JAVA

[1. 基础类型/包装类 1](#_101)

[1.1. 缓存池 1](#_102)

[1.2. 自动装箱时会使用valueOf 1](#_103)

[2. String相关 1](#_104)

[2.1. String不可变 1](#_105)

[2.2. StringBuilder可变，非线程安全 1](#_106)

[2.3. StringBuffer可变，线程安全 1](#_107)

[3. 反射 1](#_108)

[3.1. 优点 1](#_109)

[3.1.1. 可扩展性 1](#_110)

[3.1.2. 类浏览器和可视化开发环境 1](#_111)

[3.1.3. 调试器和测试工具 1](#_112)

[3.2. 缺点 1](#_113)

[3.2.1. 性能开销：JVM无法优化，非反射效率高于反射 1](#_114)

[3.2.2. 安全限制 1](#_115)

[3.2.3. 内部暴露 1](#_116)

[4. 线程 2](#_136)

[4.1. Java线程状态 2](#_164)

[4.2. 线程创建 2](#_190)

[4.3. 线程池 2](#_145)

[4.3.1. Executors 2](#_137)

[4.3.2. ForkJoinPool 2](#_146)

[4.3.3. ThreadPoolExecutor 3](#_144)

[4.4. 线程中断 3](#_148)

[4.4.1. interrupt() 3](#_149)

[4.4.2. isInterrupted() 3](#_150)

[4.4.3. interrupted() 3](#_151)

[5. I/O 3](#_163)

[5.1. 磁盘操作：File可以表示文件和目录，但不表示文件内容 4](#_165)

[5.2. 字节操作 4](#_166)

[5.2.1. InputStream 4](#_169)

[5.2.2. OutputStream 4](#_170)

[5.2.3. 基于装饰器模式 4](#_171)

[5.3. 字符操作 4](#_167)

[5.3.1. char采用UTF-16be 4](#_172)

[5.3.2. Reader和Writer 4](#_173)

[5.4. 对象操作 4](#_168)

[5.4.1. ObjectInputStream 4](#_174)

[5.4.2. ObjectOutputStream 4](#_175)

[5.5. 网络 4](#_176)

[5.6. nio 4](#_177)

[5.6.1. 以块的方式处理数据 4](#_178)

[5.6.2. FileChannel 4](#_179)

[5.6.3. SocketChannel 4](#_180)

[5.6.4. ServerSocketChannel 4](#_181)

[5.6.5. 选择器Selector 4](#_182)

[6. 问题 4](#_191)

[6.1. 接口和抽象类的区别 4](#_192)

[6.2. 字节流和字符流的区别 5](#_193)

[6.3. 自动装箱和拆箱 5](#_204)

[6.4. Scanner为什么比BufferReader慢 5](#_210)

[7. 容器 5](#_117)

[7.1. List 5](#_126)

[7.1.1. ArrayList：基于数组 5](#_127)

[7.1.2. Vector：类似ArrayList，使用synchronized实现线程安全 5](#_128)

[7.1.3. LinkedList：基于双向链表 6](#_129)

[7.1.4. CopyOnWriteArrayList 6](#_134)

[7.2. Map 6](#_122)

[7.2.1. HashTable 6](#_198)

[7.2.2. HashMap 6](#_123)

[7.2.3. LinkedHashMap 6](#_124)

[7.2.4. TreeMap 6](#_125)

[7.2.5. ConcurrentHashMap 6](#_135)

[7.3. Set 7](#_118)

[7.3.1. HashSet：底层实现是HashMap 7](#_119)

[7.3.2. LinkedHashSet：底层实现是LinkedHashMap 7](#_120)

[7.3.3. TreeSet：底层使用TreeMap 7](#_121)

[7.4. Queue 7](#_130)

[7.4.1. LinkedList：双向队列 7](#_131)

[7.4.2. PriorityQueue：基于最小堆实现 7](#_132)

[7.4.3. 线程安全版本的BlockingPriorityQueue 7](#_133)

[8. 注解 7](#_147)

[8.1. 元注解 7](#_205)

[8.1.1. Document 7](#_206)

[8.1.2. Target 7](#_207)

