# Enumeration

枚举（enumeration）类：如果一个类的对象是可数多个的，那么这个类叫做枚举类。枚举枚举对象名通常使用**全部大写**，符号常量的命名规范，枚举对象根据需要，也可以有多个属性。

## 自定义枚举类

（JDK1.5前）：

1、私有化构造器，防止在类的外部随意创建类的对象

2、在类中创建类的对象

|  |
| --- |
| **public** **class** TestEnumType {  **public** **static** **void** main(String[] args) {  Season s = Season.***SPRING***;  System.***out***.println(s);  }  }  **class** Season{  **public** **static** **final** Season ***SPRING*** = **new** Season();  **public** **static** **final** Season ***SUMMER*** = **new** Season();  **public** **static** **final** Season ***AUTUMN*** = **new** Season();  **public** **static** **final** Season ***WINTER*** = **new** Season();  **private** Season(){  }  **public** String toString(){  **if**(**this** == ***SPRING***){  **return** "SPRING";  }**else** **if**(**this** == ***SUMMER***){  **return** "SUMMER";  }**else** **if**(**this** == ***AUTUMN***){  **return** "AUTUMN";  }**else**{  **return** "WINTER";  }  }  } |

## enum关键字

使用enum定义的枚举类默认继承了**java.lang.Enum**类，且定义为final修饰的类，因此不能再继承其它类也不能被继承。

（JDK1.5后）：

1、格式：enum 类名{}

2、enum枚举类中的对象（）实例必须位于首行，多个对象之间用逗号隔开，用分号结尾；对象前自动添加public static final修饰。

3、构造器的修饰符默认且强制为private，可以省略。

4、JDK1.5之后可以在switch中使用枚举

5、使用enum定义的枚举类默认继承了java.lang.Enum类，因此枚举类就不能再继承其他类了

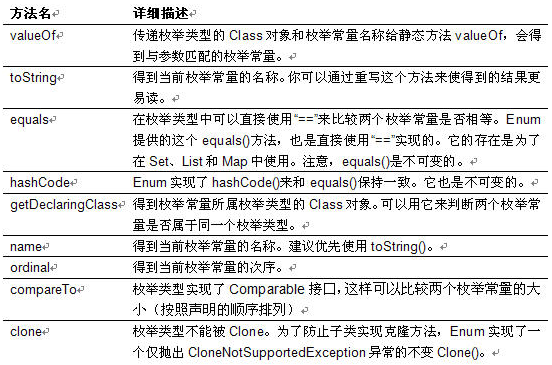
**枚举类定义属性**

枚举类对象的属性不允许被改动，应该使用private final修饰，且应在构造器中对其赋值；若枚举类显式的定义了带参数的构造器，则在列出枚举值时也必须对应的传入参数。

## 枚举类常用方法

values()：返回枚举类型的对象数组

valueOf(String str)：根据对象名获取该对象，传入的String（严格区分大小写），返回的对象。



**实现接口的枚举类**：若需要每个枚举值在调用实现的接口方法呈现出不同的行为方式, 则可以让每个枚举值分别来实现该方法。若每个枚举值在调用实现的接口方法呈现相同的行为方式，则只要统一实现该方法即可。

|  |
| --- |
| **public** **class** EnumClassTest {  **public** **static** **void** main(String[] args) {  System.***out***.println("1:"+EnumSeason.***SPRING***.getSeasonName());  System.***out***.println("1:"+EnumSeason.***SUMMER***.getSeasonName());  System.***out***.println("1:"+EnumSeason.***AUTUMN***.getSeasonName());  System.***out***.println("1:"+EnumSeason.***WINTER***.getSeasonName());  //values方法：返回枚举类型的对象数组  EnumSeason[] es = EnumSeason.*values*();  **for**(EnumSeason eseason : es) {  //JDK1.5之后可以使用在switch中  **switch**(eseason) {  **case** ***SPRING***:  System.***out***.println(EnumSeason.***SPRING***.getSeasonName());  EnumSeason.***SPRING***.getInfo();  **break**;  **case** ***SUMMER***:  System.***out***.println(EnumSeason.***SUMMER***.getSeasonName());  EnumSeason.***SUMMER***.getInfo();  **break**;  **case** ***AUTUMN***:  System.***out***.println(EnumSeason.***AUTUMN***.getSeasonName());  EnumSeason.***AUTUMN***.getInfo();  **break**;  **case** ***WINTER***:  System.***out***.println(EnumSeason.***WINTER***.getSeasonName());  EnumSeason.***WINTER***.getInfo();  **break**;  }  }  //valuesOf方法：根据对象名获取该对象，传入的String（严格区分大小写），返回的对象  EnumSeason esObject = EnumSeason.*valueOf*("SUMMER");  System.***out***.println("3:"+esObject.getSeasonName());  }  }  **interface** Info{  **void** getInfo();  }  **enum** EnumSeason **implements** Info{  //枚举类创建的对象必须位于首行，多个对象用“,”分隔，用“;”结尾，对象自动省略public static final修饰符  ***SPRING***("EnumSeason春天"){  //每一个对象都可以重写接口中的抽象方法  **public** **void** getInfo() {  System.***out***.println("this is spring");  }  },  ***SUMMER***("EnumSeason夏天"){  **public** **void** getInfo() {  System.***out***.println("this is summer");  }  },  ***AUTUMN***("EnumSeason秋天"){  **public** **void** getInfo() {  System.***out***.println("this is autumn");  }  },  ***WINTER***("EnumSeason冬天"){  **public** **void** getInfo() {  System.***out***.println("this is winter");  }  };  **private** String seasonName;  //构造器的修饰符默认且强制为private，可以省略  EnumSeason(String seasonName) {  **this**.seasonName = seasonName;  }  **public** String getSeasonName() {  **return** seasonName;  }  } |

## 枚举的扩展

java.util.EnumSet和java.util.EnumMap是两个枚举集合。

**EnumSet**保证集合中的元素不重复；

**EnumMap**中的key是enum类型，而value则可以是任意类型。

# Annotation

注解（Annotation）也被称为元数据（Metadata），是用来对类的结构（包、类、构造器、方法、属性、参数和局部变量）进行补充说明，并不会改变原有结构。注解不是程序本身，可以对程序做出解释约束，可以被其他程序读取。

## 生成文档常见注解

@author 标明开发该类模块的作者，多个作者之间使用,分割

@version 标明该类模块的版本

@see 参考转向，也就是相关主题

@since 从哪个版本开始增加的

@param 对方法中某参数的说明，如果没有参数就不能写，

@return 对方法返回值的说明，如果方法的返回值类型是void就不能写

@exception 对方法可能抛出的异常进行说明，如果方法没有用throws显式抛出的异常就不能写

@param的格式要求：@param 形参名 形参类型 形参说明

@return 的格式要求：@return 返回值类型 返回值说明

@exception的格式要求：@exception 异常类型 异常说明

@param和@exception可以并列多个

|  |
| --- |
| /\*\*  \* **@author** Irene  \* **@version** 1.0  \*/  **public** **class** TestJavadoc {  /\*\*  \* **@param** args String[]  \*/  **public** **static** **void** main(String[] args) {  }  /\*\*  \* **@param** radius double  \* **@return** double  \*/  **public** **static** **double** getArea(**double** radius){  **return** Math.***PI*** \* radius \* radius;  }  } |

## 编译时格式检查常见注解

@Override：限制重写父类的方法，如果没有重写则编译时报错，该注解只能用于方法。

@Deprecated：提醒属性方法已过时，不建议继续使用，主要为了解决兼容性的问题。

@SuppressWarnings：抑制编译器警告，与前两个注释有所不同，需要已定义的参数才能使用。多个参数可以以如下方式使用：@SuppressWarnings(value={“unchecked”,”deprecation”})

|  |  |
| --- | --- |
| **参数** | **说明** |
| deprecation | 使用了过时的类或方法的警告 |
| unchecked | 执行了为检查的转换时的警告，如使用集合未指定泛型 |
| fallthrough | 当在switch语句使用时发生case穿透 |
| path | 在类路径、源文件路径等有不存在路径的警告 |
| serial | 当可序列化的类缺少serialVersionUID定义时的警告 |
| finally | 任何finally子句不能完成时的警告 |
| all | 关于以上所有情况的警告 |
| rawtypes | 忽略类型安全 |
| unused | 忽略不使用 |
| null | 忽略空指针 |

## 自定义注解

和类、接口、枚举一个级别，同一个文件中只能有一个声明为public。[使用@interface自定义注解时，自动继承了java.lang.annotation.Annotation](mailto:使用@interface自定义注解时，自动继承了java.lang.annotation.Annotation)接口。

注解的参数在定义时以无参方法的形式来声明，其方法名和返回值定义了该参数的名字和类型，称之为配置参数。参数类型只能是八种基本数据类型、String类型、Class类型、enum类型、Annotation类型和以上所有类型的数组。

可以使用default关键字在定义注解的参数时为其指定初始值,如果只有一个参数，建议使用参数名为value。

没有参数的注解称为标记;包含参数的注解称为元数据，注意：自定义注解必须配上注解的信息处理流程才有意义。

注解定义格式：

权限修饰符 @interface 注解名{

String name() default “Tom（这个值可以随意设置为任意名称）”；//这里name理解为属性

int[] nums();

}

注解属性格式：修饰符 返回值类型 属性名() [default默认值]

1、修饰符默认是public abstract，且只能是public abstract

2、返回值类型：基本数据类型、字符串、Class、注解、枚举以及以上类型的一维数组

3、属性名：自定义

4、default默认值：可以省略

### 注解使用注意

1、注解可以没有属性，如果有属性需要使用小括号括住，如@Anno1或@Anno2()

2、属性格式为：属性名=属性值，多个属性使用逗号分隔

3、属性名为value且当前只有一个属性，value可以省略；当前有多个属性时，value不可以省略

4、属性类型为数组，设置内容格式为：{1，2，3}；如果数组的值只有一个可以省略大括号

5、一个对象上，注解只能使用一次，不能重复使用

## 元注解

元注解（**meta-annotation**）即用在注解上面的注解（元数据：用来描述数据的数据），JDK提供了4中元注解，分别为@Retention、@Target、@Documented和@Inherited。

@Retention：用来说明使用该注解的注解的生命周期

SOURCE：（废弃了）源文件到编译期间前

CLASS：编译期间到运行期间

RUNTIME：运行期间（如果想要通过反射获取注解中的信息，那么该注解的生命周期必须是RUNTIME，可以用于取代XML配置)

@Target：用来说明使用该注解的注解可以使用在哪些结构上

CONSTRUCTOR：用于描述构造器

FIELD：用于描述属性

LOCAL\_VARIABLE：用于描述局部变量

METHOD：用于描述方法

PACKAGE：用于描述包

PARAMETER：用于描述参数

TYPE：用于描述类、接口（包括注解类型）或枚举

@Documented：用来说明使用该注解的注解是否可以被javadoc所解析

@Inherited：用来说明使用该注解的注解是否可以被继承

|  |
| --- |
| **public** **class** AnnotationTest {  **public** **static** **void** main(String[] args) {  Subclass subclass = **new** Subclass();  subclass.method();  // @SuppressWarnings("unused")：抑制编译器警告。  @SuppressWarnings("unused")  **int** a = 10;  }  }  **class** SuperClass{  **public** **void** method() {  System.***out***.println("this is original");  }  }  **class** Subclass **extends** SuperClass{  //@Deprecated：提醒属性方法已过时，不建议继续使用，主要为了解决兼容性的问题。  @Deprecated  **int** ~~a~~ = 10;  String str = "string";  //Override注解：限制重写父类的方法，如果没有重写则编译时报错，该注解只能用于方法。  @Override  **public** **void** method() {  System.***out***.println("this is new");  }  } |

## Junit单元测试

JUnit是由 Erich Gamma 和 Kent Beck 编写的一个回归测试框架（regression testing framework）,供Java开发人员编写单元测试之用。多数Java的开发环境都已经集成了JUnit作为单元测试的工具。

Junit测试是程序员测试，即所谓白盒测试，因为程序员知道被测试的软件如何（How）完成功能和完成什么样（What）的功能。

要使用JUnit，必须在项目的编译路径中必须引入JUnit的库，即相关的.class文件组成的jar包。

使用JUnit测试的类必须是public的。

JUnit4常见的注解和要求：使用JUnit测试的类必须是public，类名不能叫Test，方法必须是public，无参，无返回值。

@Test：标记在非静态的测试方法上。只有标记@Test的方法才能被作为一个测试方法单独测试。一个类中可以有多个@Test标记的方法。运行时如果只想运行其中一个@Test标记的方法，那么选择这个方法名，然后单独运行，否则整个类的所有标记了@Test的方法都会被执行。

@Test(timeout=1000)：设置超时时间，如果测试时间超过了你定义的timeout，测试失败

@Test(expected)： 申明出会发生的异常，比如 @Test（expected = Exception.class）

@BeforeClass：标记在静态方法上。因为这个方法只执行一次。在类初始化时执行。

@AfterClass：标记在静态方法上。因为这个方法只执行一次。在所有方法完成后执行。

@Before：标记在非静态方法上。在@Test方法前面执行，而且是在每一个@Test方法前面都执行

@After：标记在非静态方法上。在@Test方法后面执行，而且是在每一个@Test方法后面都执行

@Ignore：标记在本次不参与测试的方法上。这个注解的含义就是“某些方法尚未完成，暂不参与此次测试”。

@BeforeClass、@AfterClass、@Before、@After、@Ignore都是配合@Test它使用的，单独使用没有意义。

|  |
| --- |
| **public** **class** TestJUnit2 {  **private** **static** Object[] *array*;  **private** **static** **int** *total*;  @BeforeClass//标记在静态方法上。因为这个方法只执行一次。在类初始化时执行。  **public** **static** **void** init(){  System.***out***.println("初始化数组");  *array* = **new** Object[5];  }  @Before//标记在非静态方法上。在@Test方法前面执行，而且是在每一个@Test方法前面都执行  **public** **void** before(){  System.***out***.println("调用之前total=" + *total*);  }  //  @Test//标记在非静态的测试方法上。只有标记@Test的方法才能被作为一个测试方法单独测试。  **public** **void** add(){  System.***out***.println("add");  *array*[*total*++] = "hello";  }  @After//标记在非静态方法上。在@Test方法后面执行，而且是在每一个@Test方法后面都执行  **public** **void** after(){  System.***out***.println("调用之前total=" + *total*);  }  @AfterClass//标记在静态方法上。因为这个方法只执行一次。在所有方法完成后执行。  **public** **static** **void** destroy(){  *array* = **null**;  System.***out***.println("销毁数组");  }  } |

### 单元测试类

**//单元测试时一定要单独创建单元测试类，千万不能在要测试的方法上进行测试！！！！！！！！！！**

**Eclipse创建单元测试类步骤**

1. 在需要测试的类上右键创建JUnit Test Case
2. 如果需要初始化属性等操作则勾选setUpBeforeClass
3. 点击next，勾选需要测试的方法

|  |
| --- |
| **class** ExampleTest {  @BeforeAll  **static** **void** setUpBeforeClass() **throws** Exception {  }  @Test  **void** testTest1() {  *fail*("Not yet implemented");  }  @Test  **void** testTest2() {  *fail*("Not yet implemented");  }  } |

## JDK1.8注解的新特性

Java 8对注解处理提供了两点改进：可重复的注解及可用于类型的注解。此外，反射也得到了加强，在Java8中能够得到方法参数的名称。这会简化标注在方法参数上的注解。

在java 8之前，注解只能是在声明的地方所使用，java8开始，注解可以应用在任何地方。

ElementType.TYPE\_PARAMETER 表示该注解能写在类型变量的声明语句中（eg：泛型声明）。

ElementType.TYPE\_USE 表示该注解能写在使用类型的任何语句中。

类型注解被用来支持在Java的程序中做强类型检查。配合第三方插件工具Checker Framework（使用Checker Framework可以找到类型注解出现的地方并检查），可以在编译的时候检测出runtime error（eg：UnsupportedOperationException； NumberFormatException；NullPointerException异常等都是runtime error），以提高代码质量。这就是类型注解的作用。

注意java 5,6,7版本是不支持注解@NonNull，但checker framework 有个向下兼容的解决方案，就是将类型注解@NonNull 用/\*\*/注释起来。

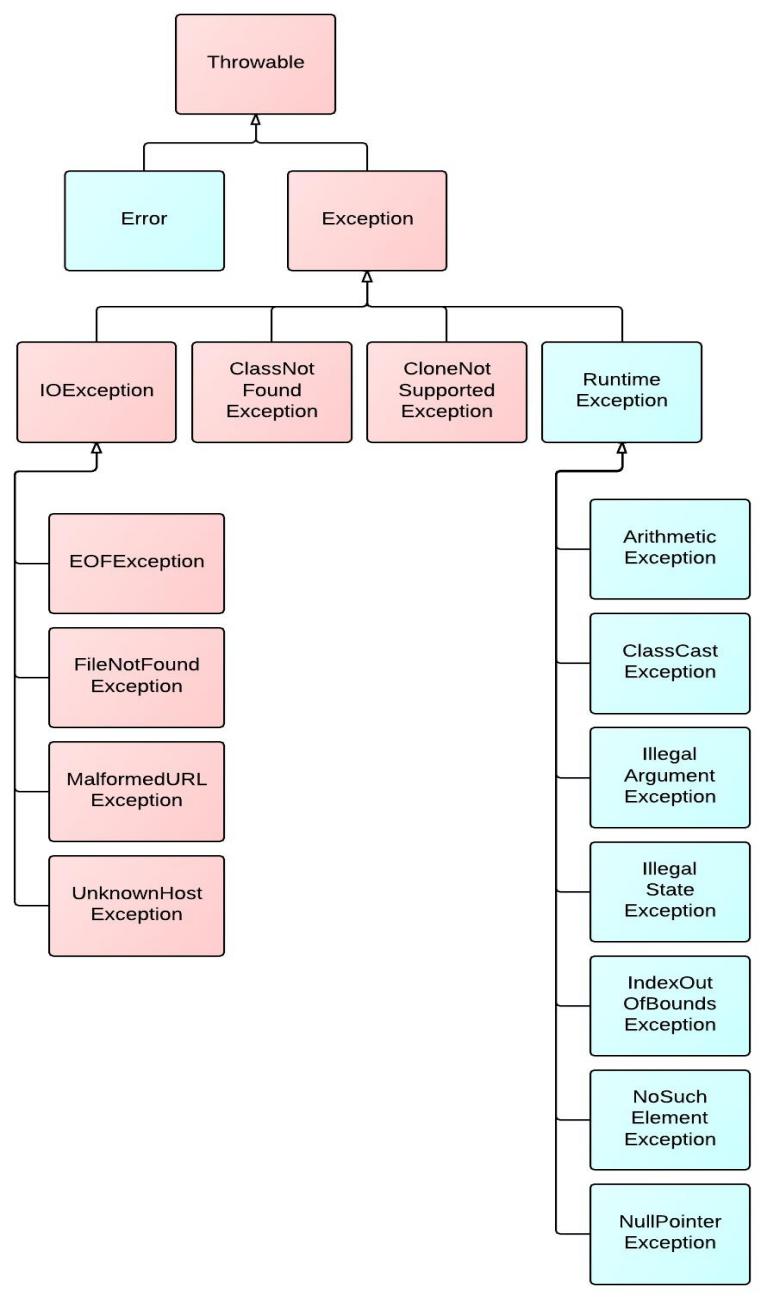
这样javac编译器就会忽略掉注释块，但用checker framework里面的javac编译器同样能够检测出@NonNull错误。

## 注解的使用

框架部分许多信息需要进行配置，有2种形式的配置方式，分为注解和配置properties或xml文件的方式。

如果配置信息不会发生修改，如servlet路径，建议使用注解的形式；如果配置信息需要频繁的修改，建议使用配置文件的方式。

# Exception



在Java语言中，将程序执行中发生的不正常情况称为“异常”。

Throwable 类是 Java 语言中所有错误或异常的超类。只有当对象是此类（或其子类之一）的实例时，才能通过 Java 虚拟机或者 Java throw 语句抛出。类似地，只有此类或其子类之一才可以是 catch 子句中的参数类型。

两个子类的实例，Error 和 Exception，通常用于指示发生了异常情况。通常，这些实例是在异常情况的上下文中新近创建的，因此包含了相关的信息（比如堆栈跟踪数据）。

Error：Java虚拟机无法解决的严重问题，如：JVM系统内部错误、资源耗尽等严重情况。

Exception：其它因编程错误或偶然的外在因素导致的一般性问题，可以使用针对性的代码进行处理。

1、(蓝色)运行时异常（非受检异常UnChecked）：是指编译器不要求强制处置的异常，由Error类和 RuntimeException类及其子类异常是运行时异常，其他都是编译时异常，一般遇到运行时异常不会处理或者抛出，而是通过修改源码的方式解决运行时异常。

2、（红色）编译时异常（受检异常Checked）：是指编译器要求必须处置的异常，即程序在运行时由于外界因素造成的一般性异常。编译器要求java程序必须捕获或声明所有编译时异常。对于这类异常，如果程序不处理，可能会带来意想不到的结果。

Error 错误，一般是指虚拟机相关的问题，如系统崩溃、虚拟机出错误、动态链接失败等，这种错误无法恢复或不可能捕获，将导致应用程序中断。通常应用程序无法处理这些错误，因此应用程序不应该试图使用 catch 块来捕获 Error 对象。由于编译器会对 Error 进行检查，不会强制要求程序员必须处理 Error，因此 Error 也被归入 unchecked 异常分类中（另外：运行时异常也属于 unchecked 异常） 。

Exception 表示一种设计或实现问题。也就是说，程序员应该对这些情况进行考虑、并提供相应的处理。

在Java中默认的异常处理方式是将异常的名称、错误原因以及异常出现的位置等信息输出到控制台，然后停止发生异常的程序。

|  |  |
| --- | --- |
| **Throwable方法摘要** | |
| String | **getMessage**()获取异常信息 |
| String | **toString**()获取异常类名和异常信息 |
| void | **printStackTrace**(overload)输出异常信息，根据参数不同输出的位置不同，空参表示输出到控制台 |

## 抓抛模型

Java中异常以对象的形式出现，不同的异常对象中封装了相应的错误信息。

**抛出异常**：当执行代码时，一旦遇到异常代码，系统会根据异常的类型创建对应的异常类的对象并抛出，抛出给方法的调用者，同时终止程序的运行。

**异常对象的生成**：

**系统向外抛异常**：由虚拟机自动生成：程序运行过程中，虚拟机检测到程序发生了问题，如果在当前代码中没有找到相应的处理程序，就会在后台自动创建一个对应异常类的实例对象并抛出——自动抛出。

**手动向外抛异常**：由开发人员手动创建：Exception exception = new ClassCastException();——创建好的异常对象不抛出对程序没有任何影响，和创建一个普通对象一样。

**捕获异常**：

try-catch-finally结构语句

throws抛出异常声明

## 异常处理方式一

格式：

try{

可能会发生异常的代码

}catch（异常的类型1 变量）{

异常的处理方式1；

}catch（异常的类型2 变量）{

异常的处理方式2；

}

……

finally{

必然会执行的代码；

该代码无论是否抛出异常都会执行，只有调用了System.exit(0)方法关闭虚拟机才会使该代码不执行。

}

当try中的代码在执行的过程中，一旦执行某行代码时发生异常，系统会创建对应的异常类型的对象向外抛；catch后面的异常类型会和抛出的异常的对象进行类型匹配，一旦匹配成功则执行相应的异常处理的代码；当catch中的代码执行完毕时再执行finally中的代码；finally中的代码执行完毕后跳出该结构执行后面的代码。

可以有多个catch语句，异常对象将从上到下依次匹配类型，一旦某个catch的异常类型匹配成功，其他的catch将不再执行，即一个异常只能进入一个catch块。因此catch后面的类型如果有子父类关系，则应子类在上，父类在下。

如果对不同类型的异常有不同的处理方式，那么不能直接写父类的异常类型；反之如果处理方式都相同，那么异常的类型可以是父类的异常类型。

无论异常是否能捕获到，或者处理异常时再次放生异常，finally都会执行，因此一般用于关闭资源和保存数据。

try不可以省略，catch和finally都可以省略，因此由try-catch结构和try-finally结构。

常用处理异常的方式:

e.printStackTrace();输出异常错误信息的名称

e.getMessage();输出异常的详细信息

就算在try或者catch中已经调用return了，finally中的代码也一定会执行。

注意：在实际开发中，一般都不会处理运行时异常，而是抛出错误后改代码。编译时异常则必须要处理，否则无法编译通过并运行。

|  |
| --- |
| **public** **class** ExceptionTest {  **public** **static** **void** main(String[] args) {  **try** {  System.***out***.println(*staticMethod*());  System.***out***.println(1/0);  }**catch** (ArithmeticException e) {  //e.getMessage()方法获取异常信息，返回字符串  System.***out***.println("Info:" + e.getMessage());  //e.printStackTrace()方法获取异常类名和异常信息，以及异常出现在程序中的位置，返回值void。  e.printStackTrace();  }**catch** (Exception e) {  e.getMessage();  }**finally** {  //finally里的信息无论无何都会执行  System.***out***.println("I have to be executed");  }  }  @SuppressWarnings("finally")  **public** **static** **int** staticMethod() {  //如果finally中有return语句，那么try-catch中的return都不会被执行  **try** {  **return** 10;  } **catch** (Exception e) {  **return** 20;  }**finally** {  **return** 30;  }  }  } |

## 异常处理方式二

格式：方法名() throws 异常类型1，异常类型2...{

方法体；

}

throws捕获到异常以后自己不处理，抛给方法的调用者进行处理。

throws和try-catch的区别：

throws自己不处理异常，将异常向上抛，抛给方法的调用者处理，但是最后还是要try-catch进行处理。

try-catch真正处理了异常。

什么时候选用throws：如果在main中调用了多个方法，而且传输了数据，一旦因为数据的错误导致的异常，中间的所有的方法都不能进行处理，这个异常必须由传数据的那个人来进行处理。

什么时候不能选用throws：异常必须要被处理的时候；如果父类被重写的方法没有抛出异常，那么子类重写的方法也不能抛出异常；子类重写的方法抛出的异常和父类被重写方法抛出的异常类型相同且不大于被重写方法抛出的异常。

**重写方法对throws异常的要求**

父类被重写的方法没有声明抛出checked受检异常，那么重写的方法也不能声明抛出异常

子类重写方法声明抛出的异常的类型和父类被重写的方法声明抛出异常类型一致

子类重写方法声明抛出的异常的类型是父类被重写的方法声明抛出异常的子类

子类重写方法可以在方法内部处理异常，而不声明抛出异常

|  |
| --- |
| **public** **class** ExceptionThrowsTest {  **public** **static** **void** main(String[] args) {  SubClass2 sc = **new** SubClass2();  sc.method1();  //异常必须要被处理的时候，就不能再继续抛异常  **try** {  sc.method2();  } **catch** (IOException e) {  e.printStackTrace();  }  }  }  **class** SuperClass2{  **public** **void** method1() { }  **public** **void** method2() **throws** Exception { }  }  **class** SubClass2 **extends** SuperClass2{  //如果父类被重写的方法没有抛出异常，那么子类重写的方法也不能抛出异常  **public** **void** method1() {}  //子类重写的方法抛出的异常不大于父类被重写方法抛出的异常  **public** **void** method2() **throws** IOException {}  } |

## 异常结构和return

try{}有return时finally{}中的语句也肯定会执行。finally{}块的代码只有在try{}块中包含遇到System.exit(0);之类的导致Java虚拟机直接退出的语句才会不执行。当程序执行 try{}遇到 return 时，程序会先执行 return 语句，但并不会立即返回——也就是把 return 语句要做的一切事情都准备好，也就是在将要返回、但并未返回的时候，程序把执行流程转去执行 finally 块，当 finally 块执行完成后就直接返回刚才 return 语句已经准备好的结果。

如果finally语句中有return语句，则不论try-catch中是否有return，都执行finally中的return。

如果finally语句中没有return语句，若catch中有return，判断是否有异常被捕获，如果有异常被捕获，则执行catch中的return语句；如果没有异常被捕获，则不会执行catch中的return语句。

try或catch中的return返回的返回值如果是数组或集合，则finally中对该数组的操作会影响到try和catch中的return返回的结果。

|  |
| --- |
| **public** **class** ExceptionReturnTest {  **public** **static** **void** main(String[] args) {  System.***out***.println(*method1*());  System.***out***.println(*method2*());  System.***out***.println(*method3*());  System.***out***.println(*method4*());  }  //如果finally语句中有return语句，则不论try-catch中是否有return，都执行finally中的return  @SuppressWarnings("finally")  **public** **static** **int** method1() {  String str = "a";  **try** {  Integer.*parseInt*(str);  **return** 1;  } **catch** (NumberFormatException e) {  System.***out***.print("抛出异常");  **return** -1;  } **finally** {  System.***out***.print("test结束");  **return** 10000;  }  }  @SuppressWarnings("finally")  **public** **static** **int** method2() {  String str = "10";  **try** {  Integer.*parseInt*(str);  **return** 1;  } **catch** (NumberFormatException e) {  System.***out***.print("抛出异常");  **return** -1;  } **finally** {  System.***out***.print("test结束");  **return** 10000;  }  }  //如果finally语句中没有return语句，若catch中有return，判断是否有异常被捕获  //如果有异常被捕获，则执行catch中的return语句  **public** **static** **int** method3() {  String str = "a";  **try** {  Integer.*parseInt*(str);  **return** 1;  } **catch** (NumberFormatException e) {  System.***out***.print("抛出异常");  **return** -1;  } **finally** {  System.***out***.print("test结束");  }  }  //如果finally语句中没有return语句，若catch中有return，判断是否有异常被捕获  //如果没有异常被捕获，则不会执行catch中的return语句  **public** **static** **int** method4() {  String str = "10";  **try** {  Integer.*parseInt*(str);  **return** 1;  } **catch** (NumberFormatException e) {  System.***out***.print("抛出异常");  **return** -1;  } **finally** {  System.***out***.print("test结束");  }  }  } |

## 手动向外抛异常

throw：手动向外抛异常

格式：throw 异常类的对象

### throws和throw的区别

throw用于抛出异常对象，后面跟的是异常对象；throw用在方法内；thorw表示一定发生了异常然后被抛出。

throws用于抛出异常类，后面跟的异常类名，可以跟多个，用逗号隔开；throws用在函数上；throws表示异常出现的一种可能性，并不一定会发生这些异常。

运行时异常没有提醒的作用，编译时异常在编译时会有提醒的作用。



throw用于主动抛出java.lang.Throwable 类的一个实例化对象，意思是说你可以通过关键字 throw 抛出一个 Error 或者一个Exception，如：throw new IllegalArgumentException(“size must be multiple of 2″)。  
 而throws 的作用是作为方法声明和签名的一部分，方法被抛出相应的异常以便调用者能处理。Java 中，任何未处理的受检查异常强制在 throws 子句中声明。

## 自定义异常

一般地，自定义异常类都是RuntimeException的子类，需要编写几个重载的构造器，只能通过throw抛出，最重要的是异常类的名字，当异常出现时，要可以根据名字判断异常类型。

1、创建一个类继承Exception（编译时异常）或者RuntimeException（运行时异常）

2、定义两个构造器，一个空参，一个有参

3、生成serialVersionUID，可以写也可以不写，如果不写，系统会自动生成一个默认的

|  |
| --- |
| **public** **class** LoginTest {  **public** **static** **void** main(String[] args) **throws** MyException {  String userName = "john";  String userPassword = "0000";  **if** ("john".equals(userName)) {  **if** ("000".equals(userPassword)) {  System.***out***.println("login success！");  }**else** {  //通过throw关键字手动抛出自定义的异常，同时在方法上声明这个异常  **throw** **new** MyException("wrong password");  }  }**else** {  **throw** **new** MyException("wrong username");  }  }  }  //自定义异常，可以继承运行时异常或编译时异常  **class** MyException **extends** Exception{  **private** **static** **final** **long** ***serialVersionUID*** = 1L;  MyException() {  }  MyException(String message){  //调用Throwable中的构造方法，返回异常的对象  **super**(message);  }  } |

## 异常小结

### 异常相关关键字

**try**：执行可能发生异常的代码。

**catch**：尝试捕获try代码块发生的异常。可以存在多个catch，如果多个catch的异常类型有继承关系，那么遵循子上父下。

**finally**：不管是否发生异常都要执行的代码，一般用于执行关闭资源的代码。

**throws**：方法声明时显示抛出异常，指定该方法可能抛出的异常类型列表。

**throw**：手动抛出异常，可以抛出系统预定异常，也可以抛出自定义异常。且自定义异常只能用throw语句抛出。~~可以代替return结束方法运行，~~**谨记不要使用异常来代替return结束方法**。

在Java7以后，JDK提供了一种更好的方式关闭资源，即使用**try-with-resources**语句，对于自动释放资源的对象必须要实现AutoCloseable接口。具体用法如下：

|  |
| --- |
| **try** (BufferedReader br = **new** BufferedReader(**new** FileReader("d:\\ hollischuang.xml"))) {  String line;  **while** ((line = br.readLine()) != **null**) {  System.***out***.println(line);  }  } **catch** (IOException e) {  } |

### 异常使用注意

避免使用异常处理代替错误处理，会降低程序的清晰性，且效率低下。

不要使用小粒度的异常处理，应将整个任务包装在一个try语句块中。

异常往往在高层处理，因此有问题优先上报，其次捕获并解决。

### 异常在继承中体现

子类在覆盖父类时，如果父类的方法抛出异常，那么子类的覆盖方法只能抛出父类的异常或者该异常的子类。

如果父类方法抛出多个异常，那么子类在覆盖该方法时，只能抛出父类异常的子集。

如果父类或者接口的方法中没有异常抛出，那么子类在覆盖方法时，也不可以抛出异常。如果子类方法发生了异常，就必须要进行try处理，绝对不能抛。

### Java异常处理过程

（1）当程序运行到某一句代码，如果发生了异常（可能是JVM判定的异常，也可能是遇到throw的），程序都会停下来，然后把异常信息封装到异常的对象中，并且“抛”出。

（2）JVM会检测在这段程序代码的外围，是否有try-catch，如果有try-catch，就判断是否有catch可以捕获它，如果捕获了，程序就进入对应的catch块进行异常处理，处理后程序继续运行try-catch之后的代码。如果没有try-catch或者是有try-catch但是捕获不住，即类型对不上，JVM都会把这个异常对象抛出上级（方法的调用者）。

（3）上级一旦接到异常对象，处理过程还是1,2。

（4）如果一直抛，一路上都没有可以捕获它，程序就崩溃了。

# Generic

为了解决集合中元素存储的安全性问题和获取元素时需要强制类型转换的问题，Java在JDK1.5后引入了泛型（**Generic**）的概念。其原理是在类声明时通过一个标识表示类中的某个属性的类型或者某个方法的返回值及参数类型，这样在类声明或实例化时只要指定好需要的具体类型即可。

泛型可以保证程序在编译时如果没有发生警告，运行时就不会产生ClassCastException异常，同时代码更简洁健壮。泛型既是对编码的约束，也是对编码的扩展。在编译后的字节码中没有泛型。

**为什么要使用泛型**

解决元素存储的安全性问题

解决获取元素时，需要类型强制转换的问题

## 泛型的使用

泛型可以在类、接口、方法中使用。

### 泛型在类中的使用

**泛型的使用**

在创建泛型类的对象时，指明泛型的类型，然后泛型类中凡是用到泛型的地方都被具体化了（指明什么类型，泛型就是什么类型）。

在创建泛型类的对象时，如果没有指明泛型的类型，那么泛型的类型默认是Object类型。

从泛型类的子类泛型类型需要具体化。

静态方法中不能使用类的泛型。

异常类不能是泛型的。

在类或接口上声明的泛型不能使用在静态成员上。

泛型的指定中不能使用基本数据类型，可以使用包装类替代。

|  |
| --- |
| **class** Person<T>{  //使用T类型定义变量  **private** T t;  //使用T类型定义一般方法  **public** T getInfo(){  **return** t;  }  **public** **void** setInfo(T t){  **this**.t = t;  }  //使用T类型定义构造器  **public** Person(){}  **public** Person(T t){  **this**.t = t;  }  //static的方法中不能声明泛型  // public static void show(T t){  // }  //不能在try-catch中使用泛型定义  // try{}  // catch(MyException<T> ex){}  } |

### 泛型在集合中的使用

**泛型在集合中使用的好处**

解决元素存储的安全问题；

解决获取数据元素时，需要类型强转的问题。

**泛型在集合中的使用**

加入集合中的对象类型必须与指定的泛型类型一致。

|  |
| --- |
| **public** **void** method() {  //JDK1.7以后有类型推断，前面声明了泛型，后面可以省略。  Collection<String> collection = **new** ArrayList<>();  collection.add("aaa");  collection.add("bbb");  System.***out***.println(collection);  } |

