

## 18.8 标准I/O

Java提供的标准I/O模型有System.in，System.out，System.err；System.out与System.err已经被包装成了PrintStream对象，System.in没有经过任何包装，只是InputStream，为了读入方便，一般将System.in包装成BufferedReader。



缓存System.out流



System类的静态方法提供了in/out/err流的重定向功能。

setIn/SetOut/setErr；



## 18.9 进程控制

在java程序内部执行操作系统的命令或者程序，通过Process来进行处理。



## 18.10 新I/O

Java1.4版本引入了java新I/O，但是老的IO类库底层也用新的IO实现过了，主要是为了提高速度；速度的提高是因为所使用的结构更接近操作系统执行I/O的方式：通道与缓冲器。

唯一与通道交互的缓冲器是ByteBuffer，里面只是存储一段字节空间，并提供了最底层的读取写入字节数据的方法；java类库中的FileStream被修改了用以产生FileChannel；因为通道就是面向字节的，所有stream可以产生通道，但是面向字符的流是不能用来产生通道的，Channels类提供了一些静态方法可以在通道中产生字符流。



字节存放于ByteBuffer中，可以使用put方法直接填入字节，也可以使用wrap方法包装字节数组到Buffer中，这样Buffer不会在分配底层数据的空间，而是直接饮用包装的数据的地址；nio的目标就是快速移动大量数据；只读访问要使用ByteBuffer的allocate来分配存储空间。



可以看到上述的代码，ByteBuffer既可以用于读也可以用于写，flip可以转换Buffer的读写状态，生成用于写的一些内部的指针等，clear则还原Buffer的状态为读状态，并将内部的指针重新指派；最好的方式是transferTo()与transferFrom()方法，可以将通道与另一个通道直接相连。

Buff容纳的是字节，要转换成字符，需要对内容在输入时编码或者输出时解码；使用Chaset类实现这个功能。



ByteBuffer虽然存储的是字节，但是可以写入读取出任何的基本类型。使用的是视图缓冲器的技术；在ByteBuffer上套一层代理，用来处理类型；ByteBuffer是将数据移进移出通道的唯一方式；nio类之间的关系图：



Buffer由数据与操作数据的4个索引组成：mark（标记）、position(位置)、limit(界限)、capacity（容量）；与这4个索引有关的方法如下：

|  |  |
| --- | --- |
| Capacity() | 返回缓冲区容量 |
| Clear() | 清空缓冲区，position=0，limit=capacity |
| Flip() | Limit=position,position=0,准备读取数据前调用 |
| Limit() | 返回limit |
| Limit(int limit) | 设置limit值 |
| Mark() | 将mark设置为position() |
| Position() | 返回position()的值 |
| Position(int pos) | 设置position的值 |
| Remaining() | 返回limit-position |
| HasRemainning | Limit-position>0 |

内存映射文件可以创建修改因为太大而不能完全放入内存的文件。

Java可以直接在操作系统层面给文件枷锁。

## 18.11 压缩

Java I/O类库支持读写压缩格式的数据流，相关的类如下：

|  |  |
| --- | --- |
| CheckedInputStream | GetCheckSum为任何的InputStream产生校验和 |
| CheckedOutputStream | GetCheckSum为任何的OutputStream产生校验和 |
| DeflaterOutputStream | 压缩类的基类 |
| ZipOutputStream | 一个DeflaterOutputStream，用于将数据压缩成Zip文件格式 |
| GZIPOutputStream | 一个DeflaterOutStream，用户将数据压缩成GZip文件格式 |
| InflaterInputStream | 解压缩类的基类 |
| ZipInputStream | 用户解压缩Zip文件格式的数据 |
| GZIPinputStream | 解压缩Gzip文件格式的数据 |

支持Zip格式的java类库更全面。



上面的例子是写入压缩文件，下面的例子是读取压缩文件。



使用ZipFile类操作文件更为方便。

Jar javaArchive也是使用的zip格式，可以压缩文件与目录进去，同时带有一个目录清单；jar程序可以自动生成这个清单；调用形式：

jar [options] destination [manifest] inputfiles

如果0选项创建的jar文件，就可以放入类路径变量CLASSPATH中，java应用就可以在类路径变量中搜索文件里面的目标类。

## 18.12 对象序列化

就是将对象写成字节流数据，放到别的地方可以还原出数据；主要的用途有RMI，数据备份；主要使用的流是ObjectXXXStream类，ObjectInputStream恢复对象时，不会调用任何的构造函数；但是恢复的类必须是在环境中，否则会爆出ClassNotFoundException异常；Externalizable接口的功能比Serializable更强大，可以对序列化的过程进行控制，writeExternal()&readExternal()方法会在序列化与反序列的过程中自动被调用，与Serializable不同的是Ser对象在恢复时不调用构造函数，直接根据二进制数据恢复对象，Ext对象在恢复时，不会是实例化任何的域信息，会在对象构造好后调用默认的构造函数，如果默认构造函数没有对实例域进行操作，所以需要在writeExternal&readExternal方法中进行实例域的处理。

