## 注解：Annotation

Java 5.0 新增注解特性，是一种特殊的接口。

注解是代码的附属信息，遵循一个基本原则：注解不能直接干扰程序代码的运行，无论增加或删除注解，代码都能够正常运行。

Java语言解释器会忽略这些注解，而由第三方工具负责对注解进行处理。

@interface修饰符定义注解，一个注解可以拥有多个成员，成员声明和接口方法声明类似，成员声明有以下几点限制：

1.成员以无入参，无抛出异常的方式声明

2.可以通过default为成员指定默认值

3.成员类型只能是基本数据类型及其包装类型、String、Class、enum、注解类型

4.如果只有一个成员，则成员的名字必须取为value()

元注解(Meta-Annotation)：Java预定义的注解，被Java编译器使用，会对注解类的行为产生影响。常用的元注解有：

@Retention(RetentionPolicy.RUNTIME) – 注解类信息保留期限

1. RetentionPolicy.SOURCE:注解信息仅保存在目标类的源码文件中，对应的字节码文件将不再保留

2. RetentionPolicy.CLASS:注解信息将进入目标类的字节码文件中，但类加载器加载字节码文件时不会将注解加载到JVM中，即运行期不能获取注解信息

3. RetentionPolicy.RUNTIME:注解信息在目标类加载到JVM后依然保留，在运行期可以通过反射机制读取类中的注解信息。

@Target(ElementType.METHOD):注解类的应用范围

## 反射：Reflect

源文件 🡪 类文件(二进制) 🡪 机器码

类对象:Class

元数据:方法、字段、构造函数、注解等

类文件需包含以下部分：

* 魔法数(CA FE BA BE)
* 使用的类文件标准版本
* 当前类的常量池
* 访问标志(abstract、public等)
* 当前类的名称
* 继承信息
* 实现的接口
* 字段
* 方法
* 属性

反射是在运行时审查、操作和修改对象的能力，可以修改对象的结构和行为。

## Swich

swich语句是一个分支语句，用来替代多层if判断的场景。使用结构如下：

swich(n){

case 1:

语句块1

break;

…

default:

语句块2

break;

}

其中n是一个表达式，这个表达式的返回值必须是：整型(int、short、char、byte)或其包装类型、String、枚举类型。

关于类型的说明：

1.为什么只能是整型，不能是float和double？

浮点型在计算机中只是一种近似的表示，总是会有精度的，因此不能很准确的进行比较。float是32位近似值，至少有6个有效数字；double是64位近似值，至少有15个有效数字。

2.long也是一种整型，为什么不能用long？

swich的本质是一种穷举，对于long类型来说，占64位，数据范围很大(19位数字)，对于int类型而已，32位已经能表示21亿的数据范围了，已经足够大了，所以根本无需使用到范围更大的long类型。

3.swich为什么能用String？

在Java 7之前，swich只支持整型和枚举，Java 7开始添加对String的支持，其本质，在内部是比较String的hashCode()返回值(int类型).

4.swich支持枚举又是何道理？

Java中的枚举是类的变种，而且运行使用的值很少，是能穷举完的。

## 枚举：Enum

Java中的枚举是类的变种，功能有限，而且允许使用的值很少。

由于枚举是特殊的类，所以可以拥有成员(字段和方法)，如果字段或方法有主体的话，那么实例列表后面必须加上分号。

枚举实例是由Java运行时创建的，在外部是不能实例化的，所以构造函数需要声明为private。

枚举有如下的特性：

1) 都隐式扩展自java.lang.Enum类

2) 不能泛型化

3) 可以实现接口

4) 不能被继承

5) 只能有一个private(或default)的构造方法

6) 如果枚举中的所有值都有实现主体，那么只能定义为抽象方法

## String

Java中的字符串是由一系列的Unicode字符组成。

在Java中，只有字符串常量是共享的，使用+或substring等操作产生的结果并不是共享的。

+ 运算符连接字符串时，实际上是先创建一个使用StringBuilder对象表现的工作区，javac会自动创建上述代码。

String是不可变类

StringBuilder可变的

StringBuffer 同步的，线程安全

当使用 s2 = "Monday" 这样的表达式创建字符串的时候，JAVA首先会在这个String缓冲池中寻找相同值的对象，找到了就将引用赋值给s2，找不到就把Monday添加到缓冲池，再把引用赋值给s2；

使用了 new 操作符，会创建一个新的String对象。

s2.intern()会将s2中值复制到常量池，如果常量池已经存在该字符串，则直接返回该字符串的引用。

String对象的三个特定：

1) 不变性：String对象一旦生成，则不能再对它进行改变。一些看起来像是修改的操作，实际上都是依靠产生新的字符串实现的。

2) 针对常量池的优化：当两个对象拥有相同值时，它们只引用常量池中的同一个拷贝。

3) 类的final定义：String对象在系统中不可能有任何子类。

String常量池的位置：1.6及之前，存放在永久区，1.7及以上，存放在堆中。

String定义成final：主要是为了“效率” 和 “安全性” 的缘故。若 String允许被继承, 由于它的高度被使用率, 可能会降低程序的性能；调用的操作系统本地的API(本地方法调用), 如果这种类可以被继承的话，我们再把它的方法重写了，往操作系统内部写入一段具有恶意攻击性质的代码什么的;所以String被定义成final。

## 基础类型

基本类型都是存在栈里面的，Int a=3;当你int b=3的时候，jvm会干嘛呢？首先，它会去栈里面找是不是已经有一个数据为3了。如果不是的话新建一个地址，是的话，就把b指向这个地址，**所以对于基本类型来说，当值相同的时候，地址也是相同的**，所以当你对基本类型==的时候，内容相同的肯定返回的是true。因为实质上，他们的地址也是相同的。

