Contents

[Java 3](#_Toc470603851)

[Java 8 3](#_Toc470603852)

[函数式编程 3](#_Toc470603853)

[流 5](#_Toc470603854)

[数值流 7](#_Toc470603855)

[调试stream 8](#_Toc470603856)

[默认方法 8](#_Toc470603857)

[Optional 8](#_Toc470603858)

[异步编程 10](#_Toc470603859)

[日期和时间 11](#_Toc470603860)

[模式匹配 12](#_Toc470603861)

[内部类和匿名类 12](#_Toc470603862)

[C++/Java 区别: 15](#_Toc470603863)

[final 关键字：数据，方法和类 17](#_Toc470603864)

[初始化及类的加载 17](#_Toc470603865)

[继承与组合 18](#_Toc470603866)

[接口与抽象类 19](#_Toc470603867)

[策略设计模式 19](#_Toc470603868)

[工厂方法设计模式： 20](#_Toc470603869)

[模板方法设计模式 21](#_Toc470603870)

[类型信息 23](#_Toc470603871)

[Groovy 25](#_Toc470603872)

[基本类型 25](#_Toc470603873)

[动态类型 25](#_Toc470603874)

[类 25](#_Toc470603875)

[函数指针（闭包） 25](#_Toc470603876)

[列表 （同构或异构） 26](#_Toc470603877)

[多维数组 26](#_Toc470603878)

[字典 26](#_Toc470603879)

[字符串 27](#_Toc470603880)

[解操作符 (Spread operator) 27](#_Toc470603881)

[范围操作符(Range operator) 27](#_Toc470603882)

[索引操作符(Subscript operator) 27](#_Toc470603883)

[命名空间 28](#_Toc470603884)

[Scala 29](#_Toc470603885)

[Scala与Java,C++区别： 29](#_Toc470603886)

[数组 30](#_Toc470603887)

[映射 31](#_Toc470603888)

[元组 32](#_Toc470603889)

# Java

参考书籍：

<Java程序语言>

<Java编程思想>

<Effective Java>

<Java程序员修炼之道>

<Java 8函数式编程>

## Java 8

Java8流支持简明的数据库查询式编程，但用的是java语法，而无需了解数据库。

行为参数化，函数式编程，无副作用函数，无状态函数

行为参数化：处理频繁变更需求的一种软件开发模式，把一个行为（一段代码）封装起来，并通过传递和使用创建的行为

例子：Comparator排序，用Runnable执行一个代码块，以及GUI事件处理

### 函数式编程

没有共享的可变数据，将方法和函数即代码传递给其他方法

#### Java 8 的常用函数式接口（<Java8实战>p46）

java.util.function.Predicate<T> T -> boolean 用于filter

.Consumer<T> T -> void 用于foreach

.Function<T, R> T -> R 用于map

…

@FunctionalInterface

public interface Predicate<T> { boolean test(T t); }

@FunctionInterface

public interface Consumer<T> { void accept(T t); }

@FunctionInterface

public interface Function<T, R> { R apply(T t); }

任何函数式接口都不允许抛出受检异常checked exception

#### Lambda表达式

(parameters) -> expression or (parameters) -> { statements; }

隐含return语句

Lambda的类型是从使用Lambda的上下文推断出来的

Java编译器会从上下文（目标类型）推断出用什么函数式接口来配合Lambda表达式，这意味着它也可以推断出适合Lambda的签名，因为函数描述符可以通过目标类型来得到。也就是说，编译器可以了解Lambda表达式的参数类型，这样就可以在Lambda语法中省去标注参数类型

(Apple a1, Apple a2) -> a1.getHeight().compareTo(a2.getWeight()) 可以省略为

(a1, a2) -> a1.getHeight().compareTo(a2.getWeight())

Lambda表达式引用的局部变量必须是final的？（匿名类也是）

因为实例变量都存储在堆中，而局部变量则保存在栈上。如果Lambda可以直接访问局部变量，而且Lambda是在一个线程上使用的，则使用Lambda的线程，可能会在分配该变量的线程将这个变量收回之后，去访问该变量。

方法引用 等效的方法引用

a -> a.getWeight() Apple::getWeight()

() -> Thread.currentThread().dumpStack() Thread.currentThread()::dumpStack()

方法引用可以视为单一方法的Lambda语法糖

静态方法的方法引用： Integer::parseInt

实例方法的方法引用： String::length

现有对象的实例方法的方法引用：obj::getValue

构造函数

(weight) -> new Apple(weight);

方法一：传递代码

public class AppleComparator implements Comparator<Apple> {

public int compare(Apple a1, Apple a2) {

return a1.getWeight().compareTo(a2.getWeight);

}

}

inventory.sort(new AppleComparator());

方法二：使用匿名类

inventory.sort(new Comparator<Apple>() {…}) ;

方法三：lambda表达式

inventory.sort((a1, a2) -> a1.getWeight().compareTo(a2.getWeight()));

方法四：方法引用

inventory.sort(comparing(Apple::getWeight));

### 流

一系列数据项，一次只生成一项。程序可以从输入流中一个一个读取数据项，然后以同样的方式将数据项写入输出流。一个程序的输出流很可能是另一个程序的输入流。如linux命令：cat file1 file2 | tr “[A-Z]” “[a-z” | sort | tail -3

