

# Thinking in Java

## 第1章 对象导论

### 面向对象

1. 封装：把描述一个对象的属性和行为的代码封装在一个模块中，也就是一个类中，属性用变量定义，行为用方法进行定义，方法可以直接访问同一个对象中的属性。
  2. 抽象：把现实生活中的对象抽象为类。分为 过程抽象 和 数据抽象
    - 数据抽象 -->鸟有翅膀,羽毛等(类的属性)
    - 过程抽象 -->鸟会飞,会叫(类的方法)
1. 继承：子类继承父类的特征和行为。子类可以有父类的方法，属性（非private）。子类也可以对父类进行扩展，也可以重写父类的方法。缺点就是提高代码之间的耦合性。
  2. 多态：多态是指程序中定义的引用变量所指向的 具体类型 和通过该引用变量 发出的方法调用 在编程时并不确定
    - 编译时多态：方法重载
    - 运行时多态：与重写有关，子类向上转型

### 抽象

“命令式”语言要求程序员建立「解空间」（对问题建模处，计算机）和「问题空间」（问题存在处，一项业务）的联系

OOP允许根据问题来描述问题，而不是根据运行解决方案的计算机来描述问题

对象具有状态、行为、标识

### 访问控制

public：所有人

private：类创建者和内部方法

protected：可以被子类访问的private

默认：包访问权限

### 继承

允许“声明多继承”，不允许“实现多继承”

只继承不做任何操作，那么子类和父类完全一样的行为和类型，没什么意义

### 多态

**编译器默认后期绑定：**编译时不确定要运行代码，只确保被调用方法存在，进行参数和返回值类型检查，运行时才根据实际类型执行对应方法（出现于父类形参，传入子类实参）。将子类看作父类的行为：upcasting（向上转型）

## 容器

也称为集合，Java有各种类型不同容器主要是因为 1.提供了不同类型接口和外部行为 2.对某些操作的不同效率

如具有相同类型接口和外部行为的 ArrayList（适合随机访问） LinkedList（适合随机插入）

## 参数化类型（范型）

Java SE 5 加入的，避免了容器在存储对象过程中的向上转型（变为Object）和取出时的向下转型（耗时且可能引发异常）

## 对象创建和生命周期

C语言为追求效率，必须显示指定对象存储空间、类型，对象存储在堆栈（创建释放快，但不灵活，必须知道存储在堆栈中数据确切生命周期，以便上下移动堆栈指针），编译器知道对象生命周期，通过编程来确定何时销毁对象

## 完全采用动态内存分配

Java在运行时才知道，且在运行时在堆（heap）内为对象动态分配内存，通过垃圾回收器自动清理不使用的对象（得益于单根继承的特性）

Java对象存储在堆，但基本数据类型是放在栈中还是放在堆中，这取决于基本类型在何处声明

## 异常处理不是面向对象独有特性

## 并发编程

Java的多线程指的是语言级别的，逻辑上的多线程，虚拟机中的线程状态，不反映任何操作系统线程状态

JDK 1.2 以前对操作系统来说是 用户级线程ULT（UserLevelThreads），操作系统认为还是只有一个 Java进程

JDK1.2及以后通过系统调用，将程序的线程交给了操作系统内核进行调度，现在的Java中线程的本质，其实就是操作系统中的线程

CPU是工厂，车间是进程，车间里的工人是线程，车间里的厕所可以理解为共享内存

## 第二章 一切都是对象（但基本数据类型不是对象）

---

String s是对象引用，对象引用是遥控器，对象是电视机

## 存储

寄存器：在Java语言中不能进行操作

**堆栈**: 存在RAM中，是部分Java数据存储的地方——包括对象引用和基本类型

**堆（通用内存池）**: 存在RAM中，存放Java对象（相较于堆栈的好处：编译器不需要知道存储数据的生命周期）

**常量存储**: 存放在代码内部

**非RAM存储（程序结束后仍保持自己状态）**: 流对象（字节流，发送给另一台机器）持久化对象（存放在磁盘）

## 作用域

变量（如对象引用）只作用于作用域结束之前

```
{  
    int x = 12;  
    {  
        int x = 12; //Illegal in Java, but legal in C/C++("隐藏"较大作用域的变量)  
    }  
}
```

但对象可以存活于作用域之外（只要需要用到的话）

```
{  
    String s = new String("a String");  
} //End of scope 作用域结束时，对象引用 s 消失，但对象还继续占用着内存空间
```

## 基本成员默认值

基本类型用作类的成员时会有默认值boolean=false, char='\u0000', 剩余数值类型默认值都为0

char的默认值 '\u0000' 是 Unicode 中的空字符。和Java中的 null 引用不是一个东西

基本类型	默认值
boolean	false
char	'\u0000'
byte	(byte)0
short	(short)0
int	0
long	0L
float	0.0f
double	0.0d

# 第三章 操作符

## 使用

=, ==, !=, 可以操作所有对象，其他操作符只能操作基本类型

**副作用：**赋值操作=, 自动递增、递减 ++, --。操作符改变了操作数自身的值（一般操作符是用于操作数生成一个新值）

## 赋值

a=b, 给对象赋值传递的是对象引用（别名现象：复制了一份b给a, 此时对象引用a和b同时指向b所指向的对象，a引用被覆盖，其原本指向的对象会交给垃圾回收器处理）

给基本类型赋值就是直接复制了数值

## 算术操作符

+、-、\*、/、%

操作符紧跟一个等号 +=, -=, 此中形式适用于Java所有「二元」操作符，只要有实际意义即可

## 一元加、减操作符

一元减号用于转换数据符号

一元加号只是为了对应一元减号（但唯一作用是把较小类型的操作数提升为int）

```
byte b = 1;
byte c = +b; // 会报错
int d = +b; // 正常运行
```

## 自动递增递减

++、--

## 关系操作符

<、>、<=、>=、==、!=

## 逻辑操作符

&&、||、!

## 短路

&&逻辑表达式产生了false值，后面部分都不进行计算

||逻辑表达式产生了true值，后面部分都不进行计算

## 直接常量

```
int i = 0x2f; //16进制  
int i2 = 0177; //8进制
```

## 指数计数法

e代表的不是我们常理解的自然对数的基数

从FORTRAN开始，到C/C++和Java延续了下来，e代表的是10

```
float f4 = 1e-43f;
```

编译器默认将指数当作double处理，这里如果不加f就会报类型的错

## 按位操作符（按位执行布尔代数运算）

二元：&、|、^、

一元：~（按位“非”，取反）

布尔值特殊情况：按位操作符和逻辑操作符同样的效果，但不会短路（即&产生了false还会计算后面部分）！！！

## 移位操作符

<<左移：低位补0

>>带符号右移：高位补符号位

>>>无符号右移：高位补0（C/C++都没有）

也可以加=号，a<<=b，a左移b次，最后值赋给a

对char byte short移位时，会先转换为int再移位，对它们进行算术运算时也是一样

## 类型转换

除布尔型之外的基本数据类型都可以进行强制类型转换

## 第四章 控制执行流程

### continue、break的label使用

label：和goto一个意思

continue label，跳转到label，但继续执行循环

break label，跳转到label，不执行循环

使用场景：嵌套循环想要从多层嵌套中break或continue

## **switch语句**

选择因子必须是整数值 (integral-selector), 如果是字符串或者浮点数, 则必须用if-else

但Java SE 5 的 enum削弱了这种限制

## **第五章 初始化与清理 (cleanup)**

---

### **构造器**

没有返回值, 和void返回值都不同, 就是没有

new 表达式倒是返回了一个对新建对象的引用

### **方法重载 (overloading)**

方法名相同, 区分通过: 参数类型列表

### **this关键字 (本质就是一个对象引用)**

只出现在非static方法内部, 对象引用, 指向调用方法的那个对象

使用场景: 将自身传递给外部方法、构造器中调用构造器

### **finalize(): 执行特殊清理工作**

使用场景:

