链滴

读书笔记 --Java 核心技术 -- 基础篇

作者：KioLuo

原文链接：https://ld246.com/article/1492567542367

来源网站：链滴

第三章　Java基本程序设计----------

Java有8种基本类型

4种整型：int, long, short, byte；长整型后缀加L，0x前缀表示十六进制，0前缀表示八进制，0b前表示二进制

2种浮点类型：float, double；后缀F表示float，后缀D表示double，默认使用double

常量Double.POSITIVE_INFINITY，Double.NEGATIVE_INFINITY，Double.NaN，表示正无穷大，无穷大，非数值。

判断是否非数值用Double.isNaN(x)

1种字符类型：char，采用UTF-16编码描述一个代码单元，最好不要在程序中使用char类型，除非需对UTF-16代码单元进行操作

1种boolean类型，整型和布尔类型不能互相转换

final定义常量，static final定义一个类常量，const为Java保留的关键字，但目前没有使用

strictfp关键字标记的方法必须使用严格的浮点计算

String类常用方法：

```
char charAt(int index);        //返回给定位置的代码单元，一般用于了解底层代码单元，较少使用

int codePointAt(int index);          //给定位置开始或结束的代码点

int offSetByCodePoints(int startIndex, int cpCount);        //返回从startIndex索引开始，位移cpCunt后的代码点索引

int compareTo(String other);        //比较字符串大小，若字符串位于other之前返回负，字符串位于ther之后返回正，如果字符串相等则为0

boolean endsWith(String suffix);        //是否以suffix结尾

boolean startsWith(String prefix);      //是否以prefix开头

boolean equals(Object other);       //判断是否相同

boolean equalsIgnoreCase(String other);         //忽略大小写判断是否相同

int indexOf(String str);

int indexOf(String str, int fromIndex);

int indexOf(int cp);

int indexOf(int cp, int fromIndex);         //返回与字符串str或代码点cp匹配的第一个子串的开始位


int lastIndexOf(String str);
```

```
int lastIndexOf(String str, int fromIndex);

int lastIndexOf(int cp);

int lastIndexOf(int cp, int fromIndex);          //返回与字符串str或代码点cp匹配的最后一个子串的始位置

int length();       //返回字符串长度

int codePointCount(int startIndex, int endIndex);          //返回startIndex和endIndex-1之间的代点数量

String replace(charSequence oldString, charSequence newString);          //用newString代替字串中所有的oldString

String substring(int beginIndex);

String substring(int beginIndex, int endIndex);          //提取从beginIndex到串尾或endIndex-1的符串

String toLowerCase();          //将所有大写转换为小写

String toUpperCase();          //小写转大写

String trim();          //返回删除了串头和串尾空格的新字符串
```

StringBuilder常用方法：

```
int length();

StringBuilder append(String str);

StringBuilder append(char c);

StringBuilder appendCodePoint(int cp);

StringBuilder setCharAt(int i, char c);

StringBuilder insert(int offset, String str);

StringBuilder insert(int offset, char c);

StringBuilder delete(int startIndex, int endIndex);

String toString();
```

从控制台读取密码：

```
Console cons = System.console();

String userName = cons.readLine("User name: ");
```

char[] passwd = cons.readPassword("Password: ");

日期与时间格式化:

%<参数索引>$<标志><宽度>t<转换字符>

%<参数索引>$<标志><宽度>.<精度><转换字符>

文件输入输出:

Scanner in = new Scanner(Paths.get(myfile.txt)); //用File对象构造一个scanner对象

PrintWriter = new PrintWriter("myfile.txt"); //构造PrintWriter对象写入文件

Scanner(File f);

Scanner(String data);

PrintWriter = new PrintWriter(String fileName);

static Path get(String pathName);

带标签的break语句，标签需放在希望跳出的最外层循环之前，并紧跟一个冒号

大数值

BigInteger和BigDecimal

BigInteger a = BigInteger.valueOf(100);

不能使用算术运算符，需要使用方法add和multiply、divide

数组

int[] b = Arrays.copyOf(a, a.length); //将数组a的内容拷贝到b中

Arrays.sort(a); //对数组a中的元素采用快速排序法进行排序

Arrays.toString(a); //将数组a转化为字符串如[1, 2, 3, ...]

Arrays.binarySearch(type[] a, type v); //二分搜索法查找值v，查找成功则返回下标值，否则回负数值r

Arrays.fill(type[] a, type v); //将数组a中的所有元素设置为值v

Arrays.equals(type[] a, type[] b); //判断两数组是否相等

Arrays.deepToString(a); //将多维数组a转化为字符串

第四章 对象与类----------

常用预定义类

java.util.GregorianCalendar 1.1

GregorianCalendar()　　//构造一个日历对象表示默认地区默认时区的当前时间

GregorianCalendar(int year, int month, int day)　　//用指定时间构造一个日历对象

int get(int field)　　//返回给定域的值

void set(int field, int value)　　//设置指定域的值

void set(int year, int month, int day)　　//设置新时间

void add(int field, int amount)　　//对指定域进行计算

int getFirstDayOfWeek()　　//获取当前用户所在地区星期的第一天

void setTime(Date time)　　//将时间设置为指定点

Date getTime()　　//获取当前时间

java.text.DateFormatSymbols 1.1

String[] getShortWeekdays()

String[] getShortMonths()

String[] getWeekdays()

String[] getMonths()

需要可变数据域的拷贝，用clone()方法

一个方法可以访问所属类的所有对象的私有数据

Java的构造器可以通过this()调用同一个类的另一个构造器，而C++不能

Java可以直接初始化实例域的初值，或者调用方法对域直接初始化；而C++不能直接在声明中直接初化，但可以用初始化列表语法

初始化块，静态初始化块

java.util.Random 1.0

Random()　　//构造一个新的随机数生成器

int nextInt(int n)　　//返回一个1 ~ n-1之间的随机数

可以为任何一类添加finalize方法，finalize方法将在垃圾回收器清除对象之前调用

设置类路径

最好采用-classpath或-cp选项指定类路径，如

java -classpath /home/user/classdir:.:/home/user/achives/achive.jar MyProg

或

java -classpath c:\classdir;.;c:\achives\achive.jar MyProg

javac编译器总是在当前目录中查找文件，而Java虚拟机仅在类路径中有"."目录时才查看当前目录，果设置类路径忘记包含"."，则程序能通过编译，却不能运行

javadoc工具可以由源文件生成一个html文档

注释以/*开始，以/结束

@author作者描述

@param变量描述

@return返回描述

@throws抛出异常描述

包与概述注释：单独的package.html或package-info.java和overview.html

注释抽取：

javadoc -d docDirectory packageName　　//一个包

javadoc -d docDirectory packageName1 packageName2...　　//多个包

javadoc -d docDirectory *.java　　//默认包

第五章 继承----------

在覆盖方法时，一定要保证返回类型的兼容性。允许子类将覆盖方法的返回类型定义为原返回类型的类型

在覆盖一个方法时，子类方法不能低于超类方法的可见性

final类和方法

final类不允许扩展，即阻止继承，没有子类，final类中的方法自动成为final方法，即不能覆盖这个方。也可以单独定义final方法

只能在继承层次内进行类型转换，在将超类转换成子类时应用instanceof进行检查

abstract抽象类和抽象方法，包含一个或多个抽象方法的类本身必须声明为抽象类，若子类没有定义部的抽象方法，则子类也必须声明为抽象类

类即使不含抽象方法，也可以声明为抽象类，抽象类不能被实例化

protected对本包和所有子类可见，默认（不用修饰符）为对本包可见

子类中的方法只能访问子类对象中被protected的域，而不能访问其他超类对象中protected的域

Object类

equals方法

如果子类能够拥有自己的相等概念，则对称性需求强制采用getClass检测；如果由超类决定相等的概，那么可以用instanceof进行检测。

对于数组类型的域，可以使用静态方法Array.equals()检测数组元素是否相等

java.util.Arrays 1.2

static boolean equals(type[] a, type[] b)　　//如果两个数组的长度相同，而且对应位置上元素也相同，则返回true，数组的元素类型可以是Object或其他基本类型

java.util.Objects 7

static boolean equals(Object a, Object b)　　//如果都为null则返回true，如果只有其中一为null，返回false，否则返回a.equals(b)

hashCode()方法

如果重新定义equals()方法，就必须重新定义hashCode()方法，这两个定义必须一致

java.lang.Object 1.0

int hashCode()　　//返回对象的散列码，两个相等的对象要求返回相等的散列码

java.lang.Objects 7.0

int hash(Object,...,Object)　　//返回一个散列码，由提供的所有对象的散列码组合得到

static int hashCode(Object a)　　//如果a为null返回0，否则返回a.hashCode()