Transient修饰字段代表，字段不能被Stream流序列化，由于Ext对象有自定义控制，所以transient与Ser搭配使用。

## 18.13 XML

# 第19章 枚举类型

枚举类型是用一些值的集合定义新的类型，很有用。

## 19.1基本的枚举特征

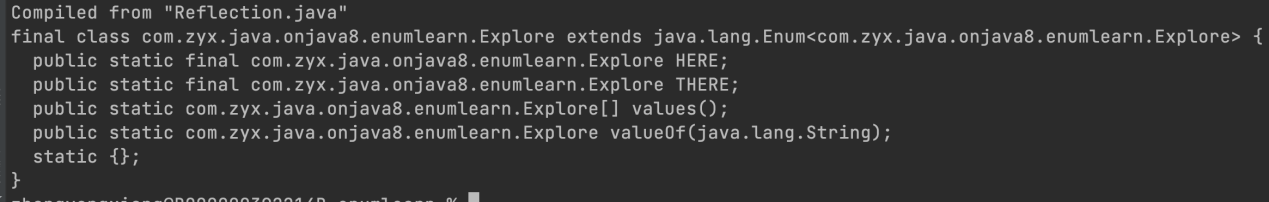
values() 方法返回所有实例的一个数组，创建enum时，编译器会生成一个继承于Enum的类；ordinal( )返回enum实例的定义的顺序，是一种int值从0开始，enum实例有equals、hashcode、comparable方法，getDeclaringClass返回枚举Class，name()返回enum的value与toString()方法相同，valueof将字符串转换为enum实例；import static导入enum，可以直接在当前类使用所有的枚举实例；enum不能被继承，基本表现就是一个类，其内可以定义属性与方法，可以有main方法，构造方法一般为private就可以，其实不是private也没有其他地方能够用到这个构造器（编译器会检查）；枚举value 相当于枚举类的实例。

如果定义自己的方法，必须在所有的实例后面加分号，而且必须定义实例，如果定义实例之前定义了任何的属性与方法，则无法编译。

可以覆盖enum内部的方法。

## 19.3 switch语句中的enum

enum天生支持swicth语句（如果case语句中没有return，可以没有default语句），最好使用静态导入的方式使用enum。



通过对枚举类型反射与反编译（javap）得知定义枚举类型时，实际是生成了一个继承于Enum类的类；编译器为这个类添加了values()与valueof()静态方法，生成的类是final的，所以无法被继承；由于继承自Enum，所以枚举实例可以向上转型为Enum类型对象；但是不能访问values()，可以通过Class.getEnumConstants()访问。

## 19.5 实现而非继承

所有的enum类都继承与Enum，java规定不能多重继承，所以定义的enum不能继承别的类了，但是可以实现多个接口；与普通的类使用基本相同。接口这种方式也可以直接在enum类中定义方法，但是还是接口的方式比较好。下面使用Enum.class的里面getEnumConstant()方法的例子。



## 19.7 使用接口组织枚举

enum不能产生子类，拓展enum元素或者对元素分组会成为问题；分类：在一个接口的内部，创建实现该接口的的枚举定义，每个枚举定义相当于一个类别；完成子类化；这种形式将枚举的作用域定义在接口内，用public的接口统一了行为，但是内部定义是分类的，实际是使用接口代替了最上面的枚举定义的功能，而使用枚举完成了子类化或者分类的功能。



可以创建枚举的枚举：



更简洁的方法是将Food接口定义在Course内部，这个也就是嵌套的枚举；Enum相当于set，但是不能添加与删除；EnumSet可以完成集合的功能，存的是enum的实例。

## 19.9 使用EnumMap

EnumMap是一种特殊的Map，键必须是enum实例；其他用法差不多；可以用来实现命令设计模式（接口定义方法，子类实现各自不同的行为）；enum一个特性是可以为enum实例编写方法，可以为每个实例编写各自不同的行为；也就是常量相关的方法，enum要定义一个或者多个abstract方法，然后常量里实现方法；这种代码风格称为“表驱动的代码“；这种多态的行为是类的特点，但是enum实例并不是类；虽然从行为上是一样的。每个enum的实例都是定义的enum类型的static final元素，而且实例内部也是可以覆盖enum定义的方法的。

责任链设计模式：多种不同的方式解决问题，将不同的链接在一起，遍历链直到链中的方案处理完该请求；可以用enum定义的实例方式定义责任链上的每个处理结点，在实例内部定义处理方法。就完成了责任链模式；枚举类型非常适合用作状态机，具有有限个状态，根据输入从一个状态到另一个状态。

## 19.11 多路分发

当操作涉及到多种类型时，程序往往会很混乱；而且Java支持单路分发，就是操作涉及到多个位置类型的对象时，Java的动态绑定机制只能处理其中一个类型，多个类型实现自动绑定的机制就是多路分发；多态只能发生在方法调用时，因为一次只能动态确定一个类型，所以有几路分发就有几个方法调用；每个方法调用决定一个未知类型；可以参考示例程序；多路分发还是很重要的，很难理解。

# 第20章 注解

注解就是向源代码中添加代码信息。也叫作元数据。SE5版本引入，注解描述程序，由编译器测试与验证，注解使用@符号，内置3个注解：@Override覆盖超类的方法，@Deprecated @SuppressWarnings 关闭不当的编译器警告信息。

（注解就是代码化的注释信息）

## 20.1 基本语法

注解使用@interface进行声明，@注解修饰java代码，就是一种修饰符；注解定义就像定义接口一样，注解定义需要元注解，就是修饰注解的注解，注解的元素看起来像方法，实际是元素属性，可以指定默认值，没有元素的注解称为标记注解。

|  |  |
| --- | --- |
| @Target | 定义的注解用于什么地方，使用的是ElementType枚举类的值；ElementType值包括CONSTRUCTOR（构造方法）、FIELD（属性）、LOCAL\_VARIABLE（局部变量声明）、METHOD（方法）、PACKAGE（包声明）、PARAMETER（参数）、TYPE（类，接口、注解定义等），具体可以看JDK的源代码。未指定Target时，就是可以应用所有类型。 |
| @Retention | 级别，约束注解的生命周期；Source注解被编译器丢弃，只会保留在源码里；CLASS注解在class文件中，但会被VM丢弃，未定义Retention时，缺省是CLASS级别；RUNTIME运行期也保留。 |
| @Documented | JAVADOC，是否出现在生成的javadoc里 |
| @Inherited | 允许子类继承 |

注解里面的元素可以是所有的基本类型、String、Class、enum、Annotation、或者上述类型的数组；

## 20.2 编写注解处理器



定义的注解类



注解的使用



定义的注解处理器

反射相关的类都实现了AnnotatedElement接口，可以获得相应类型上的注解。注解可使用的元素类型：所有基本类型、String、Class、enum、Annotation以及以上类型的数组，元素必须有值，要么是默认值，要么用时提供值；非基本类型默认值不能是NULL，可以用特定字符表示某个元素缺失。