- 用于清理一种被特殊分配内存的对象, 即调用非java代码分配了内存的对象
- 用于在验证程序终结条件, 是否有未被清理的地方

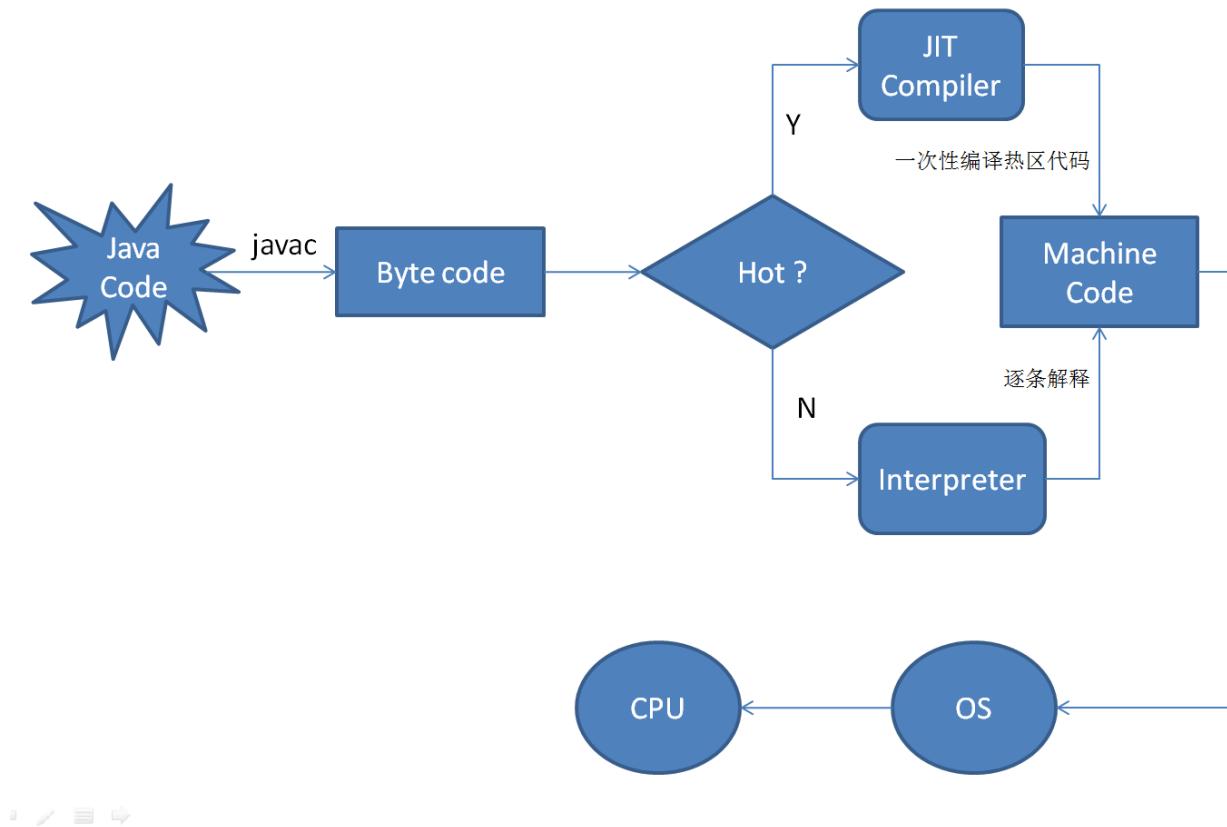
在垃圾回收发生时, 会首先调用对象finalize()方法, 并在下一次垃圾回收时才真正收回此对象占用内存

注意:

1. 对象可能不被垃圾回收 (垃圾回收并不是一定发生的)
2. 垃圾回收不等于“析构 (C/C++)”
3. 垃圾回收只和内存相关

### **JIT (just in time)**

热点代码检测, JIT判断字节码中哪些是频繁执行的, 编译成机器码。如果是用的很少, 就解释执行



**编译 Compile**: 把整个程序源代码翻译成另外一种代码，然后等待被执行，发生在运行之前，产物是「另一份代码」。 **解释 Interpret**: 把程序源代码一行一行的读懂然后执行，发生在运行时，产物是「运行结果」。

## 数组初始化

### 1. 基本数据类型数组

```
int[] a = {};
int[] a = new int[n]; 运行时创建，这时数组元素若是基本类型会自动初始化默认值
```

### 2. 对象数组（本质上是一个引用数组）

```
Integer[] a = {new Integer(1), new Integer(2), 3, };
Integer[] a = new Integer[]{new Integer(1), new Integer(2), 3, };
```

## 可变参数列表Varargs (Java SE 5添加)

只能对最后一个形参使用

**使用场景**: 在不确定方法需要处理的对象的数量时可以使用可变长参数，会使得方法调用更简单，无需手动创建数组 `new T[] {...}`

调用方式:

```

public class Varargs {
    public static void test(String... args) {
        for(String arg : args) {
            System.out.println(arg);
        }
    }
    public static void main(String[] args) {
        test(); //0个参数
        test("a"); //1个参数
        test("a", "b"); //多个参数
        test(new String[] {"a", "b", "c"}); //直接传递数组
    }
}

```

## 枚举enum (Java SE 5添加)

对象是有限且固定的类，其地位与 class、interface 相同；

枚举类是一种特殊的类，有自己的成员变量、成员方法、构造器（只能使用 **private** 访问修饰符，构造器只在构造枚举值时被调用）；

使用 enum 定义的枚举类默认继承了 java.lang.Enum 类，并实现了 java.lang.Seriable 和 java.lang.Comparable 两个接口；

所有的枚举值都是 **public static final** 的，且非抽象的枚举类不能再派生子类；

枚举类的所有实例(枚举值)必须在枚举类的第一行显式地列出，否则这个枚举类将永远不能产生实例

```

public enum WeekEnum {
    // 因为已经定义了带参数的构造器，所以在列出枚举值时必须传入对应的参数
    SUNDAY("星期日"), MONDAY("星期一"), TUESDAY("星期二"), WEDNESDAY("星期三"),
    THURSDAY("星期四"), FRIDAY("星期五"), SATURDAY("星期六");
    // 定义一个 private 修饰的实例变量
    private String date;
    // 定义一个带参数的构造器，枚举类的构造器只能使用 private 修饰
    private WeekEnum(String date) {
        this.date = date;
    }
    // 定义 get set 方法
    public String getDate() {
        return date;
    }
    public void setDate(String date) {
        this.date = date;
    }
}

```

枚举类对象两个默认的字段：**name:String**和**ordinal:int**，分别存放枚举值、枚举值对应的int（0开始）

枚举类的每个枚举值就是该枚举类的一个对象

和其他非抽象类不同，枚举类可以定义抽象方法，但其所有枚举值必须实现该方法

```
public enum Operation {  
    // 用于执行加法运算  
    PLUS { // 花括号部分其实是一个匿名内部子类  
        @Override  
        public double calculate(double x, double y) {  
            return x + y;  
        }  
    },  
    // 用于执行减法运算  
    MINUS { // 花括号部分其实是一个匿名内部子类  
        @Override  
        public double calculate(double x, double y) {  
            // TODO Auto-generated method stub  
            return x - y;  
        }  
    },  
    // 用于执行乘法运算  
    TIMES { // 花括号部分其实是一个匿名内部子类  
        @Override  
        public double calculate(double x, double y) {  
            return x * y;  
        }  
    },  
    // 用于执行除法运算  
    DIVIDE { // 花括号部分其实是一个匿名内部子类  
        @Override  
        public double calculate(double x, double y) {  
            return x / y;  
        }  
    };  
    //为该枚举类定义一个抽象方法，枚举类中所有的枚举值都必须实现这个方法  
    public abstract double calculate(double x, double y);  
}
```

## 第六章 访问权限控制

public: 所有人

private: 类创建者和内部方法

protected: 修饰的成员对于本包和其子类可见

- 基类的protected成员是包内可见的，并且对子类可见；
- 若子类与基类不在同一包中，那么在子类中，子类实例可以访问其从基类继承而来的protected方法，而不能访问基类实例的protected方法。

默认：包访问权限（有时也表示成friendly）

	Class	Package	Subclass	Subclass	World		
			(same pkg)	(diff pkg)			
public	+	+	+	+	+	+	+
protected	+	+	+	+	+	+	
no modifier	+	+	+	+			
private	+						

+ : accessible      blank : not accessible

## 类访问权限

如果不加默认是包访问权限：只有同一包下的类可以创建该类对象，但如果此类有一个public static的字段，则包外的类仍可以使用此字段

# 第七章 复用类

相比于继承，我们更优先使用组合

## 组合

类字段里包含其他类的对象引用（显示的用到其他类的行为）

## 继承

子类对象里都有一个隐藏的基类对象：初始化子类之前必须初始化基类（隐式的用到其他类的行为）

## 代理

主要作用，还是在不修改被代理对象的源码上，进行功能的增强。这在 AOP 面向切面编程领域经常见。

<https://blog.csdn.net/briblue/article/details/73928350>

## final关键字

编译期常量：static和final同时修饰的基本数据类型，定义时必须赋值（不需要对类进行初始化就可以读取）

final可以用来修饰对象引用，但是这个意义仅在于对象引用只能永远指向这个对象，但对象本身仍然可以被更改

final字段必须在定义时赋值，或者是在构造函数中赋值

final方法：为了防止被子类覆盖，private方法隐含就是final方法

**final**类：禁止继承，其所有方法都默认是final方法

## 类的加载

类是在其任何**static**成员第一次被访问时加载的，构造函数也是（**static**）

静态代码块：用**static**声明，jvm加载类时执行，仅执行一次

构造代码块：类中直接用{}定义，每一次创建对象时执行

执行顺序优先级：静态块 > main() > 实例变量初始化 > 构造代码块 > 构造函数

其实实际上 实例变量初始化和构造代码块（实例代码块）实际上是放在构造函数的超类构造函数之后，本类构造代码之前

## 第八章 多态（又名动态或后期或运行时绑定）

绑定：将方法的调用和方法主体关联起来

前期绑定：在程序执行前绑定（面向过程语言默认）

后期绑定：运行时根据对象的类型进行绑定

向上转型的使用场景：在要求基类对象的地方，传入子类方法，这样就不用为每一个子类都单独写一个实现的方法（多态去耦合）

Java中除类**static**和**final**方法（**private**属于**final**方法）之外，其他方法都是后期绑定

协变返回类型：子类重写的基类方法可以返回基类方法返回类型的子类类型

## 纯粹继承与扩展

纯粹继承（is-a关系）：导出类只覆盖基类已有的方法，导出类和基类之间可以相互替换

扩展（is-like-a关系）：导出类有额外的新方法，向上转型后这些方法就无法调用

向下转型是不安全的：java对所有转型都会进行检查

状态模式(**State Pattern**)。利用多态，状态模式可以在runtime改变内部组件的类型，从而完全改变类的行为，因此比继承更灵活。

继承来表现行为间的差异，字段来表现状态上的变化

## 第九章 接口

优先使用类而不是接口

接口和内部类提供了一种将接口和实现分离的更加结构化的方法

## 抽象类（类和接口的折中）

有抽象方法的类是抽象类，抽象类不一定有抽象方法（禁止一个类创建对象，但它又不需要抽象方法）

```
abstract void f(); //抽象方法没有方法体
```