但是对于其他类型，都是存在堆里面的动态分配内存，所以每一次新建一个数据都会动态分配一个内存地址，所以==的时候，当然内存地址是不一样的。  
要记住，**==永远是用来比较内存中的地址的，基本类型感觉上是在比较内容，实际上还是在比较地址！**

## 参数传递

\* Java参数，不管是原始类型还是引用类型，传递的都是副本(有另外一种说法是传值，但是说传副本更好理解吧，传值通常是相对传址而言)。

\* 如果参数类型是原始类型，那么传过来的就是这个参数的一个副本，也就是这个原始参数的值，这个跟之前所谈的传值是一样的。如果在函数中改变了副本的 值不会改变原始的值.(**把值复制了一份**)

\* 如果参数类型是引用类型，那么传过来的就是这个引用参数的副本，这个副本存放的是参数的地址。如果在函数中没有改变这个副本的地址，而是改变了地址中的 值，那么在函数内的改变会影响到传入的参数。如果在函数中改变了副本的地址，如new一个，那么副本就指向了一个新的地址，此时传入的参数还是指向原来的 地址，所以不会改变参数的值。

( 对象包括对象引用即地址和对象的内容)(**把地址复制了一份，地址指向的对象内容还是原来的**)

a.传递值的数据类型：八种基本数据类型和String(这样理解可以，但是事实上String也是传递的地址,只是string对象和其他对 象是不同的，string对象是不能被改变的，内容改变就会产生新对象。那么StringBuffer就可以了，但只是改变其内容。不能改变外部变量所指 向的内存地址)。

b.传递地址值的数据类型：除String以外的所有复合数据类型，包括数组、类和接口

## JDK动态代理

InvocationHandler.invoke、 Proxy.newProxyInstance

代理类继承InvocationHandler，并实现invoke，该方法会对待执行的实际方法进行代理。

代理类中通过Proxy.newProxyInstance生成指定接口的代理对象

## 函数编程：Lambda

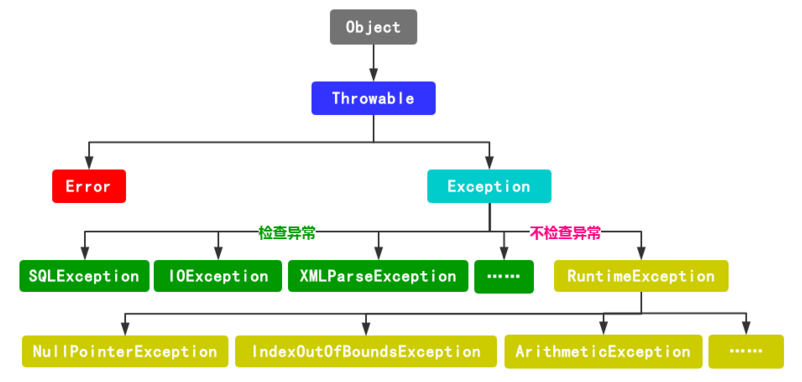
**

*// 输入两个值，返回一个值*BinaryOperator<Long> addExplicit = (Long x, Long y) -> x + y;  
System.***out***.println(addExplicit.apply(100L, 100L));  
  
*/\*\* 函数接口 \*\*/  
// 输入一个值，返回 boolean*Predicate<Integer> atLeast5 = x -> x > 5;  
System.***out***.println(atLeast5.test(10));  
  
*// 输入一个值，返回void*Consumer<Integer> consumer = x -> System.***out***.println(x);  
consumer.accept(20);  
  
*// 输入一个值，返回一个值*Function<Integer, String> ff = x -> x.toString();  
System.***out***.println(ff.apply(120));  
  
*// 无参输入，返回一个值*Supplier<Integer> supplier = () -> 5;  
System.***out***.println(supplier.get());  
  
*// 输入一个值，返回一个值*UnaryOperator<Integer> unaryOperator = x -> x;  
System.***out***.println(unaryOperator.apply(220));

*/\*\* 迭代方式 \*\*/*List<Artist> artists = **new** ArrayList<>();  
*// 内部迭代：Lambda*Long count = artists.stream()  
 .filter(artist -> artist.getOrigin() == **"London"**).count();  
  
*// 外部迭代：foreach循环(封装了迭代的语法糖)***for** (Artist artist : artists) {  
 **if** (artist.getOrigin() == **"London"**) {  
 count++;  
 }  
}  
  
*// 外部迭代：迭代器方式*Iterator<Artist> iterator = artists.iterator();  
**while** (iterator.hasNext()) {  
 Artist artist = iterator.next();  
 **if** (artist.getOrigin() == **"London"**) {  
 count++;  
 }  
}

*// 只描述Stream，最终不产生新集合的方法叫做惰性求值方法：返回值是Stream  
// 最终会从Stream产生值的方法叫做及早求值方法：返回另一个值或void*List<Integer> list = **new** ArrayList<>();  
Random random = **new** Random();  
**for** (**int** i = 0; i < 5; i++) {  
 list.add(random.nextInt(100));  
}  
**for** (Integer integer : list) {  
 System.***out***.println(integer);  
}  
  
*// 这段代码里面的输出语句是不执行的*list.stream().filter(num -> {  
 System.***out***.println(**"filter只是个惰性方法"**);  
 **return** num > 5;  
});  
list.stream().map(num -> num.toString())  
 .forEach(num -> System.***out***.println(num));  
  