### 指定泛型的范围

（1）ArrayList<String> list = new ArrayList<String>()声明集合变量或创建集合对象，指定泛型

（2）class Dog implements Comparable<Dog>{...}实现接口时，指定泛型

（3）public void test(ArrayList<Student> list){}使用泛型类或接口作为形参时，此处指定为学生类型

（4）public void test(ArrayList<?> list){}使用泛型类或接口作为形参时，此处指定为任意类型

（5）public void test(ArrayList<? **extends** Person>使用泛型类或接口作为形参时，此处指定为Person或其子类

（6）public void test(ArrayList<? **super** Son>使用泛型类或接口作为形参时，此处指定为Son或其父类

## 自定义泛型

### 自定义泛型类和接口

泛型应该但不强制用大写字母，比如K(key)、T(type)、V(value)、E(element)等，类上的泛型是创建对象时决定的。

通过子类指明父类的泛型类型

在子类继承父类时，直接指明父类的泛型类型

class B extends A<String>{}

可以通过创建子类对象时指明父类的泛型类型

class B<T> extends A<T>{}

#### 泛型在继承上的体现

泛型中没有多态的特性，泛型中类型必须完全一致，子父类无法互相转型。只要泛型类型不一样，就没有联系。

如果B是A的一个子类型（子类或者子接口），而G是具有泛型声明的类或接口，G<B>并不是G<A>的子类型。

|  |
| --- |
| **class** Father<T1, T2> {  }  //子类不保留父类的泛型  //1)没有类型 擦除  @SuppressWarnings("rawtypes")  **class** Son **extends** Father {// 等价于class Son extends Father<Object,Object>{  }  //2)具体类型  **class** Son2 **extends** Father<Integer, String> {  }  //子类保留父类的泛型  //1)全部保留  **class** Son3<T1, T2> **extends** Father<T1, T2> {  }  //2)部分保留  **class** Son4<T2> **extends** Father<Integer, T2> {  }  //子类不保留父类的泛型  //1)没有类型 擦除  @SuppressWarnings("rawtypes")  **class** BSon<A, B> **extends** Father {// 等价于class Son extends Father<Object,Object>{  }  //2)具体类型  **class** BSon2<A, B> **extends** Father<Integer, String> {  }  //子类保留父类的泛型  //1)全部保留  **class** BSon3<T1, T2, A, B> **extends** Father<T1, T2> {  }  //2)部分保留  **class** BSon4<T2, A, B> **extends** Father<Integer, T2> {  } |

### 自定义泛型方法

方法也可以被泛型化，泛型方法上的泛型不依赖于类的泛型，通过传入实参（实参传给E类型的形参）指明方法上的泛型类型。泛型方法可以用static修饰。

**泛型方法格式**

public <E> E getE<E e,int age...>{

}

访问权限 <泛型> 返回类型 方法名(泛型标识 参数名称) 抛出的异常{

}

细节：在静态方法中不能使用泛型类上的泛型

|  |
| --- |
| **public** <E> E get(**int** id, E e){  E result = **null**;  **return** result;  } |

## 通配符

通配符：<?> 可以理解成是所有泛型的父类，使用通配符的集合只能添加null，使用通配符的集合可以遍历所有元素。

只能用于声明变量|形参上，一般是形参类型上，表示可以接收任意

不能用在

创建对象，即new 后面

声明泛型类、泛型接口，即class和interface

声明泛型方法，即返回值类型前面<>中

通配符的约束

**<? extends Object>**泛型的类型只能是Object和Object的子类

**<? super Object>**泛型的类型只能是Object和Object的父类

**<? extends Comparable>**泛型的类型只能是实现了Comparable接口的类

## 泛型细节

### 编译器对泛型的处理

编译器处理泛型的方式有两种：**Code specialization**（C和C++） 和**Code sharing**（Java）。

Code sharing方式为每个泛型类型创建唯一的字节码表示，并且将该泛型类型的实例都映射到这个唯一的字节码表示上。将多种泛型类形实例映射到唯一的字节码表示是通过类型擦除（type erasue）实现的。对于Java虚拟机来说，无法识别Map<String, String> map这样的语法，需要在编译阶段通过类型擦除的方式进行解语法糖。

**类型擦除的主要过程**：

1. 将所有的泛型参数用其最左边界（最顶级的父类型）类型替换
2. 移除所有的类型参数

|  |
| --- |
| Map<String, String> map = **new** HashMap<String, String>();  map.put("name", "hollis");  map.put("wechat", "Hollis");  map.put("blog", "www.hollischuang.com");  解语法糖后  Map map = **new** HashMap<String, String>();  map.put("name", "hollis");  map.put("wechat", "Hollis");  map.put("blog", "www.hollischuang.com"); |

|  |
| --- |
| **public** **static** <A **extends** Comparable<A>> A max(Collection<A> xs) {  Iterator<A> xi = xs.iterator();  A w = xi.next();  **while** (xi.hasNext()) {  A x = xi.next();  **if** (w.compareTo(x) < 0)  w = x;  }  **return** w;  }  类型擦除后  **public** **static** Comparable max(Collection xs){  Iterator xi = xs.iterator();  Comparable w = (Comparable)xi.next();  **while**(xi.hasNext())  {  Comparable x = (Comparable)xi.next();  **if**(w.compareTo(x) < 0)  w = x;  }  **return** w;  } |

虚拟机中没有泛型，只有普通类和普通方法，所有泛型类的类型参数在编译时都会被擦除，泛型类并没有自己独有的Class类对象。比如并不存在List<String>.class或是List<Integer>.class，而只有List.class。

### 泛型和重载

在下面这段代码无法编译通过，虽然参数的泛型不同，但是编译后泛型会被擦除，变成了两个参数一样的方法，导致无法构成重载

|  |
| --- |
| **public** **static** **void** method(List<String> list) {  System.***out***.println("invoke method(List<String> list)");  }  **public** **static** **void** method(List<Integer> list) {  System.***out***.println("invoke method(List<Integer> list)");  } |

### 泛型和catch

泛型的类型参数不能用在Java异常处理的catch语句中。因为异常处理是由JVM在运行时刻来进行的。由于类型信息被擦除，JVM是无法区分两个异常类型MyException<String>和MyException<Integer>的

# Collection

Java中用来存储多个对象的容器：集合和数组，集合可以动态的将对象存入到集合中，分为Collection和Map两类，位于java.util包下。

集合的特点

1. 只能存储对象
2. 容量可以自动调节
3. 可以直接获取有效元素的个数

Collection是存放单列集合的，可以理解为直接放入一个个数据

Collection接口没有直接的实现子类， 而是有两个子接口List , Set

List 子接口的常用的实现类有 ArrayList ,LinkedList, Vector

Set子接口常用的实现类有 HashSet 和 TreeSet

Map是存放双列集合的，比如 key-value

Map有直接的实现类，常见的有 HashMap, TreeMap, Hashtable, Propeties

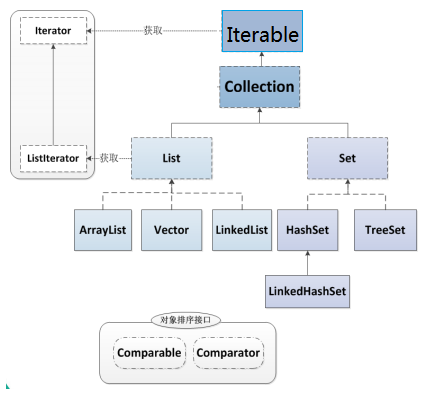
**集合和数组的区别：**

1：数组是固定长度的；集合可变长度的。

2：数组可以存储基本数据类型，也可以存储引用数据类型；集合只能存储引用数据类型。

3：数组存储的元素必须是同一个数据类型；集合存储的对象可以是不同数据类型。

## Collection接口



contains方法底层调用了对象的equals方法，为true，则说明包含这个元素。调用的是对象类重写的equals方法，比较的是内容。只要Collection集合中存储的是自定义类的对象，那么该对象所在的类必须重写equals方法。

**Collection接口提供的方法**

|  |  |
| --- | --- |
| **Collection接口方法摘要** | |
| boolean | **add**(E e) 集合中添加元素 |
| boolean | **addAll**(Collection<? extends E> c) 将c中的元素添加到当前集合中 |
| int | **size**()获取当前集合中元素的个数 |
| void | **clear**()删除当前集合中所有的元素 |
| boolean | **contains**(Object o) 判断当前集合中是否含有obj元素 |
| boolean | **containsAll**(Collection<?> c) 判断c集合是否是当前集合的子集合，c集合中重复元素视为一种元素 |
| boolean | **equals**(Object o) 集合中元素个数、顺序、内容都必须相同才返回true |
| boolean | **isEmpty**()判断当前元素个数是否为0 |
| boolean | **remove**(Object o) 删除当前集合中的指定元素，删除成功返回true |
| boolean | **removeAll**(Collection<?> c) 删除当前集合中和c集合相同的元素 |
| boolean | **retainAll**(Collection<?> c) 保留当前集合中和c集合相同的元素，删除不同元素 |
| <T> T[] | **toArray**(T[] a)将当前集合转换为一个数组，注意类型匹配 |
| Iterator<E> | **iterator**()返回Iterator接口实现类的对象，即在此接口上的迭代器，用于遍历集合中的元素 |

Arrays工具类的静态方法asList(array[])将数组转换成final修饰List集合，返回的集合既不是ArrayList也不是Vector，该集合不允许添加新的元素。

|  |
| --- |
| /\*\*  \* 未添加泛型的集合使用，可以存放不同类型的数据  \* **@author** xmm  \*/  @SuppressWarnings("rawtypes")  **public** **class** CollectionTest {  Collection collection = **new** ArrayList();    @SuppressWarnings("unchecked")  // @Test  **public** **void** initi() {  //add(Object obj)向集合中添加元素  collection.add("aaa");  collection.add("bbb");  //此时1被自动装箱为Integer类型的对象  collection.add(1);  collection.add("ddd");  collection.add("fff");  Iterator iterator = collection.iterator();  System.***out***.print("属性collection:");  **while**(iterator.hasNext()) {  System.***out***.print(iterator.next()+",");  }  System.***out***.println();  }  @SuppressWarnings("unchecked")  @Test  **public** **void** method() {  Collection collection = **new** ArrayList();  //调用initi方法给属性collection添加元素，否则其默认为空  initi();  collection.addAll(**this**.collection);  Iterator iterator = collection.iterator();  System.***out***.print("局部collection:");  System.***out***.println(iterator.hasNext());  **while**(iterator.hasNext()) {  //执行完这个迭代器循环之后，iterator的指针就会指向最后，因此后面不能再用此迭代器迭代  System.***out***.print(iterator.next()+",");  }  System.***out***.println();  //获取集合元素的个数  **int** num = collection.size();  System.***out***.println(num);  **boolean** isExist = collection.contains(**new** Integer(1));  System.***out***.println("isExist:" + isExist);  //contains()该方法也会将参数自动装箱  isExist = collection.contains(1);  System.***out***.println("测试自动装箱isExist:" + isExist);  //equals(Collection c)两个集合中的元素个数、顺序、内容都必须相同才返回true  **boolean** isSame = collection.equals(**this**.collection);  System.***out***.println("isSame:" + isSame);  //remove前后迭代器的hasNext方法结果发生了变化  System.***out***.println("????+"+iterator.hasNext());  collection.remove(1);  //移除成功，hasNext结果变为了true，如果移除失败则结果没有变化  //可能是对集合的修改会改变迭代器指针的位置  System.***out***.println("????+"+iterator.hasNext());  Iterator iterator2 = collection.iterator();  System.***out***.print("局部collection:");  System.***out***.println(iterator.hasNext());  **while**(iterator2.hasNext()) {  //继续使用iterator迭代会报ConcurrentModificationException异常，所以要用新迭代器  //某个线程在 Collection 上进行迭代时，通常不允许另一个线性修改该 Collection  //System.out.print(iterator.next()+",");  System.***out***.print(iterator2.next()+",");  }  System.***out***.println();  //retainAll(Collection c)保留调用该方法的集合中和c集合相同的元素，删除不相同的元素  **this**.collection.retainAll(collection);  //使用foreach循环遍历属性collection中的元素  **for**(Object object:**this**.collection) {  System.***out***.print(object.toString()+",");  }  System.***out***.println();  //toArray()将调用该方法的集合转换为一个数组，注意类型匹配  Object[] objects = **this**.collection.toArray();  System.***out***.println(Arrays.*toString*(objects));  //removeAll(Collection c)删除调用该方法的集合中和c集合相同的元素  **this**.collection.removeAll(collection);  //isEmpty()判断当前集合元素个数是否为0  **boolean** isEmpty = **this**.collection.isEmpty();  System.***out***.println("isEmpty:"+isEmpty);  //clear()删除当前集合中所有的元素  collection.clear();  isEmpty = **false**;  isEmpty = collection.isEmpty();  System.***out***.println("isEmpty:"+isEmpty);  }  } |
| 属性collection:aaa,bbb,1,ddd,fff,  局部collection:true  aaa,bbb,1,ddd,fff,  5  isExist:true  测试自动装箱isExist:true  isSame:true  ????+false  ????+true  局部collection:true  aaa,bbb,ddd,fff,  aaa,bbb,ddd,fff,  [aaa, bbb, ddd, fff]  isEmpty:true  isEmpty:true |

### List接口

List是元素有序且可重复的集合，继承了Collection中所有的方法。List接口有三个实现类，其共同点是都实现了List接口，且所存储的元素都是有序可重复的。

对于频繁查询和修改元素的操作，可以使用ArrayList；对于频繁插入和删除元素的操作，可以使用LinkedList。

**List独有方法**

|  |  |
| --- | --- |
| **List接口方法摘要** | |
| void | **add**(int index,E element)在index位置插入obj元素 |
| boolean | **addAll**(overload)在index位置开始添加集合c中所有的元素 |
| E | **get**(int index)获取index位置的元素 |
| int | **indexOf**(Object obj)返回obj在集合中首次出现的位置 |
| int | **lastIndexOf**(Object obj)返回obj在集合中最后一次出现的位置 |
| E/boolean | **remove**(overload)移除index位置的元素，并返回该元素 |
| E | **set**(int index,E element)将index位置的元素设置为obj |
| List<E> | **subList**(int fromIndex,int toIndex)返回从fromIndex到toIndex位置的子集合，包头不包尾 |

**List细节**

List独有的remove方法并没有重写继承的接口中的remove方法，因此形参为int型则为List的remove的方法，形参是String则为Collection的remove方法。

**List三种遍历方式**

使用迭代器iterator

使用增强for-each

使用普通for

|  |
| --- |
| **public** **class** ListTest {  Collection<String> collection = **new** ArrayList<>();  **public** **void** initi() {  collection.add("aaa");  collection.add("bbb");  collection.add("ddd");  collection.add("fff");  }  @Test  **public** **void** method() {  initi();  List<String> list = **new** ArrayList<>();  //void add(int index,Object obj)在index位置插入obj元素  list.add(0,"ccc");  //boolean addAll(int index,Collection c)在index位置开始添加集合c中所有的元素  list.addAll(1, collection);  list.add(4, "aaa");  **for**(Object object : list) {  System.***out***.print(object.toString()+" ");  }  System.***out***.println();  //Object get(int index)获取index位置的元素  System.***out***.println(list.get(3).toString());  //int indexOf(Object obj)返回obj在集合中首次出现的位置  System.***out***.println(list.indexOf("aaa"));  //int lastIndexOf(Object obj)返回obj在集合中最后一次出现的位置  System.***out***.println(list.lastIndexOf("aaa"));  //Object remove(int index)移除index位置的元素，并返回该元素  //List的remove方法不会进行自动装箱，为了防止和父类的remove方法造成混乱  System.***out***.println(list.remove(5));  //Object set(int index,Object obj)将index位置的元素设置为obj  list.set(2, "zzz");  **for**(Object object : list) {  System.***out***.print(object.toString()+" ");  }  System.***out***.println();  //List subList(int fromIndex,int toIndex)返回从fromIndex到toIndex位置的子集合，包头不包尾，左闭右开  List<String> list2 = list.subList(1, 4);  **for**(Object object : list2) {  System.***out***.print(object.toString()+" ");  }  System.***out***.println();  }  } |
| ccc aaa bbb ddd aaa fff  ddd  1  4  fff  ccc aaa zzz ddd aaa  aaa zzz ddd |

#### ArrayList类

ArrayList 是一个动态数组，实现了 List 接口以及 list相关的所有方法，它允许所有元素的插入，包括 null。另外，ArrayList 和 Vector 除了线程不同步之外，大致相等。

是List接口的主要实现类，底层是是线程不安全的动态数组，底层是数组结构，便于查询和修改，判断元素是否存在或删除元素时底层依据是equals方法。

**ArrayList的JDK1.8之前与之后的实现区别？**

JDK1.8之前：ArrayList像饿汉式，直接创建一个初始长度为10的数组。

JDK1.8之后：ArrayList像懒汉式：初始化为空数组，在添加第一个元素时初始化为长度为10的数组，如果容量满了，按照1.5倍扩容，同时将原来的数组中的元素复制到新的数组中。支持foreach和Iterator遍历。

##### ArrayList属性

|  |
| --- |
| //默认容量大小 private static final int *DEFAULT\_CAPACITY* = 10; //空数组常量 private static final Object[] *EMPTY\_ELEMENTDATA* = {}; //默认的空数组常量 private static final Object[] *DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA* = {}; //存放元素的数组，从这可以发现 ArrayList 的底层实现就是一个 Object数组 transient Object[] elementData; // non-private to simplify nested class access //数组中包含的元素个数 private int size; //数组的最大上嫌 private static final int *MAX\_ARRAY\_SIZE* = Integer.*MAX\_VALUE* - 8; |

##### ArrayList构造器

elementData 是一个大小为 0 的空数组，当指定了初始大小的时候，elementData 的初始大小就变成了所指定的初始大小了。

|  |
| --- |
| public ArrayList(int initialCapacity) {  if (initialCapacity > 0) {  this.elementData = new Object[initialCapacity];  } else if (initialCapacity == 0) {  this.elementData = *EMPTY\_ELEMENTDATA*;  } else {  throw new IllegalArgumentException("Illegal Capacity: "+  initialCapacity);  } }public ArrayList() {  this.elementData = *DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA*; }public ArrayList(Collection<? extends E> c) {  elementData = c.toArray();  if ((size = elementData.length) != 0) {  // c.toArray might (incorrectly) not return Object[] (see 6260652)  if (elementData.getClass() != Object[].class)  elementData = Arrays.*copyOf*(elementData, size, Object[].class);  } else {  // replace with empty array.  this.elementData = *EMPTY\_ELEMENTDATA*;  } } |

##### ArrayList方法

**get方法获取元素**

ArrayList 是采用数组结构来存储的，所以它的 get 方法非常简单，先是判断一下有没有越界，之后就可以直接通过数组下标来获取元素了，所以 get 的时间复杂度是 O(1)。

|  |
| --- |
| public E get(int index) {  rangeCheck(index);  return elementData(index); }  private void rangeCheck(int index) {  if (index >= size)  throw new IndexOutOfBoundsException(outOfBoundsMsg(index)); }  E elementData(int index) {  return (E) elementData[index]; } |

**add方法添加元素**

ArrayList 的 add 方法也很好理解，在插入元素之前，它会先检查是否需要扩容，然后再把元素添加到数组中最后一个元素的后面。在 ensureCapacityInternal 方法中，如果当 elementData 为空数组时，它会使用默认的大小去扩容。所以说，通过无参构造方法来创建 ArrayList 时，它的大小其实是为 0 的，只有在使用到的时候，才会通过 grow 方法去创建一个大小为 10 的数组。

第一个 add 方法的复杂度为 O(1)，虽然有时候会涉及到扩容的操作，但是扩容的次数是非常少的，所以这一部分的时间可以忽略不计。如果使用的是带指定下标的 add方法，则复杂度为 O(n)，因为涉及到对数组中元素的移动，这一操作是非常耗时的。

|  |
| --- |
| public boolean add(E e) {  ensureCapacityInternal(size + 1); // Increments modCount!!  elementData[size++] = e;  return true; }  public void add(int index, E element) {  rangeCheckForAdd(index);  ensureCapacityInternal(size + 1); // Increments modCount!!  //调用一个native的复制方法，将index位置开始的元素都往后挪一位  System.*arraycopy*(elementData, index, elementData, index + 1, size - index);  elementData[index] = element;  size++; }  private void ensureCapacityInternal(int minCapacity) {  ensureExplicitCapacity(*calculateCapacity*(elementData, minCapacity)); } private static int calculateCapacity(Object[] elementData, int minCapacity) {  if (elementData == *DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA*) {  return Math.*max*(*DEFAULT\_CAPACITY*, minCapacity);  }  return minCapacity; } private void ensureExplicitCapacity(int minCapacity) {  modCount++;   // overflow-conscious code  if (minCapacity - elementData.length > 0)  grow(minCapacity); } |

ArrayList 也采用了快速失败的机制，通过记录 modCount 参数来实现。在面对并发的修改时，迭代器很快就会完全失败，而不是冒着在将来某个不确定时间发生任意不确定行为的风险。

**set方法修改元素**

set 方法的作用是把下标为 index 的元素替换成 element，跟 get 非常类似，所以就不在赘述了，时间复杂度度为 O(1)。

|  |
| --- |
| public E set(int index, E element) {  rangeCheck(index);  E oldValue = elementData(index);  elementData[index] = element;  return oldValue; } |

**remove方法移除元素**

remove 方法与 add 带指定下标的方法非常类似，也是调用系统的 arraycopy 方法来移动元素，时间复杂度为 O(n)。

|  |
| --- |
| public E remove(int index) {  rangeCheck(index);   modCount++;  E oldValue = elementData(index);  int numMoved = size - index - 1;  if (numMoved > 0)  System.*arraycopy*(elementData, index+1, elementData, index, numMoved);  elementData[--size] = null; // clear to let GC do its work  return oldValue; }  public boolean remove(Object o) {  if (o == null) {  for (int index = 0; index < size; index++)  if (elementData[index] == null) {  fastRemove(index);  return true;  }  } else {  for (int index = 0; index < size; index++)  if (o.equals(elementData[index])) {  fastRemove(index);  return true;  }  }  return false; } |

**grow方法扩充容量**

grow 方法是在数组进行扩容的时候用到的，从中可以看见，ArrayList 每次扩容都是扩 1.5 倍，然后调用 Arrays 类的 copyOf 方法，把元素重新拷贝到一个新的数组中去。

|  |
| --- |
| private void grow(int minCapacity) {  // overflow-conscious code  int oldCapacity = elementData.length;  int newCapacity = oldCapacity + (oldCapacity >> 1);  if (newCapacity - minCapacity < 0)  newCapacity = minCapacity;  if (newCapacity - *MAX\_ARRAY\_SIZE* > 0)  newCapacity = *hugeCapacity*(minCapacity);  // minCapacity is usually close to size, so this is a win:  elementData = Arrays.*copyOf*(elementData, newCapacity); }  private static int hugeCapacity(int minCapacity) {  if (minCapacity < 0) // overflow  throw new OutOfMemoryError();  return (minCapacity > *MAX\_ARRAY\_SIZE*) ? Integer.*MAX\_VALUE* :*MAX\_ARRAY\_SIZE*; } |

**size方法**

size 方法非常简单，它是直接返回 size 的值，也就是返回数组中元素的个数，时间复杂度为 O(1)。这里要注意一下，返回的并不是数组的实际大小。

|  |
| --- |
| public int size() {  return size; } |

**indexOf和lastIndexOf方法**

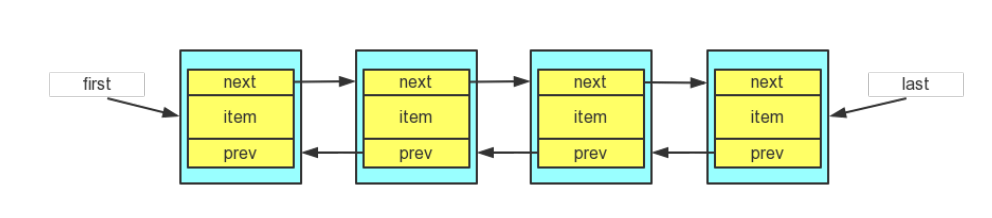
indexOf 方法的作用是返回第一个等于给定元素的值的下标。它是通过遍历比较数组中每个元素的值来查找的，所以它的时间复杂度是 O(n)。

lastIndexOf 的原理跟 indexOf 一样，而它仅仅是从后往前找起罢了。

|  |
| --- |
| public int indexOf(Object o) {  if (o == null) {  for (int i = 0; i < size; i++)  if (elementData[i]==null)  return i;  } else {  for (int i = 0; i < size; i++)  if (o.equals(elementData[i]))  return i;  }  return -1; }  public int lastIndexOf(Object o) {  if (o == null) {  for (int i = size-1; i >= 0; i--)  if (elementData[i]==null)  return i;  } else {  for (int i = size-1; i >= 0; i--)  if (o.equals(elementData[i]))  return i;  }  return -1; } |

#### LinkedList类

LinkedList：底层是一个双向链表，便于删除和添加，添加的元素可以为null，是线程不同步的。



双向链表每个结点除了数据域之外，还有一个前指针和后指针，分别指向前驱结点和后继结点（如果有前驱/后继的话）。另外，双向链表还有一个 first 指针，指向头节点，和 last 指针，指向尾节点。

**链表形式**

**单向链表**：每一个节点包含两部分，元素和指向下一个节点的地址值，其中最后一个节点只有元素。

**双向链表**：每一个节点包含三部分，元素和分别指向下一个节点和上一个节点的地址值，其中第一个节点没有指向上一个节点的地址值，最后一个节点没有指向下一个节点的地址值。

**环形链表**：每一个节点包含两部分，元素和指向下一个节点的地址值，其中最后一个节点含有指向第一个节点的地址值，因此形成闭环。

**LinkedList独有方法**

|  |  |
| --- | --- |
| **LinkedList类方法摘要** | |
| void | **addFirst**(E e)向头部添加一个元素 |
| void | **addLast**(E e)向尾部添加一个元素 |
| E | **getFirst**()获取第一个元素 |
| E | **getLast**()获取最后一个元素 |
| E | **removeFirst**()删除第一个元素，并返回该元素 |
| E | **removeLast**()删除最后一个元素，并返回该元素 |
| boolean | **offerFirst**(E e)向头部添加一个元素 |
| boolean | **offerLast**(E e)向尾部添加一个元素 |
| E | **peekFirst**()获取但不删除第一个元素，若集合中没有元素，会返回null |
| E | **peekLast**()获取但不删除最后一个元素，若集合中没有元素，会返回null |
| E | **pollFirst**()获取并且删除第一个元素，若集合中没有元素，会返回null |
| E | **pollLast**()获取并且删除最后一个元素，若集合中没有元素，会返回null |

**节点结构**

Node 是在 LinkedList 里定义的一个静态内部类，它表示链表每个节点的结构，包括一个数据域 item，一个后置指针 next，一个前置指针 prev。

|  |
| --- |
| private static class Node<E> {  E item;  Node<E> next;  Node<E> prev;  Node(Node<E> prev, E element, Node<E> next) {  this.item = element;  this.next = next;  this.prev = prev;  } } |

##### LinkedList属性

LinkedList属性非常少，只有三个。

|  |
| --- |
| //表示链表的节点个数 transient int size = 0; //指向头节点的指针 transient Node<E> first; //指向尾节点的指针 transient Node<E> last; |

##### LinkedList构造器

|  |
| --- |
| public LinkedList() { }  public LinkedList(Collection<? extends E> c) {  this();  addAll(c); } |

##### LinkedList方法

**表头位置添加元素**

对于链表这种数据结构来说，添加元素的操作无非就是在表头/表尾插入元素，又或者在指定位置插入元素。因为 LinkedList 有头指针和尾指针，所以在表头或表尾进行插入元素只需要 O(1) 的时间，而在指定位置插入元素则需要先遍历一下链表，所以复杂度为 O(n)。

当向表头插入一个节点时，很显然当前节点的前驱一定为 null，而后继结点是 first指针指向的节点，当然还要修改 first 指针指向新的头节点。除此之外，原来的头节点变成了第二个节点，所以还要修改原来头节点的前驱指针，使它指向表头节点。

|  |
| --- |
| private void linkFirst(E e) {  final Node<E> f = first;  //当前节点的前驱指向null，后继指针原来的头节点  final Node<E> newNode = new Node<>(null, e, f);  //头指针指向新的头节点  first = newNode;  //如果原来有头节点，则更新原来节点的前驱指针，否则更新尾指针  if (f == null)  last = newNode;  else  f.prev = newNode;  size++;  modCount++; } |

**表尾位置添加元素**

当向表尾插入一个节点时，很显然当前节点的后继一定为 null，而前驱结点是 last指针指向的节点，然后还要修改 last 指针指向新的尾节点。此外，还要修改原来尾节点的后继指针，使它指向新的尾节点。

|  |
| --- |
| void linkLast(E e) {  final Node<E> l = last;  //当前节点的前驱指向尾节点，后继指向null  final Node<E> newNode = new Node<>(l, e, null);  //尾指针指向新的尾节点  last = newNode;  //如果原来有尾节点，则更新原来节点的后继指针，否则更新头指针  if (l == null)  first = newNode;  else  l.next = newNode;  size++;  modCount++; } |

**指定位置添加元素**

当向指定节点之前插入一个节点时，当前节点的后继为指定节点，而前驱结点为指定节点的前驱节点。此外，还要修改前驱节点的后继为当前节点，以及后继节点的前驱为当前节点。

|  |
| --- |
| void linkBefore(E e, Node<E> succ) {  // assert succ != null;  //指定节点的前驱  final Node<E> pred = succ.prev;  //当前节点的前驱为指定节点的前驱，后继为指定的节点  final Node<E> newNode = new Node<>(pred, e, succ);  //更新指定节点的前驱为当前节点  succ.prev = newNode;  if (pred == null)  first = newNode;  else  pred.next = newNode;  size++;  modCount++; } |

**删除元素**

删除操作与添加操作大同小异，例如删除指定节点的过程如下图所示，需要把当前节点的前驱节点的后继修改为当前节点的后继，以及当前节点的后继结点的前驱修改为当前节点的前驱。

|  |
| --- |
| private E unlinkFirst(Node<E> f) {// 删除表头节点，返回表头元素的值  // assert f == first && f != null;  final E element = f.item;  final Node<E> next = f.next;  f.item = null;  f.next = null; // help GC  first = next; // 头指针指向后一个节点  if (next == null)  last = null;  else  next.prev = null; // 新头节点的前驱为null  size--;  modCount++;  return element; }  private E unlinkLast(Node<E> l) {// 删除表尾节点，返回表尾元素的值  // assert l == last && l != null;  final E element = l.item;  final Node<E> prev = l.prev;  l.item = null;  l.prev = null; // help GC  last = prev; // 尾指针指向前一个节点  if (prev == null)  first = null;  else  prev.next = null; // 新尾节点的后继为null  size--;  modCount++;  return element; }  E unlink(Node<E> x) { // 删除指定位置的节点，返回指定元素的值  // assert x != null;  final E element = x.item;  final Node<E> next = x.next; // 当前节点的后继  final Node<E> prev = x.prev; // 当前节点的前驱  if (prev == null) {  first = next;  } else {  prev.next = next; // 更新前驱节点的后继为当前节点的后继  x.prev = null;  }  if (next == null) {  last = prev;  } else {  next.prev = prev; // 更新后继节点的前驱为当前节点的前驱  x.next = null;  }  x.item = null;  size--;  modCount++;  return element; } |

**获取元素**

|  |
| --- |
| public E getFirst() { // 获取表头元素  final Node<E> f = first;  if (f == null)  throw new NoSuchElementException();  return f.item; }  public E getLast() { // 获取表尾元素  final Node<E> l = last;  if (l == null)  throw new NoSuchElementException();  return l.item; }  Node<E> node(int index) { // 获取指定下标的元素  // assert isElementIndex(index);  // 根据下标是否超过链表长度的一半，来选择从头部开始遍历还是从尾部开始遍历  if (index < (size >> 1)) {  Node<E> x = first;  for (int i = 0; i < index; i++)  x = x.next;  return x;  } else {  Node<E> x = last;  for (int i = size - 1; i > index; i--)  x = x.prev;  return x;  } } |

**常用public方法**

前面链表的添加和删除操作，那些方法都不是 public 的，LinkedList是在这些基础的方法进行操作的。

|  |
| --- |
| public E removeFirst() {//删除表头元素  final Node<E> f = first;  if (f == null)  throw new NoSuchElementException();  return unlinkFirst(f); }  public E removeLast() {//删除表尾元素  final Node<E> l = last;  if (l == null)  throw new NoSuchElementException();  return unlinkLast(l); }  public void addFirst(E e) {//插入新的表头节点  linkFirst(e); }  public void addLast(E e) {//插入新的表尾节点  linkLast(e); }  public int size() {//链表的大小  return size; }  public boolean add(E e) {//添加元素到表尾  linkLast(e);  return true; }  public boolean remove(Object o) {//删除指定元素  if (o == null) {  for (Node<E> x = first; x != null; x = x.next) {  if (x.item == null) {  unlink(x);  return true;  }  }  } else {  for (Node<E> x = first; x != null; x = x.next) {  if (o.equals(x.item)) {  unlink(x);  return true;  }  }  }  return false; }  public E get(int index) {//获取指定下标的元素  checkElementIndex(index); //先检查是否越界  return node(index).item; }  public E set(int index, E element) {//替换指定下标的值  checkElementIndex(index);  Node<E> x = node(index);  E oldVal = x.item;  x.item = element;  return oldVal; }  public void add(int index, E element) {//在指定位置插入节点  checkPositionIndex(index);   if (index == size)  linkLast(element);  else  linkBefore(element, node(index)); }  public E remove(int index) {//删除指定下标的节点  checkElementIndex(index);  return unlink(node(index)); }  public E peek() {//获取表头节点的值，表头为空返回 null  final Node<E> f = first;  return (f == null) ? null : f.item; }  public E element() {//获取表头节点的值，表头为空抛出异常  return getFirst(); }  public E poll() {//获取表头节点的值，并删除表头节点，表头为空返回 null  final Node<E> f = first;  return (f == null) ? null : unlinkFirst(f); }  public void push(E e) {//添加元素到表头  addFirst(e); }  public E pop() {//删除表头元素  return removeFirst(); } |

#### Vector类

古老的实现类（几乎不用）

Vector：底层是线程安全的动态数组（便于查询和修改），数组初始化为长度为10，如果容量满了，按照2倍扩容。除了支持foreach和Iterator遍历，还支持Enumeration迭代。

##### Vector属性

Vector比ArrayList多了一个属性capactityIncrement，这个属性是在扩容的时候用到的，它表示每次扩容只扩capacityIncrement个空间就足够了。该属性可以通过构造方法给它赋值。

|  |
| --- |
| protected int capacityIncrement; |

##### Vector构造器

从Vector构造器中可以看出Vector的默认大小也是10，而且其在初始化时就已经创建数组了，这点和ArrayList不一样。

|  |
| --- |
| public Vector(int initialCapacity, int capacityIncrement) {  super();  if (initialCapacity < 0)  throw new IllegalArgumentException("Illegal Capacity: "+  initialCapacity);  this.elementData = new Object[initialCapacity];  this.capacityIncrement = capacityIncrement; }  public Vector(int initialCapacity) {  this(initialCapacity, 0); }  public Vector() {  this(10); }  public Vector(Collection<? extends E> c) {  elementData = c.toArray();  elementCount = elementData.length;  // c.toArray might (incorrectly) not return Object[] (see 6260652)  if (elementData.getClass() != Object[].class)  elementData = Arrays.*copyOf*(elementData, elementCount, Object[].class); } |

##### Vector方法

从grow方法中我们可以发现，newCapacity默认情况下是两倍的oldCapacity，而当指定了 capacityIncrement的值之后，newCapacity 变成了oldCapacity+capacityIncrement。

|  |
| --- |
| private void grow(int minCapacity) {  // overflow-conscious code  int oldCapacity = elementData.length;  int newCapacity = oldCapacity + ((capacityIncrement > 0) ? capacityIncrement : oldCapacity);  if (newCapacity - minCapacity < 0)  newCapacity = minCapacity;  if (newCapacity - *MAX\_ARRAY\_SIZE* > 0)  newCapacity = *hugeCapacity*(minCapacity);  elementData = Arrays.*copyOf*(elementData, newCapacity); }  private static int hugeCapacity(int minCapacity) {  if (minCapacity < 0) // overflow  throw new OutOfMemoryError();  return (minCapacity > *MAX\_ARRAY\_SIZE*) ?  Integer.*MAX\_VALUE* :  *MAX\_ARRAY\_SIZE*; } |

ArrayList与Vector的区别？

底层都是数组，Vector默认**初始容量是10**，ArrayList看版本.