注解用途：framework需要额外的信息才能与源代码协同工作，XML文件可以提供这样的信息，但是一个类就有了2个信息源，需要同步修改等，这些可以用注解实现。

注解里面嵌套注解：



嵌套的注解是默认值。如果不想要默认值就像定义注解那样定义。Value元素可以使用快捷方式赋值（只给value赋值才可以这样做，其他的不可以这样做）。注解不支持继承（未来可能会实现）。

注解处理的器的实现：





## 20.3 使用apt处理注解

注解处理工具APT，帮助注解的处理过程而提供的工具，apt是操作java源文件的，apt工具会在处理完注解后编译他们；注解处理器生成新的java源文件，这个源文件作为apt的下一轮操作被处理，一轮一轮的操作，直到没有新的源文件产生。

# 第21章 并发

## 21.1 并发的多面性

能够利用多核心CPU，阻塞时提高单核CPU利用率；操作系统级别的进程，有自己独立的空间运行，彼此互不干涉，但是线程并发会共享内存与I/O等，要对共享的资源进行管理。

## 21.2 基本的线程机制

程序可以划分为多个分离的，独立运行的任务，这种任务在进程中就是一个单一的顺序控制流，每个独立的任务都是由执行线程来驱动的。

Java中代表任务的方式是Runnable接口，任务的逻辑是run方法；任务就是任务就是一个普通的类，不放在线程中不会开启线程来执行。

Java中使用Thread来创建线程驱动任务来执行，start方法进行线程初始化操作，然后调用任务的run方法，start方法会立即返回执行下面的代码，不会持有新创建线程的引用。

JDK提供了Executor来管理Thread对象，Executor在客户端与任务间提供了一个间接层，这个中介管理任务执行，不需要客户端手动创建线程并执行任务；所以为客户端与任务之间进行了解耦，提交任务与执行任务分离开，可以异步管理任务的执行，是一种生产者与消费者模式，提交是生产，执行时消费；后面有很的继承Executor的接口，ThreadPoolExecutor（线程池）实现了这个接口，通过Executors的静态方法newFixedThreadPool、newCachedThreadPool等方法可以返回这个线程池（执行器），shutdown执行后，新任务就不能提交给执行器，所有的执行器任务执行完，main所在的线程结束执行。Executor接口中的public abstract void execute(Runnable paramRunnable);方法用来提交任务，而执行任务就是Executor管理的异步了。

Ruunable是没有返回值的任务，Callable是有返回值的任务；任务方法call，必须使用ExecutorService.submit()来提交Callable任务；ExecutorService.submit()返回这个任务的Future，可以调用get()来获得call方法的结果。

Sleep()可以让执行的线程睡眠，Thread.sleep()是老的睡眠方式，TimeUnit.MILLISECONDS.sleep()是新的睡眠方式，是告诉java的线程调度器，阻塞当前线程，我们一定记住，java后面有一个我们不可见的线程调度器，接收一些通知来决定接下来的做法，睡眠就是阻塞，如果发出中断会产生InterruptedException。

Thread.currentThread()获得当前任务的Thread，可以通过set/getPriority设置优先级或者得到优先级。

Thread.yield()通知线程调度器，我运行完了，可以让别的线程来运行了。

后台线程是一种提供服务的线程，当主线程终止时，后台线程都会终止；在start()调用setDaemon(true)会将线程设置为后台线程，newFixedThreadPool等创建线程池的方法都有一个带Threadfactory的重载方法，这个是产生线程的工厂，可以在工厂里设置线程的通用属性，execute提交任务时，将会提交到工厂产生的线程里，后台线程产生的线程也是后台线程，并且后台线程的finally语句并不会得到执行。

Thread也实现了Runnable接口，所以也可以直接作为任务，当继承Thread时，可以覆写默认的run函数，可以在构造函数里start或者在外面start；还有一种形式是将Thread组合到Runnbale内部，new Thread(this)的方式传递任务进去，在构造函数或者其他的方法内start；上面都是同一种形式。实际意思就是Runnable与Thread可以任意组合。

Thread类有个join()函数，在当前线程内调用t.join()，当前线程阻塞等到t执行计数再继续执行。t中断也能使t.join失效，从而向下执行。

由于线程是独立执行的，所以线程发生的异常，除非在其本身里面处理，否则，主线程捕获不到，会抛出返回到控制台。Thread有一个setUncaughtExceptionHandler方法，可以用来设置异常处理器，建立一个Thread. UncaughtExceptionHandler对象，传入进去，就可以用来处理异常。

## 21.3 共享受限资源

并发模式解决线程冲突的方法都是序列化访问共享资源的方案。常用的方法时在使用共享资源的代码块部分加锁。保证只有一个线程访问，这种锁完成了互斥操作，也叫做互斥量。Java提供了synchronized关键字。Synchronized可以给对象加锁，也可以给Class的static方法加锁。普通方法的加锁是对象的内置锁，静态方法的加锁是Class对象的内置锁。访问共享资源的所有方法都必须同步并且必须是一个锁，否则不能起到保护的作用。

java.util.concurrent.locks包下的Lock也能达到synchronized的互斥机制，创建Lock对象后，临界区代码被lock()（trylock）与unlock()包围, unlock()写到finally中，return要在try-catch里，因为不想在return处暴露对象。

Java中的单一语句并不是原子操作，在编译后，往往被优化为多个命令，在多线程情况，这里容易出问题，所以使用同步是最稳的，同时由于存在优化缓存寄存器等存储，对共享资源的修改可能暂时存储到寄存器中，其他线程看不到修改而读取的内存的值，这带来了可视化的问题，使用volatile可以解决可视化的问题，volatile始终读取内存。Synchronized能够解决很多问题。