*/\*\* 常用的流操作 \*\*/  
// collect()*List<String> collected = Stream.*of*(**"a"**, **"b"**, **"c"**).collect(  
 Collectors.*toList*());  
  
*// map()*List<String> upperCaseList = collected.stream()  
 .map((s) -> s.toUpperCase()).collect(Collectors.*toList*());  
  
*// filter()*upperCaseList.stream().filter((s) -> s == **"A"**).count();  
  
*// flatMap():将多个Stream连接成一个*List<Integer> together = Stream.*of*(*asList*(1, 2), *asList*(3, 4))  
 .flatMap(numbers -> numbers.stream())  
 .collect(Collectors.*toList*());  
*output*(together);  
  
*// max()、min()、count()*Integer maxNum = together.stream()  
 .max(Comparator.*comparing*(num -> num)).get();  
Integer minNum = together.stream()  
 .min(Comparator.*comparing*(num -> num)).get();  
Integer count2 = (**int**) together.stream().filter((num) -> num % 2 == 0)  
 .count();  
System.***out***.println(**"max="** + maxNum + **",min="** + minNum + **",count="** + count2);  
  
*// reduce()***int** sum = Stream.*of*(1, 2, 3).reduce(0, (acc, element) -> acc + element);  
System.***out***.println(**"sum="** + sum);

## 异常体系



Error用来表示编译时和系统错误，一般我们不用关心。

Exception又分为不检查异常(运行时异常)和检查异常。

**运行时异常**类对应于编译错误，它是指Java程序在运行时产生的由解释器引发的各种异常。出现运行错误往往表示代码有错误，如：算数异常（如被0除）、下标异常（如数组越界）等。

**非运行时异常**Non\_RuntimeException类及其子类的实例，又称为检查异常。Java编译器利用分析方法或构造方法中可能产生的结果来检测Java程序中是否含有检测异常的处理程序，对于每个可能的可检测异常，方法或构造方法的throws子句必须列出该异常对应的类。在Java的标准包java.lang java.util 和 java.net 中定义的异常都是非运行异常。

## 数据库事务基础知识

### 事务的特性

数据库事务的4个特性：原子性(Atomic)、一致性(Consistency)、隔离性(Isolation)、持久性(Durability),简称ACID

原子性：组成事务的多个数据库操作是一个不可分割的原子单元；

一致性：事务操作成功后，数据库所处的状态和它的业务规则是一致的；

隔离性：并发数据操作时，不同的事务拥有各自的数据空间，操作不会对对方产生干扰；

持久性：一旦事务提交成功后，事务中的所有数据操作都必须被持久化到数据库中。

### 数据并发的问题

脏读(dirty read)、不可重复读、幻象读、第一类丢失更新、第二类丢失更新。

1.脏读：A事务读取了B事务尚未提交额更改数据，B事务最后没成功

2.不可重复读：A事务两次读取的数据不一致，两次读取中间有其他事务提交了**数据更改**。

3.幻象读：A事务两次读取的数据不一致，两次读取中间有其他事务提交了**数据新增**。

4.第一类丢失更新：A数据**撤销**时，把已经提交的B事务的更新数据覆盖了。

5.第二类丢失更新：A事务**成功提交**，覆盖了B事务已经提交的数据。

### 数据库锁

锁的粒度：表锁、页锁、行锁

锁的种类：共享锁(S:Share)、独占锁(X)、更新锁(U:Update)

更改数据，施加行独占锁

### 事务隔离级别

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 隔离级别 | 脏读 | 不可重复读 | 幻想读 | 第一类丢失更新 | 第二类丢失更新 |
| Read UnCommitted | **允许** | **允许** | **允许** | 不允许 | **允许** |
| Read Committed | 不允许 | **允许** | **允许** | 不允许 | **允许** |
| Repeatable Read | 不允许 | 不允许 | **允许** | 不允许 | 不允许 |
| Serializable | 不允许 | 不允许 | 不允许 | 不允许 | 不允许 |

**Read UnCommitted**:读未提交，就是一个事务可以读取另一个未提交事务的数据。

**Read Committed:** 读提交，就是一个事务要等另一个事务提交后才能读取数据。

**Repeatable Read：**重复读，就是在开始读取数据（事务开启）时，不再允许修改操作。

**Serializable ：**最高的事务隔离级别，事务串行化顺序执行。

**值得一提的是：大多数数据库默认的事务隔离级别是Read committed，比如Sql Server , Oracle。Mysql的默认隔离级别是Repeatable read。**

## ThreadLocal

1.2版本引入，5.0版本支持泛型

如果某个对象是非线程安全的，在多线程环境下，对对象的访问必须采用synchronized进行线程同步；线程同步会降低并发性，影响系统性能。

TreadLocal为解决多线程程序的并发问题提供一种新的思路；ThreadLocal是一个保存线程本地化对象的容器；当运行于多线程环境的某个对象使用ThreadLocal维护变量时，ThreadLocal为每个使用该变量的线程分配一个独立的变量副本。

ThreadLocal和线程同步机制都是为了解决多线程中相同变量的访问冲突问题。

在同步机制中，通过对象的锁机制保证同一时间只有一个线程访问变量，该变量是多个线程共享的；

ThreadLocal为每个线程提供一个独立的变量副本，从而隔离了多个线程对访问数据的冲突；

概况而言，对于多线程资源共享的问题，同步机制采用“时间换空间”的方式：访问串行化，对象共享化；而ThreadLocal采用“空间换时间”的方式：访问并行化，对象独立化。