ArrayList扩容增加原来的50%，Vector扩容增加原来的1倍。

ArrayList线程不安全，效率高，Vector线程安全，效率低。

Vector因为版本古老，支持Enumeration 迭代器。但是该迭代器不支持快速失败。而Iterator和ListIterator迭代器支持快速失败。如果在迭代器创建后的任意时间从结构上修改了向量（通过迭代器自身的 remove 或 add 方法之外的任何其他方式），则迭代器将抛出 ConcurrentModificationException。因此，面对并发的修改，迭代器很快就完全失败，而不是冒着在将来不确定的时间任意发生不确定行为的风险。

### Set接口

**Set相当于是对Map的封装，其底层都是Map的对象，Set接口元素取出方式只有一种：迭代器。**

Set接口存储的元素无序不可重复（所以最多只能包含一个null），是Collection的子接口，**且没有提供额外的方法。**

**Set接口的无序性**：不是指的随机性，是指由调用对象的hashCode方法得到的哈希值来决定元素存储的位置（和添加顺序无关）。

**Set接口不可重复性**：调用对象的equals方法进行比较，返回true则说明两个对象相同不能存放，因此对于自定义的对象需要重写hashCode和equals方法。

|  |
| --- |
| **public** **class** SetTest {  @Test  **public** **void** initi() {  //Set集合是Collection的子接口，且没有提供额外的方法  Set<String> set = **new** HashSet<>();  set.add("aaa");  set.add("ccc");  set.add("ddd");  set.add("bbb");  //Set中的元素是不可重复的  set.add("aaa");  //Set中元素的顺序并不是添加元素的顺序，但也不是随机的  //其顺序是根据调用对象的hashCode方法得到的哈希值决定的  System.***out***.println(set);  }  } |
| [aaa, ccc, bbb, ddd] |

#### HashSet类

HashSet是Set接口的主要实现类，**底层就是HashMap的key集合（此时HashMap的value位置是常量）。**

**HashSet特点：**不能保证元素的排列顺序，不是线程安全的，集合中的元素可以是null。

**HashSet底层实现原理**：向HashSet中添加对象时，首先会调用该对象中的hashcode方法来计算当前对象的存储的位置，如果该位置上没有其他元素，则直接存放；如果该位置上已经存在其他元素，则调用该对象的equals方法与已存在元素进行内容比较，如果返回值是true，则不能存放；如果返回值是false，则以链表的形式存放在该位置上。因此如果向Set中添加自定义类的对象，那么该对象的类必须重写equals和hashCode方法。

对于HashSet集合，判断元素是否存在或删除元素，底层依据的是hashCode和equals方法。

如果两个元素的equals方法返回true，但是其hashCode方法返回值不同，依然可以添加到hashSet中并存放到不同位置。但是这种情况下不符合hashCode方法的重写原则（两个元素的equals方法返回true时，其hashCode方法返回值也应相等），需要修改重写后的hashCode方法。

|  |
| --- |
| **public** **class** HashSetTest {  @Test  **public** **void** method() {  HashSet<SameHashCode> hashSet = **new** HashSet<>();  hashSet.add(**new** SameHashCode(1,"aaa"));  hashSet.add(**new** SameHashCode(2,"bbb"));  hashSet.add(**new** SameHashCode(3,"ccc"));  hashSet.add(**new** SameHashCode(4,"ddd"));  //因为SameHashCode类的hashCode方法对不同对象调用的结果都一样，因此即使创建的对象不同，其哈希值也是相同的  //而且该类的equals方法返回值也一直是true，因此添加元素时只能添加第一个元素，后面的元素都被认为是重复的元素  System.***out***.println(hashSet);  HashSet<DifferentHashCode> hashSet2 = **new** HashSet<>();  hashSet2.add(**new** DifferentHashCode(1, "aaa"));  hashSet2.add(**new** DifferentHashCode(2, "bbb"));  hashSet2.add(**new** DifferentHashCode(3, "ccc"));  hashSet2.add(**new** DifferentHashCode(4, "ddd"));  System.***out***.println(hashSet2);  }  }  **class** SameHashCode{  String string = "SameHashCode";  **int** num = string.length();  SameHashCode(**int** num,String string){  **this**.string = string;  }  @Override  **public** **int** hashCode() {  **return** num;  }  @Override  **public** **boolean** equals(Object obj) {  **return** **true**;  }  @Override  **public** String toString() {  **return** num + string;  }  }  **class** DifferentHashCode{  String string = "DifferentHashCode";  **int** num = string.length();  DifferentHashCode(**int** num,String string){  **this**.num = num;  **this**.string = string;  }  @Override  **public** **int** hashCode() {  **final** **int** prime = 31;  **int** result = 1;  result = prime \* result + num;  result = prime \* result + ((string == **null**) ? 0 : string.hashCode());  **return** result;  }  @Override  **public** **boolean** equals(Object obj) {  **if** (**this** == obj)  **return** **true**;  **if** (obj == **null**)  **return** **false**;  **if** (getClass() != obj.getClass())  **return** **false**;  DifferentHashCode other = (DifferentHashCode) obj;  **if** (num != other.num)  **return** **false**;  **if** (string == **null**) {  **if** (other.string != **null**)  **return** **false**;  } **else** **if** (!string.equals(other.string))  **return** **false**;  **return** **true**;  }  @Override  **public** String toString() {  **return** num + string;  }  } |
| [12aaa]  [1aaa, 2bbb, 3ccc, 4ddd] |

##### HashSet属性

|  |
| --- |
| //底层使用 HashMap 来保存 HashSet 中所有元素  private transient HashMap<E,Object> map; //定义一个虚拟的 Object 对象作为 HashMap 的 value，将此对象定义为 static final。 private static final Object *PRESENT* = new Object(); |

##### HashSet构造器

|  |
| --- |
| //默认的无参构造器，构造一个空的HashSet，实际底层会初始化一个空的 HashMap，并使用默认初始容量为 16 //和加载因子 0.75。  public HashSet() {  map = new HashMap<>(); }  //构造一个包含指定 collection 中的元素的新set  //实际底层使用默认的加载因子0.75和足以包含指定collection中所有元素的初始容量来创建一个HashMap  public HashSet(Collection<? extends E> c) {  map = new HashMap<>(Math.*max*((int) (c.size()/.75f) + 1, 16));  addAll(c); }  //指定initialCapacity和loadFactor构造一个空HashSet，实际底层以相应的参数构造一个空的 HashMap  public HashSet(int initialCapacity, float loadFactor) {  map = new HashMap<>(initialCapacity, loadFactor); }  //指定的initialCapacity构造一个空的HashSet，实际底层以相应的参数及加载因子 loadFactor 为 0.75 //构造一个空的HashMap。  public HashSet(int initialCapacity) {  map = new HashMap<>(initialCapacity); }  //以指定的 initialCapacity 和 loadFactor 构造一个新的空链接哈希集合  //此构造函数为包访问权限，不对外公开，实际只是是对 LinkedHashSet 的支持  //实际底层会以指定的参数构造一个空 LinkedHashMap 实例来实现  HashSet(int initialCapacity, float loadFactor, boolean dummy) {  map = new LinkedHashMap<>(initialCapacity, loadFactor); } |

##### HashSet方法

|  |
| --- |
| //底层实际调用底层 HashMap 的 keySet 来返回所有的 key  public Iterator<E> iterator() {  return map.keySet().iterator(); }  //底层实际调用 HashMap 的 size()方法返回 Entry 的数量，就得到该 Set 中元素的个数  public int size() {  return map.size(); }  //底层实际调用 HashMap 的 isEmpty()判断该 HashSet 是否为空  public boolean isEmpty() {  return map.isEmpty(); }  //如果此 set 包含指定元素，则返回true，更确切地讲，当且仅当此 set 包含一个满足(o==null ? e==null : //o.equals(e)) 的 e 元素时，返回 true  //底层实际调用 HashMap 的 containsKey 判断是否包含指定 key  public boolean contains(Object o) {  return map.containsKey(o); }  /\*如果此 set 中尚未包含指定元素，则添加指定元素。更确切地讲，如果此 set 没有包含满足(e==null ? e2==null : e.equals(e2))的元素 e2，则向此 set 添加指定的元素 e。如果此 set 已包含该元素，则该调用不更改 set 并返回 false。  底层实际将将该元素作为 key 放入 HashMap。由于 HashMap 的 put()方法添加 key-value 对时，当新放入 HashMap 的 Entry 中 key与集合中原有 Entry 的 key 相同（hashCode()返回值相等，通过 equals 比较也返回 true），新添加的 Entry 的 value 会将覆盖原来 Entry 的 value，但 key 不会有任何改变，因此如果向 HashSet 中添加一个已经存在的元素时，新添加的集合元素将不会被放入 HashMap 中，原来的元素也不会有任何改变，这也就满足了 Set 中元素不重复的特性。\*/  public boolean add(E e) {  return map.put(e, *PRESENT*)==null; }  /\*如果指定元素存在于此 set 中，则将其移除。更确切地讲，如果此 set 包含一个满足(o==null ? e==null : o.equals(e))的元素e，则将其移除。如果此 set 已包含该元素，则返回 true（或者：如果此 set 因调用而发生更改，则返回 true）。（一旦调用返回，则此 set 不再包含该元素）。  底层实际调用 HashMap 的 remove 方法删除指定 Entry。\*/  public boolean remove(Object o) {  return map.remove(o)==*PRESENT*; }  //底层实际调用 HashMap 的 clear 方法清空 Entry 中所有元素  public void clear() {  map.clear(); }  //返回此 HashSet 实例的浅表副本：并没有复制这些元素本身  //底层实际调用 HashMap 的 clone()方法，获取 HashMap 的浅表副本，并设置到 HashSet 中  public Object clone() {  try {  HashSet<E> newSet = (HashSet<E>) super.clone();  newSet.map = (HashMap<E, Object>) map.clone();  return newSet;  } catch (CloneNotSupportedException e) {  throw new InternalError(e);  } } |

**深入HashSet添加移除元素**

**元素对象修改HashCode和equals方法前**

|  |
| --- |
| **public** **void** method() {  HashSet<Person> set = **new** HashSet<>();  Person p1 = **new** Person(1001,"AA");  Person p2 = **new** Person(1002,"BB");  set.add(p1);  set.add(p2);  System.***out***.println("0、"+set);  //此时没有重写hashCode方法，因此p1的哈希值不因属性的改变而改变  System.***out***.println(p1.hashCode());  p1.name = "CC";  System.***out***.println(p1.hashCode());  //因为p1哈希值不变，存放的位置也不变，remove方法根据哈希值来寻找要移除元素的存放位置，移除成功  set.remove(p1);  System.***out***.println("1、"+set);  //因为此时p1已被移除，HashSet中没有相同的元素，此时存放成功  set.add(**new** Person(1001,"CC"));  System.***out***.println("2、"+set);  set.add(**new** Person(1001,"AA"));  System.***out***.println("3、"+set);  } |
| 0、[i=1001, name=AA, i=1002, name=BB]  1383547042  1383547042  1、[i=1002, name=BB]  2、[i=1002, name=BB, i=1001, name=CC]  3、[i=1002, name=BB, i=1001, name=CC, i=1001, name=AA] |

**元素对象修改HashCode和equals方法后**

如果两个元素的equals方法返回true，但它们的hashCode方法返回值不相等，HashSet将会把它们存放在不同的位置，并添加成功。

元素添加成功后哈希值发生变化，其在HashSet中存放的位置也不会发生变化。

|  |
| --- |
| **public** **void** method() {  HashSet<Person> set = **new** HashSet<>();  Person p1 = **new** Person(1001,"AA");  Person p2 = **new** Person(1002,"BB");  set.add(p1);  set.add(p2);  System.***out***.println("0、"+set);  //此时重写了hashCode方法，因此p1的哈希值因属性的改变而改变  System.***out***.println(p1.hashCode());  p1.name = "CC";  System.***out***.println(p1.hashCode());  //因为p1哈希值变化，元素实际存放的位置不变，remove方法根据哈希值来寻找要移除元素的存放位置不同于原先的位置，移除失败  set.remove(p1);  System.***out***.println("1、"+set);  //因为此时p1未被移除，但是其仍位于原先位置，因此添加新元素时，该新元素的哈希值虽然和p1相同，但是寻找的位置上没有p1  //所以在存放位置上没有与其相同的元素，结果添加成功  set.add(**new** Person(1001,"CC"));  System.***out***.println("2、"+set);  //此时添加元素的位置就是p1所存放的位置，但是因为两者哈希值不同，equals方法的结果也为false，因此也添加成功  set.add(**new** Person(1001,"AA"));  System.***out***.println("3、"+set);  } |
| 0、[i=1002, name=BB, i=1001, name=AA]  34072  34136  1、[i=1002, name=BB, i=1001, name=CC]  2、[i=1002, name=BB, i=1001, name=CC, i=1001, name=CC]  3、[i=1002, name=BB, i=1001, name=CC, i=1001, name=CC, i=1001, name=AA] |

#### LinkedHashSet类

LinkedHashSet 是具有可预知迭代顺序的 Set 接口的哈希表和链接列表实现。此实现与HashSet 的不同之处在于，后者维护着一个运行于所有条目的双重链接列表。此链接列表定义了迭代顺序，该迭代顺序可为插入顺序或是访问顺序。注意，此实现不是同步的。如果多个线程同时访问链接的哈希 Set，而其中至少一个线程修改了该 Set，则它必须保持外部同步。

对于 LinkedHashSet 而言，它继承与 HashSet、又基于 LinkedHashMap 来实现的。

LinkedHashSet 底层使用 LinkedHashMap 来保存所有元素，它继承与 HashSet，其所有的方法操作上又与 HashSet 相同，因此 LinkedHashSet 的实现上非常简单，只提供了四个构造方法，并通过传递一个标识参数，调用父类的构造器，底层构造一个 LinkedHashMap来实现，在相关操作上与父类 HashSet 的操作相同，直接调用父类 HashSet 的方法即可。

LinkedHashSet继承了HashSet，底层实现和HashSet一样，但是LinkedHashSet可以按照添加元素的顺序进行遍历（并不是按照添加顺序存放元素），因为底层维护了一张链表用来记录添加的顺序。

LinkedHashSet插入性能略低于HashSet，但在遍历所有元素时效率更高。

|  |
| --- |
| **public** **class** LinkedHashSetTest {  @Test  **public** **void** method() {  LinkedHashSet<String> linkedHashSet = **new** LinkedHashSet<>();  linkedHashSet.add("ddd");  linkedHashSet.add("ccc");  linkedHashSet.add("bbb");  linkedHashSet.add("aaa");  //LinkedHashSet集合中的元素顺序和添加顺序一致  System.***out***.println(linkedHashSet);  }  } |
| [ddd, ccc, bbb, aaa] |

##### LinkedHashSet构造器

|  |
| --- |
| public LinkedHashSet(int initialCapacity, float loadFactor) {  super(initialCapacity, loadFactor, true); }  public LinkedHashSet(int initialCapacity) {  super(initialCapacity, .75f, true); }  public LinkedHashSet() {  super(16, .75f, true); }  public LinkedHashSet(Collection<? extends E> c) {  super(Math.*max*(2\*c.size(), 11), .75f, true);  addAll(c); } |

#### TreeSet类

TreeSet可以按照添加的对象的指定属性进行排序，其底层是红黑树。TreeSet添加的元素都必须是相同类型且有序的（无序的对象可以通过实现Comparable接口或Comparator接口来使对象变为有序的），TreeSet中添加的元素只有第一个元素不会调用比较器方法。

TreeSet不能添加null元素，否则抛出NullPointedException异常。

**比较器的实现**

通过实现Comparable接口并重写该接口中的compareTo方法

通过匿名内部类的形式实现创建Comparator匿名对象并重写compare方法

**\*比较器见Java常见类和接口之比较器**

**排序方式**

**自然排序**：

创建一个类并实现Comparable接口

重写compareTo方法

在compareTo方法中按照指定的属性进行排序

向集合中添加元素

|  |
| --- |
| **public** **class** TreeSetTest {  @Test  **public** **void** method() {  TreeSet<Person> treeSet = **new** TreeSet<>();  treeSet.add(**new** Person(16,"aaa"));  treeSet.add(**new** Person(19,"bbb"));  treeSet.add(**new** Person(18,"ccc"));  treeSet.add(**new** Person(17,"ddd"));  System.***out***.println(treeSet);  }  }  **class** Person **implements** Comparable<Person>{  **int** age;  String name;  **public** Person(**int** age,String name) {  **this**.age = age;  **this**.name = name;  }  @Override  /\*  \* compareTo方法的调用者是添加的新对象，传入的参数是Set集合中已存在的对象，且在确定Set集合中没有重复元素之后才会调用compareTo方法  \* 该方法并不是和Set集合中每一个元素都比较，而是比较至合适的位置就停止比较，即其上一个比较对象和下一个比较对象所得出的比较结果不同时  \* 返回的结果大于0 ，则调用对象位于比较对象后面；返回结果小于0，则调用对象位于比较对象前面；返回结果为0，则表示两者相同  \* 两者相同时，如果没有其他属性作为比较，则认为两者相同。在整个比较过程中，并没有调用hashCode和equals方法  \*/  **public** **int** compareTo(Person o) {  **if**(o **instanceof** Person) {  System.***out***.println(**this**.toString()+o.toString());  **return** **this**.age - o.age;  }  //返回值为0则认为两者是同一个对象或者对象类型不同无法添加  **return** 0;  }  @Override  **public** String toString() {  **return** name + age;  }  } |
| aaa16aaa16  bbb19aaa16  ccc18aaa16  ccc18bbb19  ddd17ccc18  ddd17aaa16  [aaa16, ddd17, ccc18, bbb19] |

**定制排序**：

创建一个Comparator接口的实现类的对象

重写compare方法

在compare方法中按照指定的属性进行排序

将Comparator接口实现类的对象作为实参传给TreeSet的构造器中

向集合中添加元素

|  |
| --- |
| **public** **class** TreeSetInnerTest {  @Test  **public** **void** method() {  Comparator<Animal> Comparator = **new** Comparator<Animal>() {  /\*@Override  public int compare(Object o1, Object o2) {  if(o1 instanceof Animal&& o2 instanceof Animal) {  System.out.println(o1+"used"+o2);  Animal animal1 = (Animal)o1;  Animal animal2 = (Animal)o2;  //先按名称排序，如果名称相同再按年龄排序  int temp = animal1.name.compareTo(animal2.name);  if(temp == 0) {  return animal1.age - animal2.age;  }  return temp;  }  return 0;  }\*/  **public** **int** compare(Animal animal1,Animal animal2) {  **int** temp = animal1.name.compareTo(animal2.name);  **if**(temp == 0) {  **return** animal1.age - animal2.age;  }  **return** temp;  }  };  //TreeSet的构造器中传入接口Comparator的实现对象  TreeSet<Animal> treeSet = **new** TreeSet<Animal>(Comparator);  treeSet.add(**new** Animal(16,"aaa"));  treeSet.add(**new** Animal(19,"bbb"));  treeSet.add(**new** Animal(15,"aaa"));  treeSet.add(**new** Animal(18,"ccc"));  treeSet.add(**new** Animal(19,"aaa"));  treeSet.add(**new** Animal(17,"ddd"));  treeSet.add(**new** Animal(19,"bbb"));  //此时没有报错也没有将元素添加，是因为添加元素时会调用比较器，其结果返回的是0，因此添加失败  // treeSet.add("aaaa");  System.***out***.println(treeSet);  //判断是否存在也是调用的比较器，如果返回0则表示已存在  System.***out***.println(treeSet.contains(**new** Animal(19, "aaa")));  }  }  **class** Animal {  String name;  **int** age;  Animal(**int** age,String name){  **this**.age = age;  **this**.name = name;  }  @Override  **public** String toString() {  **return** name + age;  }  @Override  **public** **int** hashCode() {  System.***out***.println("not used");  **final** **int** prime = 31;  **int** result = 1;  result = prime \* result + age;  result = prime \* result + ((name == **null**) ? 0 : name.hashCode());  **return** result;  }  @Override  **public** **boolean** equals(Object obj) {  System.***out***.println("not used too");  **if** (**this** == obj)  **return** **true**;  **if** (obj == **null**)  **return** **false**;  **if** (getClass() != obj.getClass())  **return** **false**;  Animal other = (Animal) obj;  **if** (age != other.age)  **return** **false**;  **if** (name == **null**) {  **if** (other.name != **null**)  **return** **false**;  } **else** **if** (!name.equals(other.name))  **return** **false**;  **return** **true**;  }  } |
| aaa16usedaaa16  bbb19usedaaa16  aaa15usedaaa16  ccc18usedaaa16  ccc18usedbbb19  aaa19usedaaa16  aaa19usedbbb19  ddd17usedaaa16  ddd17usedbbb19  ddd17usedccc18  bbb19usedaaa16  bbb19usedbbb19  [aaa15, aaa16, aaa19, bbb19, ccc18, ddd17]  aaa19usedaaa16  aaa19usedbbb19  aaa19usedaaa19  true |

自然排序和定制排序相比，定制排序优先级更高，且定制排序更能满足个性化的需求，根据不同对象TreeSet可以有不同的排序规则。实际开发中根据实际需求来选择排序方式。

|  |
| --- |
| **public** **class** ComparableAndComparator {  @Test  **public** **void** method() {  Comparator<Man> comparator = **new** Comparator<Man>() {  /\* public int compare(Object o1,Object o2) {  if(o1 instanceof Man && o2 instanceof Man) {  Man man1 = (Man)o1;  Man man2 = (Man)o2;  return man1.name.compareTo(man2.name);  }  return 0;  }\*/  **public** **int** compare(Man man1,Man man2) {  **return** man1.name.compareTo(man2.name);  }  };  TreeSet<Man> treeSet = **new** TreeSet<>(comparator);  treeSet.add(**new** Man(16,"aaa"));  treeSet.add(**new** Man(19,"bbb"));  treeSet.add(**new** Man(15,"aaa"));  treeSet.add(**new** Man(18,"ccc"));  treeSet.add(**new** Man(19,"aaa"));  treeSet.add(**new** Man(17,"ddd"));  treeSet.add(**new** Man(19,"bbb"));  //输出结果显式定制排序优先级更高  System.***out***.println(treeSet);  }  }  **class** Man **implements** Comparable<Man>{  String name;  **int** age;  Man(**int** age,String name){  **this**.age = age;  **this**.name = name;  }  @Override  /\* public int compareTo(Object o) {  if(o instanceof Man) {  return this.age - ((Man)o).age;  }  return 0;  }\*/  **public** **int** compareTo(Man man) {  **return** **this**.age - man.age;  }  @Override  **public** String toString() {  **return** **this**.name + **this**.age;  }  } |
| [aaa16, bbb19, ccc18, ddd17] |

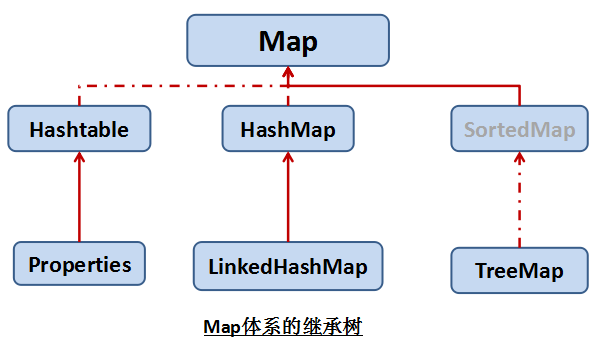
因为TreeSet中元素对象实现的Comparator或Comparable接口可以比较两个元素是否相同，替代了hashCode和equals方法的功能，导致TreeMap在调用各种方法时没有调用hashCode和equals方法，而是调用了compare或compareTo方法，因此TreeSet中存放的元素对象不需要重写hashCode和equals方法。

## Map接口

**|--Hashtable：**底层是哈希表数据结构，是线程同步的。不可以存储null键，null值。

**|--HashMap：**底层是哈希表数据结构，是线程不同步的。可以存储null键，null值。替代了Hashtable。

**|--TreeMap：**底层是二叉树结构，可以对map集合中的键进行指定顺序的排序。



Map和Collection是并列存在的，用于保存具有映射关系的数据键值对（Entry）（key-value），key和value可以是任何引用类型的数据。所有的key可以看成Set集合，其特点是无序且不可重复，其存放的key的数据（自定义对象）必须重写hashCode和equals方法；所有的value可以看成是Collection集合，集合了Set的无序特点和List的可重复的特点，其存放的value的数据（自定义对象）必须重写equals方法。

Map中的键值对可以看成是一个一个的Entry，是无序且不可重复的，在Map中的位置是由key的哈希值决定的。

**Map的遍历**

Map的遍历，不能支持foreach

（1）分开遍历：单独遍历所有key，还可以根据key获取对应value

单独遍历所有value

（2）成对遍历：遍历的是映射关系Map.Entry

Map.Entry是Map接口的内部接口。每一种Map内部有自己的Map.Entry的实现类

使用entrySet方法获取到Map中的键值对后，存放在Set中，可以通过迭代器或foreach循环获取其中的元素（即键值对）。由于获取到的元素是Object类型的对象，因此需要将获取到的元素向下转型，然后赋值给Map内的内部接口Entry，通过该接口提供的getKey()和getValue()方法可以分别获取到key和value。

for(Object object:map.entrySet()){

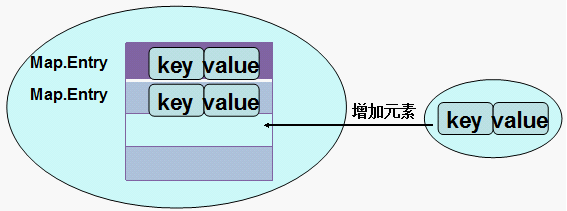
Map.Entry entry = (Entry)object;

entry.getKey();

entry.getValue();

}

在Map中存储数据，实际上是将Key---->value的数据存储在Map.Entry接口的实例中，再在Map集合中插入Map.Entry的实例化对象，如图示：



**Map接口提供的方法1**

|  |  |
| --- | --- |
| **Map接口方法摘要** | |
| V | **put**(K key,V value)向集合中添加键值对，如果key相同，后面的value会覆盖前面的value |
| V | **remove**(Object key)删除当前集合中键为key的那对元素 |
| void | **putAll**(Map<? extends K,? extends V> m)将Map集合m中的元素添加到调用该方法的Map中 |
| void | **clear**()清除调用该方法的Map中所有元素 |
| V | **get**(Object k)获取元素中key为k的value |
| int | **size**()获取集合中元素的个数 |
| boolean | **isEmpty**()当前集合是否为空 |
| boolean | **equals**(Object obj)比较两个集合中的内容，此时和两个集合中元素的添加顺序无关 |
| Set<K> | **KeySet**()获取当前集合中所有的key |
| Collection | **values**()获取当前集合中所有的value |
| Set<Map.Entry<K,V>> | **entyrSet**()获取当前集合中所有的entry（键值对） |
| boolean | **containsKey**(Object k)判断当前集合中是否包含key为k的元素 |
| boolean | **containsValue**(Object v)判断当前集合中是否包含value为v的元素 |

|  |
| --- |
| **public** **class** MapTest {  Map<String,Integer> map = **new** HashMap<>();  **public** **void** initi() {  map.put("eee", 200);  map.put("ccc", 200);  map.put("bbb", 700);  map.put("ddd", 300);  map.put("aaa", 100);  //添加时，如果key相同，则后添加的value会覆盖先添加的value  map.put("aaa", 200);  }  @Test  **public** **void** method() {  initi();  //Map是无序的，遍历顺序和添加顺序不一致，key无序不可重复，value可重复，且顺序与key一致  System.***out***.println(map);  //移除map中的指定key值的元素  map.remove("ccc");  System.***out***.println(map);  Map<String , Integer> map = **new** HashMap<>();  // map.put("fff", 100);  map.put("aaa", 300);  //将参数map中的元素全部添加到当前map中头部位置，且如果有相同key的元素则覆盖  map.putAll(**this**.map);  System.***out***.println(map);  //比较两个map是否相同，和其添加顺序和个数无关，只看存放的内容有关  System.***out***.println(map.equals(**this**.map));  //清除当前map中的所有元素  **this**.map.clear();  System.***out***.println(**this**.map);  //判断当前map是否为空  System.***out***.println(**this**.map.isEmpty());  //获取当前map中指定key的value值  System.***out***.println((**int**) map.get("ddd"));  //获取当前map中元素的个数  System.***out***.println(map.size());  //获取当前map中所有的key值并存放到一个set集合中  Set<String> set = map.keySet();  System.***out***.println(set);  //获取当前map中所有的value值并存放到一个collection集合中  Collection<Integer> collection = map.values();  System.***out***.println(collection);  //获取map中所有的键值对，并将其存放到set中  Set<Entry<String,Integer>> set2 = map.entrySet();  /\* for(Object object : set2) {  //遍历set中的元素并赋值给entry，Entry是Map内部的静态接口  Entry entry = (Entry)object;  System.out.print("key:"+entry.getKey()+" ");  System.out.println("value:"+entry.getValue());  }\*/  **for**(Entry<String, Integer> object : set2) {  //遍历set中的元素并赋值给entry，Entry是Map内部的静态接口  Entry<String , Integer> entry = object;  System.***out***.print("key:"+entry.getKey()+" ");  System.***out***.println("value:"+entry.getValue());  }  }  } |
| {aaa=200, ccc=200, bbb=700, eee=200, ddd=300}  {aaa=200, bbb=700, eee=200, ddd=300}  {aaa=200, bbb=700, eee=200, ddd=300}  true  {}  true  300  4  [aaa, bbb, eee, ddd]  [200, 700, 200, 300]  key:aaa value:200  key:bbb value:700  key:eee value:200  key:ddd value:300 |

根据containsKey和containsValue上述两个方法执行的过程来看，对于自定义的数据类型，其位于key中的数据所在的类必须重写hashCode和equals方法，value中的数据所在的类必须重写equals方法，否则无法得到预期中的结果。

|  |
| --- |
| **public** **class** MapContainsTest {  @Test  **public** **void** method() {  //没有重写hashCode方法和equals方法，很明显不是预期的结果  Map<A,A> map = **new** HashMap<>();  map.put(**new** A(10), **new** A(10));  map.put(**new** A(10), **new** A(10));  System.***out***.println(map);  Map<B,B> map2 = **new** HashMap<>();  //重写了hashCode方法和equals方法  //两者key相同，因此后添加的对象value将前一个对象覆盖了  map2.put(**new** B(10,1), **new** B(10,1));  map2.put(**new** B(10,2), **new** B(10,2));  System.***out***.println("----------");  System.***out***.println(map2);  //对于没有重写hashCode和equals方法的类A，就得不到预期中想要的结果  System.***out***.println(map.containsKey(**new** A(10)));  System.***out***.println(map.containsValue(**new** A(10)));  //判断map2中是否包含key的值为new B(10,0)的元素  //先调用了该对象的hashCode方法，结果相同，再调用equals方法，仍然相同才返回true  System.***out***.println(map2.containsKey(**new** B(10,0)));  //判断map2中是否包含value的值为new B(10,0)的元素  //调用了该对象的equals方法，相同则返回true  System.***out***.println(map2.containsValue(**new** B(10,0)));  }  }  **class** A{  **int** num;  A(**int** num){  **this**.num = num;  }  }  **class** B{  **int** num;  **int** num2;  B(**int** num,**int** num2){  **this**.num = num;  **this**.num2 = num2;  }  @Override  **public** **int** hashCode() {  System.***out***.println("override hashCode"+num2);  **final** **int** prime = 31;  **int** result = 1;  result = prime \* result + num;  **return** result;  }  @Override  **public** **boolean** equals(Object obj) {  System.***out***.println("override equals"+num2);  **if** (**this** == obj)  **return** **true**;  **if** (obj == **null**)  **return** **false**;  **if** (getClass() != obj.getClass())  **return** **false**;  B other = (B) obj;  **if** (num != other.num)  **return** **false**;  **return** **true**;  }  } |
| {exa.A@527740a2=exa.A@3108bc, exa.A@13a5fe33=exa.A@370736d9}  override hashCode1  override hashCode2  override equals2  ----------  override hashCode1  override hashCode2  {exa.B@29=exa.B@29}  false  false  override hashCode0  override equals0  true  override equals0  true |

### HashMap类

HashMap是Map的主要实现类，是线程不安全（不同步）的，**传入的key-value可以都为null。**HashMap的线程安全问题可以使用Collections的synchronizedMap(Map<K,V> m) 方法解决。

**HashMap的底层实现原理**

HashMap构造器先构造一个默认的初始长度为16的Node类型的数组，加载因子是0.75（加载因子：0.75\*16 = 12；进行添加元素时，一旦存放的元素个数超过12，就会进行扩容，可以理解为存放元素的个数一旦超过数组长度的75%，就会扩容为原来的2倍）。

添加元素(k1,v1)时，先调用k1的hashCode方法得到哈希值，根据哈希值算出该对象在数组中存放的位置。如果该位置上没有其他元素，则直接存放到该位置；如果已有其他元素(k2,v2)（说明k1和k2的哈希值相等），则调用k1的equals方法和k2比较，如果为true，则说明key相同，则v1覆盖v2；如果为false，则以链表形式将该对象存放在该位置上。

k1和k2在同一位置的情况很少出现，因为两个对象的哈希值相同的情形很少。

JDK7以前，HashMap底层是数组+链表；但是在JDK8以后，如果链表上的元素个数达到8，将链表改为红黑树，因此底层变为了数组+链表+红黑树。

Map 的这种实现方式为 get（取）和 put（存）带来了比较好的性能。但是如果涉及到大量的遍历操作的话，就尽量不要把 capacity 设置得太高（或 load factor 设置得太低），否则会严重降低遍历的效率。

影响 HashMap 性能的两个重要参数：“initial capacity”（初始化容量）和”load factor“（负载因子）。简单来说，容量就是哈希表桶的个数，负载因子就是键值对个数与哈希表长度的一个比值，当比值超过负载因子之后，HashMap 就会进行 rehash操作来进行扩容。

**为什么HashMap扩容要扩容2倍**

像15、31、63等这样的数值，它的二进制全部为1，保证可以取到容量范围内的所有索引值。

#### HashMap属性

|  |
| --- |
| //默认的初始容量为 16 static final int *DEFAULT\_INITIAL\_CAPACITY* = 1 << 4; //最大的容量上限为 2^30 static final int *MAXIMUM\_CAPACITY* = 1 << 30; //默认的负载因子为 0.75 static final float *DEFAULT\_LOAD\_FACTOR* = 0.75f; //变成树型结构的临界值为 8 static final int *TREEIFY\_THRESHOLD* = 8; //恢复链式结构的临界值为 6 static final int *UNTREEIFY\_THRESHOLD* = 6; //当哈希表的大小超过这个阈值，才会把链式结构转化成树型结构，否则仅采取扩容来尝试减少冲突 static final int *MIN\_TREEIFY\_CAPACITY* = 64; //哈希表 transient Node<K,V>[] table; //哈希表中键值对的个数 transient int size; //哈希表被修改的次数 transient int modCount; //它是通过 capacity\*load factor计算出来的，当size到达这个值时，就会进行扩容操作 int threshold; //负载因子 final float loadFactor; |

#### HashMap内部类Node

Node是 HashMap 中的一个静态内部类，哈希表中的每一个节点都是 Node 类型。我们可以看到，Node 类中有 4 个属性，其中除了 key 和value 之外，还有 hash 和 next 两个属性。hash 是用来存储 key 的哈希值的，next是在构建链表时用来指向后继节点的。

|  |
| --- |
| static class Node<K,V> implements Map.Entry<K,V> {  final int hash;  final K key;  V value;  Node<K,V> next;  Node(int hash, K key, V value, Node<K,V> next) {  this.hash = hash;  this.key = key;  this.value = value;  this.next = next;  }  public final K getKey() { return key; }  public final V getValue() { return value; }  public final String toString() { return key + "=" + value; }  public final int hashCode() {  return Objects.*hashCode*(key) ^ Objects.*hashCode*(value);  }  public final V setValue(V newValue) {  V oldValue = value;  value = newValue;  return oldValue;  }  public final boolean equals(Object o) {  if (o == this)  return true;  if (o instanceof Map.Entry) {  Map.Entry<?,?> e = (Map.Entry<?,?>)o;  if (Objects.*equals*(key, e.getKey()) &&  Objects.*equals*(value, e.getValue()))  return true;  }  return false;  } } |