java.util.Arrays 1.2

static int hashCode(type[] a)　　//计算数组a的散列码

toString()方法

getClass().getName()　　//获得类名的字符串

只要对象与一个字符串通过操作符“+”连接起来，Java编译就会自动调用toString()方法

Arrays.toString(type[] a)　　//打印数组

Arrays.deepToString(type[][] a)　　//打印多维数组

java.lang.Object 1.0

Class getClass()　　//返回包含对象信息的类对象

boolean equals()　　//比较两个对象是否相等，如果两个对象指向同一块区域为true

String toString()　　//返回描述该对象值的字符串

java.lang.Class 1.0

String getName()　　//返回类名字

Class getSuperClass()　　//返回该类的超类信息

泛型数组

ArrayList()　　//构造一个空数组列表，T不允许为基本类型

.add()　　//增加元素

.ensureCapacity(int n)　　//确定分配n个元素的数组

.size()　　//返回数组列表的当前元素数量

.trimToSize()　　//确认数组大小不再变化时，调用该方法将存储空间大小调整为当前元素数目需空间

.toArray(type[] a)　　//转换为数组并存到a

java.util.ArrayList 1.2

void set(int index, T obj)　　//设置指定位置的元素值

T get(int index)　　//返回指定位置的元素值

void add(int index, T obj)　　//向后移动元素以便插入新元素值

T remove(int index)　　//删除一个元素，并将后面的元素向前移动，返回所删除元素值

包装器，基本类型对应的类

自动装箱，自动拆箱

java.lang.Integer 1.0

int intValue()　　//以int形式返回Integer的值

static String toString(int i)　　//以一个新String对象的形式返回给定数值i的十进制表示

static String toString(int i, int radix)　　//返回数值i的给定radix参数进制的表示

static int parseInt(String s)

static int parseInt(String s, int radix)　　//返回字符串s表示的数值，给定字符串表示的是radix参进制

static Integer valueOf(String s)

static Integer valueOf(String s, int radix)

java.text.NumberFormat 1.1

Number parse(String s)　　//返回给定字符串s表示的数值

参数数量可变的方法

method(Object... args)　　//args为参数数组

枚举类

public enum Size { SMALL, MEDIUM, LARGE, EXTRA_LARGE };

所有枚举类型都是Enum类的子类，可以在枚举类中添加构造器、方法和域，构造器只是在构造枚举量时被调用

Size.SMALL.toString();　　//将返回字符串SMALL

Size s = Enum.valueOf(Size.class, "SMALL");　　//toString()的逆方法，返回Size.SMALL

Size[] values = Size.values();　　//返回包含全部枚举值的数组

Size.SMALL.ordinal();　　//将返回枚举常量的位置，这里返回0

Size.SMALL.compareTo(Size MEDIUM)　　//如果SMALL出现在枚举常量MEDIUM前，则返回个负值，相等返回0，否则返回正值

反射

Class类

java.lang.Class 1.0

e.getClass()　　//返回Class类的实例

e.getClass().getName()　　//返回类的名字

e.getClass().newInstance()　　//快速创建一个类的实例，使用默认构造器

Class.forName(className)　　//返回类名对应的Class对象，手工加载类

E.class　　//返回类型E对应的Class类的实例

java.lang.reflect.Constructor 1.1

Object newInstance(Object[] args)　　//构造一个这个构造器所属类的新实例

java.lang.Throwable 1.0

void printStackTrace()　　//打印栈的轨迹输出到标准错误流

java.lang.Class 1.0

Field[] getFields()

Field[] getDeclaredFields()　　//返回一个包含Field对象的数组，记录了这个类和其超类的公有域

Method[] getMethods()

Method getMethod(String name, Class... parameterTypes)　　//根据方法签名返回特定方法

Method[] getDeclaredMethods()　　//返回一个包含Method对象的数组，getMethods()将返所有公有方法，包括从超类继承的公有方法；getDeclaredMethods()返回这个类或接口的全部方法不包括由超类继承的方法

Constructor[] getConstructors()

Constructor[] getDeclaredConstructors()　　//返回包含Constructor对象的数组

java.lang.reflect.Field

java.lang.reflect.Method

java.lang.reflect.Constructor

Class getDeclaringClass()

Class[] getExceptionTypes()

int getModifiers()

String getName()

Class[] getParameterTypes()

Class getReturnType()

java.lang.reflect.Modifier

static String toString(int modifiers)

static boolean isAbstract(int modifiers)

static boolean isFinal(int modifiers)

static boolean isPublic(int modifiers)　　//检测对应修饰符在modifiers中的位

...

```
Class cl = harry.getClass();

Field f = cl.getDeclaredField("name");

Object v = f.get(harry)              //返回一个对象，其值为harry对象相应的该域的值

f.setAccessible(true);      //setAccessible()方法是AccessibleObject类中的一个方法，是Field，Method，Constructor的公共超类

AccessibleObject.setAccessible(fields, true)          //设置fields域可访问

f.set(obj, value)          //将obj对象的f域设置为新值
```

java.lang.reflect.Array

```
static Object get(Object array, int index)

static xxx getXxx(Object array, int index)          //xxx是基本类型的一种

static void set(Object array, int index, Object newValue)

static void setXxx(Object array, int index, xxx newValue)          //xxx是基本类型的一种

static int getLength(Object array)

static Object newInstance(Class componentType, int length)

static Object newInstance(Class componentType, int[] lengths)          //返回一个具有给定类型定维数的新数组
```

调用任意方法

建议仅仅在必要的时候才使用Method对象，最好使用接口和内部类

java.lang.reflect.Method 1.1

```
public Object invoke(Object implicitParameter, Object[] explicitParameters)              //调用这对象所描述的方法，传递给定参数，返回方法的返回值
```

第6章--接口与内部类----------

接口

接口中的方法自动地属于public，因此不必提供关键字public

java.lang.Comparable 1.0

```
int compareTo(T other)              //用这个对象和other进行比较，如果这个对象小于other返回数，相等为0，否则返回正值
```

java.util.Arrays 1.2

static void sort(Object[] a)　　//使用mergesort算法对数组a中的元素进行排序，要求数组中的素必须实现了Comparable接口的类，并且元素之间可比较

java.lang.Integer 7

static int compare(int x, int y)　　//如果x

接口中不能包含实例域或静态方法，可以包含常量，常量自动为public static final

Cloneable接口是标记接口，没有方法

javax.swing.JOptionPane 1.2

static void showMessageDialog(Component parent, Object message)　　//显示包含一条息和OK按钮的对话框，这个对话框将位于其parent组件的中央，若parent组件为null，则位于屏幕央

javax.swing.Timer 1.2

Timer(int interval, ActionListener listener)　　//构造一个定时器，每隔interval毫秒就通告listener一次

void start()

void stop()

java.awt.Toolkit 1.0

static Toolkit getDefaultToolkit()　　//获得默认的工具箱，包含GUI环境的信息

void beep()　　//发出一声铃响

内部类

内部类既可以访问自身数据域，也可以访问外围类对象的数据域

内部类的对象总有一个对外围类对象的隐式引用，称为outer

内部类可以私有，常规类只有包可见性和公有可见性

内部类语法规则

OuterClass.this　　//外围类引用的正规语法

OuterClass.InnerClass innerObject = outerObject.new InnerClass(construction parameters)　//构造内部类对象，在外围类的作用域之外引用公有内部类

局部内部类

局部类不能用public或private修饰符进行声明，它的作用域限定在声明这个局部类的块中

局部类不仅能够访问包含他们的外部类，还能访问声明为final的局部变量

final关键字可以应用于局部变量、实例变量和静态变量。创建变量之后只能赋值一次

匿名内部类

语法格式：

```
new SuperType(construction parameters)
{
    inner class methods and datas
}
```

或

```
new InterfaceType()
{
    methods and datas
}
```

利用内部类语法进行“双括号初始化”：

如

```
method(new ArrayList() {{ add("Harry"); add("Tony"); }})          //外层括号建立了ArrayList的一匿名子类，内层括号是一个对象构造块
```

静态内部类

在内部类不需要访问外围类对象的时候，应该使用静态内部类，而且只有内部类可以声明为static，态方法中必须使用静态内部类

声明在接口中的内部类自动成为static和public类

代理类

所有的代理类都扩展于Proxy类，一个代理类只有一个实例域：调用处理器，即实现了InvocationHandler接口的类对象。

代理类是在程序运行过程中创建的

java.lang.reflect.InvocationHandler 1.3

```
Object invoke(Object proxy, Method method, Object[] args)       //定义代理对象调用方法时希
```

执行的动作

java.lang.reflect.Proxy 1.3

static getProxyClass(ClassLoader loader, Class[] interfaces)　　//返回实现指定接口的代理类

static Object newProxyInstance(ClassLoader loader, Class[] interfaces, InvocationHandler handler)　　//构造一个实现指定接口的代理类的实例，所有方法都将调用给定代理处理器对象的invoke)方法

static boolean isProxyClass(Class c)　　//如果是一个代理类则返回true

第10章--部署应用程序和applet----------

JAR文件

JAR文件打包命令：

jar cvf JARFileName File1 File2 ...

添加清单文件到JAR文件中：

jar cfm JARFileName ManifestFileName ...