继承抽象类的导出类必须实现所有抽象方法，否则其本身也需要加上**abstract**关键字，成为抽象类

相比于接口，抽象类还可以有具体实现

## 接口

完全抽象的类

使用接口的核心原因：为了能够向上转型为多个基类型（以及由此带来的灵活性）

接口的域：默认就是**public static final**

接口方法：默认就是**public abstract**的

接口没有构造器

接口的多态性是指实现接口的类重写接口的抽象方法

接口可以通过**extends**接口来扩展接口

接口最常用场景：**策略设计模式**，体现在，一个方法提供一个接口类型的参数，让调用者传入有接口具体实现的类的对象，即不同类型的对象来得到了不同的实现。

jdk 1.5之前（没有**enum**关键字），接口被用来创建常量组

接口可以嵌套在类或接口里：但实现接口时，不需要实现其内部嵌套的接口。且**private**接口只能在其定义类里被实现

接口与工厂：<https://www.zhihu.com/question/30351872>

## 第十章 内部类（不同于组合）

---

### 普通内部类（成员内部类）

自动拥有一个指向外部类对象的“秘密”引用

要从外部类的非静态方法创建内部类对象，则需要使用**OuterClass.InnerClass**

不能有**static**成员

内部类具有外部类所有元素的访问权

内部类使用其外部类的对象：**OuterClass.this**

其他类直接创建内部类对象(**.new**)：先创建一个外部类对象 **outer**， **OuterClass.InnerClass inner = outer.new InnerClass();**

静态内部类（嵌套类）其实就相当于成了一个顶级类

### private 内部类

完全阻止依赖于类型的编码、隐藏了实现细节、无法访问不属于公共接口的方法

内部类向上转型：内部类实现了一个外部接口，但调用者访问其外部类提供的方法，得到一个内部类提供的向上转型的接口类型的引用，隐藏了实现细节

# 局部内部类（方法和任意作用域里的内部类）

为什么需要？

- 实现接口，创建并返回对其的引用
- 创建类，不希望其公共可用

作用域和普通变量一样

不能有public、protected、private以及static修饰符的。

什么时候用局部不用匿名内部类？：需要一个命名的构造器，需要不止一个该内部类对象

## 匿名内部类

说明：返回值的生成，与表示这个返回值的类的定义结合在一起

通过new表达式返回的引用自动向上转型

只能访问外部final对象

特点：1. 扩展or实现接口，只能做一样 2. 只能实现一个接口

工厂方法：不必再单独实现一个具体的工厂类，直接在不同类型的实现类里定义一个工厂类型的static字段即可，直接用匿名内部类来给这个字段返回具体的工厂对象

## 嵌套类（静态内部类）

特点：

- 创建嵌套类对象时，不需要外部类对象.new
- 嵌套类不能访问外部类非静态成员
- 可以有static成员

用途：

- 接口嵌套类，成为接口的一部分。用于创建公共代码，供接口的不同实现共同使用
- 嵌套类用做测试类，写测试代码

## 为什么需要内部类

内部类拥有外围类的所有元素的访问权限

最吸引人的原因：每个内部类可以单独实现一个接口，无论外部类是否已经继承了某个接口的实现，对于内部类都没有影响

内部类可以很好的实现隐藏。一般的非内部类，是不允许有private与protected权限的，但内部类可以

多重继承：不同内部类继承多个不同非接口类型（类和抽象类），或者内部类和外部类继承了不同非接口类型（类和抽象类），因为接口本身就允许多继承

多个内部类可以对应同一个接口的不同实现（控制框架）

内部类没有Is-a关系的困扰，是一个独立实体

## 闭包与回调

闭包：是一个可调用对象，记录了来自于创建它的作用域的信息，内部类是面向对象的闭包

回调：对象携带信息，在某一时刻调用初始对象

JAVA并不能显式地支持闭包，但是在JAVA中，闭包可以通过“接口+内部类”来实现。

(内部类实现外部的一个接口，这样外部就可以接受内部类)

## 控制框架

解决响应事件的需求，常见于GUI，事件驱动系统

抽象出控制Event的共有行为成接口

由不同内部类来继承该同一个接口，完成不同的具体实现

## 内部类的继承

内部类的导出类，必须实现一个带参数的构造方法，传入的是外部类的对象

因为这个导出类的基类，也就是内部类是有一个“秘密”的指向其导出类对象的引用的

必须在此构造函数第一行使用`enclosingClassReference.super()`

注意：一般导出类里使用`super`是获取一个指向基类对象的引用

```
//WithInner外部类, Inner内部类, test3继承Inner
class test3 extends WithInner.Inner{
    test3(WithInner wi){
        wi.super(); //enclosingClassReference.super(), 这里和一般导出类的super不一样
    }
}
```

至此这个导出类在创建对象时才具有基本的环境

## 内部类不能被覆盖

继承一个外部类时，其非private的内部类是不会被导出类中的同名内部类覆盖的，他们独立于不同的命名空间

## 内部类标识符（Class文件）

外部类\$内部类，如果是匿名内部类，那么内部类名是一个数字

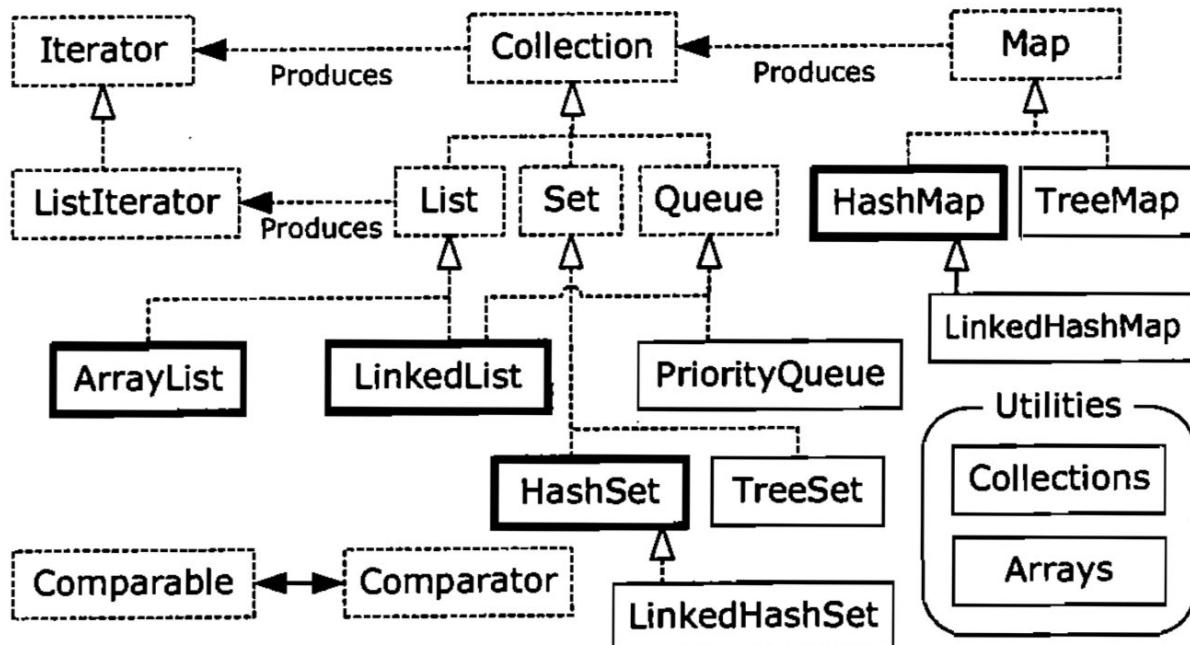
## 第十一章 持有对象

使用场景：用于保存对象引用，但不知道具体会有多少对象，而数组又有固定的尺寸，此时用到容器类（集合类）

预定义范型

```
List<Apple> apples = new ArrayList<>(); //这里使用List接收是因为，修改不同类型List只用在定义处修改
Arrays.<Apple>asList()//这里是 显式类型参数说明
```

尖括号里是类型参数（支持向上转型）



简单的容器分类

## Collection (集合，最高层接口，公共基类，继承自Iterable)

Collection对象初始化方法

```
Arrays.asList() //接收数组或一组泛型类型的可变参数的元素列表,此种情况下,底层为数组,改变List长度就会报UnsupportedOperationException错
Collections.addAll(Collection c, T... list) //接收需要初始化的Collection对象,以及数组或可变参数元素列表
Collection.addAll(Collection c) //这个是由已经定义的Collection对象调用,往自身添加另一个Collection的全部元素,没有Collections.addAll灵活
```