Java集合Collection和流Stream

集合Collection主要是为了存储和访问数据，而流Stream则主要用于描述对数据的计算。

**集合视为空间中分布的一组值；流视为时间中分布的一组值**。

集合是一个内存中的数据结构，它包含数据结构中目前所有的值—集合中的每个元素都得先算出来才能添加到集合中

流，和迭代器类似，只能遍历一次，遍历完成之后，我们就说这个流已经被消费掉了

流操作 = 中间操作 + 终端操作

**除非流水线触发一个终端操作，否则中间操作不会执行任何处理。中间操作一般可以合并起来，在终端操作时一次性全部处理。**

流的使用一般包括三件事：数据源执行一个查询；中间操作链，形成一条流的流水线；终端操作，执行流水线，并能生成结果

#### 创建流

Stream<String> stream = Stream.of(“r”, “g”, “b”);

int[] numbers = {2, 3, 4, 5};

IntStream stream = Arrays.stream(numbers);

Stream<String> lines = Files.lines(Paths.get(“data.txt”), Charset.defaultCharset())

集合.stream()

Stream.iterate(0, n->n+2).limit(10) 创建无限流

Stream.generate(Math::random).limit(5) 生成式生成流

#### 中间操作

.filter(Dish::isVegetarian) 过滤流

.distinct() 返回元素各异的流

.limit(n) 返回不超过给定长度的流

.skip(n) 返回扔掉前n个元素的流

.map(Dish::getName) 映射流

.map(w -> w.split(“”)).flatMap(Arrays::stream) 扁平化流 （流连接）

.sorted(comparing(Apple::getWeight))

#### 终端操作

.forEach(System.out::println) 返回void

.anyMatch(Dish::isVegetarian) 返回boolean

.allMatch(d -> d.getCalories()<1000) 返回boolean

.noneMatch(d -> d.getCalories()>=1000) 返回boolean

.findAny() 返回流中的任意元素

.findFirst() 查找第一个元素

.min(comparing(Transaction::getValue)) 返回最小元素

.max(comparing(Transaction::getValue)返回最大元素

.count() 返回个数

.collect(toList()) 返回列表

.collect(toSet()) 返回去重集合

.collect(counting()) 返回个数

.collect(summingInt(Dish::getCalories)) 求和

.collect(reducing(0, Dish::getCalories, Integer::sum)) 求和

.collect(averagingInt(Dish::getCalories)) 平均

.collect(maxBy(comparingInt(Dish::getCalories)) 最大值

.collect(minBy(comparingInt(Dish::getCalories)) 最小值

.collect(summarizingInt(Dish::getCalories)) 汇总操作，返回count, sum, min, average, max

.collect(joining(“,”)) 连接字符串

.groupingBy(Dish::getType) 返回Map<K, List<T>> 按Type分组

二级分组，先按Dish::getType进行分组，对每个type的List再进行按匿名函数进行第二级分组

.groupingBy(Dish::getType,

groupingBy(dish -> {

if ( dish.getCalories() <= 400) return CaloricLevel.DIET;

else if ( dish.getCalories() <= 700 return CaloricLevel.NORMAL;

else return CaloricLevel.FAT;

}

)

结果是两级map:

{

MEAT: {

DIET: [chicken],

NORMAL: [beef],

FAT: [pork]

},

FISH: {

DIET: [prawns],

NORMAL: [salmon]

},

OTHER: {

DIET: [rice, seasonal fruit],

NORMAL: [French fries, pizza]

}

}

### 数值流

.mapToInt(Dish::getCalories)

.sum() 求和

.max()

.min()

.average()

IntStream

.range(0, 100) 生成[0, 1, …, 99]

.rangeClosed(1, 100) 生成[1, …, 100]

遍历[1, … 100]范围内的勾股数对

IntStream

.rangeClosed(1, 100)

.boxed()

.flatMap( a ->

IntStream

.rangeClosed(a, 100)

.mapToObj(b -> new int[] { a, b, Math.sqrt(a\*a + b\*b) } )

.filter(t -> t[2]%1 == 0)

);

### 调试stream

不能使用forEach（终端操作），应使用peek

.stream()

.peek(x -> println(“from stream: “ + x))

.map(x -> x + 17)

.peek(x -> println(“after map: “ + x ))

.filter(x -> x % 2 == 0)

.peek(x -> println(“after filter: “ + x ))

.collect(toList())

### 默认方法

默认方法的引入是为了以兼容的方式解决像Java API这样的类库的演进问题

接口引入默认方法，同时可以多接口继承，会导致函数签名冲突，如何解决？

1. 类中的方法优先级最高：类->父类->父类->…
2. 最接近类的接口：父接口->父父接口->…
3. 若两个父接口拥有相同签名，则类必须显式覆盖或显式调用

public interface A {

void hello() { …}

}

public interface B {

void hello() {…}

}

public class C implements B, A {

void hello() {

B.super.hello(); 显示调用接口B的方法

}

}

### Optional

public class Person {

private Car car;

public Car getCar() { return car; }

}

public class Car {

private Insurance insurance;

public Insurance getInsurance() { return insurance; }

}

public class Insurance {

private String name;

public String getName() { return name; }

}

采用防御式检查减少NullPointerException

public String getCarInsuranceName(Person person) {

if (person == null) { return "Unknown"; }

Car car = person.getCar();

if (car == null) { return "Unknown"; }

Insurance insurance = car.getInsurance();

if (insurance == null) { return "Unknown"; }

return insurance.getName();

}

采用Optional方式

你可以把Optional对象看成一种特殊的集合数据，它至少包含一个元素。如果Optional包含一个值，那函数就将该值作为参数传递给flatMap,对该值进行转换，如果Optional为空，就什么也不做.