更新清单文件：

jar ufm JARFileName ManifestFileName

使用e选项指定程序的入口点

jar cvfe MyProgram.jar com.company.mypkg.MainAppClass filesToAdd

启动jar应用程序

java -jar MyProgram.jar

在windows平台中，双击jar文件，通过javaw -jar命令相关联来启动文件，与java命令不同，javaw打开shell窗口

java.lang.Class 1.0

URL getResource(String name)

InputStream getResourceAsStream(String name)　　//找到与类位于同一位置的资源，返回个可以加载资源的URL或输入流。如果没有则返回NULL

密封包：

在清单指令中加入：

Name: com/company/mypkg/

Sealed: true

第11章--异常、断言、日志和调试----------

异常

Java异常层次结构

{ Throwable { Error, Exception { IOException, RuntimeException } } }

未检查异常（unchecked），派生于Error, RuntimeException的所有异常

已检查异常（checked），所有其他异常

一个方法必须声明所有可能抛出的已检查异常，而未检查异常要么不可控制，要么就应该避免发生

java.lang.Throwable 1.0

```
Throwable()          //默认Throwable构造器
Throwable(String message)          //构造一个新的Throwable对象，带有特定的详细描述信息
Throwable(String message, Throwable cause)          //用给定的原因构造一个Throwable对象
String getMessage()          //获得Throwable对象的详细描述信息
```

如果编写一个覆盖超类的方法，而这个方法又没有抛出异常，那么这个方法就必须捕获方法代码中的一个已检查异常，不允许在子类的throws说明符中出现超过超类方法所列出的异常类范围

捕获多个异常

catch(FileNotFoundException | UnknownHostException e) {...}

捕获多个异常时，异常变量隐含为final变量

使用包装技术，抛出子系统中的高级异常，而不丢失原始异常信息

```
Throwable se = new ServletException("database error");
se.initCause(e);                    //将原始异常设置为新异常的原因
throw se;
Throwable e = se.getCause();
```

带资源的try()语句

```
try (Resource res = ...)
{
...
}
```

try块退出时，会自动调用res.close()，当close()抛出异常时，原来的异常会被重新抛出，close()的异被抑制，并通过addSuppressed()方法被增加到原来的异常中，通过getSuppressed()获取

分析堆栈跟踪（StackTrace）元素

StackTraceElement[] frames = e.getStackTrace();　　//得到StackTraceElement对象的数组

Thread.getStackTrace()　　//产生所有线程的堆栈跟踪

java.lang.StackTraceElement 1.4

String getFileName()　　//返回这个元素运行时的源文件名，不存在则返回null

int getLineNumber()　　//返回这个元素运行时的源文件行数，不存在则返回-1

String getClassName()　　//返回这个元素运行时对应的类的全名

String getMethodName()　　//返回这个元素运行时对应的方法名

boolean isNativeMethod()　　//如果这个元素运行时在一个本地方法中，返回true

String toString()　　//如果存在的话，返回一个包含类名、方法名、文件名、行数的格式化字符

断言

断言只应该用于在测试阶段确定程序内部的错误位置，当代码发布时，这些插入的检测语句会被自动走

assert 条件;

assert 条件 : 表达式;　　//若为false则抛出AssertionError异常

启用和禁用断言

在运行程序时用-enableassertions或-ea启用

java -enableassertions MyApp

java -ea:MyClass -ea:com.mycompany.mypkg... MyApp

用选项-disableassertions或-da禁用特定类和包的断言

对于系统类，需要使用-enablesystemassertions和-esa开关启用断言

java.lang.ClassLoader 1.0

void setDefaultAssertionStatus(boolean status)　　//对于通过类加载器加载的所有类来说，果没有显式的说明类或包的断言状态，则开启或禁用断言

void.setClassAssertionStatus(String className, boolean status)　　//对于给定的类和内部类启用或关闭断言

void setPackageAssertionStatus(String packageName, boolean status)　　//对于给定包和其包中的类，启用或关闭断言

void clearAssertionStatus()　　//移除所有包和类的显式断言状态设置，并禁用所有通过这个类加器加载的所有类的断言

日志

基本日志

默认日志记录器 Logger.global

Logger.getGlobal().info("...");

7个日志记录器级别：

SEVERE，WARNING，INFO，CONFIG，FINE，FINER，FINEST，默认只记录前三个级别

logger.setLevel(Level.FINE)　　//FINE和更高级别都可以记录

Level.ALL, Level.OFF 可以开启和关闭所有级别的记录

修改日志管理器配置

默认配置文件存在于：e/lib/logging.properties

若要使用另一个配置文件，要将java.util.logging.config.file设置为配置文件的存储位置，并用命令动

java -Djava.util.logging.config.file=configFile MainClass

修改默认的日志记录级别：

.level=INFO

指定自己的日志记录级别：

com.mycompany.myapp.level=FINE

要将相应日志记录发送到控制台上：

java.util.logging.ConsoleHandler.level=FINE

处理器

绕过默认配置的父处理器，安装自己的处理器：

```
Logger logger = Logger.getLogger("com.mycompany.myapp");
logger.setLevel(Level.FINE);
logger.setUseParentHandlers(false);
Handler handler = new ConsoleHandler();
```

handler.setLevel(Level.FINE);

logger.addHandler(handler);

日志API提供了两个处理器类型：FileHandler和SocketHandler

logger.setFilter(filter)　　//安装一个过滤器到日志中

logger.setFormatter(formatter)　　//安装一个格式化器到日志中

java.util.logging.Logger 1.4

Logger getLogger(String loggerName)

Logger getLogger(String loggerName, String bundleName)　　//获得给定名字的日志记录器，loggerName：具有层次结构的日志记录器名，bundleName：用来查看本地消息的资源名

void severe(String message)

void warning(String message)

void info(String message)

void config(String message)

void fine(String message)

void finer(String message)

void finest(String message)　　//记录一个由方法名和给定消息指示级别的日志记录

void entering(String className, String methodName)

void entering(String className, String methodName, Object param)

void entering(String className, String methodName, Object[] param)

void exiting(String className, String methodName)

void exiting(String className, String methodName, Object result)　　//记录一个描述进入/出方法的日志记录，其中应该包含给定参数或返回值

void throwing(String className, String methodName, Throwable t)　　//记录一个描述抛出给定异常对象的日志记录

void log(Level level, String message)

void log(Level level, String message, Object obj)

void log(Level level, String message, Object objs)

void log(Level level, String message, Throwable t)　　//记录一个给定级别和消息的日志记录

其中可以包括给定对象或者可抛出对象。要想包括对象，消息中必须包含格式化占位符，如{0}，{1}

void logp(Level level, String className, String methodName, String message)

void logp(Level level, String className, String methodName, String message, Object obj)

void logp(Level level, String className, String methodName, String message, Object objs)

void logp(Level level, String className, String methodName, String message, Throwable t)
//记录一个给定级别、准确的调用者信息和消息的日志记录，其中可以包括对象或可抛出对象

void logrb(Level level, String className, String methodName, String bundleName, String message)

void logrb(Level level, String className, String methodName, String bundleName, String message, Object obj)

void logrb(Level level, String className, String methodName, String bundleName, String message, Object objs)

void logrb(Level level, String className, String methodName, String bundleName, String message, Throwable t) //记录一个给定级别、调用者信息、资源包名和消息的日志记录，其中可以包括象

Level getLevel()

void setLevel(Level l) //获得和设置这个日志记录器的级别

Logger getParent()

void setParent(Logger l) //获得和设置这个日志记录器的父日志记录器

Handler[] getHandlers() //获得这个日志记录器的所有处理器

void addHandler(Handler h)

void removeHandler(Handler h) //增加或删除这个日志记录器中的一个处理器

boolean getUseParentHandlers()

void setUseParentHandlers(boolean b) //获得和设置“use parent handler”属性，如果true，则日志记录器会将全部的日志记录转发给它的父处理器

Filter getFilter()

void setFilter(Filter f) //获得和设置这个日志记录器的过滤器

java.util.logging.Handler 1.4

abstract void publish(LogRecord record) //将日志记录发送到希望的目的地

abstract void flush() //刷新所有已缓冲的数据

abstract void close()　　//刷新所有已缓冲的数据，并释放所有相关资源

Filter getFilter()

void setFilter(Filter f)　　//获得和设置这个处理器的过滤器

Formatter getFormatter()

void setFormatter(Formatter f)　　//获得和设置这个处理器的格式化器

Level getLevel()

void setLevel(Level l)　　//获得和设置这个处理器的级别

java.util.logging.ConsoleHandler 1.4

ConsoleHandler()　　//构造一个新的控制台处理器

java.util.logging.FileHandler 1.4

FileHandler(String pattern)

FileHandler(String pattern, boolean append)

FileHandler(String pattern, int limit, int count)

FileHandler(String pattern, int limit, int count, boolean append)　　//构造一个文件处理器
pattern：构造日志文件名的模式，limit：在打开一个新日志文件前，日志文件可以包含的近似最大节数，count：循环序列的文件数量，append：追加模式

java.util.logging.LogRecord 1.4

Level getLevel()　　//获得这个日志记录的记录级别

String getLoggerName()　　//获得正在记录这个日志记录的日志记录器的名字

ResourseBundle getresourceBundle()

String getresourceBundleName()　　//获得用于本地化消息的资源包或资源包的名字，没有则回null

String getMessage()　　//获得本地化和格式化之前的原始消息

Object[] getParameters()　　//获得参数对象，没有则返回null

Throwable getThrown()　　//获得被抛出的对象

String getSourceClassName()