#### HashMap方法

**get方法**

get方法实现步骤大致如下：

1、通过 hash 值获取该 key 映射到的桶（节点）。

2、桶上的 key 就是要查找的 key，则直接命中。

3、桶上的 key 不是要查找的 key，则查看后续节点：

（1）如果后续节点是树节点，通过调用树的方法查找该 key。

（2）如果后续节点是链式节点，则通过循环遍历链查找该key

|  |
| --- |
| public V get(Object key) {  Node<K,V> e;  return (e = getNode(hash(key), key)) == null ? null : e.value; } final Node<K,V> getNode(int hash, Object key) {  Node<K,V>[] tab; Node<K,V> first, e; int n; K k;  //如果哈希表不为空 && key 对应的桶上不为空  if ((tab = table) != null && (n = tab.length) > 0 &&  (first = tab[(n - 1) & hash]) != null) {  //是否直接命中  if (first.hash == hash && // always check first node  ((k = first.key) == key || (key != null && key.equals(k))))  return first;  //判断是否有后续节点  if ((e = first.next) != null) {  //如果当前的桶是采用红黑树处理冲突，则调用红黑树的get方法去获取节点  if (first instanceof TreeNode)  return ((TreeNode<K,V>)first).getTreeNode(hash, key);  //不是红黑树的话，那就是传统的链式结构了，通过循环的方法判断链中是否存在该 key  do {  if (e.hash == hash &&  ((k = e.key) == key || (key != null && key.equals(k))))  return e;  } while ((e = e.next) != null);  }  }  return null; } |

**put方法**

put 方法实现步骤大致如下：

1、先通过 hash 值计算出 key 映射到哪个桶。

2、如果桶上没有碰撞冲突，则直接插入。

3、如果出现碰撞冲突了，则需要处理冲突：

（1）如果该桶使用红黑树处理冲突，则调用红黑树的方法插入。

（2）否则采用传统的链式方法插入。如果链的长度到达临界值，则把链转变为红

黑树。

4、如果桶中存在重复的键，则为该键替换新值。

5、如果 size 大于阈值，则进行扩容。

|  |
| --- |
| public V put(K key, V value) {  return putVal(hash(key), key, value, false, true); }  final V putVal(int hash, K key, V value, boolean onlyIfAbsent,  boolean evict) {  Node<K, V>[] tab;  Node<K, V> p;  int n, i;  //如果哈希表为空，则先创建一个哈希表  if ((tab = table) == null || (n = tab.length) == 0)  n = (tab = resize()).length;  //如果当前桶没有碰撞冲突，则直接把键值对插入，完事  if ((p = tab[i = (n - 1) & hash]) == null)  tab[i] = newNode(hash, key, value, null);  else {  Node<K, V> e;  K k;  //如果桶上节点的 key 与当前 key 重复，那你就是我要找的节点了  if (p.hash == hash &&  ((k = p.key) == key || (key != null && key.equals(k))))  e = p;  //如果是采用红黑树的方式处理冲突，则通过红黑树的 putTreeVal 方法去插入这个键值对  else if (p instanceof TreeNode)  e = ((TreeNode<K, V>) p).putTreeVal(this, tab, hash, key, value);  //否则就是传统的链式结构  else {  //采用循环遍历的方式，判断链中是否有重复的 key  for (int binCount = 0; ; ++binCount) {  //到了链尾还没找到重复的 key，则说明 HashMap 没有包含该键  if ((e = p.next) == null) {  //创建一个新节点插入到尾部  p.next = newNode(hash, key, value, null);  //如果链的长度大于 TREEIFY\_THRESHOLD 这个临界值，则把链变为红黑树  if (binCount >= TREEIFY\_THRESHOLD - 1) // -1 for 1st  treeifyBin(tab, hash);  break;  }  //找到了重复的 key  if (e.hash == hash &&  ((k = e.key) == key || (key != null && key.equals(k))))  break;  p = e;  }  }  //这里表示在上面的操作中找到了重复的键，所以这里把该键的值替换为新值  if (e != null) { // existing mapping for key  V oldValue = e.value;  if (!onlyIfAbsent || oldValue == null)  e.value = value;  afterNodeAccess(e);  return oldValue;  }  }  ++modCount;  //判断是否需要进行扩容  if (++size > threshold)  resize();  afterNodeInsertion(evict);  return null; } |

**remove方法**

|  |
| --- |
| public V remove(Object key) {  Node<K,V> e;  return (e = removeNode(hash(key), key, null, false, true)) == null ?  null : e.value; } final Node<K,V> removeNode(int hash, Object key, Object value,  boolean matchValue, boolean movable) {  Node<K,V>[] tab; Node<K,V> p; int n, index;  //如果当前 key 映射到的桶不为空  if ((tab = table) != null && (n = tab.length) > 0 &&  (p = tab[index = (n - 1) & hash]) != null) {  Node<K,V> node = null, e; K k; V v;  //如果桶上的节点就是要找的 key，则直接命中  if (p.hash == hash && ((k = p.key) == key || (key != null && key.equals(k))))  node = p;  else if ((e = p.next) != null) {  //如果是以红黑树处理冲突，则构建一个树节点  if (p instanceof TreeNode)  node = ((TreeNode<K,V>)p).getTreeNode(hash, key);  //如果是以链式的方式处理冲突，则通过遍历链表来寻找节点  else {  do {  if (e.hash == hash &&  ((k = e.key) == key ||  (key != null && key.equals(k)))) {  node = e;  break;  }  p = e;  } while ((e = e.next) != null);  }  }  //比对找到的 key 的 value 跟要删除的是否匹配  if (node != null && (!matchValue || (v = node.value) == value ||  (value != null && value.equals(v)))) {  //通过调用红黑树的方法来删除节点  if (node instanceof TreeNode)  ((TreeNode<K,V>)node).removeTreeNode(this, tab, movable);  //使用链表的操作来删除节点  else if (node == p)  tab[index] = node.next;  else  p.next = node.next;  ++modCount;  --size;  afterNodeRemoval(node);  return node;  }  }  return null; } |

**hash方法**

在get 方法和put方法中都需要先计算 key映射到哪个桶上，然后才进行之后的操作，计算的主要代码如下：(n-1)&hash

上面代码中的 n 指的是哈希表的大小，hash 指的是 key 的哈希值，hash 是通过下面这个方法计算出来的，采用了二次哈希的方式，其中 key 的 hashCode 方法是一个native 方法：

|  |
| --- |
| static final int hash(Object key) {  int h;  return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16); } |

这个 hash 方法先通过 key 的 hashCode 方法获取一个哈希值，再拿这个哈希值与它的高 16 位的哈希值做一个异或操作来得到最后的哈希值，计算过程可以参考下图。为啥要这样做呢？注释中是这样解释的：如果当 n 很小，假设为 64 的话，那么 n-1即为 63（0x111111），这样的值跟 hashCode()直接做与操作，实际上只使用了哈希值的后 6 位。如果当哈希值的高位变化很大，低位变化很小，这样就很容易造成冲突了，所以这里把高低位都利用起来，从而解决了这个问题。

正是因为与的这个操作，决定了 HashMap 的大小只能是 2 的幂次方，想一想，如果不是2的幂次方，会发生什么事情？即使你在创建HashMap的时候指定了初始大小，HashMap 在构建的时候也会调用下面这个方法来调整大小：

|  |
| --- |
| static final int tableSizeFor(int cap) {  int n = cap - 1;  n |= n >>> 1;  n |= n >>> 2;  n |= n >>> 4;  n |= n >>> 8;  n |= n >>> 16;  return (n < 0) ? 1 : (n >= *MAXIMUM\_CAPACITY*) ? *MAXIMUM\_CAPACITY* : n + 1; } |

这个方法的作用看起来可能不是很直观，它的实际作用就是把 cap 变成第一个大于等于 2 的幂次方的数。例如，16 还是 16，13 就会调整为 16，17 就会调整为 32。

**resize方法**

HashMap 在进行扩容时，使用的 rehash 方式非常巧妙，因为每次扩容都是翻倍，与原来计算（n-1）&hash 的结果相比，只是多了一个 bit 位，所以节点要么就在原来的位置，要么就被分配到“原位置+旧容量”这个位置。

例如，原来的容量为 32，那么应该拿 hash 跟 31（0x11111）做与操作；在扩容扩到了 64 的容量之后，应该拿 hash 跟 63（0x111111）做与操作。新容量跟原来相比只是多了一个 bit 位，假设原来的位置在 23，那么当新增的那个 bit 位的计算结果为 0时，那么该节点还是在 23；相反，计算结果为 1 时，则该节点会被分配到 23+31 的桶上。

正是因为这样巧妙的 rehash 方式，保证了 rehash 之后每个桶上的节点数必定小于等于原来桶上的节点数，即保证了 rehash 之后不会出现更严重的冲突。

|  |
| --- |
| final Node<K,V>[] resize() {  Node<K,V>[] oldTab = table;  int oldCap = (oldTab == null) ? 0 : oldTab.length;  int oldThr = threshold;  int newCap, newThr = 0;  //计算扩容后的大小  if (oldCap > 0) {  //如果当前容量超过最大容量，则无法进行扩容  if (oldCap >= MAXIMUM\_CAPACITY) {  threshold = Integer.*MAX\_VALUE*;  return oldTab;  }  //没超过最大值则扩为原来的两倍  else if ((newCap = oldCap << 1) < MAXIMUM\_CAPACITY &&  oldCap >= DEFAULT\_INITIAL\_CAPACITY)  newThr = oldThr << 1; // double threshold  }  else if (oldThr > 0) // initial capacity was placed in threshold  newCap = oldThr;  else { // zero initial threshold signifies using defaults  newCap = DEFAULT\_INITIAL\_CAPACITY;  newThr = (int)(DEFAULT\_LOAD\_FACTOR \* DEFAULT\_INITIAL\_CAPACITY);  }  if (newThr == 0) {  float ft = (float)newCap \* loadFactor;  newThr = (newCap < MAXIMUM\_CAPACITY && ft < (float)MAXIMUM\_CAPACITY ?  (int)ft : Integer.*MAX\_VALUE*);  }  //新的 resize 阈值  threshold = newThr;  //创建新的哈希表  @SuppressWarnings({"rawtypes","unchecked"})  Node<K,V>[] newTab = (Node<K,V>[])new Node[newCap];  table = newTab;  if (oldTab != null) {  //遍历旧哈希表的每个桶，重新计算桶里元素的新位置  for (int j = 0; j < oldCap; ++j) {  Node<K,V> e;  if ((e = oldTab[j]) != null) {  oldTab[j] = null;  //如果桶上只有一个键值对，则直接插入  if (e.next == null)  newTab[e.hash & (newCap - 1)] = e;  //如果是通过红黑树来处理冲突的，则调用相关方法把树分离开  else if (e instanceof TreeNode)  ((TreeNode<K,V>)e).split(this, newTab, j, oldCap);  //如果采用链式处理冲突  else { // preserve order  Node<K,V> loHead = null, loTail = null;  Node<K,V> hiHead = null, hiTail = null;  Node<K,V> next;  //通过上面讲的方法来计算节点的新位置  do {  next = e.next;  if ((e.hash & oldCap) == 0) {  if (loTail == null)  loHead = e;  else  loTail.next = e;  loTail = e;  }  else {  if (hiTail == null)  hiHead = e;  else  hiTail.next = e;  hiTail = e;  }  } while ((e = next) != null);  if (loTail != null) {  loTail.next = null;  newTab[j] = loHead;  }  if (hiTail != null) {  hiTail.next = null;  newTab[j + oldCap] = hiHead;  }  }  }  }  }  return newTab; } |

在这里有一个需要注意的地方，有些文章指出当哈希表的 桶占用超过阈值时就进行扩容，这是不对的；实际上是当哈希表中的 键值对个数超过阈值时，才进行扩容的。

#### HampMap总结

HashMap中是通过红黑树的方式来处理哈希冲突。按照原来的拉链法来解决冲突，如果一个桶上的冲突很严重的话，是会导致哈希表的效率降低至 O（n），而通过红黑树的方式，可以把效率改进至 O（logn）。相比

链式结构的节点，树型结构的节点会占用比较多的空间，所以这是一种以空间换时间的改进方式。

**HashMap补充**

ArrayMap是Android SDK中的，用两个数组来模拟map，更少的内存占用空间，更高的效率。

WeakHashMap 的工作与正常的 HashMap 类似，但是使用弱引用作为 key，意思就是当 key 对象没有任何引用时，key/value 将会被回收。

### LinkedHashMap类

**LinkedHashSet的底层就是LinkedHashMap**。

LinkedHashMap的底层实现和HashMap是一样的，但是LinkedHashMap可以按照元素添加的顺序进行遍历，因为底层维护了一张链表，用来记录元素添加的顺序

从注释中，我们可以先了解到 LinkedHashMap 是通过哈希表和链表实现的，它通过维护一个链表来保证对哈希表迭代时的有序性，而这个有序是指键值对插入的顺序。另外，当向哈希表中重复插入某个键的时候，不会影响到原来的有序性。也就是说，假设你插入的键的顺序为 1、2、3、4，后来再次插入 2，迭代时的顺序还是 1、2、3、4，而不会因为后来插入的 2 变成 1、3、4、2。（但其实我们可以改变它的规则，使它变成 1、3、4、2）。

LinkedHashMap 的实现主要分两部分，一部分是哈希表，另外一部分是链表。哈希表部分继承了 HashMap，拥有了 HashMap 那一套高效的操作，所以我们要看的就是LinkedHashMap 中链表的部分，了解它是如何来维护有序性的。

#### LinkedHashMap属性

LinkedHashMap 是继承自 HashMap 的，所以它已经从 HashMap 那里继承了与哈希表相关的操作了，那么在 LinkedHashMap 中，它可以专注于链表实现的那部分，所以与链表实现相关的属性如下。

|  |
| --- |
| //LinkedHashMap 的链表节点继承了 HashMap 的节点，而且每个节点都包含 //了前指针和后指针，所以这里可以看出它是一个双向链表 static class Entry<K,V> extends HashMap.Node<K,V> {  Entry<K,V> before, after;  Entry(int hash, K key, V value, Node<K,V> next) {  super(hash, key, value, next);  } } transient LinkedHashMap.Entry<K,V> head;//头指针 transient LinkedHashMap.Entry<K,V> tail;//尾指针 //默认为 false。当为 true 时，表示链表中键值对的顺序与每个键的插入顺 //序一致，也就是说重复插入键，也会更新顺序 final boolean accessOrder; |

#### LinkedHashMap方法

LinkedHashMap实现了HashMap中的下列三个空方法，这三个方法表示的是在访问、插入、删除某个节点之后，进行一些处理，它们在 LinkedHashMap 都有各自的实现。LinkedHashMap 正是通过重写这三个方法来保证链表的插入、删除的有序性。

|  |
| --- |
| void afterNodeAccess(Node<K,V> p) { } void afterNodeInsertion(boolean evict) { } void afterNodeRemoval(Node<K,V> p) { } |

**afterNodeAccess方法**

这段代码的意思简洁明了，就是把当前节点 e 移至链表的尾部。因为使用的是双向链表，所以在尾部插入可以以 O（1）的时间复杂度来完成。并且只有当 accessOrder设置为 true 时，才会执行这个操作。在 HashMap 的 putVal 方法中，就调用了这个方法。

|  |
| --- |
| void afterNodeAccess(Node<K,V> e) { // move node to last  LinkedHashMap.Entry<K,V> last;  //当 accessOrder 的值为 true，且 e 不是尾节点  if (accessOrder && (last = tail) != e) {  LinkedHashMap.Entry<K,V> p =  (LinkedHashMap.Entry<K,V>)e, b = p.before, a = p.after;  p.after = null;  if (b == null)  head = a;  else  b.after = a;  if (a != null)  a.before = b;  else  last = b;  if (last == null)  head = p;  else {  p.before = last;  last.after = p;  }  tail = p;  ++modCount;  } } |

**afterNodeInsertion方法**

afterNodeInsertion 方法是在哈希表中插入了一个新节点时调用的，它会把链表的头节点删除掉，删除的方式是通过调用 HashMap 的 removeNode 方法。想一想，通过afterNodeInsertion 方法和 afterNodeAccess 方法，是不是就可以简单的实现一个基于最近最少使用（LRU）的淘汰策略了？当然，我们还要重写 removeEldestEntry 方法，因为它默认返回的是 false。

|  |
| --- |
| void afterNodeInsertion(boolean evict) { // possibly remove eldest  LinkedHashMap.Entry<K,V> first;  if (evict && (first = head) != null && removeEldestEntry(first)) {  K key = first.key;  removeNode(*hash*(key), key, null, false, true);  } } |

**afterNodeRemoval方法**

这个方法是当 HashMap 删除一个键值对时调用的，它会把在 HashMap 中删除的那个键值对一并从链表中删除，保证了哈希表和链表的一致性。

|  |
| --- |
| void afterNodeRemoval(Node<K,V> e) { // unlink  LinkedHashMap.Entry<K,V> p =  (LinkedHashMap.Entry<K,V>)e, b = p.before, a = p.after;  p.before = p.after = null;  if (b == null)  head = a;  else  b.after = a;  if (a == null)  tail = b;  else  a.before = b; } |

**get方法**

|  |
| --- |
| public V get(Object key) {  Node<K,V> e;  if ((e = getNode(*hash*(key), key)) == null)  return null;  if (accessOrder)  afterNodeAccess(e);  return e.value; } |

**LinkedHashMap补充**

PriorityQueue 是一个优先级队列,保证最高或者最低优先级的的元素总是在队列头部，但是 LinkedHashMap 维持的顺序是元素插入的顺序。当遍历一个 PriorityQueue 时，没有任何顺序保证，但是 LinkedHashMap 课保证遍历顺序是元素插入的顺序。

### HashTable类

HashTable（几乎不用）底层是数组+链表，线程安全的，**传入的元素key-value不可以为null。**

#### HashTable属性

|  |
| --- |
| //哈希表 private transient Entry<?,?>[] table; //记录哈希表中键值对的个数 private transient int count; //扩容的阈值 private int threshold; //负载因子 private float loadFactor; |

#### HashTable构造器

|  |
| --- |
| public Hashtable(int initialCapacity, float loadFactor) {  if (initialCapacity < 0)  throw new IllegalArgumentException("Illegal Capacity: "+  initialCapacity);  if (loadFactor <= 0 || Float.*isNaN*(loadFactor))  throw new IllegalArgumentException("Illegal Load: "+loadFactor);   if (initialCapacity==0)  initialCapacity = 1;  this.loadFactor = loadFactor;  table = new Entry<?,?>[initialCapacity];  threshold = (int)Math.*min*(initialCapacity \* loadFactor, *MAX\_ARRAY\_SIZE* + 1); }  public Hashtable(int initialCapacity) {  this(initialCapacity, 0.75f); }  public Hashtable() {  this(11, 0.75f); } |

#### HashTable方法

get方法

跟 HashMap 相比，Hashtable 的 get 方法非常简单。我们首先可以看见 get 方法使用了synchronized 来修饰，所以它能保证线程安全。并且它是通过链表的方式来处理冲突的。另外，我们还可以看见 HashTable 并没有像 HashMap 那样封装一个哈希函数，而是直接把哈希函数写在了方法中。而哈希函数也是比较简单的，它仅对哈希表的长度进行了取模。

|  |
| --- |
| public synchronized V get(Object key) {  Entry<?,?> tab[] = table;  int hash = key.hashCode();  //通过哈希函数，计算出 key 对应的桶的位置  int index = (hash & 0x7FFFFFFF) % tab.length;  //遍历该桶的所有元素，寻找该 key  for (Entry<?,?> e = tab[index] ; e != null ; e = e.next) {  if ((e.hash == hash) && e.key.equals(key)) {  return (V)e.value;  }  }  return null; } |

put方法

put 方法一开始就表明了不能有 null 值，否则就会向你抛出一个空指针异常。Hashtable的 put 方法也是使用 synchronized 来修饰。你可以发现，在 Hashtable 中，几乎所有的方法都使用了 synchronized 来保证线程安全。

|  |
| --- |
| public synchronized V put(K key, V value) {  // Make sure the value is not null  if (value == null) {  throw new NullPointerException();  }   // Makes sure the key is not already in the hashtable.  Entry<?,?> tab[] = table;  int hash = key.hashCode();  //计算桶的位置  int index = (hash & 0x7FFFFFFF) % tab.length;  @SuppressWarnings("unchecked")  Entry<K,V> entry = (Entry<K,V>)tab[index];  //遍历桶中的元素，判断是否存在相同的 key  for(; entry != null ; entry = entry.next) {  if ((entry.hash == hash) && entry.key.equals(key)) {  V old = entry.value;  entry.value = value;  return old;  }  }  //不存在相同的 key，则把该 key 插入到桶中  addEntry(hash, key, value, index);  return null; }  private void addEntry(int hash, K key, V value, int index) {  modCount++;  Entry<?,?> tab[] = table;  //哈希表的键值对个数达到了阈值，则进行扩容  if (count >= threshold) {  // Rehash the table if the threshold is exceeded  rehash();   tab = table;  hash = key.hashCode();  index = (hash & 0x7FFFFFFF) % tab.length;  }  // Creates the new entry.  @SuppressWarnings("unchecked")  Entry<K,V> e = (Entry<K,V>) tab[index];  //把新节点插入桶中（头插法）  tab[index] = new Entry<>(hash, key, value, e);  count++; } |

remove方法

|  |
| --- |
| public synchronized V remove(Object key) {  Entry<?,?> tab[] = table;  int hash = key.hashCode();  int index = (hash & 0x7FFFFFFF) % tab.length;  @SuppressWarnings("unchecked")  Entry<K,V> e = (Entry<K,V>)tab[index];  for(Entry<K,V> prev = null ; e != null ; prev = e, e = e.next) {  if ((e.hash == hash) && e.key.equals(key)) {  modCount++;  if (prev != null) {  prev.next = e.next;  } else {  tab[index] = e.next;  }  count--;  V oldValue = e.value;  e.value = null;  return oldValue;  }  }  return null; } |

rehash方法

Hashtable 的 rehash 方法相当于 HashMap 的 resize 方法。跟 HashMap 那种巧妙的 rehash方式相比，Hashtable 的 rehash 过程需要对每个键值对都重新计算哈希值，而比起异或和与操作，取模是一个非常耗时的操作，所以这也是导致效率较低的原因之一。

|  |
| --- |
| protected void rehash() {  int oldCapacity = table.length;  Entry<?,?>[] oldMap = table;  // overflow-conscious code  //扩容扩为原来的两倍+1  int newCapacity = (oldCapacity << 1) + 1;  //判断是否超过最大容量  if (newCapacity - MAX\_ARRAY\_SIZE > 0) {  if (oldCapacity == MAX\_ARRAY\_SIZE)  // Keep running with MAX\_ARRAY\_SIZE buckets  return;  newCapacity = MAX\_ARRAY\_SIZE;  }  Entry<?,?>[] newMap = new Entry<?,?>[newCapacity];  modCount++;  //计算下一次 rehash 的阈值  threshold = (int)Math.min(newCapacity \* loadFactor, MAX\_ARRAY\_SIZE + 1);  table = newMap;  //把旧哈希表的键值对重新哈希到新哈希表中去  for (int i = oldCapacity ; i-- > 0 ;) {  for (Entry<K,V> old = (Entry<K,V>)oldMap[i] ; old != null ; ) {  Entry<K,V> e = old;  old = old.next;   int index = (e.hash & 0x7FFFFFFF) % newCapacity;  e.next = (Entry<K,V>)newMap[index];  newMap[index] = e;  }  } } |

### Properties类

Properties类是HashTable的子类，其中的key和value都是String类型，一般用来读取配置文件。

**特点：1：可以持久化存储数据。2：键值都是字符串。3：一般用于配置文件。**

其实就是将读取流和指定文件相关联，并读取一行数据，因为数据是规则的key=value，所以获取一行后，通过 = 对该行数据进行切割，左边就是键，右边就是值，将键、值存储到properties集合中。

|  |  |
| --- | --- |
| **Properties类方法摘要** | |
| void | **load**(overload) 从流中读取配置信息（键和元素对） |
| void | **list**(overload)将此配置文件信息打印到指定的输出流 |
| void | **store**(overload)将集合中的数据存入properties文件中，刷新输出流 |
| String | **getProperty**(String key（,String value）)根据key或key-value的值获取数据 |
| Object | **setProperty**(String key,String value)将参数中的数据作为key-value存入 |
| Set<String> | **stringPropertyNames**()返回此属性列表中的一组key |

**读取配置文件中的内容**

|  |
| --- |
| **public** **class** PropertiesTest {  @Test  **public** **void** method() **throws** IOException {  //创建Properties对象  Properties properties = **new** Properties();  //创建一个输入流  FileInputStream fileInputStream = **new** FileInputStream(**new** File("person.properties"));  //加载流  properties.load(fileInputStream);  //读取文件内容  String name = properties.getProperty("name");  String password = properties.getProperty("password");  //设置key-value值  properties.setProperty("address", "China");  String address = properties.getProperty("address");  System.***out***.println(name+" "+ password+" "+address);  //关闭流，避免浪费资源  fileInputStream.close();  }  } |
| aaa 123321 China |

### TreeMap类

TreeSet的底层就是TreeMap，因此TreeMap的许多特点和TreeSet相同。TreeMap中的key元素也是有序的且不可重复的。Treemap的实现原理是红黑树。

排序方式：自然排序和定制排序（具体见TreeSet的排序方式，两者相同）。

HashMap的存入顺序和输出顺序无关，LinkedHashMap 则保留了键值对的存入顺序。TreeMap则是对Map中的元素进行排序。在实际的使用中我们也经常这样做：使用HashMap或者LinkedHashMap 来存放元素，当所有的元素都存放完成后，如果使用则是需要一个经过排序的Map的话，我们再使用TreeMap来重构原来的Map对象。这样做的好处是：因为HashMap和LinkedHashMap 存储数据的速度比直接使用TreeMap 要快，存取效率要高。当完成了所有的元素的存放后，我们再对整个的Map中的元素进行排序。这样可以提高整个程序的运行的效率，缩短执行时间。

这里需要注意的是，TreeMap中是根据键（Key）进行排序的。而如果我们要使用TreeMap来进行正常的排序的话，Key 中存放的对象必须实现Comparable 接口。

### 其他Map

**WeakHashMap**： WeakHashMap 是 Map 的一个特殊实现，它只用于存储对键的弱引用。当映射的某个键在 WeakHashMap 的外部不再被引用时，就允许垃圾收集器收集映射中相应的键值对。使用 WeakHashMap 有益于保持类似注册表的数据结构，其中条目的键不再能被任何线程访问时，此条目就没用了。

 **IdentifyHashMap**： Map的一种特性实现，关键属性的hash码不是由hashCode()方法计算，而是由System.identityHashCode 方法计算，使用==进行比较而不是equals()方法。

## Iterator接口

Iterator是一个接口，无法实例化，因此需要通过集合中的iterator方法获取其实现类的对象。Iterator对象称为迭代器（设计模式的一种），用于遍历集合中的元素，所有实现了Collection接口的集合类都有一个iterator()方法，用以返回一个实现了Iterator接口的对象。

Iterator接口中的方法：

|  |  |
| --- | --- |
| **Iterator接口方法摘要** | |
| boolean | **hasNext**()判断是否有更多的元素，有就返回true |
| E | **next**()指针下移，并获取指针指向的元素。 |
| default void | **remove**()从迭代器指向的 collection 中移除迭代器返回的最后一个元素（可选操作）。 |
| default void | **forEachRemaining**(Consumer<? super E> action) 执行给定的每个剩余元素的动作，直到所有的元素都被处理或操作抛出异常。 |

如果没有更多元素时再指针下移，就会发生NoSuchElementTxception异常

**Iterator使用过程中常见错误**：

不要使用不同的iterator迭代器对象调用hasNext()和next()方法；

迭代器都每次下移一位，不能上移，适合做遍历，不适合做增删，如果需要删除，可以使用iterator本身的remove方法，来删除当前的元素。

Iterator是工作在一个独立的线程中，并且拥有一个互斥（mutex） 锁。 Iterator被创建之后会建立一个指向原来对象的单链索引表，当原来的对象数量发生变化时，这个索引表的内容不会同步改变，所以当索引指针往后移动的时候就找不到要迭代的对象，所以按照 fail-fast 原则 Iterator 会马上抛出java.util.ConcurrentModificationException异常。

所以 Iterator 在工作的时候是不允许被迭代的对象被改变的。但你可以使用 Iterator 本身的方法remove()来删除对象，Iterator.remove() 方法会在删除当前迭代对象的同时维护索引的一致性。

不要用next()方法获取的元素来判断是否为空作为循环条件，再用next()方法进行操作元素。因为每调用一次next()方法，指针都会下移一次，这样相当于指针下移了两次，会抛NoSuchElementTxception异常。

|  |
| --- |
| **public** **class** IteratorTest {  Collection<String> collection = **new** ArrayList<>();  @Test  **public** **void** initi() {  //add(Object obj)向集合中添加元素  collection.add("aaa");  collection.add("bbb");  //此时1被自动装箱为Integer类型的对象  collection.add("ddd");  collection.add("fff");  Iterator<String> iterator = collection.iterator();  System.***out***.print("属性collection:");  **while**(iterator.hasNext()) {  System.***out***.print(iterator.next()+",");  }  /\*  \* 错误用法一  \*/  //此时每次循环都会返回一个新的迭代器判断是否有更过的元素，因此陷入了死循环  /\*  while(collection.iterator().hasNext()) {  collection.iterator().next();  System.out.println("loading...");  }  \*/  /\*  \* 错误用法二  \*/  //此时会报NoSuchElementException异常，因为next不仅会获取指针指向的元素，还会使指针下移  //因此每一次循环都会使指针下移2次，最后的结果只有一半，且会在最后抛异常  /\*  while(iterator.next()!=null) {  System.out.print(iterator.next()+",");  }  \*/  }  } |
| 属性collection:aaa,bbb,ddd,fff, |

**集合迭代注意问题：在迭代集合的过程中，不能对集合进行增删操作（会报并发访问异常）；可以用迭代器的方法进行操作（子类listIterator：有增删的方法）。**

### ListIterator接口

在进行list列表元素迭代的时候，如果想要在迭代过程中，想要对元素进行操作的时候，比如满足条件添加新元素。**会发生.ConcurrentModificationException并发修改异常。**

**导致的原因是：**

集合引用和迭代器引用在同时操作元素，通过集合获取到对应的迭代器后，在迭代中，进行集合引用的元素添加，迭代器并不知道，所以会出现异常情况。

**如何解决呢？**

既然是在迭代中对元素进行操作,找迭代器的方法最为合适.可是Iterator中只有hasNext,next,remove方法.通过查阅的它的子接口,**ListIterator**,发现该列表迭代器接口具备了对元素的增、删、改、查的动作。

**ListIterator是List集合特有的迭代器**。

ListIterator it = list.listIterator;//取代Iterator it = list.iterator;

add方法是将元素插入隐式光标的前面，因此添加新元素后使用previous方法会返回新元素，而调用next方法则不起作用，而是返回的添加操作前的下一个元素。

|  |  |
| --- | --- |
| **ListIterator接口方法摘要** | |
| void | [**add**](mk:@MSITStore:E:\java学习资料\手册\JDK_API_1_6_zh_CN.CHM::/java/util/ListIterator.html#add(E))([E](mk:@MSITStore:E:\java学习资料\手册\JDK_API_1_6_zh_CN.CHM::/java/util/ListIterator.html) e) 将指定的元素插入列表（可选操作）。 |
| boolean | [**hasNext**](mk:@MSITStore:E:\java学习资料\手册\JDK_API_1_6_zh_CN.CHM::/java/util/ListIterator.html#hasNext())() 以正向遍历列表时，如果列表迭代器有多个元素，则返回 true（换句话说，如果 next 返回一个元素而不是抛出异常，则返回 true）。 |
| boolean | [**hasPrevious**](mk:@MSITStore:E:\java学习资料\手册\JDK_API_1_6_zh_CN.CHM::/java/util/ListIterator.html#hasPrevious())() 如果以逆向遍历列表，列表迭代器有多个元素，则返回 true。 |
| [E](mk:@MSITStore:E:\java学习资料\手册\JDK_API_1_6_zh_CN.CHM::/java/util/ListIterator.html) | [**next**](mk:@MSITStore:E:\java学习资料\手册\JDK_API_1_6_zh_CN.CHM::/java/util/ListIterator.html#next())() 返回列表中的下一个元素。 |
| int | [**nextIndex**](mk:@MSITStore:E:\java学习资料\手册\JDK_API_1_6_zh_CN.CHM::/java/util/ListIterator.html#nextIndex())() 返回对 next 的后续调用所返回元素的索引。 |
| [E](mk:@MSITStore:E:\java学习资料\手册\JDK_API_1_6_zh_CN.CHM::/java/util/ListIterator.html) | [**previous**](mk:@MSITStore:E:\java学习资料\手册\JDK_API_1_6_zh_CN.CHM::/java/util/ListIterator.html#previous())() 返回列表中的前一个元素。 |
| int | [**previousIndex**](mk:@MSITStore:E:\java学习资料\手册\JDK_API_1_6_zh_CN.CHM::/java/util/ListIterator.html#previousIndex())() 返回对 previous 的后续调用所返回元素的索引。 |
| void | [**remove**](mk:@MSITStore:E:\java学习资料\手册\JDK_API_1_6_zh_CN.CHM::/java/util/ListIterator.html#remove())() 从列表中移除由 next 或 previous 返回的最后一个元素（可选操作）。 |
| void | [**set**](mk:@MSITStore:E:\java学习资料\手册\JDK_API_1_6_zh_CN.CHM::/java/util/ListIterator.html#set(E))([E](mk:@MSITStore:E:\java学习资料\手册\JDK_API_1_6_zh_CN.CHM::/java/util/ListIterator.html) e) 用指定元素替换 next 或 previous 返回的最后一个元素（可选操作）。 |

## Collections工具类

Collections是操作Set、List和Map等集合的工具类，提供了一系列静态方法对集合中的元素进行排序、查询和修改等操作。

**Collections工具类提供的方法**

|  |  |
| --- | --- |
| **Collections类方法摘要** | |
| static void | **reverse**(List list)反转指定list中的元素顺序 |
| static void | **shuffle**(List list)对list中的元素进行随机排序 |
| static void | **sort**(List list)对list中的元素根据自然顺序进行升序排序 |
| static void | **sort**(List list,Comparator c)根据c的规则对list中的元素进行排序 |
| static void | **swap**(List list,int i,int j)将list中的i位置和j位置的元素交换位置 |
| static Object | **max**(Collection c)根据自然顺序返回c的最大元素 |
| static Object | **max**(Collection c,Comparator com)根据比较器返回c的最大元素 |
| static Object | **min**(Collection c)根据自然顺序返回c的最小元素 |
| static Object | **min**(Collection c,Comparator com)根据比较器返回c的最小元素 |
| static int | **frequency**(Collection c,Object obj)返回c中指定元素出现的次数 |
| static void | **copy**(List dest,List src)将src中所有元素复制到dest，且dest中元素个数不小于src中的元素个数 |
| static boolean | **replaceAll**(List list,Object oldVal,Object newVal)将list中的所有的oldVal元素替换为newVal元素 |
| static List | **synchronizedList**(list)将非同步集合转成同步集合，原理同下 |
| static Map | **synchronizedMap**(map)将非同步集合转成同步集合，定义一个类，将集合所有方法加同一把锁后返回 |

|  |
| --- |
| **public** **class** CollectionsTest {  Comparator<String> comparator = **new** Comparator<String>() {  @Override  **public** **int** compare(String string1,String string2) {  **return** string1.length() - string2.length();  }  };  @Test  **public** **void** method() {  Collection<String> collection = **new** ArrayList<>();  collection.add("d");  collection.add("cccc");  collection.add("eee");  collection.add("bb");  collection.add("bb");  collection.add("bb");  collection.add("aaaaa");  collection.add("ggg");  System.***out***.println(collection);  //反转指定list中的元素顺序  Collections.*reverse*((List<String>)collection);  System.***out***.println("反转集合"+collection);  //对list中的元素进行随机排序  Collections.*shuffle*((List<String>) collection);  System.***out***.println("随机排序"+collection);  //对list中的元素根据自然顺序进行升序排序  Collections.*sort*((List<String>) collection);  System.***out***.println("自然排序"+collection);  //根据c的规则对list中的元素进行排序  Collections.*sort*((List<String>) collection, comparator);  System.***out***.println("定制排序"+collection);  //将list中的i位置和j位置的元素交换位置  Collections.*swap*((List<String>)collection, 0, collection.size()-1);  System.***out***.println("交换位置"+collection);  //根据自然顺序返回c的最大元素  System.***out***.println("自然最大"+Collections.*max*(collection));  //根据比较器返回c的最大元素  System.***out***.println("定制最大"+Collections.*max*(collection, comparator));  //根据自然顺序返回c的最小元素  System.***out***.println("自然最小"+Collections.*min*(collection));  //根据比较器返回c的最小元素  System.***out***.println("定制最小"+Collections.*min*(collection, comparator));  //返回c中指定元素出现的次数  System.***out***.println("出现次数"+Collections.*frequency*(collection, "bb"));  //将list中的所有的oldVal元素替换为newVal元素  System.***out***.println("以新换旧"+Collections.*replaceAll*((List<String>)collection, "bb", "success"));  System.***out***.println("新集合"+collection);  Collection<String> collection2 = **new** ArrayList<>();  collection2.add("111");  collection2.add("222");  collection2.add("333");  //将src中所有元素复制到dest，且dest中元素个数不小于src中的元素个数  //从0位置开始复制，并覆盖原List种的元素  Collections.*copy*((List<String>)collection, (List<String>)collection2);  System.***out***.println("复制到目标集合"+collection);  System.***out***.println("被复制集合"+collection2);  }  } |
| [d, cccc, eee, bb, bb, bb, aaaaa, ggg]  反转集合[ggg, aaaaa, bb, bb, bb, eee, cccc, d]  随机排序[eee, bb, d, aaaaa, bb, ggg, cccc, bb]  自然排序[aaaaa, bb, bb, bb, cccc, d, eee, ggg]  定制排序[d, bb, bb, bb, eee, ggg, cccc, aaaaa]  交换位置[aaaaa, bb, bb, bb, eee, ggg, cccc, d]  自然最大ggg  定制最大aaaaa  自然最小aaaaa  定制最小d  出现次数3  以新换旧true  新集合[aaaaa, success, success, success, eee, ggg, cccc, d]  复制到目标集合[111, 222, 333, success, eee, ggg, cccc, d]  被复制集合[111, 222, 333] |