声明方法接受一个Optional参数，或者将结果作为Optional类型返回，让你的同事或未来你方法的使用者，很清楚地知道它可以接受空值，或者它可能返回一个空值

Optional类和Stream接口有相似之外，不止map and flatMap方法，还有filter

注意：flatMap会用流的内容替换每个新生成的流。换句话说，由方法生成的各个流会被合并或者扁平化为一个单一的流

public class Person {

private Car car;

public Car getCar() { return car; }

public Optional<Car> getCarAsOptional() { return Optional.ofNullable(car); };

}

public class Car {

private Insurance insurance;

public Insurance getInsurance() { return insurance; }

public Optional<Insurance> getInsuranceAsOptional() { return Optional.ofNullable(insurance); }

}

public class Insurance {

private String name;

public String getName() { return name; }

}

public String getCarInsuranceName(Optional<Person> person) {

return person.flatMap(Person::getCarAsOptional)

.flatMap(Car::getInsuranceAsOptional)

.map(Insurance::getName)

.orElse("Unknown");

}

public String getCarInsuranceName(Optional<Person> person, int minAge) {

return person.filter(p -> p.getAge() >= minAge)

.flatMap(Person::getCarAsOptional)

.flatMap(Car::getInsuranceAsOPtional)

.map(Insurance::getName)

.orElse("Unknown");

}

函数式编程没有返回异常，可以将传统抛出异常的函数封装

Class OptionalUtil {

public static Optional<Integer> stringToInt(String s) {

try {

return Optional.off(Integer.parseInt(s));

} catch (NumberFormatException e) {

return Optional.empty();

}

}

}

### 异步编程

下一代网络应用都采用“混聚”mash-up的方式：它会使用来自多个来源的内容，将这些内容聚合在一起，方便用户的生活。

并发：同一个CPU上执行同个松耦合的任务，充分利用CPU的核，最大化程序的吞吐量，避免因为等待远程服务的返回，或对数据库的查询，而阻塞线程的执行，浪费宝贵的计算资源。

顺序执行异步任务 见p238

public List<String> findPrices(String product) {

List<CompletableFuture<String>> priceFutures = shops

.stream()

//CompletableFuture发起异步请求

.map(shop -> CompletableFuture.supplyAsync( () -> {

data from remote; //访问远程服务

return shop.getPrice(product, data);

})

//同步执行

.map(future -> future.thenApply(Integer::parse))

//发起异步请求

//thenCompose允许对两个异步操作进行流水线，第一个操作完成时，将其结果作为参数传递给第二个操作

.map(future -> future.thenCompose(intVal -> CompletableFuture.supplyAsync( () -> {

data from remote; //访问远程服务

return ...;

}))

.collect(toList());

return priceFutures.stream()

.map(CompletableFuture::join) //等待所有异步操作结束

.collect(toList());

}

通常而言，名称中不带Async的方法和它的前一个任务一样，在同一个线程中运行；而名称以Async结尾的方法会将后续的任务提交到一个线程池，所以每个任务是由不同的线程处理的。

合并两个相互独立的异步任务 见p241

thenCombine方法接受第二个BiFunction参数，当两个CompletableFuture对象完成计算后，对生成的两个结果进行合并

.map(shop ->

CompletableFuture.supplyAsync( () -> {

data from remote;

return shop.getPrice(product, data);

}).thenCombine(

CompletableFuture.supplyAsync( () -> {

data from remote;

return shop.getRate(product, data);

}),

(price, rate) -> price \* rate;

)

);

### 日期和时间

不可变对象，不附带时区相关的信息：

LocalDate 日期

LocalTime 时间

LocalDateTime 日期时间

// 2014-03-18T13:45:20

dateTime = LocalDateTime.of(2014, Month.MARCH, 18, 13, 45, 20)

Instant 机器的日期和时间（从Unix元年开始的秒数）

Duration, Period 间隔

TemporalAdjuster 调整日期

DateTimeFormatter 日期时间格式

带时区的信息

时区是按照一定的规则将区域划分成的标准时间相同的区间。

地区ID都为"区域/城市"的格式，这些地区集合的设定都由英特网编号分配机构IANA的时区数据库提供

ZoneId romeZone = ZoneId.of("Europe/Rome");

一旦得到一个ZoneId对象，可以结合LocalDate, LocalDateTime, Instant，构造成相对于指定时区的日期时间

ZonedDateTime zdt = dateTime.atZone(romeZone);

建议在应用中使用LocalDate, 包括存储、操作、业务规则的解读；不过如果需要将程序输入或输出本地化，再转换

### 模式匹配

函数式语言通常都会提供某种形式的模式匹配--作为switch的一种改良形式，通过这种模式匹配，你可以查询“这个值是某个类的实例吗？”或者你也可以选择递归地查询某个字段是否包含了某些值

def simplefyExpression(expr: Expr): Expr =

expr match {

case BinOp("+", e, Number(0)) => e //加0

case BinOp("\*", e, Number(1)) => e //乘以1

case BinOp("/", e, Number(1)) => e //除以1

case \_ => expr

}

## 内部类和匿名类

类

* 成员变量

匿名类

* 成员方法

匿名类（可以访问外部类的变量和方法，还可以访问该成员方法中的final变量）

匿名类作为函数的参数，或者先定义变量，然后传给函数的参数

* 内部类

成员变量

成员方法（可以访问内部类，还可以访问外部类的变量和方法）

内部类特点：

* 内部类看起来就像是一种代码隐藏机制
* 内部类了解外围类，并能与之通信
* 外部类将有一个方法，该方法返回一个指向内部类的引用。
* 创建内部类之前，必须先创建外部类，这是因为内部类对象会暗暗地连接到创建它的外部类对象上，从而内部类对象能访问其外围对象的所有成员，包括private成员。一个内部类被嵌套多少层并不重要，它能透明地访问所有它所嵌入的外围类的所有成员。