JDK提供了一些基本类型的原子操作类，在java.util.concurrent.atomic包下，原本是为了并发包下其他类所使用的，我们也可以拿来使用。

ThreadLocal对象可以提供共享变量的多个副本，因为ThreadLocal维持了一个Map，里面存储每个线程与其对应的值，当一个线程使用这个对象包裹的值，会使用这个线程对应位置上的值。

## 21.4 终结任务

本节主要讲如何终止任务。标识任务的终止状态可以设置volatile类型的全局标志，这种标志不需要同步；线程的状态：新建、就绪、阻塞、死亡。进入阻塞状态的原因：sleep()、wait()、等待对象锁、等待输入输出等；Thread类包含interrupt()方法，这个方法设置线程的中断状态，如果线程处在阻塞状态，那么会抛出InterruptedException异常，抛出异常或者调用interrupted()，这个状态复位；但是单个的Thread可以用这个方法调用，如果用Executor的话，shutdownNow()会停掉所有的这个线程池中存有的所有的线程，如果想停掉单一线程，需要使用submit()方法，因为这个方法会返回一个Future来持有线程的上下文，Future.cancel(true)可以向提交的线程内传入interrupt()方法，只有在调用sleep的时候，interrupt才能中断，当线程处于I/O阻塞或者synchronized阻塞时是不能中断的；办法是，关闭I/O资源，那么线程处就会抛出异常，达到了中断的效果。ReentrantLock具有可以被中断的功能。

## 21.5 线程之间的协作

就是规定线程间执行顺序等条件的规则，也是通过互斥来实现的。这种协作可以通过Object的wait()与notify()方法来实现。

wait()可以让当前的任务等待发生某个条件，这个条件是有其他任务维护的，并且其他任务通过notify()或者notifyAll()方法通知等待这个条件的任务，调用wait时，里面会将本任务阻塞，等待条件发生而不是空循环的方式；sleep()与yield()方法不会释放锁，但是调用wait的时候会释放锁，锁所在对象的其他synchronized方法可以获得执行以改变对象的状态。意味着，我做完了我应该做的，我要在这里等待其他人完成他们需要做的。Wait可以接受一个时间参数，过了这个时间就继续执行；wait、notify方法是在Object里面的，并且只能在同步控制方法内调用，就是必须在锁内部调用；因为他们都操作了锁；如果向对象发送nitify方法，那么就不必须先获得这个对象的锁，可以在synchronized(object)块里面操作；其实这3个方法都是操作同一个对象的锁，互斥执行来保证外部线程的执行顺序的。简而言之，就是用一个专门用于同步的对象来协调多线程的执行顺序；这个对象通常来说就是共享资源对象；有时候需要用while循环包围wait()方法，主要是避免多任务竞争时，改变共享对象状态。

# 第十二章 集合

创建多个对象并保存可以使用数据的方式，但是数组具有固定的大小，不够动态；java.util库提供了很多的集合类来保存多个对象，有List、Set、Queue与Map，称为容器类，可以动态的调整自己的大小。集合类使用范型可以实现自动类型转换，防止非法的类插入。

# 函数式编程

函数式编程语言操作代码段就像操作数据，Java允许通过Lambda与方法引用的方式进行函数式编程；OO是抽象数据，FP是抽象行为；函数式编程语言的所有的数据都是不可变的，不可变对象与无副作用解决了并发编程的共享变量的问题，通常是并行编程的解决方案。

Lambda表达式是使用最小可能语法编写的函数定义，Lambda表达式产生函数而不是类，Lambda语法尽可能少；Lambda表达式的基本语法是：

参数->方法体；没有参数用()表示，一个参数可以去掉()；

方法体多行使用{}表示，必须用return返回值，如果只有一行，可以不使用return；

方法引用：类名或者对象名::方法名称；

未绑定的方法引用是指没有关联对象的普通（非静态）方法，使用未绑定的引用时，函数式方法的签名不会与方法引用的签名匹配，必须先提供对象来调用方法；

构造函数也可以作为引用；

Lambda表达式与方法引用必须对应Java中类，因为Java的基础就是类，但是Lambda表达式是无法单纯的看出来表示的是哪个类，为此，Java8中加入了java.util.function包，里面类就是Lambda表示与方法引用类型；这个包下都是接口，每个接口都只有一个方法，称为函数式方法，并且接口使用@FunctionInterface注解，如果将方法引用或 Lambda 表达式赋值给函数式接口（类型需要匹配），Java 会适配你的赋值到目标接口。 编译器会在后台把方法引用或 Lambda 表达式包装进实现目标接口的类的实例中。

java.util.function包含完整的目标接口类型定义，并且名字有一定的规则。

* 只处理对象而非基本类型，名称是Function、Consumer、Predicate，参数类型通过泛型添加；
* 如果接收的的参数是基本类型，则类型在名字前面展示，比如：LongConsumer等；
* 如果返回值是基本类型，则用To表示，比如ToLongFunction<T>、IntToLongFunction等；
* 如果参数类型=返回值类型，则是一个Operator，单个用UnaryOperator，2个的用BinaryOperator；
* 接收参数并返回布尔值，则是一个谓词-Predicate；
* 如果接口的2个参数类型不同，则名称有一个Bi。

下面是名字的具体的列表：

|  |  |
| --- | --- |
| 特征 | 函数式方法名 |
| 无参数/无返回值 | Runnable |
| 无参数、返回类型任意 | Supplier.get() getAs类型() Callable<T> |
| 1参数，无返回值 | Consumer |
| 2参数Consumer | BiConsumer |
| 2参数Consumer | Obj基本类型Consumer |
| 1参数返回类型不同 | Function、To基本类型Function，基本类型To基本类型Function |
| 1参数，返回类型相同 | UnaryOperator |
| 2参数，返回类型相同 | BinaryOperator |
| 2参数，类型相同，返回整型 | Comparator |
| 2参数，返回布尔型 | Predicate |
| 参数基本类型返回基本类型 | 基本类型To基本类型Function |
| 2参数，类型不同 | Bi操作 |