String getSourceMethodName()　　//获得记录这个日志记录的代码区域

long getMillis()　　//获得创建时间，以毫秒为单位，从1970年开始

long getSequenceNumber()　　//获得这个日志记录的唯一序列序号

int getThreadID()　　//获得创建这个日志记录的线程的唯一ID

java.util.logging.Filter 1.4

boolean isLoggable(LogRecord record)　　//如果给定日志记录需要记录，则返回true

java.util.logging.Formatter 1.4

abstract String format(LogRecord record)　　//返回对日志记录格式化后得到的字符串

String getHead(Handler h)

String getTail(Handler h)　　//返回应该出现在包含日志记录的文档的开头和结尾的字符串

String formatMessage(LogRecord record)　　//返回经过本地化和格式化后的日志记录的消息容

第12章--泛型程序设计----------

不能用基本类型实例化类型参数，只能使用继承了Object的类

定义泛型类：

```
public class Pair {
    ...
}
```

定义泛型方法：

```
class ArrayAlg {
    public static  T getMiddle(T... a) {
        ...
    }
}
```

类型变量放在修饰符的后面，返回类型的前面

调用一个泛型方法时，在方法名前的尖括号中放入具体的类型

```
String middle = ArrayAlg.getMiddle("John", "Q");
```

类型变量的限制

```
public static  T min(T[] a) ...
```

限定类型用&分割，变量类型用逗号分割。类必须是限定列表的第一个

Java泛型转换的事实：

虚拟机中没有泛型，只有普通的类和方法

所有的类型参数都用它们的限定类型来替换

桥方法被合成用来保持多态

为保持类型安全性，必要时插入强制类型转换

运行时查询只适用于原始类型，如instanceof, getClass()

不能创建参数化类型的数组，但声明是合法的，只是不能创建，可以声明通配类型的数组，然后进行型转换。如果需要收集参数化类型，只有一种安全有效的方法：使用ArrayList。

如Pair [] table = new Pair[10];　　//ERROR

两种方法抑制Varargs警告：增加标注@SuppressWarnings("unchecked")或@SafeVarargs。

```
@SafeVarargs
public static  void addAll(Collection coll, T... ts)
```

但是可能会在别处产生错误

不能实例化类型变量，不能 new T(...)

可以通过反射调用Class.newInstance方法构造泛型对象，但是T.class不合法，需要设计API来支配Class对象：

```
public static  Pair makePair(Class cl)
{
   try { return new Pair<>(cl.newInstance(), cl.newInstance()) }
   catch (Exception ex) { return null; }
}
```

不能在静态域或方法中引用类型变量

不能抛出或捕获泛型类的实例

泛型类可以扩展或实现其他的泛型类

通配符类型

Pair, Pair是其子类型

超类型限定

？ super Manager

带有超类型限定的通配符可以向泛型对象写入，带有子类型限定的通配符可以从泛型对象读取

java.lang.Class

T newInstance()　　//返回默认构造器构造的一个新实例

T cast(Object obj)　　//如果obj为null或有可能转换成类型T，则返回obj；否则抛出BadCastException异常

T[] getEnumConstants()　　//如果T是枚举类型，则返回所有值组成的数组，否则返回null

Class getSuperclass()　　//返回这个类的超类，如果T不是一个类或Object类，返回null

Constructor getConstructor(Class... parameterTypes)

Constructor getDeclaredConstructor(Class... parameterTypes)　　//获取公有的构造，或带有给定参数类型的构造器

java.lang.reflect.Constructor

T newInstance(Object... parameters)　　//返回用指定参数构造的新实例

java.lang.Class

TypeVariable[] getTypeParameters()　　//如果这个类型被声明为泛型类型，则返回泛型类型变，否则获得一个长度为0的数组

Type getGenericSuperclass()　　//获得被声明为这一类型的超类的泛型类型

Type[] getGenericInterfaces()　　//获得类型的接口的泛型类型

java.lang.reflect.Method

TypeVariable[] getTypeParameters()　　//获得泛型类型变量

Type getGenericReturnType()　　//获得方法的泛型返回类型

Type[] getGenericParameterTypes()　　//获得方法的泛型参数类型

java.lang.reflect.TypeVariable

String getName()　　//获得类型变量的名字

type[] getBounds()　　//获得类型变量的子类限定

java.lang.reflect.WildcardType

Type[] getUpperBounds()　　//获得这个类型变量的子类（extends）限定

Type[] getLowerBounds()　　//获得这个类型变量的超类（super）限定

java.lang.reflect.ParameterizedType

Type getRawType()　　//获得这个参数化类型的原始类型

Type[] getActualTypeArguments()　　//获得这个参数化类型声明时所使用的类型参数

Type getOwnerType()　　//如果是内部类型，返回外部类型，如果是顶级类型，返回null

java.lang.reflect.GenericArrayType

Type getGenericComponentType()　　//获得声明该数组类型的泛型组合类型

第13章--集合----------

集合接口

Java类库中，集合类的基本接口是Collection接口，Collection接口扩展了Iterable接口，对于标准库任何集合都可以使用for each循环

java.util.Collection 1.2

Iterator iterator()　　//返回一个用于访问集合中每一个元素的迭代器

int size()　　//返回当前存储在集合中的元素的个数

boolean isEmpty()　　//如果集合中没有元素，返回true

boolean contains(Object obj)　　//如果集合中包含了一个与obj相等的对象，返回true

boolean containsAll(Collection other)　　//如果集合中包含了other集合中的所有元素返回true

boolean add(Object element)　　//添加一个元素

boolean addAll(Collection other)　　//添加other集合中的所有元素

boolean remove(Object obj)　　//删除与obj相等的元素

boolean removeAll(Collection other)　　//删除other集合中的所有元素相等的元素

void clear()　　//从集合中删除所有元素

boolean retainAll(Collection other)　　//删除与other集合中元素不同的元素

Object[] toArray()　　//返回这个集合的对象数组

T[] toArray(T[] arrayToFill)　　//返回这个集合的对象数组，如果arrayToFill足够则放入arrayToFill，否则新建一个数组存放

java.util.Iterator 1.2

boolean hasNext()　　//是否存在可访问的元素

E next()　　//返回将要访问的下一个对象，如果达到尾部则抛出NoSuchElement异常

void remove()　　//删除上一次访问的对象，必须紧跟在一个next()之后

Java库中的具体集合

ArrayList	一种可以动态增长和缩减的索引序列
LinkedList	一种可以在任何位置进行高效的插入和删除操作的有序序列
ArrayDeque	一种用循环数组实现的双端队列
HashSet	一种没有重复元素的无序集合
TreeSet	一种有序集
EnumSet	一种包含枚举类型值的集
LinkedHashSet	一种可以记住元素插入次序的集
PriorityQueue	一种允许高效删除最小元素的集合
HashMap	一种存储键值关联的数据结构
TreeMap	一种键值有序排列的映射表
EnumMap	一种键值属于枚举类型的映射表
LinkedHashMap	一种可以记住键值项添加次序的映射表
WeakHashMap	一种其值无用武之地后可以被垃圾回收器回收的映射表
IdentityHashMap	一种用==而不是equals比较键值的映射表

链表

Iterator接口中没有add方法，子接口ListIterator中包含add方法，可以用迭代器在特定位置添加元素

如果迭代器发现它的集合被另一个迭代器修改了，或是被集合自身的方法修改了，就会抛出一个ConcrrentMofificationException异常

java.util.List 1.2

ListIterator listIterator()　　//返回一个列表迭代器

ListIterator listIterator(int index)　　//返回一个指定位置的列表迭代器

void add(int i, E element)　　//在给定位置添加一个元素

void addAll(int i, Collection elements)　　//将集合中的所有元素添加到指定位置

E remove(int i) //删除指定位置的元素并返回这个元素

E get(int i) //获取指定位置的元素

E set(int i, E element) //用新元素取代指定位置的元素，并返回旧元素

int indexOf(Object element) //返回与指定元素相等的元素第一次出现的位置，没有返回-1

int lastIndexOf(Object element) //返回与指定元素相等的元素最后一次出现的位置

java.util.ListIterator 1.2

void add(E element) //在当前位置前添加一个新元素

void set(E element) //用新元素取代next或previous上次访问的元素，如果在next previous上次调用之后列表结构被修改了，抛出IllegalStateException异常

boolean hasPrevious() //当反向迭代列表时，若还有可访问的元素，返回true

E previous() //返回前一个对象

int nextIndex() //返回下一次调用next方法时访问的元素索引

int previousIndex() //返回下一次调用previous方法时访问的元素索引

java.util.LinkedList 1.2

LinkedList() //构造一个空链表

LinkedList(Collection elements) //构造一个链表，将集合中的所有元素添加去

void addFirst(E element)

void addLast(E element) //将某个元素添加到列表的头部或尾部

E getFirst()

E getLast() //返回列表头部或尾部的元素

E removeFirst()

E removeLast() //删除并返回列表头部或尾部的元素

数组列表

在不需要同步（一个线程访问）时使用ArrayList，需要同步（多个线程访问）时使用Vector

散列集

散列表（hash table)

散列码（hash code)

桶（bucket）

散列冲突（hash collision)

再散列（rehashed)

装填因子（load factor)