内部类实现迭代器设计模式：

interface Selector{

boolean end() ;

Object current() ;

void next() ;

}

public class Sequence{

private Object[] items ;

private int next = 0 ;

public Sequence(int size){

items = new Object[size] ;

}

public void add(Object x){

if (next < items.length){

items[next++] = x ;

}

}

private class SequenceSelector implements Selector{

private int i = 0 ;

public boolean end() {

return i == items.length ;

}

public Object current(){

return items[i] ;

}

public void next(){

if (i<items.length) i++ ;

}

}

public Selector selector(){

return new SequenceSelector() ;

}

public static void main(String[] args){

Sequence sequence = new Sequence(10) ;

for (int i = 0 ; i <10 ; i ++){

sequence.add(Integer.toString(i)) ;

}

Selector selector = sequence.selector() ;

while (!selector.end()){

selector.next() ;

}

}

}

迭代器的作用：编写通用的代码，只是使用容器，不关心容器的类型,迭代器统一了对容器的访问方式.

public static void display(Iterator<Pet> it){

while (it.hasNext){

Pet p = it.next() ;

...

}

}

public static void main(String[] args){

ArrayList<Pet> pets = Pets.arrayList(8) ;

HashSet<Pet> petsHS = new HashSet<Pet>(pets) ;

display(pets.iterator()) ;

display(petsHS.iterator());

}

如何内部类返回外部类？外部类名.this

如果创建内部类？先创建外部类，然后用外部类对象名.new 内部类()

注意与嵌套类区别：嵌套类只是静态内部类，那么它就不需要对外部类对象的引用.普通内部类的字段与方法，只能放在类的外部层次上，所以普通的内部类不能有static数据和static字段，也不能包含嵌套类

可以在方法里或作用域里定义内部类，那么内部类仅限在方法内或作用域内。

内部类就是一个面向对象的闭包，因为它不仅包含外围类对象（包括内部类的作用域）的信息，还自动拥有一个指向此外围类对象的引用，在此作用域内，内部类有权操作所有的成员，包括private成员。（闭包(closure):是一个可调用的对象，它记录了一些信息，这些信息来自于创建它的作用域。）

//匿名类在成员方法内的使用

@Override

**public** **void** onCreate(Bundle savedInstanceState) {

**super**.onCreate(savedInstanceState);

**this**.setRetainInstance(**true**);

Intent startupIntent = **new** Intent(Intent.*ACTION\_MAIN*) ;

startupIntent.addCategory(Intent.*CATEGORY\_LAUNCHER*) ;

**final** PackageManager pm = getActivity().getPackageManager() ;

List<ResolveInfo> activities = pm.queryIntentActivities(startupIntent, 0);

//匿名类作为函数的参数

Collections.*sort*(activities, **new** Comparator<ResolveInfo>() {

**public** **int** compare(ResolveInfo a, ResolveInfo b) {

**return** String.*CASE\_INSENSITIVE\_ORDER*.compare(

a.loadLabel(pm).toString(),

b.loadLabel(pm).toString());

}

});

//先定义变量，然后再函数的参数

ArrayAdapter<ResolveInfo> adapter = **new** ArrayAdapter<ResolveInfo>(

getActivity(), android.R.layout.*simple\_list\_item\_1*, activities) {

**public** View getView(**int** pos, View convertView, ViewGroup parent) {

View v = **super**.getView(pos, convertView, parent);

TextView tv = (TextView)v;

ResolveInfo ri = getItem(pos);

tv.setText(ri.loadLabel(pm));

**return** v;

}

};

setListAdapter(adapter);

## C++/Java 区别:

C++方法，默认情况下，不是动态绑定，实现虚拟机制需要用关键字virtual；Java方法，默认情况下,是动态绑定，多态

Java没有sizeof, 因为所有数据类型在所有机器中的大小都是相同的，不必考虑移植问题。C/C++有sizeof()，告诉数据项分配的字节数，需要使用sizeof()的最大原因是为了移植，因为不同的数据类型在不同的机器上可能有不同的大小。

一开始使用LinkedList构建程序，而在优化系统性能时改用ArrayList

Java特点:纯面向对象，单根继承，垃圾回收（内存自动管理）,参数化类型的容器，并发编程，网络.以上全部在语言和开发包层面就已经考虑到并集成

基本类型:

大小并不像其他大多数语言那样随机器硬件架构的变化而变化，即占存储空间大小的不变性

数组:

Java确保数组会被初始化(基本类型置0,对象引用置null)，范围检查（运行时下标检查）

发送消息给对象:调用方法的行为,如a.f(), 消息是f(),对象是a

Java和C++容器的区别：java容器有公共基类Collection, C++容器没有基类，容器之间的所有共性都是通过迭代器达成的。Java容器的共性不仅仅通过迭代器达成，也可以通过基类达成

Javadoc是用于提取注释的工具，是JDK安装的一部分。它采用了Java编译器的某些技术，查找程序内的特殊注释标签，不仅解析由这 些标签标记的信息，也将相邻注释的类名或方法名抽取出来。Javadoc输出的是一个HTML文件，可以用Web浏览器查看。

Java是建立在C++基础上，所以Java的操作符和控制执行流程，同C/C++很像，还支持枚举类型同C++/C#。

直接常量，基本类型后缀字符20L， 20.0F， 20.0D

Print(x + “” + s); //converts x and s to string

Java不允许将一个数字作为布尔值，（在C/C++允许，比如if(a)）, 必须用if (a!= 0)

Java 类似C# foreach

For (char c : “An African”.toCharArray() )

…

Switch 选择因子必须是int or char or enum那样的整数值。若将一个字符串或浮点数作为选择因子则不工作.