在使用函数接口时，名称无关紧要——只要参数类型和返回类型相同。

高阶函数是消费或者产生函数的函数，使用继承，可以为专用接口创建别名，使用Lambda表达式可以在方法中创建与返回函数；Java8提供了有限与合理的闭包支持，返回的函数中，使用到对象的属性，没有final要求，如果使用到了局部变量，则必须是final的或者等同final的，与匿名内部类的要求是一致的；等同final效果就是直接在变量声明前加上final关键字而不用更改任何其余代码，应用与对象引用的final关键字仅表示不会重新赋值引用，但是可以修改对象本身。

函数组合（Function Composition）是多个函数组成新的函数，支持的组合如下：

|  |  |
| --- | --- |
| 组合方法 | 支持接口 |
| andThen(argument)，执行原操作，再执行参数操作 | \*Function、\*Consumer、\*Operator |
| compose(argument)，执行参数操作，再执行原操作 | Function、\*Operator |
| and(\*Predicate)原谓词与参数谓词的短路逻辑与 | \*Predicate |
| or(\*Predicate)原谓词与参数谓词的短路逻辑或 | \*Predicate |
| negate()逻辑非 | \*Predicate |

柯里化求值来自于Haskell Curry，也就是来自于Haskell编程语言，柯里化意为：将一个多参数的函数转化为一系列单参数的函数，柯里化求值是用来解决什么问题？柯里化的目的是通过提供单个参数来创建新的函数，仙子啊有了一个带参函数与剩下的自由函数，

# 流式编程

流式编程的特征是内部迭代，内部迭代产生的代码可读性强，可以使用多核处理器，通过放弃对迭代过程的控制，并交给并行机制。流是懒加载的，在绝对必要时才计算。流操作的3种类型：

* 创建流；
* 修改流，中间操作，Intermediate Operations；
* 消费流，终端操作，Terminal Operations；

Stream.of()方法将元素转化为流，或者使用集合的stream()方法来生成流。Random类被流的方法增强了，Stream.generate(Supplier<T>) 可以根据supplier来生成一个流，IntStream的range()方法可以用于生成整型序列的流，编写循环更加方便，Stream.iterate()基于前一个元素来生成下一个元素，需要给出一个初始的种子。Stream.builder()是流的建造者模式。Arrays.stream()的静态方法用于把数组转换成流，Pattern的splitAsStream()方法可以把字符串根据模式分隔为字符串流。

中间操作，用于从一个流中获取对象并将对象作为另一个流从后端输出。常用的操作有：

* + peek()：帮助调试，用于无修改的查看流中的元素；
  + sorted([Comparator])；
  + distinct()：消除重复的元素；
  + filter(Predicate)：保留结果是true的元素；
  + map(Function): 转换流元素的类型；
  + flatMap(Function): 把元素流的流转化为元素流；

为了防止流为空中断流的操作，使用Optional可以避免这样的情况；常用的到的有：

* findFirst();
* findAny();
* max()/min();
* reduce();

当流为空的时候会获得Optional.empty而不是抛出异常；解包Optional的便利的函数：

* + ifPresent(Consumer)：值存在时调用Consumer；
  + orElse(otherObject)：值存在时返回，不存在时返回otherObject；
  + orElseGet(Supplier)：值存在时返回，不存在使用supplier生成的对象；
  + orElseThrow(Supplier):：值存在返回，否则使用Supplier函数生成一个异常。

创建Optional有3种方法：empty()、of(value)、ofNullable(value);

终端操作：

* toArray()：将流转换成适当类型的数组；
* toArray(generator)：生成自定义类型的数组；
* forEach(Consumer)：无序操作，并行流时才是无序的
* forEachOrdered(Consumer)：按照流的顺序操作；
* collect(Collector)：使用Collector收集流元素到结果集合中；
* collect(Supplier,BiConsumer,BiConsumer)：第一个参数创建一个初始的结果集，第二个参数用户将流的下一个元素收集到结果集中，第三个参数用于将2个结果集合并起来；
* reduce(BinaryOperator)：使用BinaryOperator组合所有流中的元素；
* reduce(identity,BinaryOperator)：使用identity作为初始值；
* reduce(identity,BiFunction,BinaryOperator)：更复杂的使用形式;
* allMatch(Predicate)：流中的每一个元素经过Predicate都是true；
* anyMatch(Predicate)：流中的任意一个元素经过Predicate为true；
* noneMatch(Predicate)：流中的每个元素经过Predicate都是false；
* findFirst()：返回一个流元素的Optional；
* findAny()：返回含有任意流元素的Optional，在非并行流中就是返回第一个元素，在并行流中返回的不一定是第一个元素；
* count()：流中元素的个数；
* max(Comparator)：决定最大元素；
* min(Comparator)：决定最小元素；
* average()：求取流元素的平均值；
* max()与min()：数值流操作，不需要Comparator;
* sum()对所有流元素求和；
* summaryStatistics()：生成可能有用的数据。

# 第十九章 类型信息

RTTI能够在程序运行时发现与使用类型信息，运行时识别对象与与类信息的2种方式：

* 传统的RTTI：编译时确定的类型信息;
* 反射机制：允许我们在运行时发现与使用类的信息;

# 第二十五章 注解

注解又叫做元数据，正式的方式为代码添加信息，将来可以使用这些信息。注解是为了解决将元数据绑定到源代码的需求，以前都是放在XML等分离的元数据配置文件中的。是对Java表达式的补充，是完整表述程序必须的信息。内置的5个注解:

* @Override:覆盖基类中的方法
* @Deprecated: 使用未来会被移除的API
* @SuppressWarning: 关闭不当的编译警告
* @SafeVarargs: 告诉编译器，在可变长参数中的泛型是类型安全的。 可变长参数是使用数组存储的，而数组和泛型不能很好的混合使用。 简单的说，数组元素的数据类型在编译和运行时都是确定的，而泛型的数据类型只有在运行时才能确定下来
* @FunctionInterface: 标记注解表明是函数式接口

还有5个元注解。注解的定义类似接口，多了一个@符号。必须有元注解@Retention与@Target。@Target定义你可以在何处应用注解，@Retention定义注解的保留范围，有3个范围源代码文件、编译后的Class文件与运行时。没有任何元素的注解就是标记注解。5个元注解:

* @Target: 注解可以应用的地方，类型是ElementType
  + CONSTRUCTOR: 构造器声明
  + FIELD: 字段声明，包括枚举常量
  + LOCAL\_VARIAVBLE: 本地变量声明
  + METHOD: 方法声明
  + PACKAGE: 包声明
  + PARAMETER: 参数声明
  + TYPE: 类或者接口或者枚举的声明
* @Retention: 注解信息可以保存多久，类型是`RetentionPolicy`
  + SOURCE: 注解会被编译器丢弃，只在源代码中
  + CLASS: 主机在类文件中可以被编译器使用，但会被虚拟机丢弃
  + RUNTIME: 虚拟机保留，反射可以获取到注解信息
* @Documented: 在Javadoc中引入该注解
* @Inherited: 允许子类继承父类注解
* @Repeatable: 可以多次应用与同一个声明

通过反射编写注解处理器或者通过javac的编译器钩子在编译时使用注解。通常通常是使用getDeclaredMethods()，Class、Method、Field等都实现了AnnotatedElement接口。注解允许的元素类型:

* 所有的基本类型
* String
* Class
* Enum
* Annotation
* 以上任何类型的数组

所有元素都要有默认值，不能是null。通常可以通过注解生成外部描述文件，比如生成SQL建表语句、生成XML文件等都是很多框架常用的。以前要程序员自己定义，现在只需要维护一个代码源就可以了。如果不放@Target，则默认可以放到所有类型上。

通过javac可以创建编译时注解。注解处理器会不断的创建新的文件，直到不再有新的文件被创建则编译源代码

# 第二十四章 并发

并发是关于正确有效的控制对共享资源的访问，表示不止一个任务正在执行，并行是使用额外的资源来更快的产生结果，表示不止一个任务同时执行。并发解决了阻塞等待问题，常见于IO密集型任务，并行解决多核处理器利用率问题，用多核处理器同时运行任务的不同部分来加快执行，常见于CPU密集型任务，并行执行IO密集型不好，因为瓶颈是IO阻塞等待，单个处理器上用并发来解决CPU密集型也没什么卵用；他们都是试图在更短的时间完成更多的工作；实现加速的方式不同。

在Java中都是使用线程来实现并发与并行，分类法：

* 纯并发：单个CPU上运行多任务，IO密集型会提升效率；
* 并发-并行：用并发编程的方式工作于多处理器上，实现并行；
* 并行-并发：用并行编程的方式工作于单处理器，可能没什么提升；
* 纯并行：并行编程多处理器。

支持并发的语言与库对并发开发都是比较脆弱的抽象，抽象的目标是提取重要的属性，忽略无关的细节，如果抽象不好，忽略的细节就会对抽象产生影响，从而导致整个抽象是不正确的。并发编程不是完美的抽象是因为，开发者永远都需要理解底层的系统、工具或者CPU是如何工作的，并发编程的本质就是程序可能在某些情况下有效，在某些情况下无效。

纯函数式语言解决了大量的并发问题，但是编程中还是要理解底层的实现机制。

并发是一系列性能技术，专注于减少等待。详细的含义：

* + 并发有很多实现方法；
  + 这些技术都是为了提升性能，使用最简单的方法满足需要的性能即可；
  + 并发提升性能的关键是减少等待，如果没有等待，则没有收益；

一般情况下，不使用并发，仅在必要的时候使用，如果没有任务会阻塞，那么在单处理器机器上使用并发是没有意义的；一些编程语言旨在将并发任务彼此隔离。这些通常被称为函数式语言；、。

Java并发的4句格言：

* 不要用它（避免使用并发）；
* 没有什么是真的，一切可能都有问题：
* 仅仅是它能运行，并不意味着它没有问题：
* 你必须理解它

由于 Java 的原始设计，代码的所有部分仍然很脆弱，你仍然必须理解这些复杂和微妙的问题。Java 中的线程绝不是简单或安全的;那种经历必须降级为另一种更新的语言。

高级Java并发结构：

* + Parallel Streams：流很容易并行化，流使用内部迭代的方式，迭代器使用的是Spliterator，这种迭代器，容易分割；并行流不一定特别有优势，流并行算法：
* 流并行将输入数据分成多个部分，算法可以应用单独的部分；
* 数组分割成本低，分割均匀并且对分割的大小有完美的掌控；
* 链表没有办法分拆，因为链表有互相的依赖关系，数组没有；
* 无状态生成器的行为类似数组，range()就是无状态的；
* 迭代生成器的行为类似链表（当前的元素依赖上一个元素），iterate()就是一个迭代生成器。
  + CompletableFutures：completedFuture()设置结果、thenApply在本线程内应用函数处理、thenApplyAsync()异步并发执行、join()阻塞当前调用线程直到CompletableFuture计算完成，get()、静态方法、then系列方法等；obtrudeValue、cancel()方法；
  + Executor：lambdas与方法引用可以通过方法签名来使用（支持结构一致性），可以配合ExecutorService来使用，中断任务又杂又乱，是为了向后兼容，任务终止的最佳方法是设置任务周期性检查标志。

CompletableFuture有各种组合CompletableFuture对象的工作方法:

* runAfterEitherAsync(CompletionStage<?>,Runnable): 当前的Future与参数的future任何一个执行完执行后面的Runbale；
* runAfterBothAsync(CompletionStage<?>,Runnable)：当前的Future与参数的future都执行完执行后面的Runbale；
* applyToEitherAsync(CompletableStage,Function): 当前的Stage与参数的Stage任何一个执行完执行后面的转化函数Function；
* acceptEitherAsync(CompletableStage,Consumer): 当前的Stage与参数的Stage任何一个执行完执行后面的消费函数；
* thenAcceptBothAsync(CompletableStage,Consumer): 当前的Stage与参数的Stage都执行完再执行后后面的Consumer；
* thenCombineAsync(CompletableStage,BiFunction): 当前的Stage与参数的Stage都执行完再执行后面的BiFunction；
* anyOf(): 返回任何一个执行完的Future；
* AllOf(): 返回一个都执行完的Future；

CompletableFuture会缓冲过程中发生的异常，只有在获取结果时才会被告知，isCompletedExceptionally方法可以检测执行过程中是否抛出了异常，抛出异常与正常结束，都算是Future结束，isDone都会返回true，可以使用exceptionally()仅在出现异常时运行，handle()调用时检查，可以返回新的类型， whenComplete()只是简单的消费，3个方法来处理异常；

Stream在没有终端操作的情况下，不做任何事情。

死锁发生的4个条件：

* + 互斥条件：任务使用的资源至少有一个不能共享；
  + 至少有一个任务持有了资源，并请求另外一个别的任务持有的资源；
  + 资源不能被抢占；
  + 必须循环等待。

Java语言规范：没必要使构造器的线程同步，因为会锁定正在构造的对象，直到构造期完成初始化后才对其他线程可见； 但是构造器有更微妙的共享方式，比如域初始化或者构造函数参数，这写都可能造成得到中间状态.

# 第二十五章 设计模式

模式是解决特定类问题的一种特别巧妙且有深刻见解的方法；模式就是添加抽象层，抽象一些东西时，就是在剥离特定的细节，动机就是“将易变的事物与不变的事物分开”；还有程序中的某些部分可能因某种原因发生变化，要保持这些变化不会引起整个代码的其他变化（对修改关闭，对扩展开放）。

设计中最难得是发现最易改变的地方，设计模式的目的是隔离代码中的更改，设计模式代表语言的失败。

* 单例模式：类只有一个对象实例(静态类的方式创建)

设计模式有23种，根据可能变化的方面分为3种类别：

* + 创建型：如何创建对象，涉及对象创建的细节，代码可以不依赖具体的兑现的类型，在添加新类型的对象时不会更改，单例模式是创建模式；
  + 构造型：设计对象以满足特定的项目约束，它们处理对象与其他对象连接的方式，以确保系统中的更改不需要更改这些连接；
  + 行为型：处理程序中特定类型的操作的对象，封装要执行的过程；

## 构建应用程序框架

从一个类或者一组类开始，创建一个新的应用程序，重用类中的大部分代码，根据需要覆盖一个或者多个方法来定制程序。

* 模板方法模式：隐藏在底层，通过调用基类中的各种防方法来驱动应用程序，特点是它是在基类中定义的，不能变更；
* 代理模式与桥接模式：完成工作的类隐藏在代理类后面，代理模式是桥接模式的特殊情况，代理模式只有一个实现，桥接模式有多个实现，允许动态更改实现；
* 状态模式：与代理模式差不多，只是代理的实现可以变更；
* 工厂模式：添加新类型时，使用多态性为这些新类型创建通用的接口，将系统中的其余代码与特定类型的信息分开，可以在不改变现有代码的情况下，添加新的类型；

# 附录：并发底层原理

并发将一个程序分为独立的任务，每个任务都由一个执行线程来驱动，简称为线程；在操作系统中，线程就是一段顺序的代码执行逻辑，这种模型简化了一个进程的并发多任务过程，好像每个线程都由自己的处理器；

Java中代表进程的是Thread类，后来因为Thread的管理比较麻烦，引入了中间层Executor等；Thread是将任务关联到处理器的一种软件概念，与普通的对象是不同的，Thread对象分配在内存中专为线程保留的区域，用于提供运行任务的一切环境，包括：

* 程序计数器，字节码指令地址；
* java方法栈，包含方法的局部变量与堆引用，每个栈大小约64K~1M；
* native code执行的栈；
* thread-local 变量的存储区域；
* 控制线程的状态管理变量；

线程必须绑定到操作系统，这样可以在合适的时机被处理器调度，java使用底层操作系统的机制来管理线程的执行。

对于计算密集型任务来说，线程的最佳数量就是物理内核数，需要注意的Intel的处理器使用了超线程（硬件技巧，单个处理器上产生快速的上下文切换）的概念，但是真实的计算核数还是没有增加；对于其他类型任务，可以不遵循这个规则。

线程的创建是有上限的，取决于-Xss(每个线程栈分配的内存大小) -Xmx（最大内存分配），超过上限OOM。

使用newWorkStealingPool创建的线程池，会根据处理器的个数创建指定大小的线程池，并且会执行任务窃取算法，底层是fork/join机制。

使用submit提交的任务抛出异常无显示，execute处理的任务抛出异常打印到控制台



线程里面的异常没有捕获，打印到控制台，使用Thread.UncaughtExceptionHandler处理线程抛出的异常，重写ThreadFactory：



Thread.setDefaultUncaughtExceptionHandler()静态方法是一个全局异常处理器，只有在当前的线程或者线程组里面没有找到异常处理器时调用。

并发的共享资源问题，导致可能产生互相干扰的问题，自增操作不是线程安全的。解决资源竞争的问题是加锁，也就是序列化访问共享资源的方式，通常用一个子句来实现锁的机制，也被称为mutex(互斥量)，java中使用**synchronized**