## 集合总结

1、ArrayList 创建时的大小为 0；当加入第一个元素时，进行第一次扩容时，默认容量大小为 10。

2、ArrayList 每次扩容都以当前数组大小的 1.5 倍去扩容。

3、Vector 创建时的默认大小为 10。

4、Vector 每次扩容都以当前数组大小的 2 倍去扩容。当指定了 capacityIncrement 之后，每次扩容仅在原先基础上增加 capacityIncrement 个单位空间。

5、ArrayList 和 Vector 的 add、get、size 方法的复杂度都为 O(1)，remove 方法的复杂度为 O(n)。

6、ArrayList 是非线程安全的，Vector 是线程安全的。

1、LinkedList 的底层结构是一个带头/尾指针的双向链表，可以快速的对头/尾节点进行操作。

2、相比数组，链表的特点就是在指定位置插入和删除元素的效率较高，但是查找的效率就不如数组那么高了

### 重写方法

**List**：自定义元素对象需要重写equals方法，因为Collection中的indexOf、remove等方法的执行依赖于equals方法。如果没有重写Object中的equals方法，最后的结果可能不是需要的结果。

**HashSet**：**（底层是Map中的key元素集合）**自定义元素对象需要重写hashCode和equals方法，因为添加新元素对象时，会调用对象的hashCode方法获取其哈希值，根据其哈希值来判断应该要存放的位置；如果有两个元素的哈希值相同，还会调用其equals方法判断内容是否相同，如果相同则不允许存放，如果不同则将哈希值相同的元素对象以链表的形式存放在Set集合中。

**TreeSet**：**（底层是TreeMap中的key元素集合）**自定义元素对象要实现Comparable接口并重写compareTo方法（自然排序）或者通过匿名内部类的形式实现Comparator接口并重写compare方法（定制排序），因为TreeSet添加元素时会将元素对象按照指定的属性进行排序，所以其添加的元素必须要是同一类型且可排序的。

**HashMap**：存放在key位置的自定义元素对象必须重写hashCode和equals方法，因为其key位置的元素对象相当于一个Set集合；存放在value位置的自定义元素对象必须重写equals方法，因为其value位置的元素对象相当于一个Collection集合。

**TreeMap**：key位置的自定义元素对象要实现Comparable接口并重写compareTo方法（自然排序）或者通过匿名内部类的形式实现Comparator接口并重写compare方法（定制排序）。

### 元素区别

**List**中元素是有序（添加顺序和存放顺序相同）可重复的，元素对象需要重写，添加元素使用add方法。

**ArrayList**中元素是以数组形式存放的，元素可以为null，线程不安全的。

**LinkedList**中元素是以链表形式存放的，元素可以为null，线程不安全的。

**Set**中元素是无序（添加顺序和存放顺序不同）不可重复的，添加元素使用add方法。

**HashSet**中元素无序不可重复，元素可以为null。

**LinkedHashSet**中元素可以按添加顺序排序，元素可以为null。

**TreeSet**中元素无序，且存放时会按照比较器的规则排序，元素不可以为null。

**Map**中存放键值对key-value，添加元素使用put方法，可以根据key获取value的值。

**HashMap**中key无序不可重复，value存放顺序跟随key，，添加元素时key不同就直接存放；key相同，如果value不同，则后者value覆盖前者value，如果value相同，则不可以存放。key-value可以为null。线程不安全的。

**LinkedHashMap**中key元素可以按照添加顺序排序，value存放顺序跟随key，添加元素时key不同就直接存放；key相同，如果value不同，则后者value覆盖前者value，如果value相同，则不可以存放。key-value可以为null。

**HashTable**中元素和HashMap基本一样，key-value不可以为null。线程安全。

**Properties**中key和value都是String类型。key-value不可以为null，但是文件中可以有null，这是因为写入文件中的是String，该null实际上是”null”。

**TreeMap**中key无序，且存放时会按照比较器的规则排序，value存放顺序跟随key，添加元素时key不同就直接存放；key相同，如果value不同，则后者value覆盖前者value，如果value相同，则不可以存放。key-value不可以为null。

### 集合排序

使用有序集合，如 TreeSet 或 TreeMap

使用有顺序的集合，如 list，然后通过 Collections.sort() 来排序。

### List、Map、Set内存

Set 集合是最接近 Collection 的集合， 因此 Set 集合几乎没有在 Collection 增加什么方法。Set 集合代表了元素无序、元素不允许重复的集合（Set 只是在 Collection 规范上增加了元素不允许重复的约束） 。

List 集合则在 Collection 的基础上为元素增加了索引的特性，因此 List 集合代表了集合元素有序、集合元素可以重复的集合。

Map 则代表了存放 key-value 对的集合，程序可以通过 key 来获取其中的 value。

就 Set 集合来说，对于开发者而言，它的集合元素是无序的，似乎显得有些杂乱、无规律，但对计算机而言这不可能，因此计算机需要快速存、取 Set 集合中的元素。Set 集合有两个实现类：HashSet 与 TreeSet，其中 HashSet 底层其实使用了一个数组来存放所有集合元素，然后通过 Hash 算法来决定每个集合元素在底层数组中存放位置，因此 HashSet 对集合元素的存、 取就是 Hash 算法+数组存、 取——也就是说 HashSet 只比数组存、 取多了些 Hash算法开销，因此性能非常快。TreeSet 底层则完全是一个红黑树，因此红黑树是折衷平衡的排序二叉树，它底层没有数组开销，存、取元素时都是基于红黑树算法的，因此内存开销较小，但性能略差。

对于 List 集合而言，主要有两个实现：ArrayList 与 LinkedList，其中 ArrayList 底层是基于数组的，而且 ArrayList 存、取元素本身就是通过元素索引来进行的，因此 ArrayList 对元素的存、取性能非常好，几乎等同于存、取数组元素。但则添加、删除元素时需要对数组元素进行“整体搬家” ，因此添加、删除元素时性能较差。而 LinkedList 底层则是基于一个链表实现的，当从链表中存、取元素时，需要定位元素的位置，系统开销较大。但添加、删除元素时，只要修改元素的引用（相当于指针）即可，因此性能非常好。

对于 Map 集合而言，其底层存、取性能与 Set 集合完全一样。其实 Set 集合本身就是基于 Map 实现的——如果我们把 Map 集合的所有 value 都当成空对象处理、只考虑 Map 集合的 key，Map 集合就变成了 Set 集合。换个角度来看，如果我们向 Set 集合中添加的对象是key-value 所组成的 Entry 对象，那么 Set 集合也就变成了 Map 集合。

### HashSet和TreeSet元素不重复

Set 只是一个接口，它的不同实现类判断元素是否相等的标准是不同的。 笼统地说， Set 里的元素是不能重复的， 判断元素重复使用 equals()。而不是==。

对于 HashSet 而言，判断两个对象是否相等是通过 equals()和 hashCode()方法，只要两个对象通过 equals()比较返回 false、或两个对象的 hashCode()不相等，那么 HashSet 就会把它们当成不相同。

对于TreeSet而言， 判断两个对象相等的唯一标准是： 两个对象通过compareTo(Object obj)比较是否返回 0，与 equals()方法无关。只要两个对象通过 compareTo(Object obj)比较没有返回 0，Java 就会把它们当成两个对象处理。

### TreeSet存放继承关系对象

根据 TreeSet 底层的实现：TreeSet 底层的实现就是红黑树，因此当程序向 TreeSet 中添加集合元素时，程序会多次调用该对象的 compareTo()方法与 TreeSet 中的集合元素进行比较，直到找到该元素在红黑树中应当所在节点位置。因此该问题的答案是：当前正在添加父类对象就多次调用父类对象的 compareTo()方法；当前正在添加子类对象就多次调用子类对象的 compareTo()方法。

至于程序是否抛出异常， 则取决于compareTo()方法的实现， 如果子类在实现compareTo()方法时，试图把被比较对象转换为子类对象之后再进行比较——如果 TreeSet 集合中已经包括了父类对象，这就会引起 ClassCastException。

### 开发集合选择

先判断存储的类型（一组对象或一组键值对）

一组对象：Collection接口

允许重复：List

增删多：LinkedList（底层维护了一个双向链表）

改查多：ArrayList（底层维护了Object类型的可变数组）

不允许重复：Set

无序：HashSet（底层维护了一个哈希表，即数组+单链表）

有序：TreeSet（底层是TreeMap，即数组+链表+红黑树）

插入和取出顺序一致：LinkedHashSet

一组键值对：Map

键无序：HashMap（底层是哈希表，数组+链表+红黑树）

键有序：TreeMap

键插入和取出顺序一致：LinkedHashMap

### fail-fast

fail-fast即快速失败，一旦发生异常，直接停止并上报。Java中指的是Java集合的一种错误检测机制。当多个线程对部分集合进行结构上的改变的操作时，有可能会产生fail-fast机制，这个时候就会抛出ConcurrentModificationException（后文用CMException代替）。

在Java中， 如果在foreach 循环里对某些集合元素进行元素的 remove/add 操作的时候，就会触发fail-fast机制，进而抛出CMException。此处实际抛出异常的是foreach中封装的iterator抛出的异常。

final void checkForComodification() {  
 if (modCount != expectedModCount)  
 throw new ConcurrentModificationException();  
}

modCount是ArrayList中的一个成员变量，它表示该集合实际被修改的次数。expectedModCount 是 ArrayList中的一个内部类Itr中的成员变量，expectedModCount表示这个迭代器预期该集合被修改的次数，其值随着Itr被创建而初始化。只有通过迭代器对集合进行操作，该值才会改变。

private void fastRemove(int index) {  
 modCount++;  
 int numMoved = size - index - 1;  
 if (numMoved > 0)  
 System.*arraycopy*(elementData, index+1, elementData, index,  
 numMoved);  
 elementData[--size] = null; // clear to let GC do its work  
}

remove方法只修改了modCount，并没有对expectedModCount做任何操作。

总结一下，之所以会抛出CMException异常，是因为我们的代码中使用了增强for循环，而在增强for循环中，集合遍历是通过iterator进行的，但是元素的add/remove却是直接使用的集合类自己的方法。这就导致iterator在遍历的时候，会发现有一个元素在自己不知不觉的情况下就被删除/添加了，就会抛出一个异常，用来提示用户，可能发生了并发修改。

在使用Java的集合类的时候，如果发生CMException，优先考虑fail-fast有关的情况，实际上这里并没有真的发生并发，只是Iterator使用了fail-fast的保护机制，只要他发现有某一次修改是未经过自己进行的，那么就会抛出异常。

### fail-safe

为了避免触发fail-fast机制，导致异常，可以使用Java中提供的一些采用了fail-safe机制的集合类。fail-save的集合容器在遍历时不是直接在集合内容上访问的，而是先复制原有集合内容，在拷贝的集合上进行遍历。java.util.concurrent包下的容器都是fail-safe的，可以在多线程下并发使用，并发修改。同时也可以在foreach中进行add/remove 。

fail-safe集合的所有对集合的修改都是先拷贝一份副本，然后在副本集合上进行的，并不是直接对原集合进行修改。并且这些修改方法，如add/remove都是通过加锁来控制并发的。所以，CopyOnWriteArrayList中的迭代器在迭代的过程中不需要做fail-fast的并发检测。（因为fail-fast的主要目的就是识别并发，然后通过异常的方式通知用户）但是，虽然基于拷贝内容的优点是避免了ConcurrentModificationException，但同样地，迭代器并不能访问到修改后的内容。

List<String> userNames = new CopyOnWriteArrayList<String>() {{  
 add("Hollis");  
 add("hollis");  
 add("HollisChuang");  
 add("H");  
}};  
Iterator it = userNames.iterator();  
for (String userName : userNames) {  
 if (userName.equals("Hollis")) {  
 userNames.remove(userName);  
 }  
}  
System.*out*.println(userNames);  
while (it.hasNext()) {  
 System.*out*.println(it.next());  
}

输出结果：

[hollis, HollisChuang, H]  
Hollis  
hollis  
HollisChuang  
H

迭代器遍历的是开始遍历那一刻拿到的集合拷贝，在遍历期间原集合发生的修改迭代器是不知道的。

**Fail-fast和Fail-safe有什么区别**

Iterator的fail-fast属性与当前的集合共同起作用，因此它不会受到集合中任何改动的影响。Java.util包中的所有集合类都被设计为fail->fast的，而java.util.concurrent中的集合类都为fail-safe的。当检测到正在遍历的集合的结构被改变时，Fail-fast迭代器抛出ConcurrentModificationException，而fail-safe迭代器从不抛出ConcurrentModificationException。

**Fail-safe集合和Vector的区别**

CopyOnWrite容器即写时复制的容器。通俗的理解是当我们往一个容器添加元素的时候，不直接往当前容器添加，而是先将当前容器进行Copy，复制出一个新的容器，然后新的容器里添加元素，添加完元素之后，再将原容器的引用指向新的容器。

CopyOnWriteArrayList中add/remove等写方法是需要加锁的，目的是为了避免Copy出N个副本出来，导致并发写。但是，CopyOnWriteArrayList中的读方法是没有加锁的。这样做的好处是我们可以对CopyOnWrite容器进行并发的读，当然，这里读到的数据可能不是最新的。因为写时复制的思想是通过延时更新的策略来实现数据的最终一致性的，并非强一致性。

所以CopyOnWrite容器是一种读写分离的思想，读和写不同的容器。而Vector在读写的时候使用同一个容器，读写互斥，同时只能做一件事儿。

## 集合未知

poll()方法和remove()方法区别？

poll() 和 remove() 都是从队列中取出一个元素，但是 poll() 在获取元素失败的时候会返回空，但是 remove() 失败的时候会抛出异常。

# IO

**IO使用优化**

使用有缓冲的IO类,不要单独读取字节或字符

使用NIO和NIO 2或者AIO,而非BIO

在finally中关闭流

使用内存映射文件获取更快的IO

## File类

File类位于java.io包下，文件和目录的抽象表现形式，与平台无关。File能新建、删除、重命名文件和目录，但不能访问文件内容，如果需要访问文件内容，则需要使用输入输出流。File对象可以作为参数传递给流的构造器。

**File类构造器**

//表示一个文件

new File(String pathname); 该路径从工程名开始

(路径中**\\ == /**，Linux中只能用/，File的静态属性String separator存储了当前系统的路径分隔符)

new File(String parent,String child);

//表示一个目录

new File(“文件夹路径”);

**File类提供的方法**

|  |  |
| --- | --- |
| **File方法摘要-文件名相关** | |
| String | **getName**()获取名称 |
| String | **getPath**()根据构造器路径获取（相对/绝对）路径 |
| File | **getAbsoluteFile**()获取绝对路径的File类型的文件 |
| String | **getAbsolutePath**()获取绝对路径 |
| String | **getParent**()获取父目录 |
| Path | **toPath**()根据构造器路径获取文件路径，返回Path类型 |
| boolean | **renameTo**(File newName)修改文件名并移动文件 |
| **File方法摘要-文件检测相关** | |
| boolean | **exists**()判断是否存在 |
| boolean | **canWrite**()判断能否写入 |
| boolean | **canRead**()判断能否读取 |
| boolean | **isFile**()判断是否是文件 |
| boolean | **isAbsolute**()判断指向的路径是否是绝对路径 |
| boolean | **isHidden**()判断是否具有隐藏属性 |
| boolean | **isDirectory**()判断是否是目录/文件夹 |
| **File方法摘要=文件操作相关** | |
| boolean | **createNewFile**()创建文件，父目录必须存在，否则创建失败 |
| boolean | **delete**()删除文件或目录 |
| boolean | **deleteOnExit**()在退出虚拟机时删除 |
| **File方法摘要-目录操作相关** | |
| boolean | **mkdir**()创建目录，父目录必须存在，否则创建失败 |
| boolean | **mkdirs**()创建目录，若父目录不存在，则连父目录一起创建 |
| String[] | **list**(overload)获取当前目录所有目录和文件，可传入过滤器作为参数过滤 |
| File[] | **listFiles**(overload)获取当前目录所有目录和文件，可传入过滤器作为参数过滤 |
| static Fled[] | **listRoots**()获取计算机中所有的盘符 |
| **File方法摘要-获取常规信息** | |
| long | **length**()获取文件的大小，返回字节大小 |
| long | **lastModified**()获取最后修改的时间，返回毫秒数 |
| **File方法摘要-获取常量** | |
| static String | **pathSeparator**返回当前系统默认的路径分隔符，window为“；” |
| static String | **Separator**返回当前系统默认的名称分隔符，windows默认为 “\” |

|  |
| --- |
| **public** **class** FileAPITest {  File file = **new** File("files/fileAPI.txt");  File file4 = **new** File("files2/fileAPICopy.txt");  /\*  \* 文件名相关  \*/  @Test  **public** **void** method() **throws** Exception {  File file3 = **new** File("D:/Eclipse/workspace/A\_B\_example/files/fileAPI.txt");  File file2 = **new** File("D:\\Eclipse\\workspace\\A\_B\_example\\files\\fileAPI.txt");  //在文件路径名中，\\等价于/，但是/适用于windows和Linux，因此推荐使用/  System.***out***.println(file2.equals(file3));  file = **new** File("files/fileAPI.txt");  System.***out***.println(file.getName());  //getPath方法可以根据File类的构造器路径获取对应的（相对/绝对）路径  System.***out***.println(file.getPath());  System.***out***.println(file2.getPath());  //获取绝对路径的File对象，如果当前对象创建时是绝对路径，则调用equals方法比较两者相同  //如果当前对象创建时是相对路径，则调用equals方法比较两者结果为false。  File newFile = file2.getAbsoluteFile();  System.***out***.println(newFile.equals(file2));  //获取当前对象的绝对路径  System.***out***.println(file.getAbsolutePath());  //获取当前对象的父目录,如果当前对象创建时是绝对路径，则返回绝对父目录  //如果当前对象创建时是相对路径，则返回相对父目录  System.***out***.println("相对"+file.getParent());  System.***out***.println("绝对"+file2.getParent());  //修改文件名并移动文件    System.***out***.println(file.renameTo(file4));  }  /\*  \* 文件检测  \*/  @Test  **public** **void** method2() {  //判读文件是否存在  System.***out***.println(file.exists());  //判断文件是否可以写入  System.***out***.println(file.canWrite());  //判断文件是否可以读出  System.***out***.println(file.canRead());  //判断当前对象是否是文件  System.***out***.println(file.isFile());  //判断当前对象是否是目录  System.***out***.println(file.isDirectory());  }  /\*  \* 文件操作相关  \*/  @Test  **public** **void** method3() **throws** Exception {  //创建文件，父目录必须存在，否则创建失败  System.***out***.println(file.createNewFile());  //删除文件  System.***out***.println(file4.delete());  }  /\*  \* 目录操作相关  \*/  @Test  **public** **void** method4(){  File file2 = **new** File("files/dir/subdir");  //创建目录，若父目录不存在则创建失败  System.***out***.println(file2.mkdir());  //创建目录，若父目录不存在则连父目录一起创建  System.***out***.println(file2.mkdirs());  //删除目录  System.***out***.println(file2.delete());  File file3 = **new** File("files");  //获取当前目录所有目录和文件，返回String数组  String[] strings = file3.list();  **for** (String string : strings) {  System.***out***.println(string);  }  //获取当前目录所有目录和文件，返回File对象数组  File[] files = file3.listFiles();  **for** (File file : files) {  System.***out***.println(file.getAbsolutePath());  }  }  /\*  \* 获取常规信息  \*/@Test  **public** **void** method5() {  //获取最后修改的时间，返回毫秒数  System.***out***.println(file.lastModified());  //获取文件的大小，返回字节大小  System.***out***.println(file.length());  }  } |
| true  fileAPI.txt  files\fileAPI.txt  D:\Eclipse\workspace\A\_B\_example\files\fileAPI.txt  true  D:\Eclipse\workspace\A\_B\_example\files\fileAPI.txt  相对files  绝对D:\Eclipse\workspace\A\_B\_example\files  true  false  false  false  false  false  true  true  true  false  true  dir  fileAPI.txt  D:\Eclipse\workspace\A\_B\_example\files\dir  D:\Eclipse\workspace\A\_B\_example\files\fileAPI.txt  1556967507515  0 |

## IO流

### 简介

IO用于设备之间的数据传输，如读取文件、网络通讯等，input读取外部数据到程序中，output将程序中的数据输出。程序中打开的文件IO流不属于内存中的资源，垃圾回收器无法回收该资源，因此需要显式的关闭IO流资源。流的操作只有**读**和**写**。

#### 流的分类

1、按操作数据单位不同分为**字节流**（8bit）和**字符流**（16bit）

2、按数据流的流向不同分为**输入流**和**输出流**

3、按流的角色不同分为**节点流**和**处理流**

**字符流由来**

因为字符每个国家都不一样，所以涉及到了字符编码问题，那么GBK编码的中文用unicode编码解析是有问题的，所以需要获取中文字节数据的同时+ 指定的编码表才可以解析正确数据。为了方便于文字的解析，所以将**字节流和编码表封装成对象**，这个对象就是字符流。只要操作字符数据，优先考虑使用字符流体系。

**字节流和字符流的区别**

1. 字节流读取数据时，读到一个字节就返回一个字节；字符流使用字节流读到了一个或多个字节（中文对应2个字节，UTF-8中是3个字节），再去查询指定的编码集，最后将查到的字符返回。
2. 字节流可以处理所有类型的数据，例如图片、视频或音频等；字符流只能处理字符文本数据。

因此在处理纯文本数据时使用字符流，除此之外都要使用字节流。

#### Java中流的种类

**4个抽象基类**

InputStream & OutputStream字节输入流和字符输入流是所有输入流的基类

Reader & Writer字节输出流和字符输出流是所有输出流的基类

**4个节点流**

FileInputStream & FileOutputStream & FileReader & FileWriter

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **分类** | **字节输入流** | **字节输出流** | **字符输入流** | **字符输出流** |
| **抽象基类** | **InputStream** | **OutputStream** | **Reader** | **Writer** |
| **访问文件** | **FileInputStream** | **FileOutputStream** | **FileReader** | **FileWriter** |
| **访问数组** | ByteArrayInputStream | ByteArrayOutputStream | CharArrayReader | CharArrayWriter |
| **访问管道** | PipedInputStream | PipedOutputStream | PipedReader | PipedWriter |
| **访问字符串** |  |  | StringReader | StringWriter |
| **缓冲流** | **BufferedInputStream** | **BufferOutputStream** | **BufferReader** | **BufferWriter** |
| **转换流** |  |  | **InputStreamReader** | **OutputStreamWriter** |
| **对象流** | **ObjectInputStream** | **ObjectOutputStream** |  |  |
|  | FilterInputStream | FilterOutputStream | FilterReader | FilterWriter |
| **打印流** |  | PrintStream |  | PrintWriter |
| **推回输入流** | PushbackInputStream |  | PushbackReader |  |
| **特殊流** | DataInputStream | DataOutputStream |  |  |

#### 流的操作规律

**1，明确源和目的。**

数据源：就是需要读取，可以使用两个体系：InputStream、Reader；

数据目标：就是需要写入，可以使用两个体系：OutputStream、Writer；

**2，操作的数据是否是纯文本数据？**

如果是：数据源：Reader

数据目标：Writer

如果不是：数据源：InputStream

数据目标：OutputStream

**3，虽然确定了一个体系，但是该体系中有太多的对象，到底用哪个呢？**

明确操作的数据设备。

数据源对应的设备：硬盘(File)，内存(数组)，键盘(System.in)

数据目标对应的设备：硬盘(File)，内存(数组)，控制台(System.out)。

**4，需要在基本操作上附加其他功能吗？**比如缓冲。

如果需要就进行装饰。

#### 流的关闭顺序

如果有多个流同时使用，则流的关闭顺序应该遵照先使用后关闭的原则，则优先关闭最后使用的流，否则会发生java.io.IOException: Stream Closed流关闭异常。

### 抽象基类

Java的IO流共涉及40多个类，都是从InputStream & OutputStream & Reader & Writer这四个抽象基类派生的，都是这四个抽象基类的子类。

因为字符流直接以字符为操作单位，因此Writer可以用字符串来替代字符数组，即以String对象为参数。

一次读取一个字符，对于不同编码中的那些由2个字节或3个字节组成的字符可以一次读出，因此读取文本就需要使用字符流（Reader Writer），其他时候就用字节流

不同编码使用InputStream读取字节的到的结果不同，Unicode码一个汉字有三个字节，所以InputStream读取三次才能读到该汉字，如果将每次读到的字节转换为char就会乱码，

一般读取文本内容的时候使用字符流，字符流复制文件会根据读取的内容进行解码，写入的编码方式可能会不同，造成读入写出不一致，因此不建议进行文件的复制，因为可能会修改原来文件的编码格式，造成文件损坏。

一般进行文件（包括文本文件）复制的时候采用字节流，读入编码集和写出编码集不同，产生的文件大小内容都会不一样，因为不同字符所占用的字节数不一样。

#### InputStream

此抽象类是表示字节输入流的所有类的超类。字符流使用的数组是字符数组，而字节流使用的数组是字节数组。

#### OutputStream

此抽象类是表示输出字节流的所有类的超类。

#### Reader

用于读取字符流的抽象类。

|  |  |
| --- | --- |
| **Reader抽象类方法摘要** | |
| int | **read**()读取一个字符，并返回该字符，如果读到末尾，返回-1 |
| int | **read**(char[])将字符读入字符数组，返回读到的字符个数，如果读至末尾，返回-1 |
| abstract void | **close**()关闭流并释放与之相关的任何系统资源 |

无论是字节流还是字符流的read方法读取时，返回的都是int类型数据。这是因为read方法读到末尾时返回的是-1，而实际操作中很容易出现连续多个1的情况，而连续读到8个1，在计算机中就被理解为-1，从而导致读取会提前停止。因此将返回的类型设置为int类型，只保留字节的8位，并在剩余二进制位补0，具体操作是byte&255或者byte&0xff。

对于write方法，可以一次写入一个字节，但接收的是一个int类型的数值，因此只写入int雷尼格的最低一个字节，即最低8位。

简单说就是read方法对读到的数据进行提升，write对操作的数据进行转换。

#### Writer

用于写入字符流的抽象类。

|  |  |
| --- | --- |
| **Writer抽象类方法摘要** | |
| (abstract）void | **write**(overload)写入数据，写入的内容根据传入的参数不同而不同 |
| abstract void | **flush**()刷新流，将流中的数据刷新到目的地中 |
| abstract void | **close**()关闭流并释放与之相关的任何系统资源，关闭前会调用flush方法 |

### 节点流

在读取文件时，必须保证该文件已经存在，否则会报异常。使用该流读取或写入文件时，如果操作的文件过大，超过了JVM的内存空间，就会发生内存溢出。

#### FileInputStream

从文件系统中的某个文件中获得输入字节。哪些文件可用取决于主机环境。FileInputStream 用于读取诸如图像数据之类的原始字节流。要读取字符流，请考虑使用 FileReader。

文件输入流

read()读取一个字节并返回其int型数值，如果没有更多内容则返回-1

read(byte[] byte)将内容读取到byte数组并覆盖byte中原有内容（数组的大小会影响到读取内容的次数，因此一般会创建大小为1024的byte数组），并返回读取的字符的长度。

|  |
| --- |
| **public** **void** method() {  FileInputStream fileInputStream = **null**;  **try** {  fileInputStream = **new** FileInputStream(**new** File("./files/exam.txt"));  **byte**[] bs = **new** **byte**[1024];  **while**((fileInputStream.read(bs))!= -1) {  System.***out***.println(**new** String(bs, 0, bs.length));  }  } **catch** (IOException e) {  name();  }**finally** {  **try** {  **if**(fileInputStream != **null**) {  fileInputStream.close();  }  } **catch** (IOException e) {  e.printStackTrace();  }  }  } |
| hello world |

#### FileOutputStream

文件输出流

FileOutputStream(File file,boolean append)如果append是true，表示在原来的内容后面进行追加；如果append是false，表示覆盖原来的内容。

write()对象调用多个write方法输出内容，只要对象没有调用close方法，都表示一次输出。

写的文件可以不存在，系统会自动创建；读的文件必须存在，否则会 抛出FileNotFoundException。

|  |
| --- |
| **public** **void** method() {  FileOutputStream fileOutputStream = **null**;  **try** {  //构造器的boolean型参数为true的话表示将输出的内容添加到原有内容后  //构造器的boolean型参数为false的话表示将输出的内容覆盖原有内容  fileOutputStream = **new** FileOutputStream(**new** File("./files/exam.txt"),**false**);  //write方法只要没有执行close方法，就视为一次输出，并不会产生覆盖  fileOutputStream.write("everyday".getBytes());  fileOutputStream.write(" wish".getBytes());  fileOutputStream.write(" you good".getBytes());  } **catch** (IOException e) {  e.printStackTrace();  }**finally** {  **try** {  **if**(fileOutputStream != **null**) {  fileOutputStream.close();  }  } **catch** (IOException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  }  } |

**FileInputStream和FileOutputStream综合使用-边读边写**

|  |
| --- |
| **public** **void** method() {  FileOutputStream fos = **null**;  FileInputStream fis = **null**;  **try** {  File file = **new** File("files/FileInputAndOutputTest.txt");  fos = **new** FileOutputStream(**new** File("./files/anotherInOutCopy"));  fis = **new** FileInputStream(file);  **byte**[] b = **new** **byte**[1024];  **while**(fis.read(b) != -1) {  fos.write(b,0,b.length);  }  } **catch** (IOException e) {  e.printStackTrace();  }**finally** {  //如果有多个流需要关闭，则应该每个close方法分别用try-catch包裹  //避免因为前面的close执行异常导致后面的close方法没有执行  **try** {  fos.close();  } **catch** (IOException e) {  e.printStackTrace();  }  **try** {  fis.close();  } **catch** (IOException e) {  e.printStackTrace();  }  }  } |

#### FileReader

FileReader类继承了InputStreamReader类，该类没有特有的方法，只有构造器。在初始化时需要指定被读取文件，如果该文件不存在则会发生FileNotFoundException。

用来读取字符文件的便捷类。此类的构造方法假定默认字符编码和默认字节缓冲区大小都是适当的。要自己指定这些值，可以先在 FileInputStream 上构造一个 InputStreamReader。

一般只有在读取文本时使用字符流。

|  |
| --- |
| **public** **void** method() {  FileReader fr = **null**;  **try** {  fr = **new** FileReader(**new** File("files/originalFile.txt"));  **char**[] charArray = **new** **char**[1024];  **while**(fr.read(charArray) != -1) {  System.***out***.println(charArray);  }  } **catch** (Exception e) {  e.printStackTrace();  }**finally** {  **try** {  **if**(fr != **null**) {  fr.close();  }  } **catch** (IOException e) {  e.printStackTrace();  }  }  } |

#### FileWriter

FileWriter类继承了OutputStreamWriter类，该类没有特有的方法，只有构造函数，其特点在于：用于处理文本数据，有默认的编码表，有临时缓冲机制。

用来写入字符文件的便捷类。此类的构造方法假定默认字符编码和默认字节缓冲区大小都是可接受的。要自己指定这些值，可以先在 FileOutputStream 上构造一个 OutputStreamWriter。

一般不用字符输出流，即在进行文本复制这种情况下一般使用字节流，因为字符流进行文本复制可能会修改原来文件的编码格式。

如果想要在文件末尾处续写数据，就要在构造函数中传入新的参数true，否则会覆盖掉已存在的数据。

FileWriter fw = new FileWriter(new File("files/test.txt"));  
fw.write("hello world");  
//FileWriter写入字符后需要进行刷新操作，否则字符无法写入  
fw.flush();

**FileReader和FileWriter综合使用-边读边写**

|  |
| --- |
| **public** **void** method() {  FileReader fr = **null**;  FileWriter fw = **null**;  **try** {  fr = **new** FileReader(**new** File("files/originalFile.txt"));  fw = **new** FileWriter(**new** File("files/Test.txt"));  **char**[] charr = **new** **char**[1024];  **while**(fr.read(charr) != -1) {  fw.write(charr);  }  } **catch** (Exception e) {  }**finally** {  **try** {  **if**(fr != **null**)  fr.close();  } **catch** (Exception e2) {  }  **try** {  **if**(fw != **null**)  fw.close();  }**catch** (Exception e) {  }  }  } |

### 缓冲流

创建内部缓冲区数组用于增加读写速度，缓冲流要套接在相应的节点流上，对读写的数据提供了缓冲的功能，提高了读写效率，并增加了新的方法。

扩大byte数组的长度也可以提高效率，构造缓冲流对象时可以设置其大小，使用缓冲流写入时需要进行刷新，即调用flush方法，同时缓冲流的close方法中也调用了flush方法。

缓冲区的出现提高了对流的操作效率，其原理时将数组进行了封装，缓冲内部使用的就是流对象的方法，然后加入了数组对数据进行临时存储。缓冲是为了增强流的功能而存在，因此建立缓冲流时要先有其他流对象存在。

缓冲流的出现就是装饰设计模式的体现，对一组对象进行功能的增强。

#### BufferInputStream

该类实现缓冲的输入流。

#### BufferOutputStream

文件输出流是用于将数据写入 File 或 FileDescriptor 的输出流。

|  |
| --- |
| @Test  **public** **void** test() **throws** Exception{  copy("files/BufferInputOutputStream.txt",  "files/originalFile.txt");  }  **public** **void** copy(String desc,String src) **throws** Exception{  //创建节点流  FileInputStream fis = **new** FileInputStream(src);  FileOutputStream fos = **new** FileOutputStream(desc);  //创建缓冲流  BufferedInputStream bis = **new** BufferedInputStream(fis);  BufferedOutputStream bos = **new** BufferedOutputStream(fos);  //一边读一边写 : 用缓冲流读和写  **byte**[] b = **new** **byte**[1024];  **int** len = 0;  **while**((len = bis.read(b)) != -1){  //写内容  bos.write(b, 0, len);  }  //关流  bos.close();  bis.close();  fos.close();  fis.close();  } |

#### BufferReader

从字符输入流中读取文本，缓冲各个字符，从而实现字符、数组和行的高效读取。 可以指定缓冲区的大小，或者可使用默认的大小。大多数情况下，默认值就足够大了。

|  |  |
| --- | --- |
| **Writer抽象类方法摘要** | |
| String | **readLine**()一次读取一行，到行标记时，将标记之前的字符数据作为字符串返回，读到末尾时返回空 |

readLine方法的原理：该方法内部调用的还是缓冲流关联的流对象的read方法，但是每次读取到一个字符，将其进行临时存储，当读取到回车标记时，将临时存储的数据一次性返回。

#### BufferWriter

将文本写入字符输出流，缓冲各个字符，从而提供单个字符、数组和字符串的高效写入。

|  |  |
| --- | --- |
| **BufferWriter抽象类方法摘要** | |
| void | **newLine**()写入一个跨平台的换行符 |

|  |
| --- |
| **public** **void** method() {  BufferedReader br = **null**;  BufferedWriter bw = **null**;  **try** {  //创建缓冲流对象：它是过滤流，是对节点流的包装  br = **new** BufferedReader(**new** FileReader("files/originalFile.txt"));  bw = **new** BufferedWriter(**new** FileWriter("files/bufferWirter.txt"));  String string = **null**;  //一次读取字符文本文件的一行字符  **while**((string = br.readLine())!= **null**) {  //一次写入一行字符串  bw.write(string);  //写入行分隔符  bw.newLine();  }  //刷新缓冲区  bw.flush();  } **catch** (Exception e) {  // **TODO**: handle exception  }**finally** {  **try** {  **if**(br != **null**)  br.close();  } **catch** (Exception e2) {  }  **try** {  **if**(bw != **null**)  bw.close();  } **catch** (Exception e2) {  }  }  } |