独立元素的序列，槽内只有一个元素，Collection包含List、Set、Queue

### 1.List (接口)

允许重复元素，按插入顺序分为：

**ArrayList** (实体类)：动态数组实现，适合随机访问

**Vector** (实体类)：和 ArrayList 类似，但它是线程安全的，过时的，不要使用

**LinkedList** (实体类，实现了Queue和List接口)：链表实现，适合随机存储，包含的操作也多于ArrayList

添加了作为Stack (栈)、Queue (队列)、double-ended queue (双端队列) 的方法

**Stack** (实体类, Java 1.0实现, 已过时) : 先进后出 (FILO)

**LinkedList**支持的方法可以产生更好的**Stack**

```
public class Stack<T>{  
    private LinkedList<T> storage = new LinkedList<>();  
    public void push(T v) {storage.addFirst(v);}  
    public T peek() {return storage.getFirst();}  
    public T pop() {return storage.removeFirst();}  
    public boolean empty() {return storage.isEmpty();}  
    public String toString() {return storage.toString();}  
}
```

## 2. Set (接口, All Set is a Collection except TreeSet)

不允许重复元素, 查找是Set最重要的操作

**HashSet**: 散列函数->查找最快O(1), 不考虑存储顺序

**TreeSet**: 红黑树->按照比较结果的升序保存对象 (可以传入**Comparator**) , 查找O(logN)

**LinkedHashSet**(实体类, 继承自**HashSet**): 链表维护插入顺序, 具有HashSet查找效率

## 3. Queue (接口, FIFO)

只允许在容器一端插入对象, 并从另一端一移除对象

可靠的将对象从程序某个区域传输到另一个区域的途径

```
Queue.offer() //将元素插入队尾, 或返回false  
Queue.peek()和Queue.element() //返回队头, 若不存在, 返回null 和  
NoSuchElementException  
Queue.poll()和Queue.remove() //移除并返回队头, 若不存在, 返回null 和  
NoSuchElementException
```

**LinkedList** (实体类, 实现了**Queue**和**List**接口) :

getFirst() 等于 element(): 返回列表第一个元素

removeFirst ()等于 remove(): 移除并返回列表第一个元素

addFirst() 等于 add() 等于 addLast(): 都将元素插入列表尾端

removeLast(): 移除并返回列表最后一个元素

**PriorityQueue** (实体类, 优先队列) : 基于堆结构实现, 可以传入**Comparator**来自定义优先级 (默认使用自然排序) , peek、poll和remove是获取队列优先级最高的元素

# Map (又名映射, 关联数组, 最高层接口, 公共基类)

除了用另一个Map, 未提供自动初始化的方式

每个槽内保存了两个对象，键值对

## 1.HashMap (实体类)

哈希实现，提供了最快的查找速度

## 2.Hashtable (实体类)

和 HashMap 类似，但它是线程安全的，这意味着同一时刻多个线程可以同时写入 HashTable 并且不会导致数据不一致。它是遗留类，不应该去使用它

## 3.TreeMap (实体类)

红黑树实现

## 4.LinkedHashMap (实体类，继承自HashMap)

链表维护插入顺序，保留了HashMap的查询速度。顺序为插入顺序或者最近最少使用 (LRU) 顺序

---

# 容器中的设计模式

## 迭代器模式

### Iterator (接口)

- 统一了容器的访问方式
- 只能单向移动
- **foreach**隐式包含了**Iterator!** ! ! ! ! !

```
Iterator.next() //获取下一元素  
Iterator.hasNext() //检查是否还有元素  
Iterator.remove() //从列表中删除迭代器最近返回的元素
```

调用remove之前必须调用next，并且调用next后还只能调用一次remove

[https://blog.csdn.net/qq\\_34115899/article/details/81978660](https://blog.csdn.net/qq_34115899/article/details/81978660) (??但这和我测试的不一样，Java 8 foreach list.remove并没有报错)

### ListIterator(接口，仅用于List)

- 更强大的Iterator子类型
- 可以向前移动
- 可以产生当前位置的前一个和后一个元素索引

## Collection和Iterator

在C++中，没有提供容器的公共基类，C++仅通过迭代器来达成容器之间的共性。

Java把这两种达成容器共性的方式绑定在一起：公共基类 (Collection)，提供统一访问模式迭代器 (Iterator)

同时实现Collection就必须实现其iterator()

Java提供了**AbstractCollection**（抽象类）作为**Collection**的默认实现

而当实现一个不是**Collection**的外部类时，实现**Collection**接口约束很多（要实现所有方法），继承**AbstractCollection**也不是一个很好的选择（因为可能已经继承了其他类），这时为这个类构建一个返回**Iterator**类型对象的方法（这个方法的返回利用了匿名内部类P243）则是将队列和消费队列的方法连接在一起耦合度最小的方式。

## Foreach和迭代器

foreach主要用于数组，但是也可以用于所有**Collection**对象

因为Java SE 5引入的**Iterable**接口，其包含了一个产生**Iterator**类型对象的**iterator()**方法

如果实现了**Iterable**接口，就可以使用foreach方法

**Collection**接口继承自**Iterable**接口（接口可以继承多个接口）

数组不是**Iterable**类型，也不存在自动转换

## 适配器模式

希望构造一个自己的**ArrayList**，并且在拥有默认的前向**Iterator**的同时，也拥有一个逆序的**Iterator**

这个时候如果直接继承**ArrayList**并改写其**iterator**方法，只能实现一个功能的**Iterator**

适配器模式：继承**ArrayList**，保留原有的**iterator**方法不变，新写一个返回逆序**Iterator**类型对象的方法（这个方法的返回利用了匿名内部类P243）

适配器模式：`java.util.Arrays#asList()`把数组类型转换为**List**类型，接收数组或一组泛型类型的可变参数的元素列表

由于是泛型类型可变参数列表，这里不能使用基本数据类型，只能使用相应的包装类型数组

**注意：** `Arrays.asList()`方法产生的**List**是使用的底层数组作为其实现，如果执行的操作不想修改这个底层数组就必须在外面再包一层，创建一个副本进行操作，如 `new ArrayList(Arrays.asList())`。

这也是为什么直接修改**Arrays.asList()**返回**List**的长度会报**UnsupportedOperationException**错的原因

---

## 总结

数组和**Collection**都是建立了**数字索引**和**对象**之间的关联

**数组**：支持基本数据类型，只能存放同一类型，长度不变

**Collection**：不支持基本数据类型，可以存放不同类型（一般不这么做），长度可变

**Map**：建立**对象**和**对象**之间的关联

**Map**和**Collection**之间唯一的重叠：`Collection<V> Map.values()`, `Set<K> keySet()`

不要使用Java 1.0过时的**Vector**, **Hashtable**, **Stack**

# 第十二章 通过异常处理错误

异常的重要目标：把错误处理代码和错误发生地点相分离

粉色是**checked exceptions**, 其必须被 try{}catch语句块所捕获,或者在方法签名里通过throws子句声明

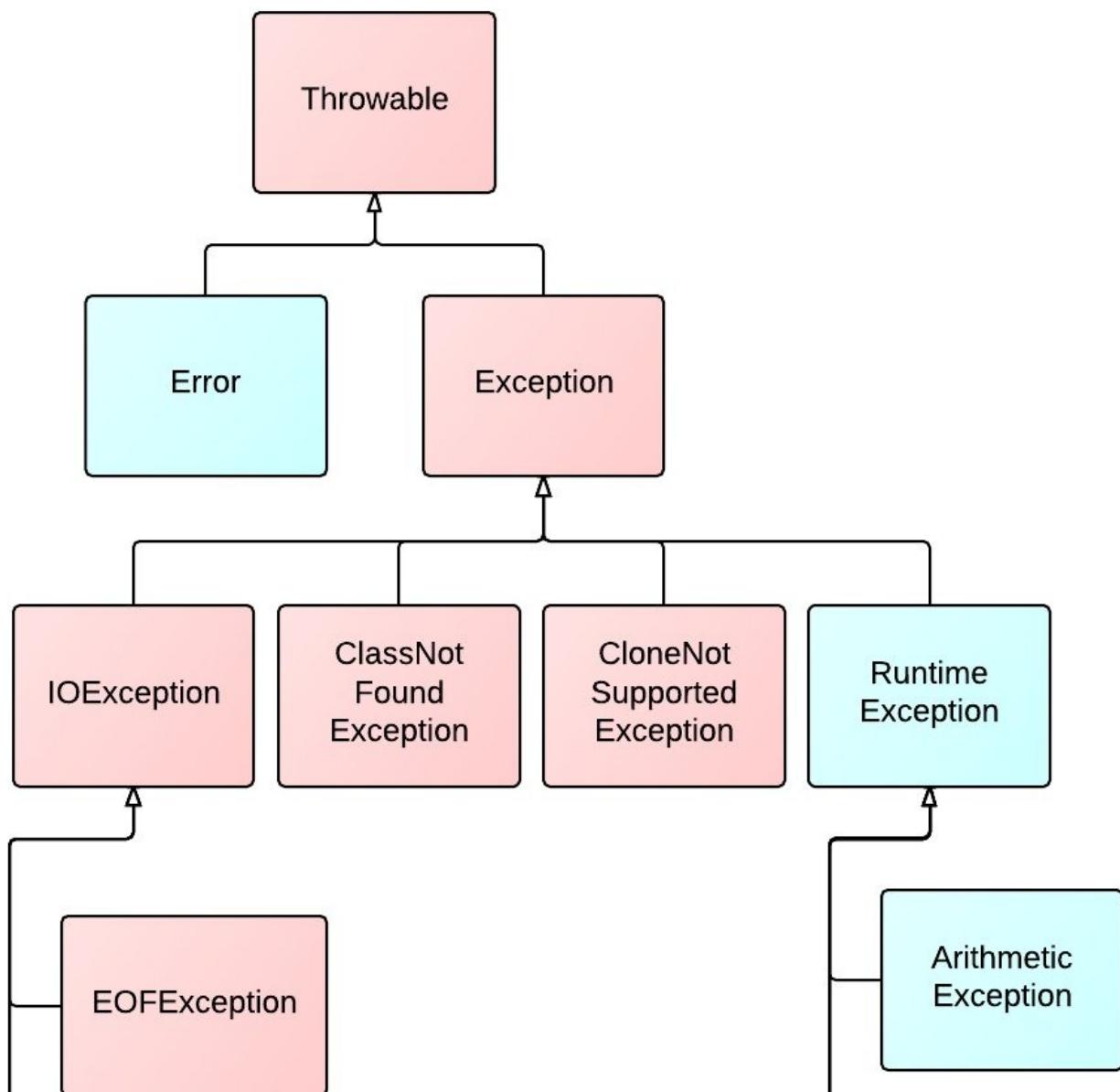
Java编译器要进行检查,Java虚拟机也要进行检查,以确保这个规则得到遵守.