Java支持重载方法，this关键字,在闭包内想引用外部类对象，可以外部类.this

初始化可以在定义类成员变量的地方为其赋值（注意在C++里不能这么做）

数组初始化: 编译器不允许指定数组的大小，为了给数组创建相应的存储空间，必须写初始化表达式.

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

Random rand = new Random(47) ;

int[] a = new int[rand.nextInt(20)] ;

可变参数列表：

static void printArray(Object… args)

{

for (Object obj : args) Fun(obj) ;

}

static void g(int…args) {}

Java解释器的运行过程：首先，找出环境变量CLASSPATH.CLASSPATH包含一个或多个目录，用作查找.class文件的根目录。从根目录开始，解释器获取包的名称并将每个句点替换成反斜杠，以从CLASSPATH根中产生一个路径名称(于是package foo.bar.baz就变成foo\bar\baz).得到路径会与CLASSPATH的各个不同的项相连接，解释器就在这些目录中查找与你所要创建的类名称相关的.class文件.

同一目录里的java文件，无package（默认包），或package名一样，类和函数没有权限修饰词，属于包访问权限

客户端访问包接口，包里的类需要加权限public，方法是public或protected

Java文件只能有一个public类，提供接口，可以提供一堆非public类，提供包访问权限。

类内private方法只能类内访问

每一个非基本类型的对象都有一个toString()方法，而且当编译器需要一个String而却只有一个对象时，该方法便会被调用。”source = “ + source ;

可以为每个类都创建一个main()方法，这种在每个类中都设置一个main()方法的技术可使每个类的单元测试都变得简单易行。而且在完成单元测试之后，也无需删除main(),可以将其留待下次测试.

继承并不只是复制基类的接口。当创建了一个导出类的对象时，该对象包含了一个基类的子对象。这个子对象与你用基类直接创建的对象是一样的。二者区别在于，后者来自于外部，而基类的子对象被包装在导出类对象内部.

初始化基类: 必须用super显式地调用基类构造器的语句.

class Game {

Game(int i ){

.i..

}

}

class BoardGame extends Game {

BoardGame(int i) {

super(i);

...

}

}

确保正确清理：Java中没有C++析构函数的概念。因此如果你想要某个类清理一些东西，就必须显式地编写一个特殊方法来做这件事，并要确保客户端程序员知晓他们必须要调用这一方法。

class Shape{

public Shape(int i) {... }

void dispose() {... }

}

class Circle extends Shape{

Circle(int i){ super(i) ; }

void dispose(){..}

}

public class CADSystem extends Shape{

private Circle c ;

private Triangle t ;

public CADSystem(int i){

super(i+i);

c = new Circle(1) ;

c = new Triangle(1) ;

}

public void dispose(){

t.dispose() ;

c.dispose() ;

super.dispose() ;

}

public static void main(String[] args){

CADSystem x = new CADSystem(47) ;

try { fun(x) ;

} finally {

x.dispose() ;

}

}

}

## final 关键字：数据，方法和类

public static final in VALUE\_THREE = 39 ; //对于基本类型，final使数值恒定不变

private static Value V2 = new Value(22) ; //对于对象引用，final使用引用恒定不变，一旦引用被初始化指向一个对象，就无法再把它改为指向另一个对象.

// java final类似C++ const，无法在方法中更改参数引用所指向的对象

Public void with(final Gizmo g){

}

//final 方法，把方法锁定，以防任何继承类修改它的含义.

//类中所有private方法都隐式地指定为final

final void f(){}

// final 类，表明你不打算继承该类，而且也不允许别人这么做.

//由于final类禁止继承，所以final类中所有的方法都隐式指定为final的，因为无法覆盖它们。

final class Dinosaur{

}

## 初始化及类的加载

类是在其任何static成员被访问时加载，构造器也是隐式static方法.

组合一般是将现有类型作为新类型底层实现的一部分来加以复用，而继承复用的是接口.不能十分确定应该使用哪一种方式时，首先选择组合，组合不会强制我们的程序设计进入继承的层次结构中。而且组合更加灵活，因为它可以动态选择类型（因此也就选择了行为）；相反，继承在编译时就需要知道确切类型.

## 继承与组合

多态通过分离做什么和怎么做，从另一角度将接口和实现分离出来。多态不但能够改善代码的组织结构和可读性，还能够创建可扩展的程序。多态的作用则是消除类型之间的耦合关系.

封装通过合并特征和行为来创建新的数据类型。实现隐藏则通过将细节私有化把接口和实现分离开来。

Java中除了static方法和final方法（private方法属于final方法）之外，其他所有的方法都是后期绑定.

编写构造器时有一条有效的准则：用尽可能简单的方法使对象进入正常状态；如果可以的话，避免调用其他方法。在构造器内唯一能够安全调用的那些方法是基类中的final方法（也适用于private方法，它们自动属于final方法）.