通常，将桶数设置为预计元素数目的75%-150%，装填因子默认为0.75

java.util.HashSet 1.2

HashSet()　　//构造一个空的散列集

HashSet(Collection elements)　　//构造一个散列表，并将集合中所有的元素添加进去

HashSet(int initialCapacity)　　//构造一个空的具有指定容量的散列集

HashSet(int initialCapacity, float loadFactor)　　//构造一个具有指定容量和装载因子空散列集

java.lang.Object 1.0

int hashCode()　　//返回这个对象的散列码，必须和equals()一致

java.util.TreeSet 1.2

TreeSet()　　//构造一个空树集

TreeSet(Collection elements)　　//构造一个空树集，并将集合中的元素添加进去

对象的比较

java.lang.Comparable 1.2

int compareTo(T other)　　//将这个对象与另一个对象other比较

java.util.Comparator 1.2

int compare(T a, T b)　　//将a与b比较

java.util.SortedSet 1.2

Comparator comparator()　　//返回用于对元素进行排序的比较器，如果元素用Comparable接口的compareTo方法则返回null

E first()　　//返回最小元素

E last()　　//返回最大元素

java.util.NavigableSet 6

E higher(E value)

E lower(E value)　　//返回大于value的最小元素，或小于value的最大元素，没有则返回null

E ceiling(E value)

E floor(E value)　　//返回大于等于value的最小元素，或小于等于value的最大元素，没有返回null

E pollFirst()

E pollLast()　　//删除并返回这个集中的最大元素或最小元素

Iterator descendingIterator()　　//返回一个按递减顺序遍历集中元素的迭代器

java.util.TreeSet 1.2

TreeSet()　　//构造一个空树集

TreeSet(Comparator c)　　//构造一个指定比较器的树集

TreeSet(SortedSet elements)　　//构造一个空树集，并将集中元素添加进去，并使用与定集相同的比较器

队列与双端队列

java.util.Queue 5.0

boolean add(E element)

boolean offer(E element)　　//添加元素到队列尾部，若队列已满则第一个方法抛出IllegalSteException，第二个方法返回false

E remove()

E poll()　　//删除并返回队列头部的元素，若队列为空，则第一个方法抛出NoSuchElemenException，第二个方法返回null

E element()

E peek()　　//返回但不删除队列头部的元素，若为空，第一个方法抛出NoSuchElementExcepion，第二个方法返回null

java.util.Deque 6.0

void addFirst(E element)

void addLast(E element)

boolean offerFirst(E element)

boolean offerLast(E element)　　//将给定对象添加到队列的头部或尾部，若队列已满，前两方法抛出异常，后两个方法返回null

E removeFirst()

E removeLast()

E pollFirst()

E pollLast()　　//删除并返回队列头部的元素，若队列为空，前两个方法抛出异常，两个方法返回null

E getFirst()

E getLast()

E peekFirst()

E peekLast()　　//返回但不删除队列头部的元素，若为空，前两个方法抛出NoSuchElementException，后两个方法返回null

java.util.ArrayDeque 6

ArrayDeque()

ArrayDeque(int initialCapacity)　　//用初始容量构造一个无限双端队列

优先级队列

优先级队列PriorityQueue使用堆结构

java.util.PriorityQueue 5.0

PriorityQueue()

PriorityQueue(int initialCapacity)

PriorityQueue(int initialCapacity, Comparator c)

映射表

java.util.Map 1.2

V get(Object key)　　//返回与键对应的值，键可以为null

V put(K key, V value)　　//将键与对应的值关系插入到映射表中

void putAll(Map entries)　　//将给定映射表中的所有条目添加到这个映射表中

boolean containsKey(Object key)　　//映射表中是否有这个键

boolean containsValue(Object value)　　//映射表中是否有这个值

Set> entrySet()　　//返回Map.Entry对象的集视图，即键值对，可以从这个集中删除元，同时也在映射表中删除该元素，但是不能添加元素

Set keySet()　　//返回所有键的集视图，可以从这个集中删除元素，同时也在映射表中删该元素，但是不能添加元素

Collection values()　　//返回所有值的集视图，可以从这个集中删除元素，同时也在映表中删除该元素，但是不能添加元素

java.util.Map.Entry 1.2

K getKey()

V getValue()　　//获得这个条目的键或值

V setValue(V newValue)　　//设置在映射表中与值对应的新值，并返回旧值

java.util.HashMap 1.2

HashMap()

HashMap(int initCapacity)

HashMap(int initCapacity, float loadFactor)

java.util.TreeMap 1.2

TreeMap(Comparator c)

TreeMap(Map entries)

TreeMap(SortedMap? extends K, ? extends V> entries)

java.util.SortedMap 1.2

Comparator comparator()　　//返回对键进行排序的比较器

K firstKey()

K lastKey()　　//返回映射表中的最小元素和最大元素

集合框架

视图与包装器

映射表类的keySet方法返回一个实现Set接口的类对象，这个类的方法对原映射表进行操作，这种集

称为视图

轻量级集包装器

子范围

不可修改视图

同步视图，Collections的syncronized方法

检查视图，Collections的checkedList方法

java.util.Collections

static Collection unmodifiableCollection(Collection c)

static List unmodifiableList(List c)

static Set unmodifiableSet(Set c)

static SortedSet unmodifiableSortedSet(SortedSet c)

static Map unmodifiableMap(Map c)

static SortedMap unmodifiableSortedMap(SortedMap c) //构造一个集合视图 其更改器方法将抛出一个UnsupportedOperationException

static Collection synchronizedCollection(Collection c)

static List synchronizedList(List c)

static Set synchronizedSet(Set c)

static SortedSet synchronizedSortedSet(SortedSet c)

static Map synchronizedMap(Map c)

static SortedMap synchronizedSortedMap(SortedMap c) //构造一个集 视图，其方法都是同步的

static Collection checkedCollection(Collection c)

static List checkedList(List c)

static Set checkedSet(Set c)

static SortedSet checkedSortedSet(SortedSet c)

static Map checkedMap(Map c)

static SortedMap checkedSortedMap(SortedMap c) //构造一个集合视图 如果插入一个错误元素，将抛出ClassCastException

static List nCopies(int n, E value)

static Set singleton(E value) //构造一个对象视图，它既可以作为一个有n个相同元素的不可修改列表，又可以作为拥有一个单一元素的集

java.util.Arrays

static List asList(E... array) //返回一个数组元素的列表视图，可修改但大小不可变

java.util.List

List subList(int firstIncluded, int firstExclued) //返回给定位置范围内的所有素的列表视图

java.util.SortedSet

SortedSet subSet(E firstIncluded, E firstExcluded)

SortedSet headSet(E firstExcluded)

SortedSet tailSet(E firstIncluded) //返回给定位置内的元素视图

java.util.NavigableSet

NavigableSet subSet(E from, boolean fromIncluded, E to, boolean toIncluded)

NavigableSet headSet(E to, boolean toIncluded)

NavigableSet tailSet(E from, boolean fromIncluded) //返回给定位置内的元视图，boolean决定是否包含边界

java.util.SortedMap

SortedMap subMap(K firstIncluded, K firstExcluded)

SortedMap headMap(K firstExcluded)

SortedMap tailMap(K firstIncluded) //返回给定范围内的键条目的映射表视图

java.util.NavigableMap

NavigableMap subMap(K from, boolean fromIncluded, K to, boolean toIncluded)

NavigableMap headMap(K from, boolean fromIncluded)

NavigableMap tailMap(K to, boolean toIncluded) //返回在给定范围内的键条目映射表视图，boolean决定是否包含边界

集合与数组间的转换

数组转集合：

Arrays.asList(E... array)

集合转数组：

Object[] values = staff.toArray()

String[] values = staff.toArray(new String[0])

staff.toArray(new String[staff.size()])

算法

排序与混排

Java中的sort方法的实现：将所有元素转入一个数组，并使用一种归并排序的变体对数组进行排序，后将排序后的数组复制回列表

java.util.Collections

static > void sort(List elements)

static void sort(List elements, Comparator) //使用稳定的排序算法，对表中的元素行排序，这种算法的时间复杂度是O(n log(n)), 其中n为列表长度

static void shuffle(List elements)

static void shuffle(List elements, Random r) //随机的打乱表中的元素，算法复杂度O(n a(n))， n为列表长度，a(n)为访问元素的平均时间

static Comparator reverseOrder() //返回一个比较器，它用与Comparable()接的compareTo()方法规定的顺序的逆序对元素进行排序

static Comparator reverseOrder(Comparator comp) //返回一个比较器，用com给定的顺序的逆序进行排序

二分查找

java.util.Collections

static int binarySearch(elements, T key)

static int binarySearch(List elements, Comparator c) //从有序列表中搜索一个键，果元素扩展了AbstractSequentialList类，则采用线性查找，否则采用二分查找。时间复杂度为O(a(n) og(n))，n为列表长度，a(n)为元素的平均访问时间，这个方法将返回这个键在列表中的索引，如果不在这个键则返回一个负值i。将这个键插入到索引-i-1的位置上，可以保持列表的有序性