[8.1.3. Inherited 7](#_208)

[8.1.4. Retention 7](#_209)

[9. 同步机制 8](#_153)

[9.1. 关键字 8](#_155)

[9.1.1. synchronized 8](#_154)

[9.1.2. volatile 8](#_157)

[9.2. Lock 8](#_156)

[9.2.1. 条件变量 8](#_158)

[9.2.2. AQS 8](#_159)

[9.2.3. 公平锁/非公平锁 9](#_194)

[9.3. 其他并发工具 9](#_201)

[9.3.1. CountDownLatch 9](#_188)

[9.3.2. CyclicBarrier 9](#_189)

[9.3.3. Exchanger 10](#_202)

[9.3.4. Semaphore 10](#_203)

[10. 1.8的改进 10](#_183)

[10.1. 接口默认方法 10](#_184)

[10.2. lambda 10](#_185)

[10.3. 流式编程 10](#_186)

[10.4. 函数式接口 10](#_187)

[11. 异常 10](#_195)

[11.1. Error 10](#_196)

[11.2. Exception 11](#_197)

# 基础类型/包装类

## 缓存池

比如new Integer(123)和Integer.valueOf(123)。Integer缓存池大小默认为-128~127。

## 自动装箱时会使用valueOf

# String相关

## String不可变

使用final修饰，防止子类继承

不提供任何改变改变方法：防止底层value数组发生变化。

成员均由private修饰：防止直接使用str.value

使用final修饰value数组：实际上没用，使用反射还是可以改变。

所以极端情况下还是能改变String底层数据。

## StringBuilder可变，非线程安全

## StringBuffer可变，线程安全

# 反射

## 优点

### 可扩展性

### 类浏览器和可视化开发环境

### 调试器和测试工具

## 缺点

### 性能开销：JVM无法优化，非反射效率高于反射

### 安全限制

### 内部暴露

# 线程

## Java线程状态

Java线程状态跟操作系统层面的不太一样，分为以下六种状态：

初始化：

运行：对应操作系统中的就绪和运行

阻塞：阻塞于锁，lock和synchronized

等待状态：该状态表示当前线程需要等待其他线程做出一些特定动作（通知或中断），如wait(),join()和park()。

超时等待状态：该状态不同于等待状态，它是可以在指定时间内返回的，如sleep()和等待状态方法的超时版本。

终止：

## 线程创建

有三种方式：

Runnable

Callable

继承Thread

## 线程池

### Executors

#### CachedThreadPool

#### FixedThreadPool

#### SingleThreadExecutor

#### ScheduledThreadPool

#### WorkStealingPool：基于ForkJoinPool

### ForkJoinPool

首先Task需要继承RecursiveTask或RecursiveAction的compute方法。在compute方法中分解子任务，并fork提交至线程池，最后通过join获取子任务的执行结果