**一条通用的准则“用继承表达行为间的差异，并用字段表达状态上的变化”**，如下例子中，两者都用到了，通过继承得到两个不同的类，用于表达act()方法的差异，而Stage通过运用组合使自己的状态发生了变化。

class Actor{

public void act(){}

}

class HappyActor extends Actor{

public void act(){}

}

class SadActor extends Actor{

public void act(){}

}

class Stage{

private Actor actor = new HappyActor() ;

public void change() {actor = new SadActor();}

public void performPlay() {actor.act();}

}

public class Transmogrify{

public static void main(String[] args){

Stage stage = new Stage() ;

stage.performPlay() ;

stage.change() ;

stage.performPlay() ;

}

}

## 接口与抽象类

interface产生一个完全抽象的类，没有提供任何具体实现，接口只提供了形式，未提供任何具体实现.接口被用来建立类与类之间的协议.接口也可以包含域，但是这些**域隐式地是static and final**

abstract class Instrument{

private int i ;

public abstract void play(Note n);

}

class Wind extends Instrument{

public void play(Note n){}

}

interface Instrument{

int VALUE = 5 ; //static and final

void play(Note n) ; //Automatically public

}

class Wind implements Instrument{

public void play(Note n){}

}

Java只能继承一个类（必须放在前面），可以继承任意多个接口，并可以向上转型为每个接口，因为每一个接口都是一个独立类型。

class Hero extends ActionCharacter

implements CanFight, CanSwim, CanFly{

public void fight(){}

public void swim(){}

public void fly(){}

}

## 策略设计模式

创建一个能够根据所传递的参数对象的不同而具有不同行为的方法。“策略”是变化，就是传递进去的参数对象。策略设计模式是接口的常见用法

适配器设计模式和代理模式

FilterAdapter代理Filter,然后生成具有你所需要的Processor接口的对象，

for example:

interface Processor{

Object process(Object input){

return input ;

}

}

class Upcase implements Processor{

String process(Object input){

return ((String)input).toUpperCase() ;

}

}

public class Apply{

public static void process(Processor p, Object s ){

p.process(s) ;

}

}

public class Filter{

public Waveform process(Waveform input){

return input ;

}

}

public class Lowpass extends Filter{

public Lowpass(double cutoff) {} ;

public Waveform process(Waveform input){

return input ;

}

}

class FilterAdapter implements Processor{

Filter filter ;

public FilterAdapter(Filter filter){

this.filter = filter ;

}

public static void main(String[] args){

Waveform w = new Waveform() ;

Apply.process(new FilterAdapter(new LowPass(1.0)), w) ;

Apply.process(new Upcase(), "haha") ;

}

}

## 工厂方法设计模式：

interface Service{

void method1() ;

void method2() ;

}

interface ServiceFactory{

Service getService() ;

}

class ServiceSupplier1 implements Service{

ServiceSupplier1() {}

public void method1() {}

public void method2() {}

}

class ServiceSupplier1Factory implements ServiceFactory{

public Service getService(){

return new ServiceSupplier1() ;

}

}

class ServiceSupplier2 implements Service{

ServiceSupplier2() {}

public void method1() {}

public void method2() {}

}

class ServiceSupplier2Factory implements ServiceFactory{

public Service getService(){

return new ServiceSupplier2() ;

}

}

public class Factories{

public static void serviceConsumer(ServiceFactory fact){

Service = fact.getService() ;

s.method1() ;

s.method2() ;

}

public static void main(String[] args){

serviceConsumer(new ServiceSupplier1Factory) ;

serviceConsumer(new ServiceSupplier2Factory) ;

}

}

**服务端，可以基于接口或基类型写出通用代码，客户端子类化工厂，提供具体服务**

## 模板方法设计模式

包含算法的基本结构，并且会调用一个或多个可覆盖的方法，以完成算法的动作。设计模式总是将变化的事物与保持不变的事物分离开，在这个模式中，模板方法是保持不变的事物，而可覆盖的方法就是变化的事物

控制框架是一类特殊的应用程序框架，它用来解决响应事件的需求。主要用来响应事件的系统被称为事件驱动系统。

public abstract class Event{

private long eventTime ;

protected final long delayTime ;

public Event(long delayTime){

this.delayTime = delayTime ;

start() ;

}

public void start(){eventTime = System.nanoTime() + delayTime ; }

public boolean ready(){ return System.nanoTime() >= eventTime ; }

public abstract void action() ;

}

public class Controller{

private List<Event> eventList = new ArrayList<Event>() ;

public void addEvent(Event c) {eventList.add(c);}

public void run(){

while(eventList.size()>0){

for (Event e : new Arraylist<Event>(eventList)){

if (e.ready()){

e.action() ;

eventList.remove(e) ;

}

}

}

}

}

public class GreenhouseControls extends Controller{

private boolean light = false ;

public class LightOn extends Event{

public LightOn(long delayTime) {super(delayTime);}

public void action(){light = true ; }

}

public class LightOff extends Event{

public LightOff(long delayTime) {super(delayTime);}

public void action(){light = false ; }

}

public class Restart extends Event{

private Event[] eventList ;

public Restart(long delayTime, Event[] eventList){

super(delayTime) ;

this.eventList = eventList ;

for (Event e : eventList) { addEvent(e) ; }

}

public void action(){

for (Event e : eventList){

e.start() ;

addEvent(e) ;

}

start() ;

addEvent(this);

}

}

}

public class GreenhouseController{

public static void main(String[] args){

GreenhouseControls gc = new GreenhouseControls() ;

gc.addEvent(gc.new LightOn(200)) ;

Event[] eventList = {gc.new LightOn(200), gc.new LightOff(200)} ;

gc.addEvent(gc.new Restart(2000, eventList)) ;

gc.run() ;

}

}

在目前的设计中你并不知道Event到底做了什么，这正是此设计的关键所有，“使变化的事特与不变的事物相互分离”。“变化向量”就是各种不同的Event对象所具有的不同行为，而你通过创建不同的Event子类来表现不同的行为.