### 转换流

转换流是字符流体系中的成员，是字节流和字符流之间的桥梁，该流对象可以对读取到的字节数据进行指定编码集的编码转换。当操作字节流和字符流的转换或操作的数据需要进行编码集的指定时，就可以使用转换流。

FileReader fr = new FileReader("a.txt");  
InputStreamReader isr = new InputStreamReader(new FileInputStream("a.txt"),"gbk");

以上两句代码功能一致，因为读取文本文件是如果不指定编码集，则使用的是系统默认的编码集，而在张国默认的编码集为gbk。如果a.txt文件中的字符数据是通过utf-8形式编码的，则需要在转换流指定编码集的位置将gbk修改为utf-8。

操作文件的字符流是转换流的子类，转换流已经完成了编码转换的操作，因此对于直接操作文本文件分FileReader而言就不用再重新定义了。注意使用FileReader操作文本数据时，使用的是默认的编码集，如果要使用指定编码集，就必须要使用转换流。

#### InputStreamReader

是字节流通向字符流的桥梁：它使用指定的 [charset](mk:@MSITStore:D:\shi\java学习资料\JDK_API_1_6_zh_CN.CHM::/java/nio/charset/Charset.html) 读取字节并将其解码为字符。它使用的字符集可以由名称指定或显式给定，或者可以接受平台默认的字符集。

#### OutputStreamWriter

是字符流通向字节流的桥梁：可使用指定的 [charset](mk:@MSITStore:D:\shi\java学习资料\JDK_API_1_6_zh_CN.CHM::/java/nio/charset/Charset.html) 将要写入流中的字符编码成字节。它使用的字符集可以由名称指定或显式给定，否则将接受平台默认的字符集。

可以将字节流和字符流通过转换流互相转换，将**读取时的字节流转成字符流**，将**写入时的字符流转成字节流**

可以将读取文件内容的编码集，在写入内容时，将写入的内容转成另一种编码集，在构造器中设置编码集。

**注意**：读取时设置的编码集要和读取文件的编码集一致。

|  |
| --- |
| **public** **void** testMyInput() **throws** Exception {  FileInputStream fis = **new** FileInputStream("files/originalFile.txt");  FileOutputStream fos = **new** FileOutputStream("files/ChangeIo.txt");  //因为读取的原文件编码格式为utf-8，因此这里的编码集因该设置为utf-8  InputStreamReader isr = **new** InputStreamReader(fis, "utf-8");  //这里可以将目标文件设置为任意编码集，然后以对应的编码集打开  OutputStreamWriter osw = **new** OutputStreamWriter(fos, "Unicode");  BufferedReader br = **new** BufferedReader(isr);  BufferedWriter bw = **new** BufferedWriter(osw);  String str = **null**;  **while** ((str = br.readLine()) != **null**) {  bw.write(str);  bw.newLine();  bw.flush();  }  bw.close();  br.close();  } |

### 管道流

PipedInputStream、PipedOutputStream、PipedReader、PipedWriter

管道读取流和管道写入流可以像管道一样对接上，管道读取流就可以读取管道写入流写入的数据。连接方式有两种，通过两个流对象的构造器或两个对象的connect方法。

两个管道流使用时需要加入多线程技术，让读写同时进行。因为单线程先执行read，会发生死锁，因为read方法是阻塞式的，没有数据的read方法会让线程等待。

public static void main(String[] args) throws IOException {  
 PipedInputStream pipin = new PipedInputStream();  
 PipedOutputStream pipout = new PipedOutputStream();  
 pipin.connect(pipout);  
 new Thread(new Input(pipin)).start();  
 new Thread(new Output(pipout)).start();  
}

### 对象流

用于存储和读取基本数据类型数据和对象的处理流，可以将Java对象写入到数据源中，也能把对象从数据源中还原回来。该对象必须要可序列化，即实现了Serializable接口。

#### ObjectInputStream

**反序列化**：用ObjectInputStream读取基本数据类型或对象的机制。

|  |  |
| --- | --- |
| **ObjectInputStream类方法摘要** | |
| Object | **readObject**()从ObjectInputStream读取一个对象 |

#### ObjectOutputStream

**序列化**：用ObjectOutputStream保存基本类型数据或对象的机制。

|  |  |
| --- | --- |
| **ObjectOutputStream类方法摘要** | |
| void | **writeObject**(Object obj)将指定的对象写入ObjectOutputStream，被写入的对象必须实现Serializable接口，否则会抛出NotSerializableException |

public void method() throws Exception {  
 ObjectOutputStream oos = new ObjectOutputStream(  
 new FileOutputStream("files/ObjectIOTest.txt"));  
 Person p1 = new Person("xmm",26);  
 oos.writeObject(p1);  
 oos.flush();  
 oos.close();  
}  
public void method2() throws Exception {  
 ObjectInputStream ois = new ObjectInputStream(  
 new FileInputStream("files/ObjectIOTest.txt"));  
 Person p2 = (Person)ois.readObject();  
 System.*out*.println(p2.toString());  
 ois.close();  
}  
class Person implements Serializable {  
 private static final long *serialVersionUID* = 10000L;  
 String name;  
 int age;  
 public Person(String name,int age) {  
 this.name = name;  
 this.age = age;  
 }  
}

#### PrintStream

字节打印流，System.out对应的类型就是PrintStream，其构造器可以接收字符串路径、File对象和OutputStream。

当目的是一个字节输出流时，可以在printStream对象上加入一个**true**参数。这样对于println方法可以进行自动的刷新，而不是等待缓冲区满了再刷新。最终print方法都将具体的数据转成字符串，而且都对IO异常进行了内部处理。因为操作的数据都转成了字符串，所以使用PrintWriter更好一些。

**特点**

1：提供了更多的功能，比如打印方法。可以直接打印任意类型的数据。

2：它有一个自动刷新机制，创建该对象，指定参数，对于指定方法可以自动刷新。

3：它使用的本机默认的字符编码。

4：该流的print方法不抛出IOException。

#### PrintWriter

字符打印流，用于将基本数据类型的数据格式转化为字符串输出。其构造器可以接收字符串路径、File对象、OutputStream和Writer，对于前两种可以指定编码集；对于后两种可以指定自动刷新，自动刷新值为true时只有println、printf和format三个方法可以使用。

如果想要自动刷新和指定编码集，可以使用下列方式：

PrintWriter pw = new PrintWriter(new OutputStreamWriter(  
 new FileOutputStream("a.txt"),"utf-8"),true);

中间还可以添加缓冲流提高效率，如下：

PrintWriter pw = new PrintWriter(new BufferedWriter(new OutputStreamWriter(  
 new FileOutputStream("a.txt"),"utf-8")),true);

提供了一系列重载的print和println方法，用于多种数据类型的输出。PrintStream和PrintWriter的输出不会抛出异常，拥有自动flush功能，System.out返回的是PrintStream的实例。

**标准输入输出流**

System.in和System.out分别代表了标准输入输出流，在jvm启动时已经存在，当jvm关闭，这两个流就关闭。如果使用close方法关闭时，这两个流提前结束了。

System.in的类型是InputStream，System.out的类型是PrintSream，其是OutputStream的子类和FilterOutputStream的子类

可以通过System类的setIn和setOut方法对默认设备进行改变：

void setIn(InputStream in);

void setOut(PrintStream out);

|  |
| --- |
| **public** **void** method() {  PrintStream ps = **null**;  **try** {  FileOutputStream fos = **new** FileOutputStream(**new** File("files/PrintIOPTT.txt"));  // 创建打印输出流,设置为自动刷新模式(写入换行符或字节 '\n' 时都会刷新输出缓冲区)  ps = **new** PrintStream(fos, **true**);  **if** (ps != **null**) {// 把标准输出流(控制台输出)改成文件  System.*setOut*(ps);  }  **for** (**int** i = 1; i <= 255; i++) { // 输出ASCII字符  System.***out***.print(i+"\t");  **if** (i % 51 == 0 ) { // 每50个数据一行  System.***out***.println(); // 换行  }  }  } **catch** (FileNotFoundException e) {  e.printStackTrace();  } **finally** {  **if** (ps != **null**) {  ps.close();  }  }  } |
| 1 2 3 4 5 6 7 8 9 10  11 12 13 14 15 16 17 18 19 20  21 22 23 24 25 26 27 28 29 30  31 32 33 34 35 36 37 38 39 40  41 42 43 44 45 46 47 48 49 50 |

### 数据流

为了方便的操作Java中的基本数据类型，可以使用数据流。数据流有两个类DataInputStream和DataOutputStream，使用时分别套接在InputStream和OutputStream节点流上。

#### DataInputStream

|  |  |
| --- | --- |
| **DataInputStream类方法摘要** | |
| Xxx | **readXxx**()一次读取对应数据类型的字节个数，并将其转换为该基本数据类型 |
| String | **readUTF**()按照UTF-8修改版读取字符，只能读取writeUTF()写入的字符数据 |
| void | **readFully**(byte[] b) |

#### DataOutputStream

|  |  |
| --- | --- |
| **DataOutputStream类方法摘要** | |
| Xxx | **writeXxx**()一次写入对应数据类型的字节个数，并将其转换为该基本数据类型 |
|  | **writeUTF**()按照UTF-8修改版写入字符 |

### 随机存取文件流

#### RandomAccessFile

该类不属于流体系，但是其内部封装了字节流和一个缓冲区（字节数组），通过内部指针来操作数组中的数据。该对象可以读写数据，写数据时如果该位置已有数据，则会发生数据覆盖（修改）。在使用该对象时，最好数据都是有规则的，或者是分段的。

该类实例化时如果操作的文件不存在就会自动建立；如果存在的文件有数据，在没有指定指针位置的情况下写入数据就会将文件开头的数据覆盖。

**该类可以用于多线程的下载，即通过多线程往一个文件中同时存储数据。**

**RandomAccessFile特点**

1. 只能操作文件，因此其构造器接收两种类型的参数：字符串路径和File对象，内部封装了字节读取流和字节写入流。
2. 该对象既可以对文件进行读取，也可以对文件进行写入。可以读取文件的部分内容，也可以向写入部分内容。因此在进行对象实例化时，必须要指定该对象的操作模式，如r、rw等。
3. 该对象中定义了一个大型的byte数组，通过指针来操作这个数组，通过getFilePointer获取指针的位置，通过seek方法设置指针的位置。

|  |  |
| --- | --- |
| **RandomAccessFile类方法摘要** | |
| int | **skipBytes**(int n)跳过指定的字节数 |
| void | **seek**(long pos)指定指针的位置 |
| long | **getFilePointer**()获取指针的位置 |

//随机写入数据，可以实现已有数据的修改。  
private static void randomWrite() throws IOException {  
 RandomAccessFile raf = new RandomAccessFile("random.txt", "rw");  
 raf.seek(8 \* 4);  
 System.*out*.println("pos :" + raf.getFilePointer());  
 raf.write("xmm".getBytes());  
 raf.writeInt(102);  
 raf.close();  
}  
private static void read() throws IOException {  
 RandomAccessFile raf = new RandomAccessFile("random.txt", "r");//只读模式。  
 //指定指针的位置。  
 raf.seek(8);//实现随机读取文件中的数据。注意：数据最好有规律。  
 System.*out*.println("pos1 :" + raf.getFilePointer());  
 byte[] buf = new byte[4];  
 raf.read(buf);  
 String name = new String(buf);  
 int age = raf.readInt();  
 System.*out*.println(name + "::" + age);  
 System.*out*.println("pos2 :" + raf.getFilePointer());  
 raf.close();  
}  
private static void write() throws IOException {  
 //rw：当这个文件不存在，会创建该文件。当文件已存在，不会创建。所以不会像输出流一样覆盖。  
 RandomAccessFile raf = new RandomAccessFile("random.txt", "rw");//rw读写模式  
 //往文件中写入人的基本信息，姓名，年龄。  
 raf.write("cap".getBytes());  
 raf.writeInt(97);  
 raf.close();  
}

#### 访问模式

"r" 以只读方式打开。调用结果对象的任何 write 方法都将导致抛出 IOException。

"rw" 打开以便读取和写入。如果该文件尚不存在，则尝试创建该文件。

"rws" 打开以便读取和写入，对于 "rw"，还要求对文件的内容或元数据的每个更新都同步写入到底层存储设备。

"rwd" 打开以便读取和写入，对于 "rw"，还要求对文件内容的每个更新都同步写入到底层存储设备。

### 序列流

#### SequenceInputStream

该对象可以将多个读取流合并成一个流，便于操作。其原理就是将每一个读取流对象存储到一个集合中，最后一个流对象结尾作为这个流的结尾。

它从输入流的有序集合开始，并从第一个输入流开始读取，直到到达文件末尾，接着从第二个输入流读取，依次类推，直到到达包含的最后一个输入流的文件末尾为止。

该对象的构造函数参数是枚举，想要获取枚举，需要有Vector集合，但不高效。需用ArrayList，但ArrayList中没有枚举，只有自己去创建枚举对象。因为枚举操作的是具体集合中的元素，所以无法具体实现，但是枚举和迭代器功能相同，所以可用迭代替代枚举。

**合并原理：**多个读取流对应一个输出流。

**切割原理：**一个读取流对应多个输出流。

public class SplitFileDemo {  
 private static final String *CFG* = ".properties";  
 private static final String *SP* = ".part";  
  
 public static void main(String[] args) throws IOException {  
 File file = new File("c:\\0.bmp");  
 File dir = new File("c:\\partfiles");  
 *meger*(dir);  
 }  
 //数据的合并。  
 private static void meger(File dir) throws IOException {  
 if (!(dir.exists() && dir.isDirectory()))  
 throw new RuntimeException("指定的目录不存在，或者不是正确的目录");  
 File[] files = dir.listFiles(new SuffixFilter(*CFG*));  
 if (files.length == 0)  
 throw new RuntimeException("扩展名.proerpties的文件不存在");  
 //获取到配置文件  
 File config = files[0];  
 //获取配置文件的信息。  
 Properties prop = new Properties();  
 FileInputStream fis = new FileInputStream(config);  
 prop.load(fis);  
 String fileName = prop.getProperty("filename");  
 int partcount = Integer.*parseInt*(prop.getProperty("partcount"));  
 File[] partFiles = dir.listFiles(new SuffixFilter(*SP*));  
 if (partFiles.length != partcount)  
 throw new RuntimeException("缺少碎片文件");  
 ArrayList<FileInputStream> al = new ArrayList<FileInputStream>();  
 for (int x = 0; x < partcount; x++) {  
 al.add(new FileInputStream(new File(dir, x + *SP*)));  
 }  
 Enumeration<FileInputStream> en = Collections.*enumeration*(al);  
 SequenceInputStream sis = new SequenceInputStream(en);  
 File file = new File(dir, fileName);  
 FileOutputStream fos = new FileOutputStream(file);  
 byte[] buf = new byte[1024];  
 int len = 0;  
 while ((len = sis.read(buf)) != -1) {  
 fos.write(buf, 0, len);  
 }  
 fos.close();  
 sis.close();  
 }  
 //带有配置信息的数据切割。  
 public static void splitFile(File file) throws IOException {  
 //用一个读取流和文件关联。  
 FileInputStream fis = new FileInputStream(file);  
 //创建目的地。因为有多个。所以先创建引用。  
 FileOutputStream fos = null;  
 //指定碎片的位置。  
 File dir = new File("c:\\partfiles");  
 if (!dir.exists())  
 dir.mkdir();  
 //碎片文件大小引用。  
 File f;  
 byte[] buf = new byte[1024 \* 1024];  
 //因为切割完的文件通常都有规律的。为了简单标记规律使用计数器。  
 int count = 0;  
 int len;  
 while ((len = fis.read(buf)) != -1) {  
 f = new File(dir, (count++) + ".part");  
 fos = new FileOutputStream(f);  
 fos.write(buf, 0, len);  
 fos.close();  
 }  
 //碎片文件生成后，还需要定义配置文件记录生成的碎片文件个数。以及被切割文件的名称。  
 //定义简单的键值信息，可是用Properties。  
 String filename = file.getName();  
 Properties prop = new Properties();  
 prop.setProperty("filename", filename);  
 prop.setProperty("partcount", count + "");  
 File config = new File(dir, count + ".properties");  
 fos = new FileOutputStream(config);  
 prop.store(fos, "");  
 fos.close();  
 fis.close();  
 }  
}  
class SuffixFilter implements FileFilter {  
 private String suffix;  
 SuffixFilter(String suffix) {  
 this.suffix = suffix;  
 }  
 public boolean accept(File file) {  
 return file.getName().endsWith(suffix);  
 }  
}

### 数组访问流

对于数组元素操作无非两种操作：设置（写）和获取（读），而这两操作正好对应流的读写操作。这两个对象就是使用了流的读写思想来操作数组。

对于数组访问流而言，其源是内存，目的也是内存，因而没有调用系统资源，也就不需要关闭。由于数组本身没有方法，只有一个length属性，因此为了便于数组的操作，将数组进行封装，对外提供方法操作数组中的元素。

#### ByteArrayInputStream

该对象实例化时需要明确一个数据源，因此要传入相应的数组。

#### ByteArrayOutputStream

该对象实例化时可以使用空参数，因为其内置了一个可变长度的数组作为缓冲区。

### 压缩文件流

#### ZipInputStream

#### ZipOutputStream

# Thread

## 多线程简介

### 程序组成

**程序**（**program**）：指一段静态的代码，静态对象

**进程**（**process**）：是程序的一次执行过程，或是正在运行的程序

**线程**（**thread**）：进程可进一步细化为线程，是一个程序内部的一条执行路径

进程是程序运行和资源分配的基本单位，一个程序至少有一个进程，一个进程至少有一个线程。进程在执行过程中拥有独立的内存单元，而多个线程共享内存资源，减少切换次数，从而效率更高。

线程是进程的一个实体，是cpu调度和分派的基本单位，是比程序更小的能独立运行的基本单位。同一进程中的多个线程之间可以并发执行。

多线程扩展了多进程的概念，使得同一个进程可以同时并发处理多个任务。线程（**Thread**）也被称作轻量级进程（**Lightweight Process**） ，线程是进程的执行单元。就像进程在操作系统中的地位一样，线程在程序中是独立的、并发的执行流。当进程被初始化后，主线程就被创建了。对于绝大多数的应用程序来说，通常仅要求有一个主线程，但我们也可以在该进程内创建多条顺序执行流，这些顺序执行流就是线程，每条线程也是互相独立的。

线程是进程的组成部分，一个进程可以拥有多个线程，一个线程必须有一个父进程。线程可以拥有自己的堆栈、自己的程序计数器和自己的局部变量，但不再拥有系统资源，它与父进程的其他线程共享该进程所拥有的全部资源。因为多个线程共享父进程里的全部资源，因此编程更加方便；但必须更加小心，我们必须确保线程不会妨碍同一进程里的其他线程。

### Java程序运行原理

java 命令会启动 java 虚拟机，启动 JVM，等于启动了一个应用程序，也就是启动了一个进程。该进程会自动启动一个 “主线程” ，然后主线程去调用某个类的 main 方法。所以 main方法运行在主线程中。在此之前的所有程序都是单线程的。

Java虚拟机至少启动了垃圾回收线程和主线程，因此JVM是多线程的。

**守护线程和非守护线程**

程序运行完毕，JVM会等待非守护线程完成后关闭，但是JVM不会等待守护线程。守护线程最典型的例子就是GC线程。

**多线程和异常**

在执行多线程时如果发生异常且这个异常没有被捕获的话，这个线程就停止执行了。如果这个线程持有某个对象的监视器，那么这个对象监视器会被立即释放。

**使用多线程的优点**

提高应用程序的响应，对图形化界面更有意义；

提高计算机系统CPU的利用率；

改善程序结构，将复杂冗长的进程分为多个线程独立运行，便于理解和修改。

提高应用程序的使用率。

### 线程调度

#### 分时调度模型

所有线程轮流使用CPU的使用权，平均分配每个线程占用CPU的时间片。

#### 抢占式调度模型

一个线程用完CPU之后，操作系统会根据线程优先级、线程饥饿情况等数据算出一个总的优先级并分配下一个时间片给某个线程执行。

由于Java采用抢占式的线程调度算法，因此可能会出现某条线程常常获取到CPU控制权的情况，为了让某些优先级比较低的线程也能获取到CPU控制权，可以使用Thread.sleep(0)手动触发一次操作系统分配时间片的操作，这也是平衡CPU控制权的一种操作。

**多线程开发的优化**

1、给线程命名

2、最小化同步范围

3、优先使用volatile

4、尽可能使用更高层次的并发工具而非wait和notify()来实现线程通信,如BlockingQueue,Semeaphore

5、优先使用并发容器而非同步容器.

6、考虑使用线程池

### 并行和并发

并行指的是逻辑上同时发生，某一个时间内同时运行多个程序。

并发指的是物理上同时发生，某一个时间点同时运行多个程序。可以通过调度和控制多个CPU来实现并发。

## 多线程的实现

创建一个新的线程共有下列四种方式：

1. 继承Thread类
2. 实现Runnable接口
3. 实现Callable接口
4. 线程池

### 继承Thread类

继承Thread并重写run方法，最后创建对象并调用父类中的start方法。

通过匿名内部类的对象创建线程

new Thread(){

public void run(){

执行代码;

}

}.start();

start的作用有两个：开启一个分线程，然后调用run方法

|  |
| --- |
| **public** **class** ExtendsThreadTest {  **public** **static** **void** main(String[] args) {  Test test = **new** Test();  //test线程  test.start();  //主线程  **for** (**int** i = 0; i < 100; i++) {  System.***out***.println(Thread.*currentThread*().getName() + " " + i);  }  }  }  **class** Test **extends** Thread{  **public** **void** run() {  **for** (**int** i = 0; i < 100; i++) {  System.***out***.println(Thread.*currentThread*().getName() + " " + i);  }  }  } |

### 实现Runnable接口

实现Runnable接口，并重写run方法，创建该类的对象并将该对象作为参数传递到Thread的构造器中并运行start方法。

传递参数到构造器中是为了将实现该接口的类和线程进行关联并明确要运行的run方法。

实现接口的方式避免了单继承的局限性；同时多个线程可以共享同一个接口实现类的对象，适合多个相同的线程来处理同一份资源。

|  |
| --- |
| **public** **class** ImplementsRunnableTest {  **public** **static** **void** main(String[] args) {  Test2 test2 = **new** Test2();  Thread thread = **new** Thread(test2);  //test2线程  thread.start();  //主线程  **for** (**int** i = 0; i < 100; i++) {  System.***out***.println(Thread.*currentThread*().getName() + " " + i);  }  }  }  **class** Test2 **implements** Runnable{  **public** **void** run() {  **for** (**int** i = 0; i < 100; i++) {  System.***out***.println(Thread.*currentThread*().getName() + " " + i);  }  }  } |

**主线程**：main方法执行的内容

**分线程**：main方法中其他的线程内容

开启多个分线程则需要创建多个Thread或其子类的对象并调用start方法

### 继承和实现优劣

从**单继承多实现**的角度看，实现Runnable接口更好一些。如果使用继承Thread方式，那么将不能再继承其它的类。

从**共享数据**的角度看，继承Thread的方式，在Thread的子类中属性必须使用static修饰（多个对象共同使用一份类变量），因为Thread要开启多个线程需要创建多个对象。实现Runnable则不用特殊处理，使用比较方便，因为Runnable实现类的对象只需要创建一个。

从**线程同步**的角度看，继承Thread这种方式，同步代码块的监视器不能使用this而只能使用当前类的运行时对象或其他共享对象，同步方法必须使用static修饰（此时监视器是当前类的运行时对象，存在于方法区中，多个对象只有一个运行时对象）。实现Runnable，同步代码块的监视器可以使用this，同步方法不用使用static修饰。

### 实现Callable接口

实现Callable接口，重写call方法（返回值为Object），call方法有Object类型的返回值；

将实现接口的类的对象传递到FutureTask的构造器中创建FutureTask类的对象，再将FutureTask类的对象作为参数传递到Thread的构造器中创建Thread类的对象，通过Thread类的对象调用start方法创建线程。

主线程需要分线程的结果，可以通过FutureTask对象的get方法获取call方法返回的结果，get方法除了获取返回值，还会阻塞主线程，等待分线程执行完毕返回结果。

|  |
| --- |
| public class CallableTest {  public static void main(String[] args) throws InterruptedException, ExecutionException {  MyThread myThread = new MyThread();  FutureTask<String> futureTask = new FutureTask<>(myThread);  FutureTask<String> futureTask2 = new FutureTask<>(myThread);  Thread thread1 = new Thread(futureTask);  thread1.start();  thread1.setName("one");  Thread thread2 = new Thread(futureTask2);  thread2.start();  thread2.setName("two");  String string = futureTask.get();  System.*out*.println(string);  String string2 = futureTask2.get();  System.*out*.println(string2);  } } class MyThread implements Callable<String> {  private static int *sum* = 0;  public synchronized String call() {  for(int i = 1; i<50; i++) {  *sum* =*sum* + i;  System.*out*.println(Thread.*currentThread*().getName()+*sum*);  }  return "success";  } } |

### Runnable和Callable的区别

Runnable接口中的run()方法的返回值是void，它做的事情只是纯粹地去执行run()方法中的代码而已；Callable接口中的call()方法是有返回值的，是一个泛型，和Future、FutureTask配合可以用来获取异步执行的结果。

这其实是很有用的一个特性，因为多线程相比单线程更难、更复杂的一个重要原因就是因为多线程充满着未知性，某条线程是否执行了？某条线程执行了多久？某条线程执行的时候我们期望的数据是否已经赋值完毕？无法得知，我们能做的只是等待这条多线程的任务执行完毕而已。而Callable+Future/FutureTask却可以方便获取多线程运行的结果，可以在等待时间太长没获取到需要的数据的情况下取消该线程的任务。

## Thread类及其API

Thread()创建新的Thread对象

Thread(Runnable target)指定创建线程的目标对象，该对象实现了Runnable接口中的run方法

Thread(Runnable target,String)创建新的Thread对象

|  |  |
| --- | --- |
| **Thread方法摘要** | |
| void | **run**()线程在被调度时执行的操作 |
| void | **start**()启动线程，并执行对象的run()方法 |
| String | **getName**()返回线程的名称 |
| void | **setName**(String name)设置线程的名称 |
| static Thread | **currentThread**()返回当前线程 |
| staitc void | **yield**()线程让步，暂停当前正在执行的线程，让出当前线程本次执行的机会，把执行机会让给优先级相同或更高的线程，若队列中没有相同优先级的线程，则无视本方法 |
| void | **join**()本线程暂停执行，让调用该方法的线程全部执行完毕后再执行本线程 |
| static void | **sleep**(long millis)让当前线程进入睡眠（让出CPU资源），当睡眠结束再继续执行（注意：不影响其他线程） |
| void | **~~stop~~**~~()~~强制线程生命期结束（已过时，禁止使用） |
| void | **interrupt**()打断另一个线程，相比于stop，interrupt更灵活优雅 |
| static boolean | **isInterrupted**()接受interrupt的信息，用来判断当前线程是否需要停止，为true就表示要结束线程了 |
| boolean | **isAlive**()判断线程是否还活着 |
| String | **toString**()返回该线程的字符串表示形式，包括线程名称、优先级和线程组。 |
| void | **setDaemon**（boolean on）将该线程标记为守护线程或用户线程。当正在运行的线程都是守护线程时，Java 虚拟机退出。该方法必须在启动线程前调用。 |
| ThreadGroup | **getThreadGroup**()获取该线程所属的线程组 |

当A线程执行到了B线程的join方式。A线程处于冻结状态，释放了执行权，B开始执行。A什么时候执行呢？只有当B线程运行结束后，A才从冻结状态恢复运行状态执行。

当另一个线程调用本线程的interrupt方法，如果该线程正在sleep睡眠，那么就会抛出异常，此时可以在catch中直接终止本线程。

**start方法和run方法的区别**

start()方法被用来启动新创建的线程，而且start()内部调用了run()方法，这和直接调用run()方法的效果不一样。当你调用run()方法的时候，只会是在原来的线程中调用，没有新的线程启动，start()方法才会启动新线程。

start()方法表示创建新的线程并就绪，只要获得CPU执行权就可以调用内部的run()方法运行线程。run()方法表示线程的运行，并没有创建新的线程，所有的线程类都必须实现run()方法。

除了使用setNmae方法设置线程的名称，也可以通过调用父类的构造器设置线程的名称

public MyThread(String name){

super(name);

}

|  |
| --- |
| public class MyThreadTest {  public static void main(String[] args) {  Thread thread = new Thread(new MyThread());  thread.setName("我的线程：");  thread.start();  for (int i = 0; i < 100; i++) {  System.*out*.println(Thread.*currentThread*().getName() + " " + i);  if (i == 50) {  //暂停当前线程，让出执行机会，把执行机会让给优先级相同或更高的线程，  //若队列中没有相同优先级的线程，则无视本方法  Thread.*yield*();  }  if(i == 75) {  //打断另一个线程，相比于stop，interrupt更灵活优雅  thread.interrupt();  }  }  } } class MyThread implements Runnable{  public void run() {  for (int i = 0; i < 100; i++) {  System.*out*.println(Thread.*currentThread*().getName() + " " + i);  //接受interrupt的信息，用来判断当前线程是否需要停止，为true就表示要结束线程了  if(Thread.*currentThread*().isInterrupted()) {  //判断线程是否还活着  System.*out*.println(Thread.*currentThread*().isAlive()+"-----------------------");  return;  }  }  } } |

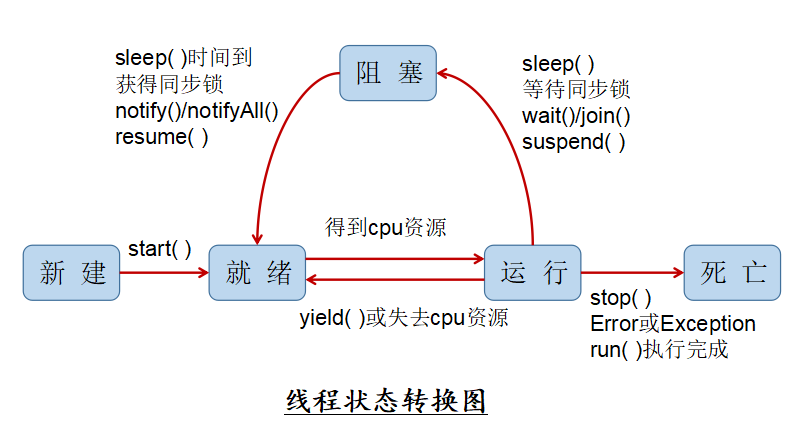
### 线程的优先级控制相关API

线程创建时继承父线程的优先级

|  |  |
| --- | --- |
| **方法摘要** | |
| int | **getPriority**()返回线程优先级的值 |
| void | **setPriority**(int newPriority)设置线程优先级的值 |
| Thread.**MAX\_PRIORITY** = 10; |  |
| Thread.**MIN\_PRIORITY** = 1; |  |
| Thread.**NORM\_PRIORITY** = 5; | 线程默认优先级 |

## 线程的生命周期

一个完整的线程生命周期包含下列物种状态：



### 新建

当一个Thread类或其子类的对象被声明创建时，新的线程对象处于新建状态。

### 就绪

处于新建状态的线程调用start方法后，进入线程队列等待CPU时间片，此时该线程已经具备了运行的条件。调用run方法是线程的线程执行体，也就是线程将要完成的事。

### 运行

当就绪的线程被调度并获得处理器资源时，便进入运行状态，run方法定义了线程的操作和功能。

### 阻塞

在特殊情况下，被认为的挂起或执行输入输出操作时，让出CPU并临时中止自身的执行，进入阻塞状态。

sleep()：方法允许以毫秒为单位的一段时间作为参数，使得线程在指定时间内进入阻塞状态，不能得到CPU资源，指定的时间一过，线程重新进入可执行状态。

~~suspend()resume()~~（该方法已过时，不要使用）：suspend()使得线程进入阻塞状态，并且不会自动恢复，必须其对应的resume()被调用，才能使得线程重新进入可执行状态。

yield()：使当前线程放弃已经获得的CPU资源，但不使当前线程阻塞（即线程仍处于可执行状态），随时可能再次获得CPU资源。

wait()、notify()和notifyAll()：wait()方法使线程进入阻塞状态，该方法有指定以毫秒为单位的一段时间作为参数和空参两种方式，前者需要对应的notify()或notifyAll方法被调用或超出指定时间时线程重新进入可执行状态，后者则必须要notify()或notifyAll方法被调用。

#### 唤醒阻塞的线程

如果线程是因为调用了wait()、sleep()或者join()方法而导致的阻塞，可以中断线程，并且通过抛出InterruptedException来唤醒它；如果线程遇到了IO阻塞，无能为力，因为IO是操作系统实现的，Java代码并没有办法直接接触到操作系统。

### 死亡

线程完成了它的全部工作或线程被提前强制性的中止。

线程的停止：通过stop方法就可以停止线程。但是这个方式过时了。

停止线程：原理就是：让线程运行的代码结束，也就是结束run方法。

怎么结束run方法？一般run方法里肯定定义循环。所以只要结束循环即可。

第一种方式：定义循环的结束标记。

第二种方式：如果线程处于了冻结状态，是不可能读到标记的，这时就需要通过Thread类中的interrupt方法，将其冻结状态强制清除。让线程恢复具备执行资格的状态，让线程可以读到标记，并结束。

## 多线程的同步

线程安全问题的原因在于当一个线程正在操作共享数据时，另一个线程也进来进行操作，那么就会发生线程安全问题。

1，多个线程使用共享数据。

2，有多条语句对共享数据进行操作。

解决线程安全的方式（同步加锁）

当一个线程在操作共享数据时，其他的线程不允许操作这个数据。可以使用synchronized关键字解决这个问题。synchronized共有两种使用方式：同步代码块和同步方法。

定义线程同步的前提

必须要有两个或者两个以上的线程，才需要同步

多个线程必须保证使用同一个锁

### 同步代码块

用来操作共享数据的代码都放在代码块中，在使用时要注意哪些代码需要存放于代码块中。

#### 继承Thread

synchronized(同步监视器){

共享数据操作代码块

}

继承Thread需要同步监视器（锁）：可以是任意类的对象,多个线程必须使用的是同一个对象（同一把锁）。因为创建多个线程会创建多个子类对象，因此同步监视器不可以使用this关键字

共享数据必须要用static修饰，自定义的监视器也要用static修饰，因为要确保多个线程使用的是同一个监视器

|  |
| --- |
| **class** SynTest1 **extends** Thread{  **private** **static** **int** *num* = 100;  **private** **static** Object *obj* = **new** Object();  **public** **void** run() {  **while**(**true**) {  **synchronized** (*obj*) {  **if**(*num* > 0) {  System.***out***.println(Thread.*currentThread*().getName() + " " + *num*);  *num*--;  }**else** {  **break**;  }  }  }  }  } |
| **public** **void** method1() {  SynTest1 synTest1 = **new** SynTest1();  SynTest1 synTest2 = **new** SynTest1();  SynTest1 synTest3 = **new** SynTest1();  synTest1.start();  synTest2.start();  synTest3.start();  } |

#### 实现Runnable

synchronized(同步监视器){

共享数据操作代码块

}

实现接口因为只会创建一个类的对象，所以可以用this关键字为同步监视器。

共享数据不必用static修饰，可以不用自定义监视器，若自定义监视器也不必用static修饰。

|  |
| --- |
| **class** SynTest3 **implements** Runnable{  **private** **int** num = 100;  **public** **void** run() {  **while**(**true**) {  **synchronized** (**this**) {  **if**(num > 0) {  System.***out***.println(Thread.*currentThread*().getName() + " " + num);  num--;  }**else** {  **break**;  }  }  }  }  } |
| **public** **void** method3() {  SynTest3 synTest3 = **new** SynTest3();  Thread t1 = **new** Thread(synTest3);  Thread t2 = **new** Thread(synTest3);  Thread t3 = **new** Thread(synTest3);    t1.start();  t2.start();  t3.start();  } |