## Java 集合框架



|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 类 | 内部表现 | 首次出现版本 | 元素顺序 | 成员限制 | 基本操作 | 迭代性能 | 备注 |
| HashSet | 哈希表 | 1.2 | 无 | 无 | O(1) | O(capacity) | 最佳通用实现 |
| LinkedHashSet | 哈希链表 | 1.2 | 插入顺序 | 无 | O(1) | O(n) | 保留插入顺序 |
| EnumSet | 位域 | 5.0 | 枚举声明 | 枚举类型 | O(1) | O(n) | 值不能为null |
| TreeSet | 红黑树 | 1.2 | 升序排列 | 可比较 | O(log(n)) | O(n) | 元素需要实现Comparable或Comparator接口 |
| CopyOnWriteArraySet | 数组 | 5.0 | 插入顺序 | 无 | O(n) | O(n) | 线程安全 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 类 | 表示方式 | 首次出现版本 | 随机访问 | 备注 |
| ArrayList | 数组 | 1.2 | 能 | 最佳全能实现 |
| LinkedList | 双向链表 | 1.2 | 否 | 高效插入和删除 |
| CopyOnWriteArrayList | 数组 | 5.0 | 能 | 线程安全,遍历快,修改慢 |
| Vector | 数组 | 1.0 | 能 | 已过时,同步 |
| Stack | 数组 | 1.0 | 能 | 扩展Vector,添加了push()/pop()/peek(),过时，用Deque替代 |

列表中的元素在集合里是有位置的(下标)

数组和数组列表的缺点:从数组的中间位置删除元素要付出很大的代价，原因是数组中处于被删除元素之后的所有元素都要向数据的前端移动。

Java中的链表都是双向列表，删除和添加元素效率很高，不支持随机访问，指定访问链表中第n个元素，需要越过前面n-1个元素。

使用列表唯一的理由是：尽可能减少在列表中间插入或删除元素所服务的代价，如果列表元素不多，完全可以使用ArrayList。

Vector的所有方法都是同步的，多线程环境下可以安全的访问Vector对象，但同步操作需要耗费大量时间；而ArrayList方法不是同步的。

HashSet为对象计算hashCode，然后再存到对应的地方(称为桶)，基于链表实现，每个链表成为桶。要想查找表中对象的位置，先计算对象的hashCode，然后与桶的总数取余，得到的结果就是保存这个元素的桶的索引。

TreeSet是一个有序的集合，可以以任意顺序将元素插入到集合中，每个值将自动按照元素排序后的顺序呈现。

队列，从尾部添加元素，头部删除元素，Java 6引入Deque接口和ArrayDeque、LinkedList类实现



|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 类 | 表现方式 | 出现版本 | null键 | null值 | 备注 |
| HashMap | 哈希表 | 1.2 | 是 | 是 | 通用实现 |
| ConcurrentHashMap | 哈希表 | 5.0 | 否 | 否 | 通用线程安全实现 |
| ConcurrentSkipListMap | 哈希表 | 6.0 | 否 | 否 | 专用的线程安全实现 |
| EnumMap | 数组 | 5.0 | 否 | 是 | 键是枚举类型 |
| LinkedHashMap | 哈希表+列表 | 1.4 | 是 | 是 | 保留插入或访问顺序 |
| TreeMap | 红黑树 | 1.2 | 否 | 是 | 按键排序 |
| IdentityHashMap | 哈希表 | 1.4 | 是 | 是 | 比较时使用==,而不是equals() |
| WeakHashMap | 哈希表 | 1.2 | 是 | 是 | 不会阻止垃圾回收键 |
| HashTable | 哈希表 | 1.0 | 否 | 否 | 过时，同步 |
| Properties | 哈希表 | 1.0 | 否 | 否 | 使用String类的方法扩展Hashtable |

Map，HashMap与TreeMap，TreeMap会对Key进行排序。

最终的集合框架

获取长度：

// args.length; 数组长度

// "".length(); 字符串长度

// list.size() 集合框架长度

// set.size() 集合框架长度

// map.size() 集合框架长度

对于集合，需要重点关注下面问题：

1).是否允许空

2).是否允许重复数据

3).是否有序:读取数据的顺序和存放数据的顺序是否一致

4).是否线程安全

### ArrayList

以数组形式实现的集合。能动态扩容

允许空元素、允许重复数据、数据有序、非线程安全。

ArrayList的优点：

1、ArrayList底层以数组实现，是一种随机访问模式，再加上它实现了RandomAccess接口，因此查找也就是get的时候非常快。

2、ArrayList在顺序添加一个元素的时候非常方便，只是往数组里面添加了一个元素而已

ArrayList的缺点：

1、删除元素的时候，涉及到一次元素复制，如果要复制的元素很多，那么就会比较耗费性能

2、插入元素的时候，涉及到一次元素复制，如果要复制的元素很多，那么就会比较耗费性能

因此，ArrayList比较适合顺序添加、随机访问的场景。

ArrayList是线程非安全的，因为ArrayList中所有的方法都不是同步的，在并发下一定会出现线程安全问题。

想要使用ArrayList并且让它线程安全可以用Collections.synchronizedList方法把ArrayList变成一个线程安全的List。

另一个方法就是Vector，它是ArrayList的线程安全版本，其实现90%和ArrayList都完全一样，区别在于：

1、Vector是线程安全的，ArrayList是线程非安全的

2、Vector可以指定增长因子，如果该增长因子指定了，那么扩容的时候会每次新的数组大小会在原数组的大小基础上加上增长因子；如果不指定增长因子，那么就给原数组大小\*2，源代码是这样的：

int newCapacity = oldCapacity + ((capacityIncrement > 0) ?

capacityIncrement : oldCapacity);