其他简单算法

java.util.Collections

static > T min(Collection elements)

static > T max(Collection elements)

static min(Collection elements, Comparactor c)

static max(Collection elements, Comparator c)　　//返回集合中最小的或最大的元素

static void copy(List to, List from)　　//将原列表中的所有元素复制到目标列表的应位置上。目标列表的长度至少与原列表一样

static void fill(List l, T value)　　//将列表中的所有位置设置为相同的值

static boolean addAll(Collection c, T... values)　　//将所有的值添加到集合中，如集合改变了返回true

static boolean replaceAll(List l, T oldValue, T newValue)　　//用newValue取代所有值oldValue的元素

static int indexOfSubList((List l, List s)

static int lastIndexOfSubList(List l, List s)　　//返回l中第一个或最后一个等于s子表的索引。如果l中不存在等于s的子列表，则返回-1

static void swap(List l, int i, int j)　　//交换给定偏移量的两个元素

static void reverse(List l)　　//逆置列表中元素的顺序，时间复杂度为O(n)

static void rotate(List l, int d)　　//旋转列表中的元素，将索引i的条目移动到置(i + d)%l.size()。时间复杂度为O(n)

static int frequency(Collection c, Object o)　　//返回c中与对象o相同的元素个数

boolean disjoint(Collection c1, Collection c2)　　//如果两个集合没有共同的元素，返回true

遗留的集合

Hashtable类

枚举

java.util.Enumeration

boolean hasMoreElements()　　//如果还有更多的元素可以查看，则返回true

E nextElement()　　//返回被检测的下一个元素

java.util.Hashtable

Enumeration keys()　　//返回一个遍历散列表中键的枚举对象

Enumeration elements()　　//返回一个遍历散列表中元素的枚举对象

java.util.Vector

Enumeration elements()　　//返回遍历向量中元素的枚举对象

属性映射表（property map）

实现属性映射表的Java平台类称为Properties

java.util.Properties

Properties()　　//创建一个空的属性映射表

Properties(Properties defaults)　　//创建一个带有一组默认值的空的属性映射表

String getProperty(String key)　　//获得属性的对应关系，返回与键对应的字符串
如果在映射表中不存在，返回默认表中与这个键对应的字符串

String getProperty(String key, string defaultValue)　　//获得在键没有找到时具有的
认值属性，将返回与键对应的字符串，如果在映射表中不存在，就返回默认的字符串

void load(InputStream in)　　//从InputStream加载属性映射表

void store(OutoutStream out, String commentString)　　//把属性映射表存储到Out
utStream

栈

Stack类扩展为Vector类

java.util.Stack

E push(E item)　　//将item压入栈并返回item

E pop()　　//弹出并返回栈顶的item

E peek()　　//返回栈顶元素，但不弹出

位集

java.util.BitSet

BitSet(int initialCpacity)　　//创建一个位集

int length()　　//返回位集的逻辑长度

boolean get(int bit)　　//获得一个位

void set(int bit)　　//设置一个位，开

void clear(int bit)　　//清除一个位，关

```
void and(BitSet set)                    //这个位集与另一个位集进行逻辑“AND"
void or(BitSet set)                     //这个位集与另一个位集进行逻辑“OR”
void xor(BitSet set)                    //逻辑“XOR”
void andNot(BitSet set)                 //清除这个位集中对应另一个位集中设置的所有位
```

第14章--多线程----------

每个进程拥有自己的一整套变量，而线程则共享数据

也可以通过构建一个Thread的子类来定义一个线程，如

```
class MyThread extends Thread {
    public void run() {
        task code
    }
}
```

然后构建一个子类调用start方法

不要调用Thread类或Runnable对象的run()方法，只会执行run()中的任务，不会启动新线程

java.lang.Thread

```
static void sleep(long millis)          //休眠给定的毫秒数
Thread(Runnable target)                 //构造一个新线程，用于调用给定target的run()方法
void start()            //启动线程，将调用run()方法。这个方法立即返回，并且新线程并行运行
void run()              //调用关联Runnable()的run()方法
```

java.lang.Runnable

```
void run()              //必须覆盖这个方法，在这个方法中提供要执行的指令
```

中断线程

如果在每次工作迭代之后都调用sleep()方法（或其他的可中断方法），isInterrupted检测既没必要也用处

java.lang.Thread

```
void interrupt()                        //向线程发送中断请求，线程的中断状态将被设置为true，如
```

目前线程被一个sleep()阻塞，将会抛出InterruptedException异常

static boolean interrupted()　　//测试当前线程（正在执行这一命令的线程）是否被中，并将会将当前线程的中断状态设置为false

boolean isInterrupted()　　//测试线程是否被终止，不改变线程中断状态

static Thread currentThread()　　//返回代表当前执行线程的Thread对象

线程状态（6种）

New（新创建）

Runnable（可运行）

Blocked（被阻塞）

Waiting（等待）

Timed waiting（计时等待）

Terminated（被终止）

调度器使用时间片机制，抢占式调度系统给每一个可运行线程一个时间片来执行任务，当时间片用完操作系统将剥夺该线程的运行权，并给另一个线程运行机会

java.lang.Thread

void join()　　//等待终止指定的线程

void join(long millis)　　//等待指定的线程死亡或者经过指定的毫秒数

Thread.State getState()　　//得到这一线程的状态

void stop()　　//停止该线程，方法已过时

void suspend()　　//暂停这一线程的执行，方法已过时

void resume()　　//恢复线程，仅在suspend()之后调用，方法已过时

线程属性

线程优先级

java.lang.Thread

void setPriority(int newPriority)　　//设置线程的优先级，必须在Thread.MIN_PRIORITY Thread.MAX_PRIORITY之间，一般用Thread.NORM_PRIORITY优先级

static int MIN_PRIORITY　　//线程的最小优先级，值为1

static int NORM_PRIORITY　　//线程的默认优先级，值为5

static int MAX_PRIORITY　　//线程的最高优先级，值为10

static void yield()　　//导致当前执行线程处于让步状态。如果有其他的可运行线具有至少与此线程同样高的优先级，那么这些线程接下来会被调度。

守护线程

java.lang.Thread

void setDaemon(boolean isDaemon)　　//标识该线程为守护线程或用户线程。这方法须在线程启动之前调用

未捕获异常处理器

线程的run方法不能抛出任何被检测的异常

该处理器必须属于一个实现Thread.UncaughtExceptionHandler接口的类

如果不安装默认的处理器，磨人的处理器为空。如果不为独立的线程安装处理器，此时处理器就是该程的ThreadGroup对象

java.lang.Thread

static void setDefaultUncaughtExceptionHandler(Thread.UncaughtExceptionHandler handler)

static Thread.UncaughtExceptionHandler getDefaultUncaughtExceptionHandler()　　//设置获取未捕获异常的默认处理器

void setUncaughtExceptionHandler(Thread.UncaughtExceptionHandler handler)

Thread.UncaughtExceptionHandler getUncaughtExceptionHandler()　　//设置或获取未获异常的处理器

java.lang.Thread.UncaughtExceptionHandler

void uncaughtException(Thread t, Throwable e)　　//当一个线程因未捕获异常而终止，按定要将客户报告记录到日志中

java.lang.ThreadGroup

void uncaughtException(Thread t, Throwable e)　　//如果有父线程组，调用父线程组的这一法；或者，如果Thread类有默认处理器，调用该处理器，否则，输出栈踪迹到标准错误流上（但如果是一个ThreadDeath对象，栈踪迹是被禁用的）

锁对象

ReentrantLock类实现了Lock接口

java.util.concurrent.locks.Lock

void lock()　　//获取这个锁，如果锁同时被另一个线程拥有则发生阻塞

```
void unlock()                    //释放这个锁
```

java.util.concurrent.locks.ReentrantLock

```
ReentrantLock()                  //构建一个可以被用来保护临界区的可重入锁

ReentrantLock(boolean fair)                    //构建一个带有公平策略的锁。一个公平锁偏爱等待
间最长的线程。但是会降低性能
```

条件对象

java.util.concurrent.locks.Lock

```
Condition newCondition()                    //返回一个与该锁相关的条件对象
```

java.util.concurrent.locks.Condition

```
void await()                    //将该线程放到条件的等待集中

void signalAll()                    //解除该条件的等待集中的所有线程的阻塞状态

void signal()                    //从该条件的等待集中随机地选择一个线程，解除其阻塞状态
```

synchronized关键字

```
public synchronized void method()
{
   method body
}
```

等价于

```
public void method()
{
   this.intrinsicLock.lock();
   try
   {
      method body
   }
```

```
    finally { this.intrinsicLock.unlock(); }
}
```

每个对象有一个内部锁，并且该锁只有一个内部条件

将静态方法声明为synchronized也是合法的，当该方法被调用时，相关的类对象的内部锁被锁住，因没有其他线程可以调用同一个类的这个或任何其他的同步静态方法

使用优先情况：阻塞队列 > synchronized > Lock/Condition

java.lang.Object

```
void notifyAll()            //解除那些在该对象上调用wait方法的线程的阻塞状态，只能在同步方法或步块内部调用
void notify()               //随机选择一个在该对象上调用wait方法的线程，解除其阻塞状态。
void wait()                 //导致线程进入等待状态直到它被通知。该方法只能在一个同步方法中调。
void wait(long millis)
void wait(long millis, int nanos)                   //导致线程进入等待状态直到它被通知或者经过指的时间
```

同步阻塞

```
synchronized (obj)
{
    critical section
}               //获得obj的锁
```