### ThreadPoolExecutor

参数：

corePoolSize：核心线程数量

maximumPoolSize：最大线程数量

keepAliveTime：超时时间

timeunit：时间单位

workQueue：任务队列

ThreadFactory：线程工厂

rejectedHandler：拒绝策略，一般在线程数达到maxPoolSize并且队列已满 或 线程池调用shutdown后触发

添加任务流程：

1. 线程数未达到核心数，创建核心线程并执行

2. 队列未满，则加入队列

3. 未达到maxPoolSize，创建线程

4. 采取拒绝策略

## 线程中断

### interrupt()

设置中断标志，这种方式是让一个线程终止的最好方法。

stop方法虽然也可以关闭线程，但是会造成数据不一致和部分资源得不到释放的问题。

其他的线程终止方法还有：设置终止标志位，线程内部不断轮询。

### isInterrupted()

是否设置了中断标志

### interrupted()

判断是否中断了，并清除标记位

# I/O

## 磁盘操作：File可以表示文件和目录，但不表示文件内容

## 字节操作

### InputStream

### OutputStream

### 基于装饰器模式

## 字符操作

### char采用UTF-16be

### Reader和Writer

## 对象操作

### ObjectInputStream

### ObjectOutputStream

## 网络

## nio

### 以块的方式处理数据

### FileChannel

### SocketChannel

### ServerSocketChannel

### 选择器Selector

# 问题

## 接口和抽象类的区别

语法上的区别：

1. 抽象类可以提供成员方法的实现细节，而接口中只能存在public abstract方法（1.8后可以使用default关键字）

2. 抽象类中的成员变量可以是各种类型，而接口中的成员变量只能是public static final类型。

3. 接口中不能含有静态代码块及静态方法，而抽象类可以有。

4. 一个类只能继承一个抽象类，而一个类可以实现多个接口。

设计层面：

1. 抽象类是对一类事物的抽象，而接口则是对行为的抽象。抽象类对整个类整体进行抽象，包括属性、行为，但是接口却是对类局部（行为）进行抽象。比如飞机和鸟都是抽象类，飞行则是接口。

2. 抽象类作为很多子类的父类，是一种模板式设计。而接口是一种行为规范，是一种辐射式设计。

## 字节流和字符流的区别

Java默认采用UTF-16。即无论中文还是英文都用2个字节来表示。

字节流读取字节，字符流读取字符。

## 自动装箱和拆箱

自动装箱和拆箱都是编译期完成的。

从1.5开始就是编译期自动替换完成，装箱使用valueOf()，拆箱使用xxxValue()。

## Scanner为什么比BufferReader慢

1. Scanner的缓冲区比BufferReader小。

2. Scanner对数据进行正则解析，而BufferReader则是简单地读取字符序列。

# 容器

## List

### ArrayList：基于数组

### Vector：类似ArrayList，使用synchronized实现线程安全

### LinkedList：基于双向链表

### CopyOnWriteArrayList

是线程安全的，底层使用0-1目录原理。

优点：

通过锁控制并发写，但是允许并发读。

缺点：

内存占用大，2n

数据不一致，写操作不是立即同步的，操作可见性会受影响。

## Map

### HashTable

线程安全的，key和value都不允许是null。

### HashMap

通过哈希链表解决哈希冲突

遍历：根据结点位置进行遍历，与插入顺序和访问顺序均无关。

put：当某hash桶上的结点过多超出阈值（或哈希链长度大于8并且数组长度小于64），会进行树化。

扩容：达到了capcity\*load-factor时会进行扩容，扩容除了构造更长（2倍）的hash桶之外，还需要对结点进行迁移。

普通链结点：根据(hash & oldCap) == 0划分为两段链表，为0的在原位置，为1的所属桶与原先的桶偏移oldCap。

树化结点：遍历树，并同样根据（hash & oldCap) == 0划分为2个链表，再根据链表长度判断是否需要树化。

HashMap接收key为null。

### LinkedHashMap

使用双向链表来维护元素的顺序（插入顺序或访问顺序）

可以用来实现LRU算法，可根据创建是设定的遍历顺序来有序遍历。

### TreeMap

使用红黑树作为底层数据结构。

可以null做为key，只不过自己需要定义一个Comparator支持null。

### ConcurrentHashMap

CAS和synchronized控制并发写，尾插法保证了并发读写。

并发扩容，当其他线程修改该map时，会从右到左依次分配16个槽交给该线程进行扩容

## Set

### HashSet：底层实现是HashMap

### LinkedHashSet：底层实现是LinkedHashMap

### TreeSet：底层使用TreeMap

## Queue

### LinkedList：双向队列

### PriorityQueue：基于最小堆实现

### 线程安全版本的BlockingPriorityQueue

# 注解

## 元注解

### Document

从注释可以看到 @Document注解用途主要是标识类型是否要被收入javadoc

### Target

标识注解的使用范围，可以赋值为ElementType类型

### Inherited

@Inherited阐述了某个被标注的类型是被继承的。如果一个使用了@Inherited修饰的annotation类型被用于一个class，则这个annotation将被用于该class的子类。