Vector虽然是线程安全的，但是只是一种相对的线程安全而不是绝对的线程安全，它只能够保证增、删、改、查的单个操作一定是原子的，不会被打断，但是如果组合起来用，并不能保证线程安全性。

### LinkedList

LinkedList是基于双向链表实现的。

只保留第一个元素、最后一个元素、以及链表的长度。

LinkedList允许空元素、允许重复元素、数据有序、非线程安全。

LinkedList和ArrayList的对比：

1、顺序插入速度ArrayList会比较快，因为ArrayList是基于数组实现的，数组是事先new好的，只要往指定位置塞一个数据就好了；LinkedList则不同，每次顺序插入的时候LinkedList将new一个对象出来，如果对象比较大，那么new的时间势必会长一点，再加上一些引用赋值的操作，所以顺序插入LinkedList必然慢于ArrayList

2、基于上一点，因为LinkedList里面不仅维护了待插入的元素，还维护了Entry的前置Entry和后继Entry，如果一个LinkedList中的Entry非常多，那么LinkedList将比ArrayList更耗费一些内存

3、数据遍历的速度，结论是：使用各自遍历效率最高的方式，ArrayList的遍历效率会比LinkedList的遍历效率高一些

4、有些说法认为LinkedList做插入和删除更快，这种说法其实是不准确的：

（1）LinkedList做插入、删除的时候，慢在寻址，快在只需要改变前后Entry的引用地址

（2）ArrayList做插入、删除的时候，慢在数组元素的批量copy，快在寻址

所以，如果待插入、删除的元素是在数据结构的前半段尤其是非常靠前的位置的时候，LinkedList的效率将大大快过ArrayList，因为ArrayList将批量copy大量的元素；越往后，对于LinkedList来说，因为它是双向链表，所以在第2个元素后面插入一个数据和在倒数第2个元素后面插入一个元素在效率上基本没有差别，但是ArrayList由于要批量copy的元素越来越少，操作速度必然追上乃至超过LinkedList。

### CopyOnWriteArrayList

首先提两点：

1、CopyOnWriteArrayList位于java.util.concurrent包下，可想而知，这个类是为并发而设计的

2、CopyOnWriteArrayList，顾名思义，Write的时候总是要Copy，也就是说对于CopyOnWriteArrayList，任何可变的操作（add、set、remove等等）都是伴随复制这个动作的。

CopyOnWriteArrayList允许空元素、允许重复元素、数据有序、线程安全。

CopyOnWriteArrayList内部有一把可重入锁、用来保存元素的数组使用了volatile

CopyOnWriteArrayList的缺点，就是修改代价十分昂贵，每次修改都伴随着一次的数组复制；但同时优点也十分明显，就是在并发下不会产生任何的线程安全问题，也就是绝对的线程安全。

CopyOnWriteArrayList这个并发组件，其实反映的是两个十分重要的分布式理念：

（1）读写分离

我们读取CopyOnWriteArrayList的时候读取的是CopyOnWriteArrayList中的Object[] array，但是修改的时候，操作的是一个新的Object[] array，读和写操作的不是同一个对象，这就是读写分离。这种技术数据库用的非常多，在高并发下为了缓解数据库的压力，即使做了缓存也要对数据库做读写分离，读的时候使用读库，写的时候使用写库，然后读库、写库之间进行一定的同步，这样就避免同一个库上读、写的IO操作太多

（2）最终一致

对CopyOnWriteArrayList来说，线程1读取集合里面的数据，未必是最新的数据。因为线程2、线程3、线程4四个线程都修改了CopyOnWriteArrayList里面的数据，但是线程1拿到的还是最老的那个Object[] array，新添加进去的数据并没有，所以线程1读取的内容未必准确。不过这些数据虽然对于线程1是不一致的，但是对于之后的线程一定是一致的，它们拿到的Object[] array一定是三个线程都操作完毕之后的Object array[]，这就是最终一致。最终一致对于分布式系统也非常重要，它通过容忍一定时间的数据不一致，提升整个分布式系统的可用性与分区容错性。当然，最终一致并不是任何场景都适用的，像火车站售票这种系统用户对于数据的实时性要求非常非常高，就必须做成强一致性的。

最后总结一点，随着CopyOnWriteArrayList中元素的增加，**CopyOnWriteArrayList的修改代价将越来越昂贵，因此，CopyOnWriteArrayList适用于读操作远多于修改操作的并发场景中。**

### HashMap

Key和value都允许为空；key重复会覆盖、value允许重复；数据是无序的；非线程安全。

内部实现是一个数组，数组元素是单向链表。

HashMap，它的顺序是基于HashCode，HashCode是一个随机性很强的数字，所以HashMap中的Entry完全是随机存放的。

HashMap中对Key的HashCode要做一次rehash，防止一些糟糕的Hash算法生成的糟糕的HashCode，那么为什么要防止糟糕的HashCode？

糟糕的HashCode意味着的是Hash冲突，即多个不同的Key可能得到的是同一个HashCode，糟糕的Hash算法意味着的就是Hash冲突的概率增大，这意味着HashMap的性能将下降，表现在两方面：

1、有10个Key，可能6个Key的HashCode都相同，另外四个Key所在的Entry均匀分布在table的位置上，而某一个位置上却连接了6个Entry。这就失去了HashMap的意义，HashMap这种数据结构性高性能的前提是，Entry均匀地分布在table位置上，但现在确是1 1 1 1 6的分布。所以，我们要求HashCode有很强的随机性，这样就尽可能地可以保证了Entry分布的随机性，提升了HashMap的效率。

2、HashMap在一个某个table位置上遍历链表的时候的代码：

1

if (e.hash == hash && ((k = e.key) == key || key.equals(k)))