客户端锁定

监视器概念

监视器是只包含私有域的类，每个监视器类的对象有一个相关的锁，使用该锁对所有的方法进行加锁该锁可以有任意多个相关条件

volatile域

volatile关键字为实例域的同步访问提供了一种免锁机制，如果声明一个域为volatile，那么编译器和拟机就知道该域是可能被另一个线程并发更新的，但是volatile变量不能提供原子性

线程局部变量

例如SimpleDateFormat类，为每一个线程构造一个实例

```
public static final ThreadLocal dateFormat =
    new ThreadLocal() {
        protected SimpleDateFormat initialValue() {
            return new SimpleDateFormat("yyyy-MM-dd");
        }
    };
```

java.lang.ThreadLocal

```
T get()                    //得到这个线程的当前值。如果是首次调用get，会调用initialize来得到这个值
protected initialize()                    //应覆盖这个方法来提供一个初始值。默认情况下，返回null
void set(T t)                  //为这个线程设置一个新值
void remove()                    //删除对应这个线程的值
```

java.util.concurrent.ThreadLocalRandom

```
static ThreadLocalRandom current()                    //返回特定于当前线程的Random类实例
```

锁测试与超时

java.util.concurrent.locks.Lock

```
boolean tryLock()                    //尝试获得锁而没有发生阻塞；如果成功返回真。这个方法会抢夺用的锁，即使该锁有公平加锁策略，即便其他线程已经等待很久也是如此
boolean tryLock(long time, TimeUnit unit)          //尝试获得锁，阻塞时间不会超过给定的值；果成功返回true
void lockInterruptibly()          //获得锁，但是会不确定的发生阻塞，如果线程被中断，抛出InterrptedException异常
```

java.util.concurrent.locks.Condition

```
boolean await(long time, TimeUnit unit)                    //进入该条件的等待集，直到线程从等待集移除或等待了指定的时间之后才解除阻塞，如果因为等待时间到了而返回就返回false，否则返回true
void awaitUninterruptibly()                //进入该条件的等待集，直到线程从等待集移出才解除阻塞如果线程被中断，该方法不会抛出InterruptedException异常
```

读/写锁

java.util.concurrent.locks.ReentrantReadWriteLock

Lock readLock()　　//得到一个可以被多个读操作公用的读锁，但会排斥所有写操作

Lock writeLock()　　//得到一个写锁，排斥所有其他的读操作和写操作

阻塞队列

阻塞队列方法

| add | 添加一个元素 | 如果队列满，抛出IllegalStateException异常 |
| element | 返回队列的头元素 | 如果队列空，抛出NoSuchElementException异常 |
| offer | 添加一个元素并返回true | 如果队列满，返回false
|
| peek | 返回队列的头元素 | 如果队列空，返回null
|
poll	移出并返回队列的头元素	如果队列空，返回null
put	添加一个元素	如果队列满，则阻塞
remove	移出并返回头元素	如果队列空，抛出NoSuchElementException异常
take	移出并返回头元素	如果队列空，则阻塞

java.util.concurrent.ArrayBlockingQueue

ArrayBlockingQueue(int capacity)

ArrayBlockingQueue(int capacity, boolean fair)　　//构造一个带有指定的容量和公平性设置的阻队列。该队列用循环数组实现

java.util.concurrent.LinkedBlockingQueue

LinkedBlockingQueue()

LinkedBlockingDeque()　　//构造一个无上限的阻塞队列或双向队列，用链表实现

LinkedBlockingQueue(int capacity)

LinkedBlockingDeque(int capacity)　　//根据指定容量构建一个有限的阻塞队列或双向队，用链表实现

java.util.concurrent.DelayQueue

DelayQueue()　　//构造一个包含Delayed元素的无界的阻塞时间有限的阻塞队列。只有那些延已经超过时间的元素可以从队列中移出

java.util.concurrent.Delayed

long getDelay(TimeUnit unit)　　//得到该对象的延迟，用给定的时间单位进行度量

java.util.concurrent.PriorityBlockingQueue

PriorityBlockingQueue()

PriorityBlockingQueue(int initialCapacity)

PriorityBlockingQueue(int initialCapacity, Comparator comparator) //构造一个无边界阻塞优队列，用堆实现

java.util.concurrent.BlockingQueue

void put(E element) //添加元素，在必要时阻塞

E take() //移出并返回头元素，必要时阻塞

boolean offer(E element, long time, TimeUnit unit) //添加给定的元素，如果成功返回true如果必要时阻塞，直至元素已经被添加或超时

E poll(long time, TimeUnit unit) //移出并返回头元素，必要时阻塞，直至元素可用或超时用。失败时返回null

java.util.concurrent.BlockingDeque

void putFirst(E element)

void putLast(E element) //添加元素，必要时阻塞

E takeFirst()

E takeLast() //移出并返回头元素或尾元素，必要时阻塞

boolean offerFirst(E element, long time, TimeUnit unit)

boolean offerLast(E element, long time, TimeUnit unit) //添加给定的元素，成功时返回tru，必要时阻塞直至元素被添加或超时

E pollFirst(long time, TimeUnit unit)

E pollLast(long time, TimeUnit unit) //移动并返回头元素或尾元素，必要时阻塞，直至元可用或超时，失败时返回null

java.util.concurrent.TransferQueue

void transfer(E element)

boolean tryTransfer(E element, long time, TimeUnit unit) //传输一个值，或者尝试在给定超时时间内传输这个值，这个调用将阻塞，直到另一个线程将元素删除。第二个方法会在调用成功时回true

线程安全的集合

高效的映射表、集合和队列

这些集合返回弱一致性（weakly consisteut)的迭代器，意味着迭代器不一定能反映出它们被构造之

的所有的修改，不会抛出ConcurrentModificationException异常

java.util.concurrent.ConcurrentLinkedQueue

ConcurrentLinkedQueue()　　//构造一个可以被多线程安全访问的无边界非阻塞的队列

ConcurrentSkipListSet()

ConcurrentSkipListSet(Comparator comp)　//构造一个可以被多线程安全访问的有序集，第一个求元素实现Comparable接口

java.util.concurrent.ConcurrentHashMap

java.util.concurrent.ConcurrentSkipListMap

ConcurrentHashMap()

ConcurrentHashMap(int initialCapacity)

ConcurrentHashMap(int initialCapacity, float loadFactor, int concurrencyLevel)　　//构造一可以被多线程安全访问的散列映射表

ConcurrentSkipListMap()

ConcurrentSkipListSet(Comparator comp)　//构造一个可以被多线程安全访问的有序的映射表

V putIfAbsent(K key, V value)　　//如果该键没有在映射表中出现，则将给定的值同给定的键关起来，并返回null，否则返回与该键关联的现有值

boolean remove(K key, V value)　　//如果给定的键与给定的值关联，删除给定的键与值并返回，否则返回false

boolean replace(K key, V oldValue, V newValue)　　//如果给定的键当前与oldValue相关联，用与newValue关联，否则返回false

写数组的拷贝

CopyOnWriteArrayList和CopyOnWriteArraySet，所有的修改线程对底层数组进行复制

任何集合类可以使用同步包装器（synchronization wrapper)变成线程安全的：

```
List synchArrayList = Collections.synchronizedList(new ArrayList());
```

如果在另一个线程可能进行修改时要对集合进行迭代，仍然需要使用“客户端”锁定：

```
synchronized (synchHashMap)
{
    Iterator iter = synchHashMap.keySet().iterator();
```

```
    while (iter.hasNext()) ...;
}
```

最好使用java.util.concurrent包中定义的集合，不使用同步包装器中的

java.util.Collections

```
static  Collection synchronizedCollection(Collection c)
static  List synchronizedList(List c)
static  Set synchronizedSet(Set c)
static  SortedSet synchronizedSortedSet(SortedSet c)
static  Map synchronizedMap(Map c)
static  SortedMap synchronizedSortedMap(SortedMap c)      //构建集合视图，该集合的方法是步的
```

Callable与Future

Callable与Runnable类似，但是有返回值，类型参数是返回值的类型

Future保存异步计算的结果，可以启动一个计算，将Future计算交给某个线程，Future对象的所有者结果计算好之后就可以获得它

FutureTask包装器是一种非常便利的机制，可将Callable转换成Future和Runnable，它同时实现二的接口，例如：

```
Callable<Integer> myComputation = ...;
FutureTask<Integer> task = new FutureTask<Integer>(myComputation);
Thread t = new Thread(task); //it's a Runnable
t.start();
...
Integer result = task.get(); //it's a Future
```

java.util.concurrent.Callable

```
V call()              //运行一个将产生结果的任务
```

java.util.concurrent.Future

```
V get()
V get(long time, TimeUnit unit)          //获取结果，如果没有结果可用，则阻塞直到真正得到结果过指定的时间为止。如果不成功，第二个方法抛出TimeoutException异常
boolean cancel(boolean mayInterrupt)              //尝试取消这一任务的运行，如果任务已经开始，且mayInterrupt参数值为true，它就会被中断，如果成功执行了取消操作，返回true
```

boolean isCancelled()　　//如果任务在完成前被取消了，返回true

boolean isDone()　　//如果任务结束，无论是正常结束、中途取消或发生异常，都返回true。

java.util.concurrent.FutureTask

FutureTask(Callable task)