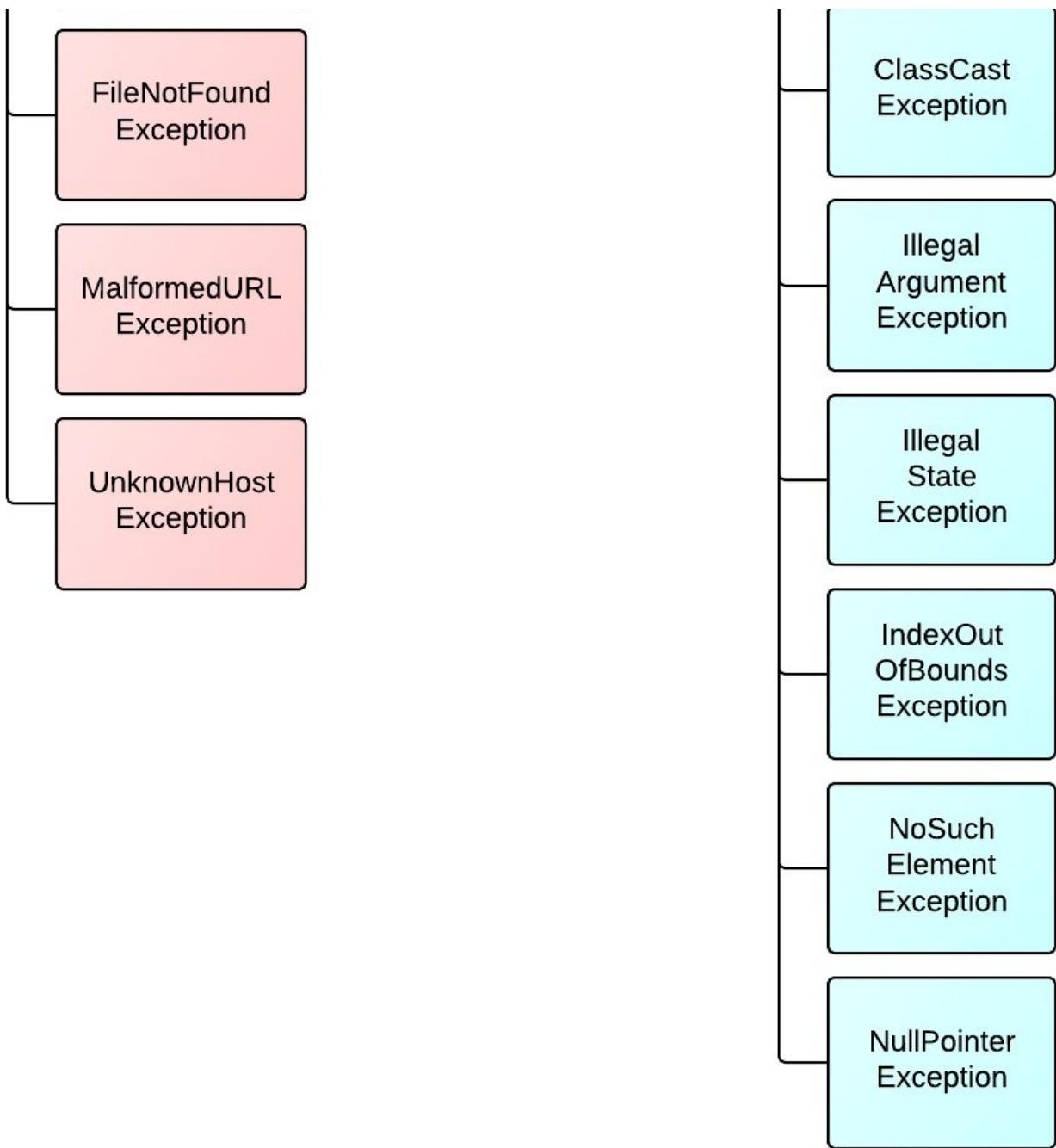
## Error和Exception的区别

**Error类**: 一般是指与JVM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

**runtime exceptions**: 需要程序员自己分析代码决定是否捕获和处理, 比如 空指针, 被0除... 而声明为 Error的, 则属于严重错误, 需要根据业务信息进行特殊处理, Error不需要捕捉。





## 异常参数

标准异常类有两个构造器

- 默认构造器
- 接受字符串参数的构造器

**Throwable**是异常类型的根类

## 捕获异常

监控区域 (guarded region) : 一段可能产生异常的代码，并且后面跟着处理这些异常的代码

异常处理程序 (catch块)

**终止与恢复模型：**Java与C++都是用终止模型。即异常发生后不会在处理后尝试再次执行异常程序，好处是不需要了解异常抛出的地点，相比恢复模型降低了耦合

e.printStackTrace()方法打印从方法调用处直到异常抛出处的方法调用序列， 默认打印到标准错误流

e.printStackTrace(System.out)打印到控制台

## 异常说明

跟在方法名后的**throws**关键字描述的异常

这样做好处是，不用暴露源码也能提醒使用者处理异常

这种在编译时被强制检查的异常成为**被检查的异常（checked exceptions）**

### unchecked and checked exceptions

**未检查的异常（unchecked exceptions）**：Error和RuntimeException 及其子类

**检查了的异常（checked exceptions）**：所有其他的Exception类都是，通常是从一个可以恢复的程序中抛出来的，并且最好能够从这种异常中使用程序恢复。

## 捕获所有异常

printStackTrace()打印的信息可以通过getStackTrace()得到

getStackTrace()返回的是一个由**Java虚拟机栈（VM Stack, JVM运行时数据区之一）**返回的栈轨迹

## 重新抛出异常

如果在catch块里重新抛出当前异常对象，则异常抛出点不会变，异常抛给上一级程序

如果是在catch块里调用，**fillInStackTrace()**方法返回了一个Throwable对象，则调用此方法的这一行变成了一场抛出点（这和在**catch**块里发生了一个新的异常是同样的效果）

## 异常链

把原始异常信息传递给新的异常

Java 5之后，Throwable的子类在构造器中可以接收一个**cause**对象（表示原始异常）作为参数，用于传递给新异常

但只有**Error, Exception, RuntimeException**三个子类提供带cause参数的构造器，链接其他类型异常需要使用**initCause()**方法

## 使用finally进行清理

finally处理内存回收之外（内存由GC处理）的清理：已打开的文件和网络连接，屏幕上图形等

**finally**永远会执行（除非这是dameon线程run方法里的**finally**，那么有可能无法执行）：

- 即使try块中抛出了catch块无法catch的异常
- 即使try块中有break和continue
- 即使try块中有return

## 异常丢失

- 在finally块中return：没有任何输出
- 在finally块中抛出新异常：只会存在finally块中的异常

## 异常的限制（为满足里氏代换原则）

当继承类或实现接口时，至多只允许抛出基类方法异常说明中包含的异常类型（即派生类override的基类方法最少可以不抛出任何异常，但最多只能抛出基类方法包含的异常，或者是这些异常的派生类异常）