看到，由于采用了”&&”运算符，因此先比较HashCode，HashCode都不相同就直接pass了，不会再进行equals比较了。HashCode因为是int值，比较速度非常快，而equals方法往往会对比一系列的内容，速度会慢一些。Hash冲突的概率大，意味着equals比较的次数势必增多，必然降低了HashMap的效率了。

HashMap和Hashtable的区别：

HashMap和Hashtable是一组相似的键值对集合，它们的区别也是面试常被问的问题之一，我这里简单总结一下HashMap和Hashtable的区别：

1、Hashtable是线程安全的，Hashtable所有对外提供的方法都使用了synchronized，也就是同步，而HashMap则是线程非安全的

2、Hashtable不允许空的value，空的value将导致空指针异常，而HashMap则无所谓，没有这方面的限制

3、上面两个缺点是最主要的区别，另外一个区别无关紧要，我只是提一下，就是两个的rehash算法不同，Hashtable的是：

private int hash(Object k) {

    // hashSeed will be zero if alternative hashing is disabled.

    return hashSeed ^ k.hashCode();

}

这个hashSeed是使用sun.misc.Hashing类的randomHashSeed方法产生的。HashMap的rehash算法上面看过了，也就是：

static int hash(int h) {

// This function ensures that hashCodes that differ only by

// constant multiples at each bit position have a bounded

// number of collisions (approximately 8 at default load factor).

h ^= (h >>> 20) ^ (h >>> 12);

return h ^ (h >>> 7) ^ (h >>> 4);

}

### LinkedHashMap

通过维护一个运行于所有条目的双向链表，LinkedHashMap保证了元素迭代的顺序。

LinkedHashMap key和value都允许空；key重复会覆盖、value允许重复；数据是有序的；非线程安全。

关于LinkedHashMap，先提两点：

1、LinkedHashMap可以认为是HashMap+LinkedList，即它既使用HashMap操作数据结构，又使用LinkedList维护插入元素的先后顺序

2、LinkedHashMap的基本实现思想就是—-多态。可以说，理解多态，再去理解LinkedHashMap原理会事半功倍；反之也是，对于LinkedHashMap原理的学习，也可以促进和加深对于多态的理解。