### Retention

可以赋值 RetentionPolicy类型

SOURCE

CLASS

RUNTIME

# 同步机制

## 关键字

### synchronized

### volatile

#### 可见性、有序性

#### lock指令，生成内存屏障

#### 强制写入当前缓存，并引起其他CPU或内核无效化cache

## Lock

### 条件变量

内置的条件变量：wait/notify

Condition：await/signal

注意：在Object监视器模型上，一个对象拥有一个同步队列和等待队列。而并发包中的Lock，则拥有一个同步队列和多个等待队列。

### AQS

AQS的主要原理（分为独占锁和共享锁）：

独占锁：

获得锁：首先调用自定义同步器实现的tryAcquire(int arg)方法，该方法（通过getState(),setState(int arg)和compareAndSetState(int state, int update)）保证线程安全的获取同步状态，如果同步状态获取失败，则构造同步结点（独占式Node，EXCLUSIVE，同一时刻只能有一个线程成功获取同步状态）并通过addWaiter(Node node)方法将该结点加入到同步队列的尾部（一开始的话会添加一个空的Node，将其waitState改为Signal），最后通过acquireQueued(Node node, int arg)方法，使得该结点自旋地获取同步状态。如果获取不到，则阻塞线程。只有当前驱节点释放锁或者线程被中断，才能被唤醒。

释放锁：会唤醒头结点的后继节点。

共享锁：

获得锁：当tryAcquireShared(int arg)返回值大于0时，表示能够获取到同步状态。

释放锁：需要线程安全地释放，因为可能有多个线程同时释放资源。

#### ReentrantLock

是可重入锁，通过判断当前线程是否为获得锁的线程来决定获取操作是否成功。

又可分为公平锁和非公平锁。默认情况为非公平锁，因为能减少线程唤醒次数，开销更少。

公平锁：如果队列中有等待着的Node，那么进行等待。

非公平锁：不管队列中是否有等待Node，先尝试CAS获得锁，如果失败，再入队。

#### ReentrantReadWriteLock

将state分为两部分：前16位代表读，后16位代表写。

读锁是共享锁，写锁是独占锁。还有锁降级。

### 公平锁/非公平锁

公平锁：在有线程等待某个锁时，加入到等待队列，进行挂起。

非公平锁：不管有没有线程等待，先插队，插队失败才加入等待队列。

非公平锁容易造成饥饿现象，但是能减少线程唤醒次数。

## 其他并发工具

### CountDownLatch

等待多线程完成。

使用countDown和await方法。

内部使用Sync(继承AQS)，通过共享锁的方式进行同步。

1. 获取共享状态：状态为0时表示其他线程都完成了任务，await()。

2. 释放共享状态：countDown()方法

### CyclicBarrier

同步屏障，让一组线程到达一个屏障时被阻塞，直到最后一个线程到达屏障时，屏障才会开门。所有阻塞的线程继续工作。

通过await方法阻塞等待。屏障可通过reset方法重用。

内部通过lock(ReentrantLock)加锁，并将线程加入条件队列，当最后一个线程运行到await处，执行预定义方法，然后唤醒条件队列加入到等待队列，从而唤醒其他线程。

### Exchanger

可交换两个对象（可用于遗传算法）

V exchanger.exchange(V v)

### Semaphore

通过new Semaphore(num)设定固定数量的信号量。

通过acquire和release来获取。

# 1.8的改进

## 接口默认方法

## lambda

## 流式编程

相关方法：

1. 创建流：stream()

2. 过滤：filter(() -> { return true;})

3. 聚合成集合：collect(Collectors.toList());

4. 映射：map(i -> i \* i)

## 函数式接口

# 异常

## Error

Error类一般是指与虚拟机相关的问题，如系统崩溃，虚拟机错误，内存空间不足，方法调用栈溢等。对于这类错误的导致的应用程序中断，仅靠程序本身无法恢复和预防，遇到这样的错误，建议让程序终止。

## Exception

Exception类表示程序可以处理的异常，可以捕获且可能恢复。遇到这类异常，应该尽可能处理异常，使程序恢复运行，而不应该随意终止异常。

Exception类又分为运行时异常（Runtime Exception）和受检查的异常(Checked Exception )，运行时异常如ArithmaticException,IllegalArgumentException，编译能通过，但是一运行就终止了，程序不会处理运行时异常，出现这类异常，程序会终止。而受检查的异常，要么用try...catch捕获，要么用throws字句声明抛出，交给它的父类处理，否则编译不会通过。