这个限制的原因：保证派生类对象向上转型的时候，对基类对象的方法调用能够正常运行

1. 派生类**override**方法异常说明  $\leq$  基类对应方法异常说明

以上限制和构造器无关

但由于派生类构造器会调用基类构造器，所以

2. 派生类构造器异常说明  $\geq$  基类构造器异常说明

## 构造器

构造器中异常，用finally来清理，可能在异常抛出时清理对象还没初始化，而finally却对其清理。

最安全的处理方式：使用嵌套的try catch子句

## 异常匹配

**catch** 基类异常可以捕获其所有派生类异常

## 其他

harmful if swallowed（吞噬则有害）

即catch块里不做任何操作，这样会让异常消失。

一种处理方式是，直接在**catch**块里**throw RunTimeException(e)**

## 第十三章 字符串

### 不可变String（只读）

任何会修改String值的方法，都是创建了一个全新的String对象

注意字符串常量池的概念，代码里所有“test”字符串都会在编译期确定，放入字符串常量池

```
String s1 = "first";
String s2 = new String("second");
s1:中的"first" 是字符串常量，在编译期就被确定了，先检查字符串常量池中是否含有"first"字符串，若没有则添加"first"到字符串常量池中，并且直接指向它。所以s1直接指向字符串常量池的"first"对象。
s2:用new String() 创建的字符串不是常量，不能在编译期就确定，所以new String() 创建的字符串不放入常量池中，它们在堆里有自己的地址空间，然后堆里的这个String对象指向常量池的"second"。但是"second"字符串常量在编译期也会被加入到字符串常量池
```

参数就本应是给方法提供信息的，而不是让方法改变自己的

## 重载 + 和 StringBuilder (Java SE 5引入，之前 StringBuffer 线程安全)

重载：一个操作符在应用于特定的类时，被赋予特殊的意义（String的+和+=是Java仅有的重载过的操作符）

Java不允许程序员重载任何操作符，C++允许

如下，编译器会优化操作符，自动引入StringBuilder，使用其append，避免多次new String带来的开销

```
String s = "a" + "b" + 47;
```

但当在循环里使用重载操作符的时候，**StringBuilder**是在循环体内部构造的，每一次都会new一个，效率低

```
String s = "";
for(String field : fields){
    s += field;
}
```

所以应该在循环内部显示的使用StringBuilder

## 无意识的递归

重写类的toString方法时，如果要打印此类对象内存地址，使用了this，那么会引发递归调用

```
public String toString(){
    return this;
}
```

这里this返回的是本类的一个对象引用，但返回值是String，那么会调用此对象的toString，导致了递归调用

应该使用Object.toString()，即用super.toString()

## 格式化输出 (Java SE 5)

来源于C语言的printf()方法

## System.out.format() <=> System.out.printf()

可用于PrintStream和PrintWriter对象

### Formatter类

构造器可以接受多种输出目的地，常用PrintStream、OutputStream和File

- 提供对空格和对齐的强大控制力
- 类型转换

**String.format:** String类的静态方法，接收和Formatter.format一样的参数，返回格式化的String，内部实现也是用一个Formatter对象

## 正则表达式

注意Java String中的转义字符和正则表达式转义字符的区别

Java String中的“\\\" = \\(正则中的\)

### String中的正则表达式

- String.matches(String regex): 检查String是否和正则表达式匹配，返回true or false
- String.split(String regex ): 就是将字符串从正则表达式匹配的地方切开
- String.replaceAll, String.replaceFirst()

如果不是只用一次，String以外的正则表达式具有更佳的性能

### Pattern (java.util.regex)

Pattern.compile功能更强大的正则表达式对象

## 第十四章 类型信息

### RTTI (Run-time type identification) 运行时状态识别

#### 多态

使用基类引用调用基类方法，但此方法在所有派生类里都有不同实现，并且是动态绑定的，所以都能产生正确的行为，这就是多态。

## Class对象（运行时类型信息的表示，Java依赖其完成RTTI）

每编译一个新类就会为其生成Class对象，保存在.class文件

无论我们对引用进行怎样的类型转换，对象本身所对应的Class对象都是同一个。由于Class对象的存在，Java不会因为类型的向上转换而迷失。这就是多态的原理。

### Class类（所有Class对象都属于这个类）的静态方法

**Class.forName("全限定ClassName")** 获得该Class对象的引用，但返回值通常被忽略，相当于手动加载此类

Object.getClass()获取当前类的Class对象

Class.newInstance()实现虚拟构造器的一种方式

为了使用类做的准备工作

**加载：**类加载器执行。查找字节码.class，并创建一个Class对象

**链接：**验证类中的字节码，为静态域分配空间，有必要的话解析这个类创建的对其他类的引用

**初始化：**如果有超类，对其初始化，执行静态初始化器和静态初始化块

## 类字面常量（也是获得Class对象的引用，但不会初始化类）

**ClassName.class**，可以用于普通类，接口，数组以及基本数据类型（等价于基本类型包装器的TYPE字段）

！！不同于**Class.forName()**，使用类字面常量的时候不会自动初始化Class对象！！，被延迟到了对静态域首次引用时才执行

如果一个域是static而非final，那么读取前必须要进行链接和初始化

## 泛化的Class引用

Java SE 5加入的泛型

```
Class<Integer> intClass = int.class; //泛型使得编译器强制执行了额外的类型检查  
Class<Number> numberClass = int.class;//报错，即使Integer继承于Number，但Integer  
Class不是 Number Class的子类
```

泛型通配符？，搭配extends使用

```
Class<? extends Number> numberClass = int.class;//正确，使得Class引用numberClass可以接受Number类及其所有导出类的Class对象
```

## 新的转型语法

转型多指向下转型，编译器检查向下转型正确性，向上转型不需要检查（编译器允许自由的向上转型）

Java SE 5添加的转型语法，用于普通转型无法使用的情况

Class对象引用的方法

**Class.cast(Object obj)**：将参数obj对象转换成Class引用的类型

普通转型：(ClassName)obj

## RTTI三种形式

1. 传统类型转换，如 (Shape)
2. 代表对象类型的Class对象
3. 关键字instance of（等价于Object.isInstanceOf()）

## instanceof和Class的等价性

查询类型信息时， instanceof()和直接比较Class对象有区别

equals和==直接比较Class对象是否相等， 比较的是是否为确切的这个类型

而instanceof指的是， 你是这个类吗， 或者你是这个类的派生类吗

## 反射（Class类和java.lang.reflect共同支持）

产生背景： RTTI的限制， 类型必须在编译时已知， 才能够通过RTTI来识别

JavaBeans， RMI

但如果我们想要在运行时， 获取编译时不知道的类（未知类）的信息， 需要用到反射

RTTI： 在编译时获取.class文件

反射： 在运行时获取.class文件

**java.lang.reflect**（类库）： 包含Field、 Method、 Constructor

## 动态代理（P338）

### 空对象（没太看懂）

## 第十五章 泛型（Java SE 5引入的重大变化）

产生背景： 要编写可以应用于多种类型的代码

实现了参数化类型的概念

### 元组（数据传送对象， 信使）

```
public class TwoTuple<A,B>(){
    public final A a;
    public final B b;
    public TwoTuple(A a, B b){
        this.a = a;
        this.b = b;
    }
    public String toString(){
        return "(" + a + ", " + b + ")";
    }
}
```

## 生成器（P358）

工厂设计模式的一种应用： 只是工厂方法需要提供参数， 但生成器不需要

## 泛型方法（泛型参数置于返回值前）

泛型方法和其所属类是否为泛型类没有关系，优先使用泛型方法而不是泛化整个类

**static**方法无法访问泛型类的类型参数（因为泛型类的类型是在创建对象时指定的，而**static**方法在此之前就可以被调用）

```
public static <T> ArrayList<T> f(T a){}
```

## 泛型方法的显示类型参数说明（很少用）

在点操作符与方法名之间插入尖括号说明类型 `Arrays.<Apple>asList()`

## 擦除

在泛型代码内部，无法获得任何有关泛型参数类型的信息

泛型→具体类型信息→被擦除

## 边界（泛型的参数类型→设置限制条件）

无界通配符`<?>`

超类型通配符（参数是下界）`<? super T>`：某个特定类的任何基类

`<T extends HasF>`：限制为HasF的类型子集

## 泛型最大的价值

使用容器类的地方

Java SE 5之前容器类里对象都是Object，放入时会丢失类型信息，取出时必须向下转型

## 第十六章 数组

在使用时，除非能确定效率和数组有关，否则，优先使用容器而不是数组

和其他种类容器区别：效率、类型、保存基本类型的能力

效率最高的存储和随机访问对象引用序列的方式

泛型出现后，数组最大的优点就是效率

对象数组元素默认值为**null**，基本类型数组默认值为其作为类成员时的默认值一样

## 数组是第一级对象

数组标识符是一个引用，指向堆中创建的一个真实数组对象，这个数组对象保存指向其他对象的引用

**length**：数组对象唯一一个可以访问的域

## 多维数组

粗糙数组：数组中构成矩阵的每个向量长度可以不一致

`Arrays.deepToString()`: 将多维数组转换成多个String

## 数组与泛型

不能实例化具有参数化类型的数组（但可以创建一个参数化类型数组的引用，并实例化一个具体类型的数组转型赋值给该引用）

```
List<String>[] ls = new ArrayList<>()[] //错误  
  
List<String>[] ls; //正确  
List[] la = new List[10];  
ls = (List<String>[])la;
```