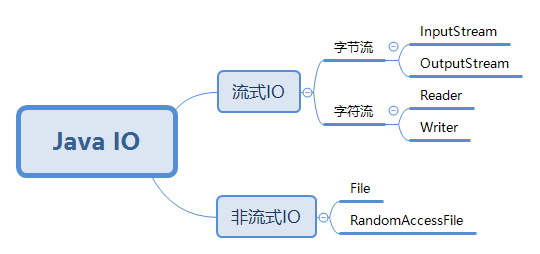
<http://www.importnew.com/25049.html>

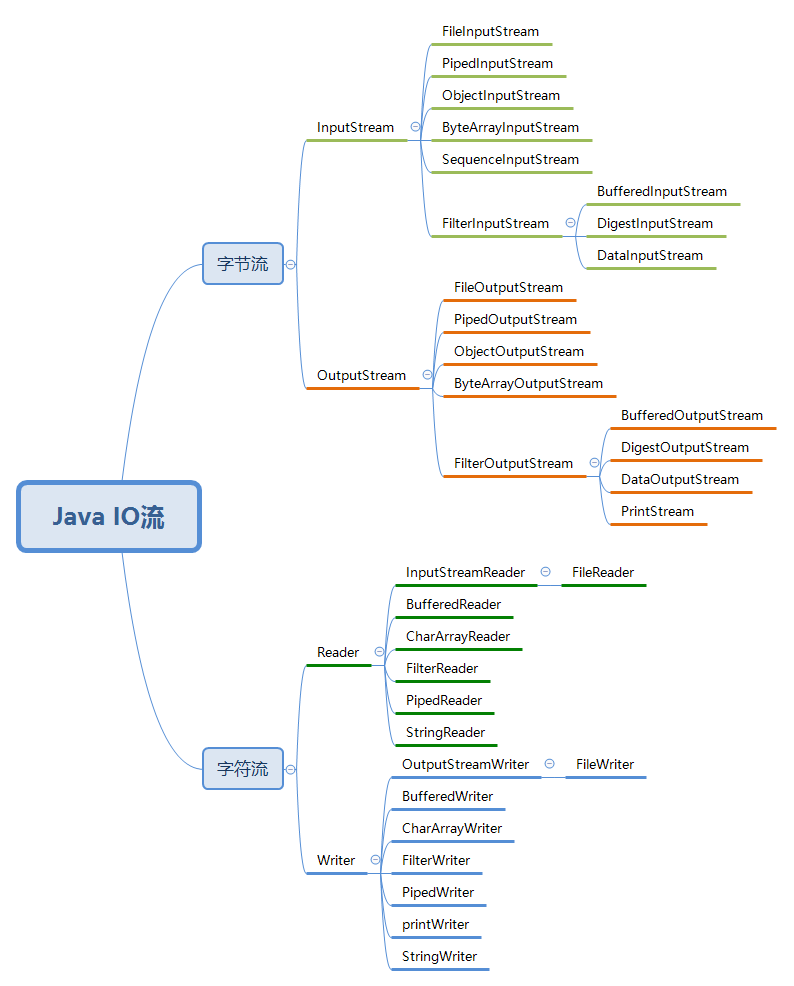
<http://www.importnew.com/23907.html>

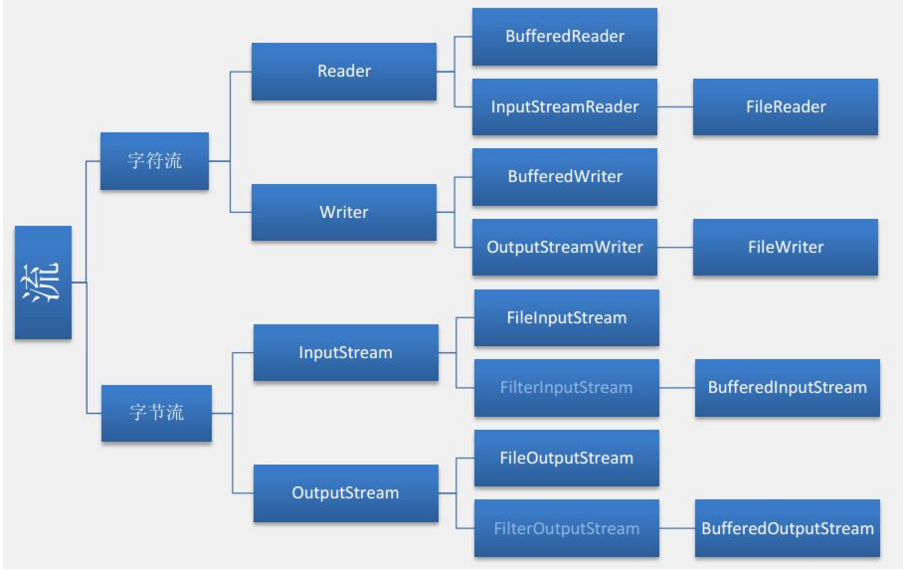
并发编程 – Concurrent 用户指南：<http://www.importnew.com/26461.html>

Java集合系列：<http://www.cnblogs.com/skywang12345/p/3310835.html>

## Java IO体系







**那么字节流和字符流又有什么区别呢？**

1.字节流也称为原始数据，需要用户读入后进行相应的编码转换。而字符流的实现是基于自动转换的，读取数据时会把数据按照JVM的默认编码自动转换成字符。

2.字符流处理的单元为2个字节的Unicode字符，分别操作字符、字符数组或字符串，而字节流处理单元为1个字节，操作字节和字节数组。

所以字符流是由Java虚拟机将字节转化为2个字节的Unicode字符为单位的字符而成的。

3.字节流可用于任何类型的对象，包括二进制对象，而字符流只能处理字符或者字符串，字节流提供了处理任何类型的IO操作的功能，但它不能直接处理Unicode字符，而字符流就可以；

**基于以上的区别，那么什么情况下用字符流，什么情况下用字节流呢？**

如果是音频文件、图片、歌曲，就用字节流好点；如果是中文（文本）的，用字符流更好；

## Java命令行工具

* javac
* java
* jar
* javadoc
* jdeps
* jps
* jstat
* jstatd
* jinfo
* jstack
* jmap
* javap

**javac**:**java源码编译器，把.java源码文件编译成字节码.class文件**

javac some/package/MyClass.java

**java:启动Java虚拟机的可执行文件**

java some.package.MyClass

java –jar my-packaged.jar

**jar:操作.jar文件：创建、更新、索引、列表、提取**

jar [选项] my.jar someDir/

**javadoc:从java源码中生成文档**

javadoc some.package

**jdeps:分析包或类的依赖**

jdeps com.me.MyClass

**jps:列出本地所有活动的JVM进程**

jps

**jstat:指定java进程的一些基本信息**

jstat <pid>

**jstatd:能让本地的JVM的信息通过网络传出去，通过RMI实现**

**jinfo:显示Java进程的系统属性和JVM选项**

jinfo <pid>

**jstack:输出进程中每个Java线程的堆栈跟着**

jstack <pid>

**jmap:查看Java进程的内存分配情况**

jmap <pid>

**javap:java类的反汇编程序，能查看类文件内容的工具**

javap <classname>

VisualVM:聚合上面的工具，并提供图形化界面。Java 6引入。

## 线程池

<https://www.cnblogs.com/aaron911/p/6213808.html>

<https://www.cnblogs.com/dongguacai/p/6030187.html>

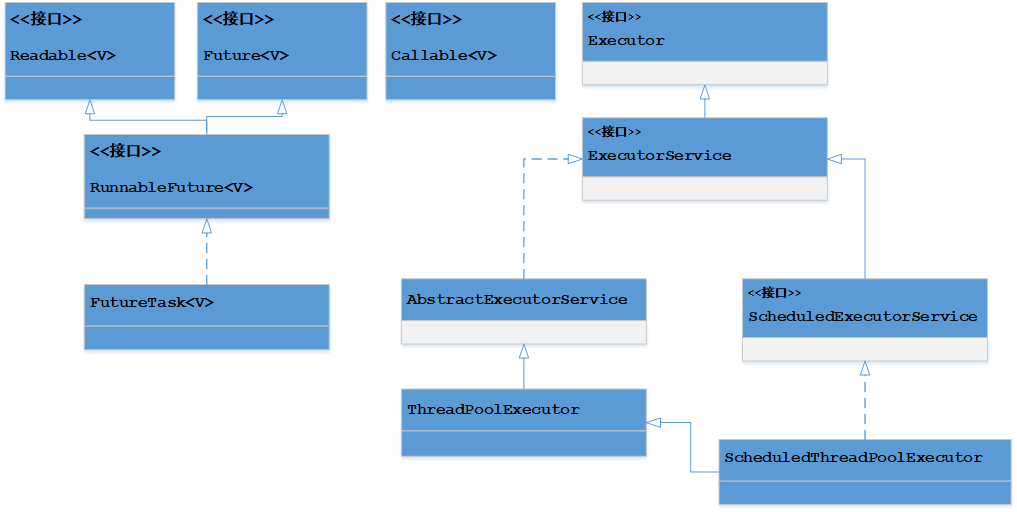
<https://www.cnblogs.com/dongguacai/p/6038960.html>