子句表示，加的锁是对象的monitor锁；一个变量可能被别的线程读取或者读取别的线程做的修改，就需要进行锁同步。

使用volatile的3个理由：

* + 字分裂，因为大数据类型，比如long与double是分为2个32bit的字来写入或者读取的，多线程的情况下，中间可能读取到脏数据，volatile修饰可以防止字分裂；
  + 可见性，每个任务都由自己的处理器，也有本线程的快速缓存，执行时不必每次都去主内存中读取，但是写入不是及时的，这就是缓存一致性，volatile可以防止编译器优化，读写变量直接是内存，不会被缓存；

使用volatile的变量特点：

* 该变量被多个任务访问；
* 访问中至少又一次是写操作；
* 尝试不使用同步的方式做并发安全。

多个线程对变量写入时，如果使用到旧值决定新值，一定要使用同步的方式，因为volatile也不能保证并发安全；同步也会让主内存变量刷新。

只要不会影响结果，jvm会重排指令来优化性能，但是重排会影响处理器缓存与主内存的交互，volatile关键字可以阻止重排volatile变量周围的读写指令；这称为happens before担保原则；volatile变量之前的指令一定在volatile变量读写之前完成，之后的也是之后完成；此时volatile称为memory barrier（内存屏障），happens before担保确保变量读写不能跨过内存屏障进行重排。同步也会让主内存刷新，所以如果一个变量完全由 synchronized 的方法或代码段(或者 java.util.concurrent.atomic 库里类型之一)所保护，则不需要让变量用 volatile。

happens before还保证，当写入volatile变量时，之前的所有变量会刷新到主内存，读取一个volatile变量时，其他变量与volatile变量会从主内存刷新到缓存。目前不建议使用volatile关键字了，更好的方式是使用atomic的原子类，以较低的成本保证了并发安全。

java中的单一操作，并不是原子操作，因为可能被分解为多个CPU指令，特别是long，double这种多字的，所以不是原子的。

Goetx测试：如果你可以编写用于现代微处理器的高性能JVM，那么就有资格考虑是否可以避免同步；

原子操作不能被线程机制中断，所以可以用来编写无锁代码（lock-free-code），不需要同步，但是多核处理器上原子性不能避免可见性，同步可以保证可见性，使用原子操作访问不稳定的处于中间状态的对象也容易造成同步问题，这时访问的对象也需要同步，最重要的是不要有共享变量。

java5引入了原子变量类，Atomic\*类，这些类保证共享对象不会处于中间的不稳定状态。这些无锁的操作性能比synchronized 要好很多；利用了现代机器CPU的机器级原子性。

synchronized 不仅可以用于方法，也可以用于代码块，包含的代码部分被成为临界区，代码块被称为同步控制块，同步代码块性能比同步方法好很多。

java.util.concurrent包下有显示定义的同步锁类，代码虽然不如synchronized 优雅，但是处理问题更加灵活；synchronized 失败后会抛出异常，不会释放锁，Lock类可以在finally中清理，保持正常状态。

java.util.concurrent里面有很多用于处理并发问题的类，但是是比并行流与CompletableFuture更底层的机制。

* + DelayQueue：无界阻塞队列，用于放置Delayed接口的对象，只有对象到期时才会进入到可用状态的子队列中，Delayed接口的getDelay方法返回0时代表对象到期，对象没有到期时，是一种中间状态，此时poll拿不到，拿到的是null，所以这个队列不能存null值，因为没办法判断是否是此时队列可用对象为空；策略模式：算法的一部分是作为参数传递进来的；
  + PriorityBlockingQueue：按照优先级在队列中排序；类似SortedSet;

无琐集合：早期的集合并法安区是直接使用的synchronized ，这造成性能比较差；无琐集合的特性：只要读取者仅能看到已完成修改的结果，对集合的修改就可以同时发生在读取发生时。实现策略有2种：

* 复制策略：修改是在不可见的副本上完成，修改完成后使用原子操作与主题数据交换，CopyOnWriteArrayList与CopyOnWriteArraySet与ConcurrentHashMap都使用类似的技术；
* CAS：

编写并法程序的步骤：

* + 避免写并法程序；
  + 使用现代高级的并法工具；
  + 不要在任务间共享变量，在任务之间必须传递的任何信息都应该使用 Java.util.concurrent 库中的并发数据结构；
  + 如果必须在任务之间共享变量，请使用 java.util.concurrent.atomic 里面其中一种类型，或在任何直接或间接访问这些变量的方法上应用 synchronized；

# 附录：shutdownhook

有的JVM进程需要优雅的关闭，JVM关闭分为2种方式:

* 运行流程结束;
* 突然的关闭;

运行流程结束的情况:

* 最后的非daemon线程终止，当main线程执行完毕，则JVM开启关闭处理;
* 从OS发送中断信号，比如ctrl+C;
* 在代码中调用System.exit();

JVM进程能够优雅的关闭最好，但是也会以不可预料的方式突然的关闭.

* 从OS发送kill信号，比如kill -9 <jvm\_pid>
* 在java代码中调用Runtime.getRuntime().halt()
* 宿主机操作系统终止允信，比如断电或者操作系统错误

JVM可以注册关闭前要运行的任务，这样的任务非常适合用来释放资源或者是类似的house-keeping任务的好地方，在JVM术语中，这些关闭任务叫做shutdownhook，shutdownhook是一个已经初始化但是还没开始运行的线程，当JVM开始执行关闭处理时，JVM会以一个未定义的顺序启动所有的hook，运行完hook后，JVM停止。

使用Runtime.getRuntime().addShutdownHook()添加hook，比如

**Thread** printingHook = **new** **Thread**(() -> System.out.println("In the middle of a shutdown")); Runtime.getRuntime().addShutdownHook(printingHook);

JVM负责start hook线程，如果hook线程已经启动过了，那么addShutdownHook会抛出异常，已经注册的hook也可以通过remove的方式移除。只有JVM是正常终止的才会运行hook，Runtime.getRuntime().halt(129)的终止方式也不会运行hook.