### 同步方法

同步方法中的代码同时只能有一个线程进行操作。

#### 继承Thread

public static synchronized void test(){

共享数据操作代码块

}

继承Thread时的同步方法需要用static修饰，因为使用static修饰后该方法中的默认监视器是该类的**运行时的对象。**

**运行时对象：**一个类不论创建多少个对象（实例）都只有一个运行时对象，该对象的类型为Class类型，可以用类名.class获取，该对象存放于方法区中。

|  |
| --- |
| **class** SynTest2 **extends** Thread{  **private** **static** **int** *num* = 100;  **private** **boolean** test = **true**;  **public** **void** run() {  **while**(test) {  test = *test*();  }  }  **public** **static** **synchronized** **boolean** test() {  **if**(*num* > 0) {  System.***out***.println(Thread.*currentThread*().getName() + " " + *num*);  *num*--;  **return** **true**;  }**else** {  **return** **false**;  }  }  } |
| **public** **void** method2() {  SynTest2 synTest1 = **new** SynTest2();  SynTest2 synTest2 = **new** SynTest2();  SynTest2 synTest3 = **new** SynTest2();  synTest1.start();  synTest2.start();  synTest3.start();  } |

#### 实现Runnable

public synchronized void test(){

共享数据操作代码块

}

实现接口时同步方法的监视器默认是this

|  |
| --- |
| **class** SynTest4 **implements** Runnable{  **private** **int** num = 100;  **private** **boolean** test = **true**;  **public** **void** run() {  **while**(test) {  test = test();  }  }  **public** **synchronized** **boolean** test() {  **if**(num > 0) {  System.***out***.println(Thread.*currentThread*().getName() + " " + num);  num--;  **return** **true**;  }**else** {  **return** **false**;  }  }  } |
| **public** **void** method4() {  SynTest4 synTest4 = **new** SynTest4();  Thread t1 = **new** Thread(synTest4);  Thread t2 = **new** Thread(synTest4);  Thread t3 = **new** Thread(synTest4);    t1.start();  t2.start();  t3.start();  } |

### 同步代码块和同步方法的区别

同步代码块使用的锁可以是任意对象。

**同步函数使用的锁是this，静态同步函数的锁是该类的字节码文件对象**。

**在一个类中只有一个同步，可以使用同步函数。如果有多同步，必须使用同步代码块，来确定不同的锁。所以同步代码块相对灵活一些。**

如果锁对象是this，可以使用同步方法，否则能使用同步代码块就尽量使用同步代码块。

### 线程同步的优缺点

**线程同步的好处**：可以保证线程的安全（多个线程操作共享数据不会出现问题）。

**线程同步的缺点**：操作共享数据的代码同时只能有一个线程执行，因此效率低一些，还可能出现死锁。

## 同步监视器（锁）

**检测一个线程是否持有对象监视器**

Thread类提供了一个holdsLock(Object obj)方法，当且仅当对象obj的监视器被某条线程持有的时候才会返回true，注意这是一个static方法，这意味着”某条线程”指的是当前线程。

### 互斥锁

Java中引入了对象互斥锁(**monitor**)的概念，来保证共享数据操作的完整性。

每个对象都对应于一个可称为“互斥锁”的标记，这个标记用来保证在任一时刻，只能有一个线程访问该对象。

关键字synchronized 来与对象的互斥锁联系。当某个对象用synchronized修饰时，表明该对象在任一时刻只能由一个线程访问。

同步的局限性：导致程序的执行效率要降低

同步方法（非静态的）的锁为this。

同步方法（静态的）的锁为当前类本身。

### 死锁

不同的线程分别占用对方需要的同步资源不放弃，都在等待对方放弃自己需要的同步资源，就形成了线程的死锁。

**死锁的产生条件**

1.互斥条件：一个资源每次只能被一个进程使用。   
2.请求与保持条件：一个进程因请求资源而阻塞时，对已获得的资源保持不放。   
3.不剥夺条件:进程已获得的资源，在末使用完之前，不能强行剥夺。   
4.循环等待条件:若干进程之间形成一种头尾相接的循环等待资源关系。

多个线程尽量不要使用同一把锁,可以使用方法在线程等待的某段时间中结束本线程

JDK中的jconsole工具可以帮助检测死锁

|  |
| --- |
| **public** **class** TestDeadLock {  **static** StringBuffer *s1* = **new** StringBuffer();  **static** StringBuffer *s2* = **new** StringBuffer();  **public** **static** **void** main(String[] args) {  **new** Thread() {  **public** **void** run() {  System.***out***.println("deadLock");  **synchronized** (*s1*) {  **try** {  Thread.*sleep*(5);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  *s2*.append("A");  **synchronized** (*s2*) {  *s2*.append("B");  System.***out***.print(*s1*);  System.***out***.print(*s2*);  }  }  }  }.start();  **new** Thread() {  **public** **void** run() {  **synchronized** (*s2*) {  *s2*.append("C");  **synchronized** (*s1*) {  *s1*.append("D");  System.***out***.print(*s2*);  System.***out***.print(*s1*);  }  }  }  }.start();  }  } |

### Lock同步锁

Lock接口：多线程在JDK1.5版本升级时，推出一个接口Lock接口。

Lock是控制多个线程对共享资源进行访问的工具。锁提供了对共享资源的独占访问，每次只能有一个线程对Lock对象加锁，线程开始访问共享资源之前应先获得Lock对象。

在实现线程安全的控制中，比较常用的是ReentrantLock(可重入锁)，可以显式加锁、释放锁。

解决线程安全问题使用同步的形式，(同步代码块，要么同步函数)其实最终使用的都是锁机制。

到了后期版本，直接将锁封装成了对象。线程进入同步就是具备了锁，执行完，离开同步，就是释放了锁。

在后期对锁的分析过程中，发现，获取锁，或者释放锁的动作应该是锁这个事物更清楚。所以将这些动作定义在了锁当中，并把锁定义成对象。

所以同步是隐示的锁操作，而Lock对象是显示的锁操作，它的出现就替代了同步。在实现线程安全的控制中，比较常用的是ReentrantLock(可重入锁)，可以显式加锁和释放锁。

在之前的版本中使用Object类中wait、notify、notifyAll的方式来完成的。那是因为同步中的锁是任意对象，所以操作锁的等待唤醒的方法都定义在Object类中。

而现在锁是指定对象Lock。所以查找等待唤醒机制方式需要通过Lock接口来完成。而Lock接口中并没有直接操作等待唤醒的方法，而是将这些方式又单独封装到了一个对象中。这个对象就是Condition，将Object中的三个方法进行单独的封装。并提供了功能一致的方法 await()、signal()、signalAll()体现新版本对象的好处。

|  |
| --- |
| **class** BoundedBuffer {  **final** Lock lock = **new** ReentrantLock();  **final** Condition notFull = lock.newCondition();  **final** Condition notEmpty = lock.newCondition();  **final** Object[] items = **new** Object[100];  **int** putptr, takeptr, count;  **public** **void** put(Object x) **throws** InterruptedException {  //上锁，同时只能有一个线程执行锁住后的代码  lock.lock();  **try** {  **while** (count == items.length)  notFull.await();  items[putptr] = x;  **if** (++putptr == items.length)  putptr = 0;  ++count;  notEmpty.signal();  } **finally** {  //这句放入finally代码块中保证必须解锁  lock.unlock();  }  }  **public** Object take() **throws** InterruptedException {  lock.lock();  **try** {  **while** (count == 0)  notEmpty.await();  Object x = items[takeptr];  **if** (++takeptr == items.length)  takeptr = 0;  --count;  notFull.signal();  **return** x;  } **finally** {  lock.unlock();  }  }  } |

### ReentrantLock类

synchronized是和if、else、for、while一样的关键字，ReentrantLock是类，这是二者的本质区别。既然ReentrantLock是类，那么它就提供了比synchronized更多更灵活的特性，可以被继承、可以有方法、可以有各种各样的类变量，ReentrantLock比synchronized的扩展性体现在几点上：

（1）ReentrantLock可以对获取锁的等待时间进行设置，这样就避免了死锁   
（2）ReentrantLock可以获取各种锁的信息   
（3）ReentrantLock可以灵活地实现多路通知

二者的锁机制其实也是不一样的:ReentrantLock底层调用的是Unsafe的park方法加锁，synchronized操作的应该是对象头中mark word。

#### Lock和synchronized的区别

主要相同点：Lock 能完成 synchronized 所实现的所有功能

主要不同点：Lock 有比 synchronized 更精确的线程语义和更好的性能。synchronized 会自动释放锁，而 Lock 一定要求程序员手工释放，并且必须在 finally 从句中释放。Lock 还有更强大的功能，例如，它的 tryLock 方法可以非阻塞方式去拿锁。

### 锁的种类

自旋锁: 自旋锁在JDK1.6之后就默认开启了。基于之前的观察，共享数据的锁定状态只会持续很短的时间，为了这一小段时间而去挂起和恢复线程有点浪费，所以这里就做了一个处理，让后面请求锁的那个线程在稍等一会，但是不放弃处理器的执行时间，看看持有锁的线程能否快速释放。为了让线程等待，所以需要让线程执行一个忙循环也就是自旋操作。在jdk6之后，引入了自适应的自旋锁，也就是等待的时间不再固定了，而是由上一次在同一个锁上的自旋时间及锁的拥有者状态来决定。

偏向锁: 在JDK1.之后引入的一项锁优化，目的是消除数据在无竞争情况下的同步原语。进一步提升程序的运行性能。 偏向锁就是偏心的偏，意思是这个锁会偏向第一个获得他的线程，如果接下来的执行过程中，改锁没有被其他线程获取，则持有偏向锁的线程将永远不需要再进行同步。偏向锁可以提高带有同步但无竞争的程序性能，也就是说他并不一定总是对程序运行有利，如果程序中大多数的锁都是被多个不同的线程访问，那偏向模式就是多余的，在具体问题具体分析的前提下，可以考虑是否使用偏向锁。

轻量级锁: 为了减少获得锁和释放锁所带来的性能消耗，引入了“偏向锁”和“轻量级锁”，所以在Java SE1.6里锁一共有四种状态，无锁状态，偏向锁状态，轻量级锁状态和重量级锁状态，它会随着竞争情况逐渐升级。锁可以升级但不能降级，意味着偏向锁升级成轻量级锁后不能降级成偏向锁。

### 乐观锁和悲观锁

乐观锁：乐观锁认为竞争不总是会发生，因此它不需要持有锁，将比较-替换这两个动作作为一个原子操作尝试去修改内存中的变量，如果失败则表示发生冲突，那么就应该有相应的重试逻辑。

悲观锁：悲观锁认为竞争总是会发生，因此每次对某资源进行操作时，都会持有一个独占的锁，就像synchronized，不管三七二十一，直接上了锁就操作资源了。

## 多线程的通信

### 多线程的上下文切换

多线程的上下文切换指的是CPU控制权由一个已经正在运行的线程切换到另外一个就绪并等待获取CPU执行权的线程的过程。

### 操作线程涉及的方法

这三个方法只能在同步代码块和同步方法中使用，因为这三个方法在调用前都必须先获得对象的锁。A锁上的线程被wait了,那这个线程就相当于处于A锁的线程池中，只能A锁的notify唤醒。

wait()方法

令当前线程挂起并放弃CPU、同步资源，使别的线程可访问并修改共享资源，而当前线程排队等候再次对资源的访问。wait方法应该在循环调用，因为当线程获取到 CPU 开始执行的时候，其他条件可能还没有满足，所以在处理前，循环检测条件是否满足会更好。

在当前线程中调用方法：对象名.wait()

使当前线程进入等待（某对象）状态 ，直到另一线程对该对象发出 notify (或notifyAll) 为止。

调用方法的必要条件：当前线程必须具有对该对象的监控权（加锁）

调用此方法后，当前线程将释放对象监控权 ，然后进入等待

在当前线程被notify后，要重新获得监控权，然后从断点处继续代码的执行。

notify()方法

唤醒正在排队等待同步资源的线程中优先级最高者结束等待

notifyAll()方法

唤醒正在排队等待资源的所有线程结束等待

在当前线程中调用方法： 对象名.notify()

功能：唤醒等待该对象监控权的一个线程。

调用方法的必要条件：当前线程必须具有对该对象的监控权（加锁）

#### wait()和notify()/notifyAll()方法在放弃对象监视器时的区别

wait方法立即释放对象监视器；notify/notifyAll则会等待线程剩余代码执行完毕才会放弃对象监视器。

#### wait(),notify()和suspend(),resume()之间的区别

suspend方法阻塞时不会释放占用的锁（如果占用了的话），而wait方法则相反。上述的核心区别导致了一系列的细节上的区别。

首先，suspend和resume方法隶属于 Thread 类，wait和notify方法却直接隶属于 Object 类，也就是说，所有对象都拥有这一对方法。因为wait方法阻塞时要释放占用的锁，而锁是任何对象都具有的，调用任意对象的wait方法导致线程阻塞，并且该对象上的锁被释放。而调用任意对象的notify方法则导致从调用该对象的wait方法而阻塞的线程中随机选择的一个解除阻塞（但要等到获得锁后才真正可执行）。

其次，suspend和resume方法可在任何位置调用，但是wait和notify方法却必须在 synchronized 方法或块中调用，因为只有在synchronized 方法或块中当前线程才占有锁，才有锁可以释放。同样的道理，调用这一对方法的对象上的锁必须为当前线程所拥有，这样才有锁可以释放。因此wait和notify方法调用必须放置在这样的 synchronized 方法或块中，该方法或块的上锁对象就是调用这一对方法的对象。若不满足这一条件，则程序虽然仍能编译，但在运行时会出现IllegalMonitorStateException 异常。

wait和 notify方法的上述特性决定了它们经常和synchronized关键字一起使用，将它们和操作系统进程间通信机制作一个比较就会发现它们的相似性，synchronized方法或块提供了类似于操作系统原语的功能，它们的执行不会受到多线程机制的干扰，而这一对方法则相当于 block 和wakeup 原语（这一对方法均声明为 synchronized）。它们的结合使得我们可以实现操作系统上一系列精妙的进程间通信的算法（如信号量算法），并用于解决各种复杂的线程间通信问题。

suspend方法和不指定超时期限的wait方法的调用都可能产生死锁。Java 并不在语言级别上支持死锁的避免，因此在编程中必须小心地避免死锁。

关于 wait() 和 notify() 方法两点说明

第一：调用 notify() 方法导致解除阻塞的线程是从因调用该对象的 wait() 方法而阻塞的线程中随机选取的，我们无法预料哪一个线程将会被选择，所以编程时要特别小心，避免因这种不确定性而产生问题。

第二：除了 notify()，还有一个方法 notifyAll() 也可起到类似作用，唯一的区别在于，调用 notifyAll() 方法将把因调用该对象的 wait() 方法而阻塞的所有线程一次性全部解除阻塞。当然，只有获得锁的那一个线程才能进入可执行状态。

以上对 Java 中实现线程阻塞的各种方法作了一番分析，重点分析了wait和notify方法，因为它们的功能最强大，使用也最灵活，但是这也导致了它们的效率较低，容易出错。实际使用中我们应该灵活使用各种方法，以便更好地达到我们的目的。

#### 操作线程的方法定义在Object类

原因是JAVA提供的锁是对象级而不是线程级，每个对象都有锁，通过线程获得。如果线程需要等待某些锁那么调用对象中的wait()方法就有意义了。如果wait()方法定义在Thread类中，线程正在等待的是哪个锁就不明显了。简单的说，由于wait，notify和notifyAll都是锁级别的操作，所以把他们定义在Object类中因为锁属于对象。

#### wait()和sleep()的区别？

sleep睡着时释放执行权但不会释放锁(即同步资源)；wait睡着时释放执行权且会释放锁(即同步资源)。

sleep必须指定时间然后自动唤醒；wait可以指定时间也可以不指定时间，不指定时间需要需要被其他线程通过notify或notifyAll方法唤醒。

sleep是Thread类的静态方法，由Thread直接调用；wait是Object中的方法，由同步监视器调用。

### 释放锁的操作

1. 当前线程的同步代码块或同步方法执行结束
2. 当前线程在同步代码块或同步方法中遇到break、return终止了该代码块
3. 当前线程在同步代码块或同步方法中出现了未处理的Error或Exception，导致异常结束线程执行
4. 当前线程在同步代码块或同步方法中执行了锁对象的wait()方法，当前线程暂停，并释放锁

### 不会释放锁的操作

1. 线程执行同步代码块或同步方法时，程序调用sleep()方法或yield()方法暂停当前程序的执行
2. 线程执行同步代码块时，其它线程调用了该线程的suspend()方法将该线程挂起，该线程不会释放锁。

### 线程间共享数据

通过在线程之间共享对象就可以了，然后通过wait/notify/notifyAll、await/signal/signalAll进行唤起和等待，比方说阻塞队列BlockingQueue就是为线程之间共享数据而设计的。

## 线程组

Java中使用ThreadGroup来表示线程组，它可以对一批线程进行分类管理，Java允许程序直接对线程组进行控制。默认情况下，所有的线程都属于主线程。

## 线程池

因为线程在创建和销毁时会大量的消耗内存，所以需要使用线程池来提高性能。线程池一个项目中只会有一个，使用单例模式创建。

使用线程池可以有效地控制系统中并发线程的数量。避免因并发创建的线程过多，导致系统性能下降，JVM崩溃。

系统启动一个新线程的成本是比较高的，因为它涉及与os交互。这种情况下，系统启动时即创建大量空闲的线程，就可以很好地提高性能，尤其是当程序需要创建大量生存期很短暂的线程时。

除此之外，使用线程池可以有效地控制系统中并发线程的数量。避免因并发创建的线程过多，导致系统性能下降，JVM崩溃。

### 提交任务线程池队列已满

如果使用的LinkedBlockingQueue，也就是无界队列的话，没关系，继续添加任务到阻塞队列中等待执行，因为LinkedBlockingQueue可以近乎认为是一个无穷大的队列，可以无限存放任务。

如果你使用的是有界队列比方说ArrayBlockingQueue的话，任务首先会被添加到ArrayBlockingQueue中，ArrayBlockingQueue满了，则会使用拒绝策略RejectedExecutionHandler处理满了的任务，默认是AbortPolicy。

Java 5以前，需要手动创建自己的线程池；Java 5开始，新增了Executors工厂类产生线程池。

使用线程池执行线程任务

调用Executors 类的静态方法newFixedThreadPool(int nThreads)，创建一个可重用的、具有固定线程数的线程池ExecutorService对象 创建Runnable实例，作为线程执行任务 调用ExecutorService对象的submit()提交Runnable实例 调用ExecutorService对象的shutDown()方法关闭线程池。

public static ExecutorService newCachedThreadPool()

创建一个具有缓存功能的线程池

缓存：百度浏览过的信息再次访问

public static ExecutorService newSingleThreadExecutor()

创建一个只有单线程的线程池，相当于上个方法的参数是1

## 线程实例

### 生产者消费者

生产者(Productor)将产品交给店员(Clerk)，而消费者(Customer)从店员处取走产品，店员一次只能持有固定数量的产品(比如:20），如果生产者试图生产更多的产品，店员会叫生产者停一下，如果店中有空位放产品了再通知生产者继续生产；如果店中没有产品了，店员会告诉消费者等一下，如果店中有产品了再通知消费者来取走产品。

可能出现两个问题：

生产者比消费者快时，消费者会漏掉一些数据没有取到。

消费者比生产者快时，消费者会取相同的数据。

#### 产品

|  |
| --- |
| **public** **class** Product {  **public** **static** Integer *count* = 0;  **public** **synchronized** **void** addProduct() {  **if**(*count* >= 20) {  **try** {  wait();  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }**else** {  *count*++;  System.***out***.println("生产了第" + *count* + "个产品");  notifyAll();  }  }  **public** **synchronized** **void** getProduct() {  **if**(*count* < 1) {  **try** {  wait();  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }**else** {  System.***out***.println("消费了第" + *count* + "个产品");  *count*--;  notifyAll();  }  }  } |

#### 店员

|  |
| --- |
| **public** **class** Clerk {  **public** **static** **void** main(String[] args) {  Product product = **new** Product();  Productor productor = **new** Productor(product);  **new** Thread(productor).start();  Customer consumer = **new** Customer(product);  **new** Thread(consumer).start();;  }  } |

#### 生产者

|  |
| --- |
| **public** **class** Productor **implements** Runnable {  Product product;  **public** Productor(Product product) {  **this**.product = product;  }  **public** **void** run() {  **while**(**true**) {  **try** {  Thread.*sleep*((**long**)(Math.*random*()\*500));  } **catch** (InterruptedException e) {  e.printStackTrace();  }  product.addProduct();  }  }  } |

#### 消费者

|  |
| --- |
| **public** **class** Customer **implements** Runnable{  Product product;  **public** Customer(Product product) {  **this**.product = product;  }  **public** **void** run() {  **while**(**true**) {  **try** {  Thread.*sleep*((**long**)(Math.*random*()\*500));  } **catch** (InterruptedException e) {  e.printStackTrace();  }  product.getProduct();  }  }  } |

**生产者消费者模型的作用**

（1）通过平衡生产者的生产能力和消费者的消费能力来提升整个系统的运行效率，这是生产者消费者模型最重要的作用。

（2）解耦，这是生产者消费者模型附带的作用，解耦意味着生产者和消费者之间的联系少，联系越少越可以独自发展而不需要收到相互的制约。

### 银行取钱

1.定义一个Account类

1）该Account类封装了账户编号（String）和余额（double）两个属性

2）设置相应属性的getter和setter方法

3）提供无参和有两个参数的构造器

4）系统根据账号判断与用户是否匹配，需提供hashCode()和equals()方法的重写

2.提供一个取钱的线程类

1）提供了Account类的account属性和double类的取款额的属性

2）提供带线程名的构造方法

3）run()方法中提供取钱的操作

3.在主类中创建线程进行测试。考虑线程安全问题。

#### 账户类

getter、setter、空参构造器、有参构造器、equals和hashCode方法可以自动生成。

|  |
| --- |
| **public** **class** Account {  **private** String account;  **private** BigDecimal balance;  } |

#### 取钱类

|  |
| --- |
| **public** **class** GetMoney **extends** Thread{  Account account;  BigDecimal money;  **public** GetMoney(String name,Account account,BigDecimal money) {  **super**(name);  **this**.account = account;  **this**.money = money;  }  **public** **void** run() {  **while**(**true**) {  **synchronized** (account) {  **if**(account.getBalance().compareTo(money) == 1) {  System.***out***.println(Thread.*currentThread*().getName() + "取了" + money + "元");  **try** {  Thread.*sleep*(50);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  account.setBalance(account.getBalance().subtract(money));  }**else** {  System.***out***.println("超出余额，取款失败");  }  System.***out***.println("现在余额为：" + account.getBalance());  }  }  }  } |

#### 测试类

|  |
| --- |
| **public** **class** GetMoneyTest {  **public** **static** **void** main(String[] args) {  Account account = **new** Account("123456",**new** BigDecimal(100000));  Thread t1 = **new** GetMoney("老公", account, **new** BigDecimal(100));  Thread t2 = **new** GetMoney("老婆", account, **new** BigDecimal(1000));  t1.start();  t2.start();  }  } |

## 多线程小结

### stop和suspend方法

反对使用 stop()，是因为它不安全。它会解除由线程获取的所有锁定，而且如果对象处于一种不连贯状态， 那么其他线程能在那种状态下检查和修改它们。 结果很难检查出真正的问题所在。suspend()方法容易发生死锁。调用 suspend()的时候，目标线程会停下来，但却仍然持有在这之前获得的锁定。此时，其他任何线程都不能访问锁定的资源，除非被"挂起"的线程恢复运行。对任何线程来说，如果它们想恢复目标线程，同时又试图使用任何一个锁定的资源，就会造成死锁。所以不应该使用 suspend()，而应在自己的 Thread 类中置入一个标志，指出线程应该活动还是挂起。若标志指出线程应该挂起，便用 wait()命其进入等待状态。若标志指出线程应当恢复，则用一个 notify()重新启动线程。

### 同步和异步

如果有一个资源需要被一个或多个线程共享，这个资源就变成了“竞争”资源，此时多条线程必须按某种既定的规则、依次访问、修改这个“竞争”资源，当一条线程正在访问、修改该“竞争”资源时，其他线程不能同时修改这份“竞争”资源，这就是同步处理。

对于一个银行账户， 如果有多个线程试图去访问这个账户时， 如果不对多个线程进行同步控制，有可能账户余额只有 1000 块，但多个线程都试图取款 800 块时，这些线程同时判断余额之后，都会显示余额足够，从而导致每个线程都取款成功。这显然不是我们希望看到结果。

当程序试图执行一个耗时操作时， 程序不希望阻塞当前执行流， 因此程序也不应该试图立即获取该耗时操作返回的结果，此时就使用异步编程了，典型的应用场景就是 Ajax。当浏览器通过 JavaScript 发出一个异步请求之后，JavaScript 执行流并不会停下来，而是继续向下执行，这就是异步。程序会通过监听器来监听远程服务器响应的到来。

### synchronized同步

**当一个线程进入一个对象的一个 synchronized 方法后，其它线程是否可进入此对象的其它方法方法后，其它线程是否可进入此对象的其它方法?**

当一个线程进行一个对象的 synchronized 方法之后， 其他线程完全有可能再次进入该对象的其他方法。不过要分几种情况来看：

1、如果其他方法没有使用 synchronized 关键字修饰，则可以进入。

2、如果当前线程进入的 synchronized 方法是 static 方法，其他线程可以进入其他synchronized 修饰的非静态方法； 如果当前线程进入的 synchronized 方法是非 static 方法， 其他线程可以进入其他 synchronized 修饰的静态方法。

3、如果两个方法都是静态方法、或者都是非静态方法，并且都使用了 synchronized 修饰，但只要在该方法内部调用了同步监视器的 wait()，则其他线程依然可以进入其他使用synchronized 方法修饰的方法。

4、如果两个方法都是静态方法、或者都是非静态方法，并且都使用了 synchronized 修饰，而且没有在该方法内部调用了同步监视器的 wait()，则其他线程不能进入其他使用synchronized 方法修饰的方法。

## 多线程未知

FutureTask是什么

这个其实前面有提到过，FutureTask表示一个异步运算的任务。FutureTask里面可以传入一个Callable的具体实现类，可以对这个异步运算的任务的结果进行等待获取、判断是否已经完成、取消任务等操作。当然，由于FutureTask也是Runnable接口的实现类，所以FutureTask也可以放入线程池中。

什么是线程局部变量ThreadLocal

线程局部变量是局限于线程内部的变量，属于线程自身所有，不在多个线程间共享。Java提供ThreadLocal类来支持线程局部变量，是一种实现线程安全的方式。但是在管理环境下（如 web 服务器）使用线程局部变量的时候要特别小心，在这种情况下，工作线程的生命周期比任何应用变量的生命周期都要长。任何线程局部变量一旦在工作完成后没有释放，Java 应用就存在内存泄露的风险。

ThreadLoal的作用是什么

简单说ThreadLocal就是一种以空间换时间的做法在每个Thread里面维护了一个ThreadLocal.ThreadLocalMap把数据进行隔离，数据不共享，自然就没有线程安全方面的问题了。

什么是CAS

CAS，全称为Compare and Swap，即比较-替换。假设有三个操作数：内存值V、旧的预期值A、要修改的值B，当且仅当预期值A和内存值V相同时，才会将内存值修改为B并返回true，否则什么都不做并返回false。当然CAS一定要volatile变量配合，这样才能保证每次拿到的变量是主内存中最新的那个值，否则旧的预期值A对某条线程来说，永远是一个不会变的值A，只要某次CAS操作失败，永远都不可能成功。

ConcurrentHashMap的并发度是什么?

ConcurrentHashMap的并发度就是segment的大小，默认为16，这意味着最多同时可以有16条线程操作ConcurrentHashMap，这也是ConcurrentHashMap对Hashtable的最大优势，任何情况下，Hashtable能同时有两条线程获取Hashtable中的数据吗？

ConcurrentHashMap的工作原理

ConcurrentHashMap在jdk 1.6和jdk 1.8实现原理是不同的。

jdk 1.6:

ConcurrentHashMap是线程安全的，但是与Hashtablea相比，实现线程安全的方式不同。Hashtable是通过对hash表结构进行锁定，是阻塞式的，当一个线程占有这个锁时，其他线程必须阻塞等待其释放锁。ConcurrentHashMap是采用分离锁的方式，它并没有对整个hash表进行锁定，而是局部锁定，也就是说当一个线程占有这个局部锁时，不影响其他线程对hash表其他地方的访问。   
具体实现:ConcurrentHashMap内部有一个Segment.

jdk 1.8

在jdk 8中，ConcurrentHashMap不再使用Segment分离锁，而是采用一种乐观锁CAS算法来实现同步问题，但其底层还是“数组+链表->红黑树”的实现。

CyclicBarrier和CountDownLatch区别

这两个类非常类似，都在java.util.concurrent下，都可以用来表示代码运行到某个点上，二者的区别在于：

CyclicBarrier的某个线程运行到某个点上之后，该线程即停止运行，直到所有的线程都到达了这个点，所有线程才重新运行；CountDownLatch则不是，某线程运行到某个点上之后，只是给某个数值-1而已，该线程继续运行。

CyclicBarrier只能唤起一个任务，CountDownLatch可以唤起多个任务

CyclicBarrier可重用，CountDownLatch不可重用，计数值为0该CountDownLatch就不可再用了。

关于volatile关键字

可以创建Volatile数组吗?

Java 中可以创建 volatile类型数组，不过只是一个指向数组的引用，而不是整个数组。如果改变引用指向的数组，将会受到volatile 的保护，但是如果多个线程同时改变数组的元素，volatile标示符就不能起到之前的保护作用了。

volatile能使得一个非原子操作变成原子操作吗?

一个典型的例子是在类中有一个 long 类型的成员变量。如果你知道该成员变量会被多个线程访问，如计数器、价格等，你最好是将其设置为 volatile。为什么？因为 Java 中读取 long 类型变量不是原子的，需要分成两步，如果一个线程正在修改该 long 变量的值，另一个线程可能只能看到该值的一半（前 32 位）。但是对一个 volatile 型的 long 或 double 变量的读写是原子。

一种实践是用 volatile 修饰 long 和 double 变量，使其能按原子类型来读写。double 和 long 都是64位宽，因此对这两种类型的读是分为两部分的，第一次读取第一个 32 位，然后再读剩下的 32 位，这个过程不是原子的，但 Java 中 volatile 型的 long 或 double 变量的读写是原子的。volatile 修复符的另一个作用是提供内存屏障（memory barrier），例如在分布式框架中的应用。简单的说，就是当你写一个 volatile 变量之前，Java 内存模型会插入一个写屏障（write barrier），读一个 volatile 变量之前，会插入一个读屏障（read barrier）。意思就是说，在你写一个 volatile 域时，能保证任何线程都能看到你写的值，同时，在写之前，也能保证任何数值的更新对所有线程是可见的，因为内存屏障会将其他所有写的值更新到缓存。

volatile类型变量提供什么保证?

volatile 主要有两方面的作用:1.避免指令重排2.可见性保证.例如，JVM 或者 JIT为了获得更好的性能会对语句重排序，但是 volatile 类型变量即使在没有同步块的情况下赋值也不会与其他语句重排序。 volatile 提供 happens-before 的保证，确保一个线程的修改能对其他线程是可见的。某些情况下，volatile 还能提供原子性，如读 64 位数据类型，像 long 和 double 都不是原子的(低32位和高32位)，但 volatile 类型的 double 和 long 就是原子的。

# Socket

## 网络编程概述

**Java网络编程**

Java是 Internet 上的语言，它从语言级上提供了对网络应用程序的支持，程序员能够很容易开发常见的网络应用程序。

Java提供的网络类库，可以实现无痛的网络连接，联网的底层细节被隐藏在 Java 的本机安装系统里，由 JVM 进行控制。并且 Java 实现了一个跨平台的网络库，程序员面对的是一个统一的网络编程环境。

**计算机网络**

把分布在不同地理区域的计算机与专门的外部设备用通信线路互连成一个规模大、功能强的网络系统，从而使众多的计算机可以方便地互相传递信息、共享硬件、软件、数据信息等资源。

**网络编程的目的**

直接或间接的通过网络协议与其他计算机进行通讯。

**网络编程主要解决的问题**

如何准确地定位网络上的一台或多台主机和找到主机后如何可靠高效的进行数据传输。

## 通讯要素

**通信双方地址**

**一定的通信规则（协议）**

**OSI参考模型：**即Open System Interconnection，过于理想化，未能广泛推广

**TCP/IP参考模型：**实际上的国际标准

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **OSI参考模型** | **TCP/IP参考模型** | **TCP/IP参考模型各层对应协议** | **数据传输方式** | |
| 应用层 | 应用层 | HTTP、ftp、telnet、DNS… | 数据封装  ∣Segment  ∣Packet  ∣Frame  ↓Bits | 数据拆封  ↑Segment  ∣Packet  ∣Frame  ∣Bits |
| 表示层 |
| 会话层 |
| 传输层 | 传输层 | TCP、UDP… |
| 网络层 | 网络层 | IP、ICMP、ARP… |
| 数据链路层 | 物理+数据链路层 | Link | -------------------🡪  🡨------------------ | |
| 物理层 |

**物理层和数据链路层**涉及物理介质访问和二进制数据流传输。

**网络层**的主要协议有IP（Internet protocol）、ICMP（Internet Control Message Protocol，互联网控制报文协议）、IGMP（Internet Group Management Protocol，互联网组管理协议）、ARP（Address Resolution Protocol，地址解析协议）和RARP（Reverse Address Resolution Protocol，反向地址解析协议）等，涉及寻址和路由选择。

**传输层**的基本功能是为两台主机间的应用程序提供端到端的通信。传输层从应用层接受数据，并且在必要的时候把它分成较小的单元，传递给网络层，并确保到达对方的各段信息正确无误。

**应用层**提供应用程序的网络接口。

### IP和端口号

#### IP地址

位于网络中的计算机具有唯一的IP地址（**InetAddress**），用于唯一的标识Internet上的计算机，这样不同的主机可以互相区分

本地回环地址（hostAddress）：127.0.0.1

主机名（hostName）：localHost

#### 端口号

端口号被规定为一个16位的整数，范围为0~65535，用于标识计算机上运行的进程，不同的进程有不同的端口号。其中，0~1024被预先定义的服务通信占用（如http占用端口80，Tomcat占用端口8080，MySql占用端口3306,Oracle占用端口1521等）。除非我们需要访问这些特定服务，否则，就应该使用 1024~65535 这些端口中的某一个进行通信，以免发生端口冲突。

**端口号与IP地址的组合得出一个网络套接字：Socket。**

### 网络通信协议

网络通信协议

计算机网络中实现通信必须有一些约定，即通信协议，对速率、传输代码、代码结构、传输控制步骤、出错控制等制定标准。

通信协议分层的思想

由于结点之间联系很复杂，在制定协议时，把复杂成份分解成一些简单的成份，再将它们复合起来。最常用的复合方式是层次方式，即同层间可以通信、上一层可以调用下一层，而与再下一层不发生关系。各层互不影响，利于系统的开发和扩展。

传输层协议中有两个非常重要的协议

传输控制协议TCP(Transmission Control Protocol)

用户数据报协议UDP(User Datagram Protocol)。

TCP/IP 以其两个主要协议：传输控制协议(TCP)和网络互联协议(IP)而得名，实际上是一组协议，包括多个具有不同功能且互为关联的协议。

IP(Internet Protocol)协议是网络层的主要协议，支持网间互连的数据通信。

TCP/IP协议模型从更实用的角度出发，形成了高效的四层体系结构，即物理链路层、网络层、传输层和应用层。

#### TCP协议

TCP协议用于实现面向连接的会话。

使用TCP协议前，须先建立TCP连接，形成传输数据通道

传输前，采用“三次握手”方式，是可靠的

TCP协议进行通信的两个应用进程：客户端、服务端

在连接中可进行大数据量的传输

传输完毕，需释放已建立的连接，效率低

#### UDP协议

将数据、源、目的封装成数据包，不需要建立连接

每个数据报的大小限制在64K内

因无需连接，故是不可靠的

发送数据结束时无需释放资源，速度快

## InetAddress类

**java 中ip对象：InetAddress.**

Java 中有关网络方面的功能都定义在 java.net 程序包中。Java 用 InetAddress 对象表示 IP 地址，该对象里有两个字段：主机名(String) 和 IP 地址(int)。