## 类型信息

在运行时识别对象和类的信息：RTTI；反射机制（运行更加动态的编程风格）

RTTI和反射之间的区别：对RTTI来说，编译器在编译时打开和检查.class文件（查寻对象的类型信息）；对反射来说，.class文件在编译时不可获取，是在运行时打开和检查.class文件。

场景：

1. 假设你从磁盘文件，或者网络连接中获取一串字节，并且你被告知这个字节代表一个类，而这个类在编译器为你的程序生成代码之后很久才会出现。
2. **基于构件的编程**：可视化编程方法，将不同组件的图标drag到表单中来创建程序，然后在编程时通过设置构件的属性值来配置它们。（Java通过JavaBeans提供基于构件的编程架构）
3. 希望提供在**跨网络的远程平台上创建和运行对象**的能力，即远程方法调用，允许一个Java程序将对象分布到多台机器上。

如何使用反射？

Class类与Java.lang.reflect类库一起对反射的概念进行了支持。该类库包含Field, Method, Constructor类，这些类型的对象是由JVM在运行时创建，用以表示未知类里对应的成员。用Constructor创建新的对象，用get(),set()方法读取和修改与Field对象关联的字段，用invoke()方法调用 与Method对象关联的方法。

Class对象：每当编写并编译一个新类，就会产生一个Class对象（保存在类名.class文件中）

使用类的三个步骤：

1. 加载

当程序创建对类的静态成员的引用时（注：构造器也是类的静态方法），就会加载这个类，如下是过程：类加载器首先检查这个类的Class对象是否已经加载，如果尚未加载，默认的类加载器就会根据查找类名.class文件，将类的Class对象载入内存，然后用Class对象来创建这个类的所有对象。

1. 链接

在链接阶段将验证类中的字节码，为静态域分配存储空间，并且如果需要的话，将解析这个类创建的对其他类的所有引用。

1. 初始化

如果该类具有超类，则对其初始化，执行静态初始化器和静态初始化块。（注：静态初始化块可以是对静态成员的操作）

//创建Class对象

Class cc = 类名.class ;

Class.forName(“类名”);

obj.getClass() ;

//获取类型信息

cc.getName() ;

cc.isInterface() ;

cc.getSimpleName() ;

cc.getCanonicalName() ;

cc.getSuperClass() ;

instanceof保持了类型概念，它指的是“你是这个类吗？或者你是这个类的派生类吗？”；==比较实际的Class对象，就没有考虑继承。

# Groovy

参考书籍：

<Groovy程序语言>

## 基本类型

最好指示类型如byte, char, short, int, long, BigInteger, float, double, BigDecimal

## 动态类型

def表示动态类型，同javascript var

## 类

**class** User {

String name

User(String name) { **this**.name = name}

String getName() { "Name: $name" }

}

**def** user = **new** User('Bob')

**assert** user.name == 'Name: Bob'

this.name直接访问字段,而user.name外部访问字段，将会触发user.getName()

The user.name call triggers a call to the property of the same name, that is to say, here, to the getter for name. If you want to retrieve the field instead of calling the getter, you can use the direct field access operator:

assert user.@name == 'Bob' // direct field access operator

user.&getName // method pointer operator 函数指针类型是groovy.lang.Closure

## 函数指针（闭包）

**def** transform(List elements, Closure action) {

**def** result = []

elements.*each* {

result << action(it)

}

result

}

String describe(Person p) {

"$p.name is $p.age"

}

**def** action = **this**.&describe

**def** list = [

**new** Person(name: 'Bob', age: 42),

**new** Person(name: 'Julia', age: 35)]

**assert** transform(list, action) == ['Bob is 42', 'Julia is 35']

类，抽象类，接口，属性权限等同java

除了公有字段，groovy会自动添加setter&getter; final字段，groovy会自动添加getter

## 列表 （同构或异构）

def numbers = [1, 2, 3]

assert numbers instanceof java.util.List

def heterogeneous = [1, "a", true]

assert heterogeneous instanceof java.util.ArrayList

def linkedList = [2, 3, 4] as LinkedList

assert linkedList instanceof java.util.LinkedList

LinkedList otherLinked = [3, 4, 5]

assert otherLinked instanceof java.util.LinkedList

assert numbers[0, 2] == [1, 3]

assert numbers[0..2] == [1, 2, 3]

## 多维数组

def multi = [[0, 1],

[2, 3]]

assert multi[1][0] == 2

## 字典

def map = [:]

map.'single quote'

map."double quote"

map.'''triple single quote'''

map."""triple double quote"""

map./slashy string/

map.$/dollar slashy string/$

**def** key = 'name'

**def** person = [key: 'Guillaume'] //map person有一个key是’key’

**assert** !person.containsKey('name')

**assert** person.containsKey('key')

person = [(key): 'Guillaume'] //map person有一个key是’name’

**assert** person.containsKey('name')

**assert** !person.containsKey('key')

## 字符串

String 同所有语言，不变量。GString惰性求值(called interpolated strings)

java.lang.String str = ‘a single quoted string’

java.lang.String str = ‘’’a triple quoted string’’’

groovy.lang.GString gStr = “hello ${str}”