可以参数化数组本身的类型，如 `T[]`

## 创建测试数据

### `Arrays.fill()`

用单一数据填充数组

### 数据生成器（P443）

Generator中嵌套多个不同类，实现不同的生成器逻辑

嵌套类使得可以使用和其要生成类型相同的名字，如嵌套类`Integer`，不会和外部的`Integer`冲突

可以为不同的类型数据选择不同的生成器类型（策略设计模式的实例，每个不同生成器都是一个策略）

如为基本数据类型的包装器类型构造的`CountingGenerator`，内部用不同嵌套类实现不同类型生成器

调用时就可以 `new CountingGenerator.Integer()`

## Arrays实用功能

`equals()`: 比较数组是否相等

`sort()`: 对数组排序，基本类型：快速排序，引用类型：稳定归并排序

`binarySearch()`: 对排序后数组查找元素（只能用于已排序数组）

`toString()`: 产生数组的String表示

`hashCode()`: 产生数组的哈希值

`asList()`: 接收一个可变参数列表，生成List（但是此List的长度不能改变）

### `System.arraycopy()`（复制数组，比for循环快很多）

复制数组，支持基本类型数组和对象数组，但不进行自动装箱和拆箱

## 数组排序

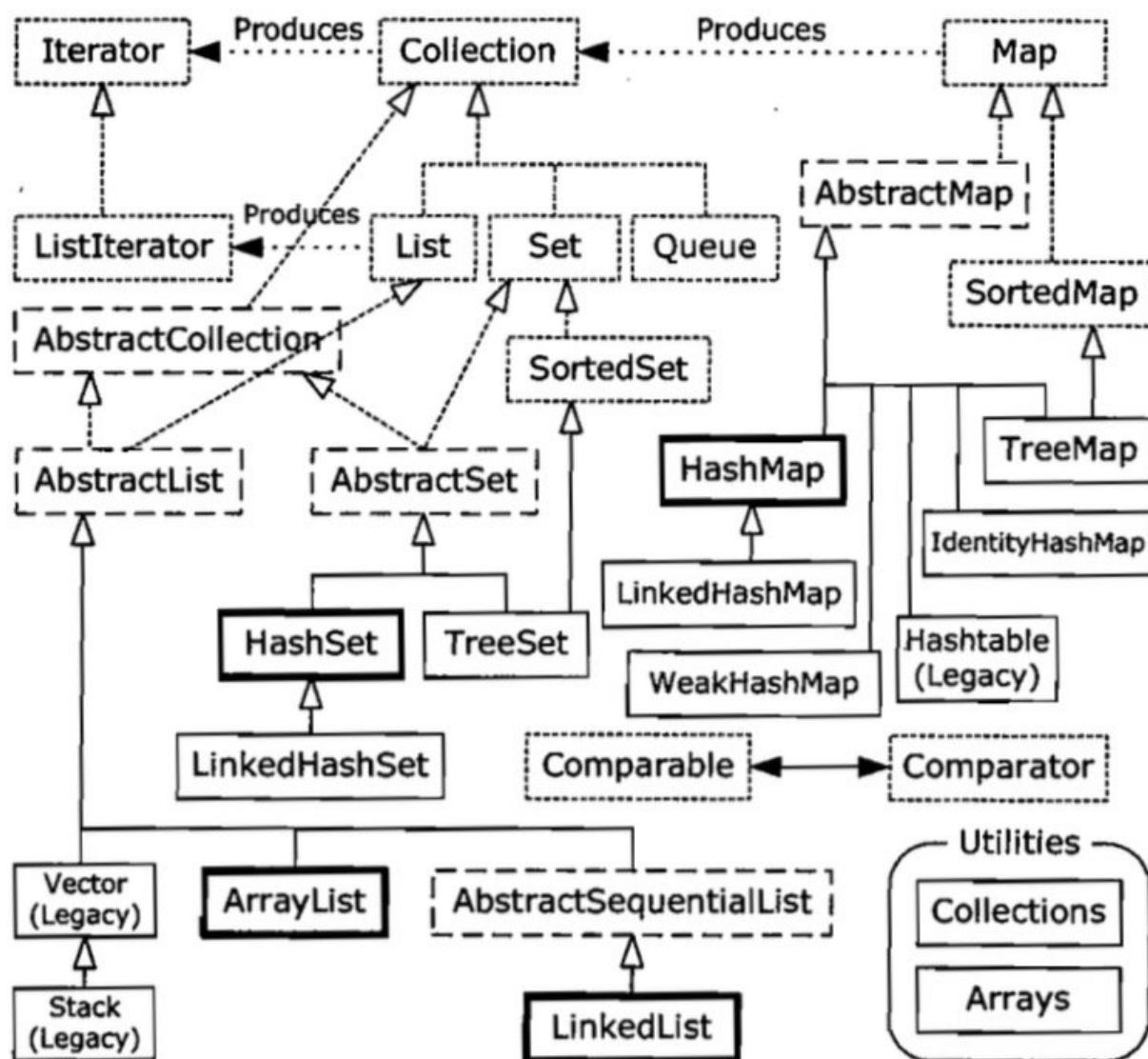
`Arrays.sort()`默认只支持基本类型数组排序

若要对对象数组排序，要求该类型实现了 Comparable 接口或者有相关联的 Comparator

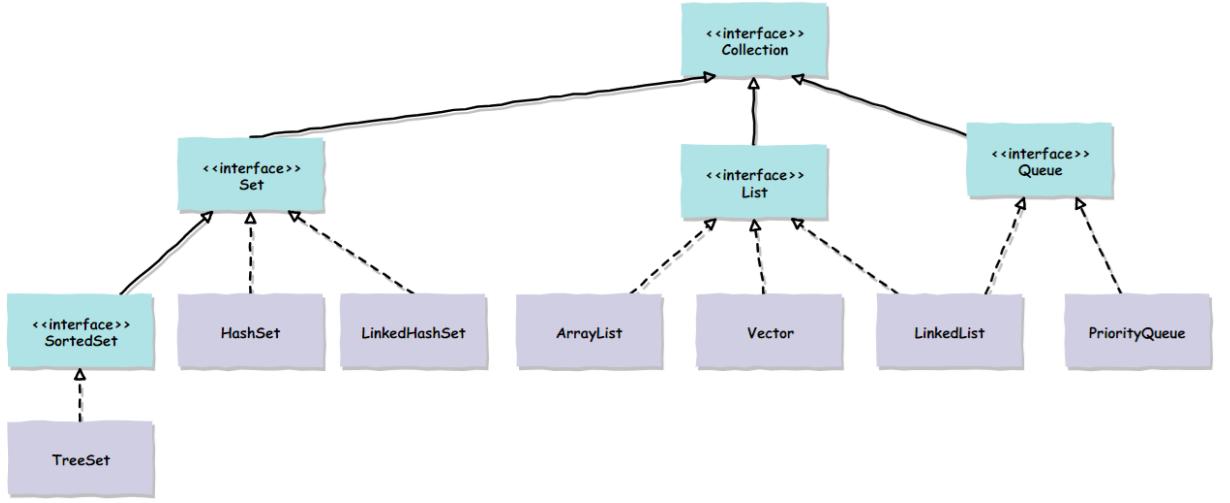
Arrays.sort(a)，要求对象数组a实现了 Comparable 接口

Arrays.sort(b, String.CASE\_INSENSITIVE)， b是一个String数组，传入了String的一个Comparator

## 第十七章 容器深入研究



Full Container Taxonomy



@Cyc2018

## 填充容器

所有的Collection子类都有一个构造器，接收一个Collection对象，用接受的Collection对象来填充

**Collections.fill()**: 类似Arrays对应方法，用单一数据填充List容器（仅适用List容器）

**Collections.nCopies(int n, T o)**: 创建一个List, 里面包含n个T类型的对象o

**Generator**: 可以编写一个适配器，将Generator适配到Collection 的构造器上

## Map生成器

需要一个二元组类，因为Map构造器每次生成的应该是一个<K,V>对

## 使用抽象类

每个java.util容器都有自己的Abstract类

## 可选操作和UnsupportedMethodException

Collection接口的添加和移除都是可选操作，意味着实现类不需要为这些方法提供实现

为什么需要可选操作？：防止出现接口爆炸的情况

- 因为在接口的有些实现类里，可选方法可能是完全没有意义的，调用该方法会抛出UnsupportedMethodException
- 如果创建了一个新的Collection类，但没有实现所有方法，它仍然适合现有的类库
- 从设计的角度来说，如果一个接口方法为可选的，表明这个方法只为某类特定的实现而设计的

比如Arrays.asList()产生了一个基于固定大小的数组的List，那么它的add, addAll, retain, retainAll方法都是没有意义的，调用则会引发UnsupportedMethodException异常

## Set和存储顺序

### Set (接口)

存入元素必须唯一，必须定义equals()方法，不保证维护元素的次序

**HashSet** (默认使用，对速度进行了优化，查找的时间复杂度为 O(1))

必须定义**hashCode()**, **equals()**

**TreeSet** (按对象的比较函数保持次序的Set，是SortedSet接口唯一实现，查找 O(logN))

红黑树实现，元素必须实现**Comparable**接口，**equals()**

```
Comparator comparator() //返回当前Set使用的comparator, null则为自然排序  
Object first() //返回第一个元素  
Object last() //返回末尾元素  
SortedSet subSet(fromElement, toElement) //获取子集  
SortedSet headSet(toElement) //获取小于toElement的子集  
SortedSet tailSet(fromElement) //获取大于等于fromElement的子集 前闭后开
```