FutureTask(Runnable task, V result)　　//构造一个既是Future又是Runnable的对象

执行器

执行者工厂方法

newCachedThreadPool	必要时创建新线程；空闲线程会被保留60秒
newFixedThreadPool	该池包含固定数量的线程：空闲线程一直被保留
newSingleThreadExecutor	只有一个线程的池，该线程顺序执行每一个提交的任务
newScheduledThreadPool	用于预定执行而构建的固定线程池，替代java.util.Timer
newSingleThreadScheduledExecutor	用于预定执行而构建的单线程池

连接池使用：

1. 调用Executors类中静态的方法newCachedThreadPool或newFixedThreadPool
2. 调用submit提交Runnable或Callable对象
3. 如果想要取消一个任务，或如果提交Callable对象，那就要保存好返回的Future对象
4. 当不再提交任何任务时，调用shutdown

java.util.concurrent.Executors

ExecutorService newCachedThreadPool()　　//返回一个带缓存的线程池

ExecutorService newFixedThreadPool(int threads)　　//返回一个指定线程数的线程池

ExecutorService newSingleThreadExecutor()　　//返回一个执行器，它在一个单个的线程中次执行各个任务

java.util.concurrent.ExecutorService

Future submit(Callable task)

Future submit(Runnable task, T result)

Future submit(Runnable task)　　//提交指定的任务去执行

void shutdown()　　//关闭服务，会先完成已经提交的任务而不再接收新的任务

java.util.concurrent.ThreadPoolExecutor

int getLargestPoolSize()　　//返回线程池在该执行器生命周期中的最大尺寸

预定执行

ScheduledExecutorService接口具有为预定执行或重复执行任务而设计的方法

可以预定Runnable或Callable在初始的延迟之后只运行一次，也可以预定一个Runnable对象周期性运行

java.util.concurrent.Executors

ScheduledExecutorService newScheduledThreadPool(int threads)　　//返回一个线程池，它用给定的线程数来调度任务

ScheduledExecutorService newSingleThreadScheduledExecutor()　　//返回一个执行器，在一个单独线程中调度任务

java.util.concurrent.ScheduledExecutorService

ScheduledFuture schedule(Callable task, long time, TimeUnit unit)

ScheduledFuture schedule(Runnable task, long time, TimeUnit unit)　　//预定在指定的时之后执行任务

ScheduledFuture scheduleAtFixedRate(Runnable task, long initialDelay, long period, TimeUnit unit) //预定在初始的延迟结束后，周期性地运行给定的任务，周期长度是period

ScheduledFuture scheduleWithFixedDelay(Runnable task, long initialDelay, long delay, TimeUnit unit)　//预定在初始的延迟结束后周期性的给定的任务，在一次调用完成和下一次调用开始之间有度为delay的延迟

控制任务组

invokeAny方法提交所有对象到一个Callable对象的集合中，并返回某个已经完成了的任务的结果

invokeAll方法提交所有对象到一个Callable对象的集合中，并返回一个Future对象的列表，代表所有务的解决方案

可以用ExecutorCompletionService来对结果按可获得的顺序进行排列，如下：

```
ExecutorCompletionService service = new ExecutorCompletionService(executor);
for (Callable<T> task : tasks) service.submit(task);
for (int i = 0; i < tasks.size(); i++) processFurther(service.take().get());
```

java.util.concurrent.ExecutorService

T invokeAny(Collection> tasks)

T invokeAny(Collection> tasks, long timeout, TimeUnit unit)　　//执行给定的任务，返回其中

个任务的结果。第二个方法若发生超时，抛出一个Timeout Exception异常

List> invokeAll(Collection> tasks)

List> invokeAll(Collection> tasks, long timeout, TimeUnit unit)　　//执行给定的任务，返回有任务的结果。第二个方法若发生超时，抛出一个TimeoutException异常

java.util.concurrent.ExecutorCompletionService

ExecutorCompletionService(Executor e)　　//构建一个执行器完成服务来收集给定执行器的结果

Future submit(Callable task)

Future submit(Runnable task, T result)　　//提交一个任务给底层的执行器

Future take()　　//移除下一个已完成的结果，如果没有任何已完成的结果可用则阻塞

Future poll(long time,TimeUnit unit)　　//移除下一个已完成的结果，如果没有任何已完成结可用则返回null。第二个方法将等待给定的时间

Fork-Join框架

分解任务，并行运行，采用框架可用的一种方式完成递归计算，需要提供一个扩展RecursiveTask的（返回T类型）或者提供一个扩展RecursiveAction（无返回）的类，这两个为ForkJoinTask的子类，盖compute方法来生成并调用子任务，然后合并其结果，如：

```
class Counter extends RecursiveTask<Integer> {
   ...
   protected Integer compute() {
      if (to - from < THRESHOLD) {
         solve problem directly
      }
      else {
         int mid = (from + to) / 2;
         Counter first = new Counter(values, from, mid, filter);
         Counter second = new Counter(values, mid, to, filter);
         invokeAll(first, second);
         return first.join() + second.join();
      }
   }
}
```

在这里，invokeAll方法接收到很多任务并阻塞，直到所有这些任务都已经完成，join方法将生成结果

fork-join框架后台的方法：工作密取（work stealing)

ForkJoinTask需要通过ForkJoinPool来执行，任务分割出的子任务会添加到当前工作线程所维护的双队列中，进入队列的头部。当一个工作线程的队列里暂时没有任务时，它会随机从其他工作线程的队的尾部获取一个任务

如：

```
ForkJoinPool pool = new ForkJoinPool()
pool.invoke(counter);
System.out.println(counter.join());
```

同步器

公用集结点模式，预置功能

| CyclicBarrier | 允许线程集等待直至其中预定数目的线程到达一个公共障栅，然后可以选择执行一个理障栅的动作 | 当大量的线程需要在它们的结果可用之前完成时 |

| CountDownLatch | 允许线程集等待直到计数器减为0 | 当一个或多个线程需要等待直到指定数目的件发生 |

| Exchanger | 允许两个线程在要交换的对象准备好时交换对象 | 当两个线程工作在同一数据结构的两实例上的时候，一个向实例添加数据而另一个从实例清除数据 |

| Semaphore | 允许线程集等待直到被允许继续运行为止 | 限制访问资源的线程总数，如果许可数是1常常阻塞线程直到另一个线程给出许可为止 |

| SynchronousQueue | 允许一个线程把对象交给另一个线程 | 在没有显式同步的情况下，当两个线准备好将一个对象从一个线程传递到另一个时 |

信号量，管理着许多的许可证，同步原语

倒计时门栓（CountDownLatch)

障栅，使用如下：

```
CyclicBarrier barrier = new CyclicBarrier(nthreads);
public void run() {
    doWork();
    barrier.await();
}
//或者加入超时参数
barrier.await(100, TimeUnit.MILLISECONDS);
//加入障栅动作
Runnable barrierAction = ...;
CyclicBarrier barrier = new CyclicBarrier(nthreads, barrierAction);
```

交换器（Exchanger)，当两个线程在同一个数据缓冲区的两个实例上工作的时候，就可以使用交换器

同步队列，将生产者与消费者线程配对，当一个线程调用SynchronousQueue的put方法时，它会阻直到另一个线程调用take方法。它不是一个队列，没有包含任何元素，数据仅仅沿一个方向传递

线程与Swing

两个原则：

如果一个动作需要花费很长时间，在一个独立的工作器线程中做这件事不要在事件分配线程中做

除了事件分配线程，不要在任何线程中接触Swing组件

假定想在一个线程中周期性地更新标签来表明进度。不可以从自己的线程中调用label.setText，而应使用EventQueue类的invokeLater方法(异步执行)和invokeAndWait方法使所调用的方法在事件分配程中执行，如：

```
EventQueue.invokeLater(new
   Runnable()
   {
      public void run()
      {
         label.setText(percentage + "% complete");
      }
   });
```

java.awt.EventQueue

static void invokeLater(Runnable runnable)　　//在待处理的线程被处理之后，让runnable对象的un方法在事件分配线程中执行

static void invokeAndWait(Runnable runnable)　　//在待处理的线程被处理之后，让runnabl对象的run方法在事件分配线程中执行，该调用会阻塞，知道run方法终止

static boolean isDispatchThread()　　//如果执行这一方法的线程是事件分配线程，返回true

Swing工作线程

javax.swing.SwingWorker　//产生类型为T的结果以及类型为V的进度数据

abstract T doInBackground()　　//覆盖这一方法来执行后台的任务并返回这一工作的结果

void process(List data)　　//覆盖这一方法来处理事件分配线程中的中间进度数据

void publish(V... data)　　//传递中间进度数据到事件分配线程中。从doInBackground调用这法

void execute()　　//为工作器线程的执行预定这个工作器

SwingWorker.StateValue getState()　　//得到这个工作器线程的状态，值为PENDING、STARTD或DONE之一

单一线程规则

每一个Java应用程序都开始于主线程中的main方法，在Swing程序中，main方法的生命周期很短，在事件分配线程中规划用户界面的构造然后退出。

一些线程安全的方法：

JTextComponent.setText

JTextArea.insert

JTextArea.append

JTextArea.replaceRange

JComponent.repaint

JComponent.revalidate