Internet上的主机有两种方式表示地址：

域名(hostName)：www.atguigu.com

IP 地址(hostAddress)：202.108.35.210

InetAddress类主要表示IP地址，两个子类：Inet4Address、Inet6Address。

InetAddress类对象含有一个Interne主机地址的域名和IP地址，如：www.atguigu.com 和 202.108.35.210。

连接网络输入一个域名后，先寻找位于C:\Windows\System32\drivers\etc目录下的hosts文件中是否存在该域名地址，如果已存在，则跳过DNS服务器直接和主机建立连接；如果不存在，再通过DNS，将域名转化为IP地址（该行为称之为**域名解析**），进而和主机建立连接。

|  |  |
| --- | --- |
| **InetAddress类方法摘要** | |
| static InetAddress | **getByName**(String host)在指定主机名的情况下确定主机的IP地址 |
| static InetAddress | **getbyAddress**(byte[] addr)在指定原始IP地址的情况下返回InetAddress对象 |
| String | **getHostAddress**()返回IP地址字符串 |
| String | **getHostName**()返回此IP地址的主机名 |
| static InetAddress | **getLocalHost**()返回本地主机 |
| boolean | **isReachable**(int timeout)测试是否可以到达该地址 |
| String | **getCanonicalHostName**()获取此IP地址的完全限定域名 |

|  |
| --- |
| **public** **class** InetAdressTest {  **public** **static** **void** main(String[] args) **throws** UnknownHostException{  //通过名称(ip字符串or主机名)来获取一个ip对象，可能会抛出java.net.UnknownHostException异常  InetAddress ip = InetAddress.*getByName*("www.baidu.com");  System.***out***.println("addr:"+ip.getHostAddress());  System.***out***.println("name:"+ip.getHostName());  }  } |

## Socket编程

客户端－服务器是一种最常见的网络应用程序模型。服务器是一个为其客户端提供某种特定服务的硬件或软件。客户机是一个用户应用程序，用于访问某台服务器提供的服务。端口号是对一个服务的访问场所，它用于区分同一物理计算机上的多个服务。套接字用于连接客户端和服务器，客户端和服务器之间的每个通信会话使用一个不同的套接字。

利用套接字(**Socket**)开发网络应用程序早已被广泛的采用，以至于成为事实上的标准。通信的两端都要有Socket作为两台机器间通信的端点，网络通信其实就是Socket间的通信。Socket允许程序把网络连接当成一个流，数据在两个Socket间通过IO传输。一般主动发起通信的应用程序属客户端，等待通信请求的 为服务端

|  |  |
| --- | --- |
| **Socket类方法摘要** | |
| InetAddress | **getLocalAdress**()返回对方Socket中的IP的InetAddress对象 |
| InetAddress | **getInetAdress**()返回对方Socket中IP地址 |
| int | **getLocalPort**()返回本地Socket中的端口号 |
| int | **getPort**()返回对方Socket中的端口号 |
| void | **close**()关闭Socket，不可在以后的网络连接中使用，除非创建新的套接字 |
| InputStream | **getInputStream**()获取与Socket相关联的字节输入流，用于从Socket中读数据 |
| OutputStream | **getOutStream**()获取与Socket相关联的字节输出流，用于向Socket中写数据 |

|  |  |
| --- | --- |
| **ServerSocket类方法摘要** | |
| InetAddress | **getInetAdress**()返回此服务器套接字的本地地址 |
| Socket | **accept**()等待客户端的连接请求，返回与该客户端进行通信用的Socket对象 |
| void | **setSoTimeout**(int timeout) 设置accept()方法等待连接的时间为timeout毫秒。若时间已到，还没有客户端连接，则抛出InterruptedIOException异常，accept()方法不再阻塞，该倾听Socket可继续使用。若timeout值为0，则表示accept()永远等待。该方法必须在倾听Socket创建后，在accept()之前调用才有效 |
| void | **close**()关闭监听Socket |
| int | **getLocalPort**()返回此套接字在其上监听的端口号 |
| SocketAddress | **getLocalSocketAddress**()返回此套接字绑定的端点的地址 |

类 Socket 和 ServerSocket 实现了基于TCP协议的客户端－服务器程序。Socket是客户端和服务器之间的一个连接，连接创建的细节被隐藏了。这个连接提供了一个安全的数据传输通道，这是因为 TCP 协议可以解决数据在传送过程中的丢失、损坏、重复、乱序以及网络拥挤等问题，它保证数据可靠的传送。

### TCP网络编程

**TCP传输：**两个端点的建立连接后会有一个传输数据的通道，这通道称为流，而且是建立在网络基础上的流，称之为socket流。该流中既有读取，也有写入。

**tcp的两个端点：**一个是客户端，一个是服务端。

客户端：对应的对象，Socket

服务端：对应的对象，ServerSocket

**客户端Socket的工作过程包含以下四个基本的步骤**

创建 Socket：根据指定服务端的 IP 地址或端口号构造Socket类对象。若服务器端响应，则建立客户端到服务器的通信线路。若连接失败，会出现异常。

打开连接到Socket的输入/出流：使用getInputStream()方法获得输入流，使用getOutputStream()方法获得输出流，进行数据传输

按照一定的协议对Socket进行读/写操作：通过输入流读取服务器放入线路的信息（但不能读取自己放入线路的信息），通过输出流将信息写入线程。

关闭Socket：断开客户端到服务器的连接，释放线路

#### ServerSocket类

该类用于表示服务端，实例化时需要指定端口号。

|  |  |
| --- | --- |
| **ServerSocket类方法摘要** | |
| Socket | **accept**()接收Socket连接，阻塞式方法 |
| InetAddress | **getInetAddress**()获取服务器的IP地址 |

#### Socket类

该类用于表示客户端，实例化时需要指定服务器IP地址和端口号。

|  |  |
| --- | --- |
| **Socket类方法摘要** | |
| int | **getPort**()获取与服务器连接的端口号 |
| InetAddress | **getLocalAddress**()获取本地IP地址并将其封装为InetAddress类型对象返回 |
| void | **close**()关闭Socket连接，结束本次通信。（关闭连接前应释放所有资源） |
| InputStream | **getInputStream**()获取输入流对象，如果是服务器端返回的则读取客户端发送的数据，反之则读取服务端发送的数据。 |
| OutputStream | **getOutputStream**()获取输出流对象，如果是服务器端返回的则读取客户端发送的数据，反之则读取服务端发送的数据。 |

#### TCP客户端

1，建立tcp的socket服务，最好明确具体的地址和端口。这个对象在创建时，就已经可以对指定ip和端口进行连接(三次握手)。

2，如果连接成功，就意味着通道建立了，socket流就已经产生了。只要获取到socket流中的读取流和写入流即可，只要通过getInputStream和getOutputStream就可以获取两个流对象。

3，关闭资源。

|  |
| --- |
| **public** **void** client() **throws** Exception, Exception {  /\*  \* 创建一个客户端 第一个参数 ：InetAddress.getLocalHost()服务器的地址 第二参数 ： 端口号  \*/  Socket socket = **new** Socket(InetAddress.*getLocalHost*(), 6666);  // 获取向服务器发送数据的流  OutputStream os = socket.getOutputStream();  // 向服务器端发送数据  os.write("abcdef".getBytes());  socket.shutdownOutput(); //通知服务器数据已经发送完毕  //获取读取服务器数据的流  InputStream is = socket.getInputStream();  // 接收数据  **byte**[] b = **new** **byte**[1024];  **int** len = 0;  **while** ((len = is.read(b)) != -1) {  System.***out***.println("客户端 ： " + **new** String(b, 0, len));  }  // 关闭流  is.close();  os.close();  socket.close();  } |

服务器程序的工作过程包含以下四个基本的步骤

调用ServerSocket(int port)：创建一个服务器端套接字，并绑定到指定端口上。用于监听客户端的请求。

调用accept()：监听连接请求，如果客户端请求连接，则接受连接，返回通信套接字对象。

调用该Socket类对象的getOutputStream()和getInputStream ()：获取输出流和输入流，开始网络数据的发送和接收。

关闭ServerSocket和Socket对象：客户端访问结束，关闭通信套接字。

#### TCP服务端

1，创建服务端socket服务，并监听一个端口。

2，服务端为了给客户端提供服务，获取客户端的内容，可以通过accept方法获取连接过来的客户端对象。

3，可以通过获取到的socket对象中的socket流和具体的客户端进行通讯。

4，如果通讯结束，关闭资源。注意：要先关客户端，再关服务端。

|  |
| --- |
| **public** **void** server() **throws** Exception {  // 创建一个服务器端 : 参数：端口号  ServerSocket serverSocket = **new** ServerSocket(6666);  // 接收客户端的请求  Socket socket = serverSocket.accept();  // 读取客户端发送的内容的流  InputStream is = socket.getInputStream();  // 接收数据  **byte**[] b = **new** **byte**[1024];  **int** len = 0;  **while** ((len = is.read(b)) != -1) {  System.***out***.println("服务器 ： " + **new** String(b, 0, len));  }  // 获取给客户端写数据的流  OutputStream os = socket.getOutputStream();  // 写内容  os.write("cccc".getBytes());  socket.shutdownOutput();//通知客户端数据已经发送完毕  // 关闭资源  os.close();  is.close();  //如果通讯结束，关闭资源。注意：要先关客户端，在关服务端  socket.close();  serverSocket.close();  } |

### UDP网络编程

只要是网络传输，必须有socket 。数据一定要封装到数据包中，数据包中包括目的地址、端口、数据等信息。因为无法直接操作UDP，所以要将UDP封装成对象，这个对象就是DatagramSocket，封装了UDP协议的socket对象。

类 DatagramSocket 和 DatagramPacket 实现了基于 UDP 协议网络程序。

UDP数据报通过数据报套接字 DatagramSocket 发送和接收，系统不保证UDP数据报一定能够安全送到目的地，也不能确定什么时候可以抵达。

DatagramPacket 对象封装了UDP数据报，在数据报中包含了发送端的IP地址和端口号以及接收端的IP地址和端口号。

UDP协议中每个数据报都给出了完整的地址信息，因此无须建立发送方和接收方的连接

因为数据包中包含的信息较多，为了操作这些信息方便，也一样会将其封装成对象。这个数据包对象就是：DatagramPacket.通过这个对象中的方法，就可以获取到数据包中的各种信息。

DatagramSocket具备发送和接受功能，在进行udp传输时，需要明确一个是发送端，一个是接收端，发送端与接收端是两个独立的运行程序。利用DatagramSocket建立发送端和接收端来传输数据，利用DatagramPacket建立数据包封装数据。

#### DatagramSocket类

该类的作用类似于码头，其实例对象可以发送和接收DatagramPacket数据包。其发送端构造器可以不用端口号作为参数，此时系统会默认分配一个没有被其他网络程序使用的端口号；也可以手动传入一个指定的端口号，用来接收发送到该端口号的数据。其接收端构造器则必须要指定端口号，用于指定发送数据的方向。

|  |  |
| --- | --- |
| **DatagramSocket类方法摘要** | |
| void | **receive**(DatagramPacket p)接收数据包 |
| void | **send**(DatagramPacket p)发送数据包 |

#### DatagramPacket类

该类的实例对象相当于一个集装箱，用于封装UDP通信中发送或者接收的数据。其接收端构造器需要接收一个字节数组及发送的长度，而发送端的构造器不但需要接收字节数组、发送起始位置和发送数据长度，还要指定发送端IP地址和端口号。

|  |  |
| --- | --- |
| **DatagramPacket类方法摘要** | |
| InetAddress | **getAdress**()获取发送端或接收端的IP地址 |
| int | **getPort**()获取发送端或接收端的端口号 |
| byte[] | **getData**()获取数据缓冲区 |
| int | **getLength**()获取发送或接收到的数据长度 |

#### udp接收端

1，创建UDP的Socket服务，即DatagramSocket对象，必须要明确一个端口，只有发送到这个端口的数据才会被接收

2，定义数据包，即DatagramPacket对象，用于存储接收到数据

3，通过Socket服务的接收方法将收到的数据存储到数据包中

4，通过数据包的方法获取数据包中的具体数据内容，比如ip、端口、数据等等

5，关闭资源

|  |
| --- |
| **public** **class** UDPReceTest {  **public** **static** **void** main(String[] args) **throws** Exception{  //1，创建udp的socket服务，指定一个端口，只有发送到这个端口的数据才会被接收  DatagramSocket ds = **new** DatagramSocket(10123);  //2，定义数据包，用于存储接收到数据。先定义字节数组，数据包会把数据存储到字节数组中。  **byte**[] buf = **new** **byte**[1024];  /\*  \* 第一个参数 ： 用来接收数据的组组  \* 第二个参数 ：可以使用数组的多大长度来接收  \*/  DatagramPacket dp = **new** DatagramPacket(buf,buf.length);  //3，通过socket服务的接收方法将收到的数据存储到数据包中。  ds.receive(dp);//该方法是阻塞式方法。  //4，通过数据包的方法获取数据包中的具体数据内容，比如ip，端口，数据等等。  String ip = dp.getAddress().getHostAddress();  **int** port = dp.getPort();  //dp.getLength() : 接收的内容的长度  String text = **new** String(dp.getData(),0,dp.getLength());//将字节数组中的有效部分转成字符串。  System.***out***.println(ip+":"+port+"--"+text);  //5，关闭资源。  ds.close();  }  } |

#### udp发送端

1，建立UDP的Socket服务，即创建DatagramSocket对象，创建时若没有明确端口，系统会自动分配一个未被使用的端口

2，明确要发送的具体数据

3，将数据封装成了数据包，即创建DatagramPacket对象

4，用socket服务的send方法将数据包发送出去

5，关闭资源

|  |
| --- |
| **public** **class** UDPSendTest {  **public** **static** **void** main(String[] args) **throws** Exception {  //1，建立udp的socket服务。  DatagramSocket ds = **new** DatagramSocket(8888);//指定发送端口，不指定系统会随机分配。  //2，明确要发送的具体数据。  String text = "udp传输演示哥们来了";  **byte**[] buf = text.getBytes();  //3，将数据封装成了数据包。  /\*  \* DatagramPacket(byte buf[], int offset, int length, InetAddress address, int port)  \* 第一个参数 ：要发送的内容  \* 第二个参数 ：从数组中的哪个位置开始发送  \* 第三个参数 ：从数组中的哪个位置开始，内容的长度为length  \* 第四个参数 ：接收数据的地址  \* 第五个参数 ：端口号  \*/  DatagramPacket dp = **new** DatagramPacket(buf,buf.length,InetAddress.*getByName*("localHost"),10123);  //4，用socket服务的send方法将数据包发送出去。  ds.send(dp);  //5，关闭资源。  ds.close();  }  } |

## URL编程

URL(Uniform Resource Locator)：统一资源定位符，它表示 Internet 上某一资源的地址。通过 URL 我们可以访问 Internet 上的各种网络资源，比如最常见的 www，ftp 站点。浏览器通过解析给定的 URL 可以在网络上查找相应的文件或其他资源。

### URL的基本结构

<传输协议>://<主机名>:<端口号>/<文件名>

例如: <http://192.168.1.100:8080/helloworld/index.jsp>

http://127.0.0.1:8080/examples/hello.txt

### URL类

为了表示URL，java.net中实现了类URL，通过其构造器来初始化一个URL对象。

|  |
| --- |
| **URL类构造器摘要** |
| URL(String spec)通过一个表示URL地址的字符串可以构造一个下URL对象。例如：URL url = new URL ("http://www.atguigu.com/"); |
| URL(URL context,String spec)：通过基URL和相对URL构造一个URL对象。例如：URL downloadUrl = new URL(url, “download.html") |
| URL(String protocol,String host,String file);例如：new URL("http", "www.atguigu.com", “download. html"); |
| URL(String protocol,String host,int port,String file); 例如:URL gamelan = new URL("http", "www.atguigu.com", 80,“download.html"); |

类URL的构造方法都是public修饰且声明抛出非运行时异常，必须要对这一异常进行处理，通常是用 try-catch 语句进行捕获。一个URL对象生成后，其属性是不能被改变的，但可以通过它给定的方法来获取这些属性。

|  |  |
| --- | --- |
| **URL类方法摘要** | |
| String | getProtocol()获取该URL的协议名 |
| String | getHost()获取该URL的主机名 |
| String | getPort()获取该URL的端口号 |
| String | getPath()获取该URL的文件路径 |
| String | getFile()获取该URL的文件名 |
| String | getQuery()获取该URL的查询名 |
| InputStreams | openStream()能从网络上读取数据 |

### URLConnection类

#### URLConnection类简介

若希望输出数据，例如向服务器端的CGI程序发送一些数据，则必须先与URL建立连接，然后才能对其进行读写，此时需要使用 URLConnection 。

针对HTTP协议的URLConnection类，表示到URL所引用的远程对象的连接。当与一个URL建立连接时，首先要在一个 URL 对象上通过方法 openConnection() 生成对应的 URLConnection 对象。如果连接过程失败，将产生IOException。

通过URLConnection对象获取的输入流和输出流，即可以与现有的CGI程序进行交互。

公共网关接口CGI（**Common Gateway Interface**）：是用户浏览器和服务器端的应用程序进行连接的接口。

#### 获取URLConnection对象的两种方式

通过URLConnection类的构造器

protected [URLConnection](mk:@MSITStore:D:\Eclipse\JDK_API_1_6_zh_CN.CHM::/java/net/URLConnection.html#URLConnection(java.net.URL))([URL](mk:@MSITStore:D:\Eclipse\JDK_API_1_6_zh_CN.CHM::/java/net/URL.html) url)构造一个到指定 URL 的 URL 连接

通过URL类的openConnection()方法

URL netchinaren = new URL ("http://www.atguigu.com/index.shtml");

URLConnectonn u = netchinaren.openConnection( );

URLConnection类

|  |  |
| --- | --- |
| **URLConnection类方法摘要** | |
| Object | getContent()获取该URL的协议名 |
| int | getContentLength()返回content-length头字段的值 |
| String | getContentType()获取content-type头字段的值 |
| long | getDate()返回date头字段的值 |
| long | getLastModified()返回last-modified头字段的值 |
| InputStream | getInputStream()返回从此打开的连接读取的输入流 |
| OutputSteram | getOutputStream()返回写入到此连接的输出流 |

类URL和URLConnection提供了最高级网络应用。URL网络资源的位置来统一表示Internet上各种网络资源。通过URL对象可以创建当前应用程序和URL表示的网络资源之间的连接，这样当前程序就可以读取网络资源数据，或者把自己的数据传送到网络上去。

|  |
| --- |
| **public** **class** URLTest {  **public** **static** **void** main(String[] args) **throws** Exception {  URL url = **new** URL("http://192.168.10.23:8080/txt/123.txt");  //打开连接 (多态)  URLConnection openConnection = url.openConnection();  //向下转型  HttpURLConnection conn = (HttpURLConnection) openConnection;  conn.connect(); //连接服务器  //访问服务器的响应码  **int** responseCode = conn.getResponseCode();  **if**(responseCode == 200){//访问服务器成功  FileOutputStream fos = **new** FileOutputStream("123.txt");  //读取内容  InputStream is = conn.getInputStream();  **byte**[] b = **new** **byte**[1024];  **int** len = 0;  **while**((len = is.read(b)) != -1){  fos.write(b,0,len);  }  is.close();  fos.close();  }  }  } |

# Reflect

## 反射机制概述

反射（**Reflection**）其实就是动态加载（被视为动态语言的关键）一个指定的类，并获取该类中的所有信息。将字节码文件封装成对象，并将字节码文件中的内容都封装成对象，这样便于操作这些成员。简单说：**反射技术可以对一个类进行解剖。反射的好处**在于大大的增强了程序的扩展性。

**反射相关的主要API**

java.lang.Class代表一个类

java.lang.reflect.Method代表类的方法

java.lang.reflect.Field代表类的属性

java.lang.reflect.Constructor代表类的构造方法

……

反射的基本步骤

获得Class对象，就是获取到指定的名称的字节码文件对象；

实例化对象，获得类的属性、方法或构造函数；

访问属性、调用方法、调用构造函数创建对象。

**Java反射机制的作用**

反射就是动态加载对象，并对对象进行剖析。Java反射机制的作用：

1、在运行时判断任意一个对象所属的类

2、在运行时构造任意一个类的对象

3、在运行时判断任意一个类所具有的成员变量和方法

4、在运行时调用任意一个对象的成员变量和方法

5、，生成动态代理

## Class类及对象

Class是反射的源头，其本身也是一个类，其对象只能由系统建立，一个Class对象对应的是一个加载到JVM中的一个.class文件，一个类在 JVM 中只会有一个Class对象，通过Class可以完整地得到一个类中的完整结构 。

**运行时对象**

加载字节码文件时，内存中加载的字节码文件（存放于方法区中）就是Class对象，即运行时对象，该字节码文件所对应的那个类叫做运行时类。

字节码文件只加载一次，方法区中如果已经存在运行时类，那么就不会再加载该类所对应的字节码文件。

Class类的对象包含了某个被加载类的结构，一个被加载类对应一个Class对象。当一个class被加载或类加载器（classloader）的defineClass()被JVM调用，JVM便自动产生一个Class对象。

### 获取运行时类的对象

1. 通过类名.class获取

前提：已知具体的类

优点：最安全可靠，程序性能最高

缺点：必须要先明确该类

1. 通过对象名.getClass();获取，该方法是Object中定义的final修饰的方法，因此所有类都有该方法且不可被重写。

前提：已获取类的对象

缺点：必须要创建该类对象，才可以调用getClass方法

前两种方式不利于程序的扩展，因为都需要在程序使用具体的类来完成

1. 通过Class.forName(“全类名”);获取

前提：已知类的全类名

优点：这种方式扩展性最强，因此用的最多

1. 通过类加载器Class clazz = 任意对象.getClass().getClassLoader().loadClass(“全类名”)

|  |
| --- |
| **public** **class** ReflectionTest {  @SuppressWarnings({ "rawtypes", "unused" })  **public** **static** **void** main(String[] args) **throws** ClassNotFoundException {  Person person = **new** Person();  Class class1 = Person.**class**;  Class class2 = person.getClass();  Class class3 = Class.*forName*("test509.Person");  Class class4 = **new** ReflectionTest().getClass().getClassLoader().loadClass("test509.Person");  }  } |

### 创建类的对象

#### 通过无参构造器

通过运行时对象.newInstance()方法获取类的对象，要求该类必须有一个无参的构造器和构造器的访问权限必须足够。

|  |
| --- |
| **public** **class** ReflectionTest {  **public** **static** **void** main(String[] args) **throws** Exception {  Class<?> clazz = Class.*forName*("test509.Person");  Object object = clazz.newInstance();  Person person2 = (Person) object;  System.***out***.println(person2.name);  }  } |

#### 通过有参构造器

通过Class类的getDecalaredConstructor(Class...parameterTypes)取得本类的指定形参类型的构造器，再向构造器的形参中传递一个对象数组，里面包含了构造器所需要的各个参数。

1、调用指定参数结构的构造器，生成Constructor的实例（该方法只能调用public修饰的构造器，其他权限修饰符修饰的构造器需要使用getDeclaredConstructor()方法调用）

Constructor con = clazz.getConstructor(String.class,Integer.class);

2、通过Constructor的实例创建对应类的对象，并初始化类属性

Person p2 = (Person)con.newInstance("Peter",20);

|  |
| --- |
| **public** **void** method2() **throws** Exception {  //通过全类名获取运行时对象  Class<?> clazz = Class.*forName*("test509.Person2");  //该方法返回指定参数的构造器，没有该构造器则会报NoSuchMethodException异常  Constructor<?> constructor = clazz.getDeclaredConstructor(String.**class**,**int**.**class**);  //使用构造器类的对象的方法创建Person2类的对象  Person2 person2 = (Person2) constructor.newInstance("xmm",26);  System.***out***.println(person2.name +" " + person2.age);  } |

这种方式必须要先明确具体的构造器的参数类型，不便于扩展。所以一般情况下，被反射的类，内部通常都会提供一个公有空参构造器。

## 反射的使用

### 获取运行时类的完整结构

**FIled、Method、Constructor、SuperClass、Interface、Annotation**

#### 获取并调用属性

获取属性信息

|  |  |
| --- | --- |
| **Class类方法摘要** | |
| Field | **getField**(String name) 获取Class对象所表示的类或接口的指定public属性 |
| Field [] | **getFields**()获取Class对象所表示的类或接口的public的属性 |
| Field | **getDeclaredField**(String name)获取Class对象所表示的类或接口的指定属性，包括private、default、protected和public修饰的属性 |
| Field [] | **getDeclaredFields**()获取Class对象所表示的类或接口的全部属性 |

|  |
| --- |
| **public** **void** method1() **throws** Exception {  Class<?> clazz = Class.*forName*("test509.Person1");  //通过属性名获取对应属性，如果该属性不存在则会抛出异常  Field age = clazz.getDeclaredField("sex");  System.***out***.println(age);  //获取所有属性并返回属性数组  Field[] fields = clazz.getDeclaredFields();  **for** (Field field : fields) {  System.***out***.println(field);  }  } |
| private int test509.Person1.age  java.lang.String test509.Person1.name  private int test509.Person1.age |

|  |  |
| --- | --- |
| **Field类方法摘要** | |
| <T extends Annotation>T | **getAnnotation**<Class<T> annotationClass)获取属性的注解 |
| int | **getModifiers**()以整数形式返回此属性的修饰符 |
| Class<?> | **getType**()得到属性的属性类型 |
| String | **getName**()返回属性的名称 |
| Object | **get**(Object obj) 取得指定对象obj上此Field的属性内容 |
| void | **set**(Object obj,Object value) 设置指定对象obj上此Field的属性内容 |
| void | **setAccessible**(true)访问私有属性时，让这个属性可见 |

|  |
| --- |
| **public** **void** method2() **throws** Exception {  Class<?> clazz = Class.*forName*("test509.Person1");  Field[] fields = clazz.getDeclaredFields();  **for** (Field field : fields) {  //获取属性的注解，返回注解集合  Annotation[] annotations = field.getAnnotations();  System.***out***.println(Arrays.*toString*(annotations));  //获取属性的权限修饰符，返回int型数值，可以通过Modifier类的toString方法转换为String  **int** modifiers = field.getModifiers();  System.***out***.print(Modifier.*toString*(modifiers) + " ");  //获取属性的类型  Class<?> type = field.getType();  System.***out***.print(type + " ");  //获取属性的名称  String name = field.getName();  System.***out***.println(name);  }  } |
| [@test509.MyAnn(name=name)]  public class java.lang.String name  []  private int age |

调用属性

在反射机制中，可以直接通过Field类操作类中的属性，通过Field类提供的set()和get()方法就可以完成设置和取得属性内容的操作。

在类中属性设置为private的前提下，使用set()和get()方法时，首先要使用Field类中的setAccessible(true)方法将需要操作的属性设置为可以被外部访问。

|  |
| --- |
| **public** **void** method3() **throws** Exception {  Class<?> clazz = Class.*forName*("test509.Person1");  Person1 person = (Person1) clazz.newInstance();  Field[] fields = clazz.getDeclaredFields();  **for** (Field field : fields) {  **if**("class java.lang.String".equals(field.getType().toString())) {  //设置person对象的该属性为Smith  field.set(person, "Smith");  //获取person属性的值  }**else** **if**("int".equals(field.getType().toString())) {  //设置private属性可见，否则无法调用和修改  field.setAccessible(**true**);  //修改person属性age的值为30  field.setInt(person, 30);  }  System.***out***.println(field.get(person));  }  } |
| 30  Smith |

#### 获取并调用方法

获取方法信息

|  |  |
| --- | --- |
| **Class类方法摘要** | |
| Method | **getMethod**(String name,Class<?>... parameterTypes) 获取Class对象所表示的类或接口的指定public方法 |
| Method[] | **getMethods**()获取Class对象所表示的类或接口的public方法 |
| Method | **getDeclaredMethod**(String name,Class<?>... parameterTypes)获取Class对象所表示的类或接口的指定方法，包括private、default、protected和public修饰的方法 |
| Method[] | **getDeclaredMethods**()获取Class对象所表示的类或接口的所有方法 |

|  |
| --- |
| **public** **void** method() **throws** Exception {  Class<?> clazz = Class.*forName*("test509.Person1");  //通过方法名获取指定方法，如果方法有形参，则传入形参的运行时对象  Method method = clazz.getDeclaredMethod("method1",String.**class**);  System.***out***.println(method);  //返回所有的方法，包括非public修饰的方法  Method[] methods = clazz.getDeclaredMethods();  **for** (Method method2 : methods) {  System.***out***.println(method2);  }  } |
| public void test509.Person1.method1(java.lang.String)  public void test509.Person1.method1(java.lang.String)  private void test509.Person1.method2() throws java.lang.Exception |

|  |  |
| --- | --- |
| **Method类方法摘要** | |
| <T extends [Annotation](mk:@MSITStore:D:\Eclipse\JDK_API_1_6_zh_CN.CHM::/java/lang/annotation/Annotation.html)>T | **getReturnType**()获取全部的返回值 |
| int | **getModifiers**()获取方法权限修饰符 |
| Class<?> | **getReturnType**()获取全部的返回值 |
| String | **getName**()返回方法名称 |
| Class<?>[] | **getParameterTypes**()获取全部的参数 |
| Class<?>[] | **getExceptionTypes**()获取方法的异常信息 |
| String | **toGenericString**()以String形式返回方法的全部信息 |
| Object | **invoke**(Object obj,Object...args)通过对象obj调用该方法，并返回该方法的返回值 |

getModifiers()方法的返回值有1、2、4、0等多种，0表示default；1表示public；2表示private；4表示protected；10表示private static。

|  |
| --- |
| **public** **void** method2() **throws** Exception {  Class<?> clazz = Class.*forName*("test509.Person1");  Method[] methods = clazz.getDeclaredMethods();  **for** (Method method2 : methods) {  //获取方法的注解，返回注解数组  Annotation[] annotations = method2.getAnnotations();  System.***out***.println(Arrays.*toString*(annotations));  //获取方法的权限修饰符，返回int型，可以使用Modifier的静态toString方法转换为String型  **int** perm = method2.getModifiers();  System.***out***.print(Modifier.*toString*(perm)+" ");  //获取方法的返回值类型  Class<?> returnType = method2.getReturnType();  System.***out***.print(returnType + " ");  //获取方法名称  String name = method2.getName();  System.***out***.print(name + " ");  //获取方法的参数类型，返回参数数组  Class<?>[] parameterTypes = method2.getParameterTypes();  System.***out***.print("("+Arrays.*toString*(parameterTypes)+") ");  //获取方法的异常信息，返回异常数组  Class<?>[] exceptionTypes = method2.getExceptionTypes();  System.***out***.println(Arrays.*toString*(exceptionTypes));  }  } |
| [@test509.MyAnn(name=xmm)]  public void method1 ([class java.lang.String]) []  [@test509.MyAnn(name=cap)]  private void method2 ([]) [class java.lang.Exception] |

调用方法

获取方法之后使用Object invoke(Object obj, Object[] args)进行调用，并向方法中传递要设置的obj对象的参数信息。

1、Object 对应原方法的返回值，若原方法无返回值，此时返回null

2、原方法若为静态方法，此时形参Object obj可为null

3、原方法形参列表为空，则Object[] args省略不写

4、原方法声明为private,则需要在调用invoke()方法前调用方法对象的setAccessible(true)方法，才可访问private的方法。

|  |
| --- |
| **public** **void** method3() **throws** Exception {  Class<?> clazz = Class.*forName*("test509.Person1");  Person1 person = (Person1) clazz.newInstance();  Method[] methods = clazz.getDeclaredMethods();  **for** (Method method : methods) {  **if**("private".equals(Modifier.*toString*(method.getModifiers()))) {  //如果要调用的方法权限修饰符为private，则需要调用该方法并传入参数true  method.setAccessible(**true**);  }  **if**(method.getParameterTypes().length == 0) {  //调用空参方法，传入调用该方法的对象即可  method.invoke(person);  }**else** {  //调用有参方法，传入调用该方法的对象和形参  method.invoke(person,"ABC");  }  }  } |
| Person2 private method  Person1 public method ABC |

#### 获取构造器

|  |  |
| --- | --- |
| **Class类方法摘要** | |
| Constructor<T> | [**getConstructor**](mk:@MSITStore:D:\Eclipse\JDK_API_1_6_zh_CN.CHM::/java/lang/Class.html#getConstructor(java.lang.Class...))([Class](mk:@MSITStore:D:\Eclipse\JDK_API_1_6_zh_CN.CHM::/java/lang/Class.html)<?>...parameterTypes) 获取指定参数的public构造器 |
| Constructor<?>[] | **getConstructors**()获取所有的public构造器 |
| Constructor<T> | **getDeclaredConstructor**([Class](mk:@MSITStore:D:\Eclipse\JDK_API_1_6_zh_CN.CHM::/java/lang/Class.html)<?>...parameterTypes)获取指定参数的构造器 |
| Constructor<?>[] | **getDeclaredConstructors**()获取所有的构造器 |

|  |
| --- |
| **public** **void** method1() **throws** Exception {  Class<?> clazz = Class.*forName*("test509.Person1");  //获取所有的构造器，并返回构造器数组  Constructor<?>[] declaredConstructors = clazz.getDeclaredConstructors();  **for** (Constructor<?> constructor : declaredConstructors) {  //返回描述该构造器的字符串  System.***out***.println(constructor.toGenericString());  }  } |
| test509.Person1()  private test509.Person1(java.lang.String) |

|  |  |
| --- | --- |
| **Constructor类方法摘要** | |
| int | **getModifiers**()获取构造器权限修饰符 |
| String | **getName**()获取方法名称 |
| Class<?>[] | **getParameterTypes**()获取得参数的类型 |

|  |
| --- |
| **public** **void** method2() **throws** Exception {  Class<?> clazz = Class.*forName*("test509.Person1");  Constructor<?> constructor = clazz.getDeclaredConstructor(String.**class**);  constructor.setAccessible(**true**);  //获取构造器的权限修饰符  **int** num = constructor.getModifiers();  System.***out***.println(Modifier.*toString*(num));  //获取构造器名称  String name = constructor.getName();  System.***out***.println(name);  //获取构造器的参数类型  Class<?>[] parameterTypes = constructor.getParameterTypes();  System.***out***.println(Arrays.*toString*(parameterTypes));  } |
| private  test509.Person1  [class java.lang.String] |

#### 获取父类

|  |  |
| --- | --- |
| **Class类方法摘要** | |
| Class<? super T> | [**getSuperclass**](mk:@MSITStore:D:\Eclipse\JDK_API_1_6_zh_CN.CHM::/java/lang/Class.html#getConstructor(java.lang.Class...))()获取此Class对象所表示的类或接口的父类的Class |

#### 获取接口

|  |  |
| --- | --- |
| **Class类方法摘要** | |
| Class<?>[] | **getInterfaces**()获取此Class对象所表示的类或接口实现的接口 |

#### 获取注解

|  |  |
| --- | --- |
| **Class类方法摘要** | |
| <A extends Annotation>A | [**getAnnotation**](mk:@MSITStore:D:\Eclipse\JDK_API_1_6_zh_CN.CHM::/java/lang/Class.html#getConstructor(java.lang.Class...))(Class<A> annotationClass) 获取指定类型的注解，如果该注解不存在，则返回 null。 |
| Annotation[] | **getAnnotations**()返回此元素上存在的所有注解 |
| Annotation[] | **getDeclaredAnnotations**()返回此元素上直接存在的所有注解 |
| boolean | **isAnnotationPresent**(Class <? extends Annotation>)当前对象是否有注解 |

通过反射获取注解必须要将该注解的元注解的RetentionPolicy设置为RUNTIME。

//下面的元注解必须有，否则通过反射获取不到

@Retention(RetentionPolicy.***RUNTIME***)

#### 获取泛型

Java采用泛型擦除的机制来引入泛型，其泛型仅仅是给编译器javac使用，用来确保数据的安全性和免去强制类型转换的麻烦。但是一旦编译完成，所有的和泛型有关的类型全部擦除。

为了通过反射操作这些类型以迎合实际开发的需要，Java新增了ParameterizedType、GenericArrayType、TypeVariable和WildcardType几种类型来代表不能被归一到Class类中的类型但是又和原始类型齐名的类型。

ParameterizedType表示一种参数化的类型，比如Collection<String>

GenericArrayType表示一种元素类型是参数化类型或者类型变量的数组类型

TypeVariable是各种类型变量的公共父接口

WildcardType代表一种通配符类型表达式，如？、？ extends Number等，wildcard就是通配符的意思。

|  |  |
| --- | --- |
| **Class类方法摘要** | |
| Type | **getGenericSuperclass**()获取此Class对象所表示的类或接口的父类的类型 |
| Type[] | **getGenericInterfaces**()获取此Class对象所表示的类或接口实现的接口的类型 |

ParameterizedType接口表示参数化泛型类型，是Type接口的子接口，其getActualTypeArguments()可以获取实际的泛型类型参数并以数组形式返回。

|  |
| --- |
| **public** **void** test8(){  Class clazz = SubClass.**class**;  //获取带泛型的父类，返回的虽然是Type类型的对象，但实际是ParameterizedType接口的实现类的对象  Type type = clazz.getGenericSuperclass();  //将父接口转成子接口 - 因为要调用ParameterizedType接口中的方法  ParameterizedType pt = (ParameterizedType) type;  //获取父类中的所有的泛型的类型  Type[] actualTypeArguments = pt.getActualTypeArguments();  System.***out***.println(Arrays.*toString*(actualTypeArguments));  } |

#### 获取包

|  |  |
| --- | --- |
| **Class类方法摘要** | |
| Package | **getPackage**()获取此类的包 |