<http://ifeve.com/java-threadpool/>

<https://blog.csdn.net/youanyyou/article/details/80405873>

### 为什么使用线程池

### JAVA线程池体系



### JAVA线程池原理

提交一个任务到线程池中，线程池的处理流程如下：

1、判断线程池里的核心线程是否都在执行任务，如果不是（核心线程空闲或者还有核心线程没有被创建）则创建一个新的工作线程来执行任务。如果核心线程都在执行任务，则进入下个流程。

2、线程池判断工作队列是否已满，如果工作队列没有满，则将新提交的任务存储在这个工作队列里。如果工作队列满了，则进入下个流程。

3、判断线程池里的线程是否都处于工作状态，如果没有，则创建一个新的工作线程来执行任务。如果已经满了，则交给饱和策略来处理这个任务。



### 线程池优缺点

### 线程池数量设置

开发中我们经常会使用到线程池来处理一些业务，而在不新增设备的情况下，我们所能使用的线程资源又不是无限的，那么高并发、任务执行时间短的业务怎样使用线程池？还有并发不高、任务执行时间长的业务怎样使用线程池？并发高、业务执行时间长的业务怎样使用线程池？

1：高并发、任务执行时间短的业务，线程池线程数可以设置为CPU核数+1，减少线程上下文的切换

2：并发不高、任务执行时间长的业务这就需要区分开看了：

a）假如是业务时间长集中在IO操作上，也就是IO密集型的任务，因为IO操作并不占用CPU，所以不要让所有的CPU闲下来，可以适当加大线程池中的线程数目，让CPU处理更多的业务

b）假如是业务时间长集中在计算操作上，也就是计算密集型任务，这个就没办法了，和（1）一样吧，线程池中的线程数设置得少一些，减少线程上下文的切换

3：并发高、业务执行时间长，解决这种类型任务的关键不在于线程池而在于整体架构的设计，看看这些业务里面某些数据是否能做缓存是第一步，我们的项目使用的时redis作为缓存（这类非关系型数据库还是挺好的）。增加服务器是第二步（一般政府项目的首先，因为不用对项目技术做大改动，求一个稳，但前提是资金充足），至于线程池的设置，设置参考 2 。最后，业务执行时间长的问题，也可能需要分析一下，看看能不能使用中间件（任务时间过长的可以考虑拆分逻辑放入队列等操作）对任务进行拆分和解耦。

---------------------

多线程的底层机制是由操作系统实现的，当一个线程遇到IO阻塞时，例如读写文件，操作系统可能会暂时挂起该线程，从而让其他线程优先执行，也就是将多出来的时间片切分给其他的线程，直到等待该线程的IO操作返回，再重新调度该线程运行。

### CPU密集型(CPU-bound)

CPU密集型也叫计算密集型，指的是系统的硬盘、内存性能相对CPU要好很多，此时，系统运作大部分的状况是CPU Loading 100%，CPU要读/写I/O(硬盘/内存)，I/O在很短的时间就可以完成，而CPU还有许多运算要处理，CPU Loading很高。

在多重程序系统中，大部份时间用来做计算、逻辑判断等CPU动作的程序称之CPU bound。

CPU bound的程序一般而言CPU占用率相当高。这可能是因为任务本身不太需要访问I/O设备，也可能是因为程序是多线程实现因此屏蔽掉了等待I/O的时间。

---------------------

### IO密集型(I/O bound)

IO密集型指的是系统的CPU性能相对硬盘、内存要好很多，此时，系统运作，大部分的状况是CPU在等I/O (硬盘/内存) 的读/写操作，此时CPU Loading并不高。

I/O bound的程序一般在达到性能极限时，CPU占用率仍然较低。这可能是因为任务本身需要大量I/O操作，而pipeline做得不是很好，没有充分利用处理器能力。

---------------------

把任务分为计算密集型和IO密集型。

计算密集型任务的特点是要进行大量的计算，消耗CPU资源，比如计算圆周率、对视频进行高清解码等等，全靠CPU的运算能力。这种计算密集型任务虽然也可以用多任务完成，但是任务越多，花在任务切换的时间就越多，CPU执行任务的效率就越低，所以，要最高效地利用CPU，计算密集型任务同时进行的数量应当等于CPU的核心数。

计算密集型任务由于主要消耗CPU资源，因此，代码运行效率至关重要。Python这样的脚本语言运行效率很低，完全不适合计算密集型任务。对于计算密集型任务，最好用C语言编写。

第二种任务的类型是IO密集型，涉及到网络、磁盘IO的任务都是IO密集型任务，这类任务的特点是CPU消耗很少，任务的大部分时间都在等待IO操作完成（因为IO的速度远远低于CPU和内存的速度）。对于IO密集型任务，任务越多，CPU效率越高，但也有一个限度。常见的大部分任务都是IO密集型任务，比如Web应用。

IO密集型任务执行期间，99%的时间都花在IO上，花在CPU上的时间很少，因此，用运行速度极快的C语言替换用Python这样运行速度极低的脚本语言，完全无法提升运行效率。对于IO密集型任务，最合适的语言就是开发效率最高（代码量最少）的语言，脚本语言是首选，C语言最差。

总之，计算密集型程序适合C语言多线程，I/O密集型适合脚本语言开发的多线程。

---------------------

## JAVA IO模型