## 解操作符 (Spread operator)

* 对集合中的每个元素进行解操作

**class** Car {

String make

String model

}

**def** cars = [

**new** Car(make: 'Peugeot', model: '508'),

**new** Car(make: 'Renault', model: 'Clio')]

**def** makes = cars\*.make

**assert** makes == ['Peugeot', 'Renault']

* 对列表进行解操作

**int** function(**int** x, **int** y, **int** z) {

x\*y+z

}

**def** args = [4,5,6]

**assert** function(\*args) == 26

* 对字典进行解操作

**def** m1 = [c:3, d:4]

**def** map = [a:1, b:2, \*:m1]

**assert** map == [a:1, b:2, c:3, d:4]

## 范围操作符(Range operator)

**def** range = 0..5

**assert** (0..5).*collect*() == [0, 1, 2, 3, 4, 5]

**assert** (0..<5).*collect*() == [0, 1, 2, 3, 4]

**assert** (0..5) **instanceof** List

索引操作符(Subscript operator)

The subscript operator is a short hand notation for getAt or putAt, depending on whether you find it on the left hand side or the right hand side of an assignment:

**class** User {

Long id

String name

**def** getAt(**int** i) {

**switch** (i) {

**case** 0: **return** id

**case** 1: **return** name

}

**throw** **new** IllegalArgumentException("No such element $i")

}

**void** putAt(**int** i, **def** value) {

**switch** (i) {

**case** 0: id = value; **return**

**case** 1: name = value; **return**

}

**throw** **new** IllegalArgumentException("No such element $i")

}

}

**def** user = **new** User(id: 1, name: 'Alex')

**assert** user[0] == 1

**assert** user[1] == 'Alex'

user[1] = 'Bob'

**assert** user.name == 'Bob'

## 命名空间

**package** com.philips

**import** groovy.xml.MarkupBuilder

**import** **static** java.lang.Math.\*

# Scala

参考书籍：

<快学Scala>

<深入理解Scala>

## Scala与Java,C++区别：

val greeting: String = “Hello World”;//常量定义

var counter = 0 ; //变量定义

//在scala中，我们用方法，而不是强制类型转换，来做数值类型之间的转换

99.44.toInt == 99

//两种方法等价

a.方法(b)

a 方法 b

如1.to(10), 1 to 10

不带参数的scala方法通常不使用(),一般来讲，没有参数且不改变当前对象的方法不带().

Scala没有提供++,--自增自减操作符

import scala.math.\_ //在scala中,\_字符是“通配符”，类似Java中的\*

在C++中，你会写s[i];在Java中，你会写s.charAt(i).在scala中，你会写s(i)，背后的运行原理：

s.apply(i). 如BigInt(“123”)等价于简写BigInt.apply(“123”)

在Java&C++中，表达式有值，而语句执行动作。在Scala中，几乎所有构造出来的语法结构都有值。如下：

val s = if (x>0) 1 else -1

var distance = {val dx=x-x0; val dy=y-y0; sqrt(dx\*dx + dy\*dy)}

{r = r\*n; n -= 1} //返回Unit类型等于Java/C++的void

//在scala中，对循环的使用不多，通常可以通过序列中的所有值应用某个函数的方式来处理它们

for (i <- 0 集合) {}

for I <- 0 until s.length) {} //i的最后一个取值是s.length-1

for (i <- 1 to 10) yield i%3 //该循环会构造出一个集合，每次迭代生成集合中的一个值

for (i <- 1 to 3; j <- 1 to 3 if i != j) {} //生成器

def abs(x: Double) = if (x>0) x else –x //普通函数

def fac(n: Int): Int = if (n<=0) 1 else n\*fac(n-1) //递归函数

def box(s: String) { …} //过程

默认参数，带名参数，变长参数

val words = scala.io.Source.fromFile(“test.data”).mkString //在words被定义时即被取值

lazy val words = scala.io.Source.fromFile(“test.data”).mkString //在words被首次使用时取值

def words = scala.io.Source.fromFile(“test.data”).mkString //在每一次words被使用时取值

try {…}

catch {…}

finally {…}

## 数组

val a = Array(2, 3, 4) ; //固定数组

val b = ArrayBuffer[Int](); //等同于Java的ArrayList,C++的vector

b += 1;

b += (1,2,3,4);

for (elem <- b) println(elem)

var result = for (elem <- b if elem%2==0) yield 2\*elem

b.filter(\_ %2==0).map(2\*\_) //函数式编程，filter用于过滤集合元素，map用于映射

Array(1,2,3,4).sum

ArrayBuffer(“Mary”, “had”,”a”,”little”).max

val bSorted = ArrayBuffer(1,2,3,4).sorted

val a = Array(1,2,3,4)

scala.util.Sorting.quickSort(a)

a.mkString(“ and “); //显示数组或数组缓冲的内容，允许用指定元素之间的分隔符

val matrix = Array.ofDim[Double](3, 4)

matrix(row)(column) = 4

## 映射

val scores = Map(“Alice”->10, “Bob”->3, “Cindy”->8) //不可变映射

val scores = scala.collection.mutable.Map(“Alice”->10,”Bob”->3) //可变映射

映射 = 数据结构 = 函数key->value

scores(“Bob”)

scores.getOrElse(“Bob”, 0)

scores += (“Bob”->10, “Fred”->7)

scores -= “Alice”

for ((key, value) <- scores) {}

val scores = scala.colleciton.immutable.SortedMap(“Alice”->10, “Bob”->19)

val symbols = Array(“<”, “-“, “>”)

val counts = Array(2, 10, 2)

val map = symbols.zip(counts) //将key集合和value集合zip为一个映射

## 元组

val t = (1, 2.14, “Bob”) 元组

t.\_2 //获取元组的元素

val (first, second, third) = t