**LinkedHashSet** (保持次序的HashSet，双向链表)

必须定义**hashCode()**, **equals()**，内部用链表维护插入元素的顺序

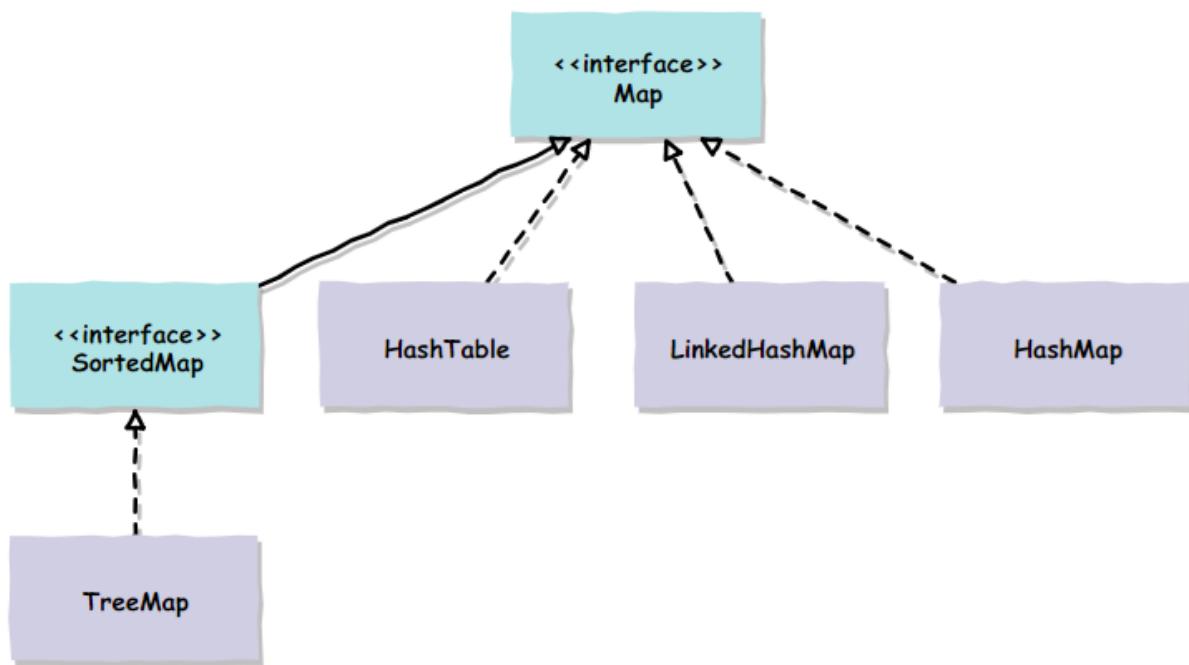
## 队列 (Queue)

Queue在Java SE 5中仅有两个实现**LinkedList**和**PriorityQueue**，两者差异在于排序行为而不是性能

**优先级队列 (PriorityQueue)** 基于堆实现

**双向队列 (Double-ended queue)** : ArrayDeque (JDK 8) , LinkedList也包含支持双向队列的方法

## 理解Map (又名映射表，关联数组)



@Cyc2018

任何插入Map的对象都必须有**equals**方法，如果用于HashMap，则要求有恰当的**hashCode**方法

## **HashMap (不关心顺序)**

插入和查询键值对的开销是固定的

构造器设置容量和负载因子调整性能

容量：哈希表中的桶位数

初始容量：创建表时的桶位数

尺寸：当前存储项数

负载因子：尺寸/容量（默认0.75的时候会进行再哈希，如果知道自己要存多少数据，可以设置合理尺寸来避免再哈希）

## **LinkedHashMap (有序的，继承自HashMap)**

插入和查询比HashMap慢一点点，但迭代访问时反而更快

取出顺序是插入顺序，或者是LRU次序

## **TreeMap (有序的，由Comparable或Comparator决定)**

唯一有subMap()方法的Map，可以返回一个子树

## **WeakHashMap**

保存WeakReference

value只保存一份实例，节省存储空间

允许释放映射所指的对象

如果映射之外，没有引用指向某个“键”，那么此“键”可以被GC回收（不管内存是否够用）

## **ConcurrentHashMap (线程安全)**

不涉及同步加锁

## **IdentityHashMap (无序，key值可以重复，因为是==比较)**

使用==代替equals对“键”进行比较的HashMap。专为解决特殊问题

1. IdentityHashMap有其特殊用途，比如序列化或者深度复制。或者记录对象代理。
2. 举个例子，jvm中的所有对象都是独一无二的，哪怕两个对象是同一个class的对象，而且两个对象的数据完全相同，对于jvm来说，他们也是完全不同的，如果要用一个map来记录这样jvm中的对象，你就需要用IdentityHashMap，而不能使用其他Map实现。

## **哈希和hashCode**

哈希的目的：使用一个对象来查找另一个对象

哈希的价值：用空间换时间，维护了一个固定大小的哈希表（通常是个数组），通过计算的hashCode作为数组下标，然后用不同的策略解决冲突，开放定址法：线行探查法，平方探查法，链地址法，再哈希法，建立公共溢出区

哈希表中的每一个位置通常称为桶（bucket），桶的数量用2的整数次方，可以减少求余的操作

## String的hashCode

如果两个String对象的字符串相同，那么他们指向的是同一块内存区域（字符串常量池某个位置），**hashCode也相同**

## 重写hashCode

1. 同一个对象调用hashCode必须产生相同的值
2. 速度快，且必须有意义
3. 能够产生均匀分布的hashCode

**Effective Java**中的推荐做法：

初始化一个非零常量值，如17

为对象内每个有意义的域都计算出一个int哈希码，然后全部相加，得到最后的哈希值

## 选择接口的不同实现

### 对List的选择

需要注意，List的**LinkedList**和**LinkedHashSet**、**LinkedHashMap**很不一样

背后用数组支撑的**List**和**ArrayList**，随机访问不受列表大小影响

**LinkedList**访问速度受列表大小影响

但随机插入的表现则正好相反

### 对Set的选择

HashSet性能基本上总比TreeSet好，特别是在添加和查询元素

#### TreeSet迭代比HashSet快

！！！ **LinkedHashSet**插入比**HashSet**慢，这是由于维护链表的开销造成的（这和List不一样）

### 对Map的选择

除了**IdentityHashMap**，所有Map的插入都会随着容量增大而变慢，但查找比插入代价少的多

Hashtable和HashMap性能相当

TreeMap通常比HashMap要慢

！！！ **LinkedHashMap**插入比**HashMap**慢一点，但是迭代要快

## Collection和Map的同步控制

**Collections自动同步整个容器（synchronized多线程中重要部分）**

`Collections.synchronizedList`

`Collections.synchronizedLMap`

Collections.synchronizedSet

...

## 快速报错 (ConcurrentModificationException)

保护机制，防止多个进程同时修改同一个容器的内容

## 持有引用

Java.lang.ref类库，这些类为垃圾回收提供更大的灵活性，当存在可能用尽内存的大对象的时候，这些类会很有用

### 继承自抽象类Reference

当GC考察的对象，只能通过某个Reference对象才能获得的时候，以下类提供了不同级别的指示

以下三种引用对应的“可获得性”级别由强到弱

### SoftReference (内存不足就回收)

实现内存敏感的高速缓存，在JVM报告内存不足情况之前将清除所有的软引用。注意：对象是否被释放取决于垃圾收集器的算法以及垃圾收集器运行时可用的内存数量。

### WeakReference (GC看见就回收，不管内存是否充足)

为实现规范映射 (canonicalized mapping) 而设计，它不妨碍GC回收映射的键 (或值)。规范映射的对象实例可以在程序中多处被使用，以节省存储空间。如果对象只剩下一个weak引用，那GC的时候就会回收 (不管内存是否够用)。和 SoftReference 都可以用来实现cache

### PhantomReference (必须与 ReferenceQueue 类一起使用，可用来替换 finalize)

#### get方法永远返回null

用于调度回收前的清理工作，比Java finalize更灵活

SoftReference、WeakReference放入ReferenceQueue (用作回收前清理) 是可选的

但PhantomReference必须依赖ReferenceQueue

## Java 中的 finalize 有哪些问题？

1. 影响 GC 性能，可能会引发 OutOfMemoryException
2. finalize 方法中对异常处理不当会影响 GC
3. 子类中未调用 super.finalize 会导致父类的 finalize 得不到执行

总结一下就是：实现 finalize 对代码的质量要求非常高，一旦使用不当，就容易引发各种问题。

## Java 1.0/1.1 的容器

Vector：Java 1.0/1.1中唯一可扩展序列

Enumeration (接口)：迭代器

Hashtable: 类似HashMap

Stack: 继承自Vector

BitSet: 最小长度是long 64, 用于存储大对象

## 第十八章 Java I/O系统

---

### File 类

特定文件的名称

一个目录下一组文件的名称

目录过滤器: File.list(FilenameFilter f), 参数是FilenameFilter接口的一个实现 (可以用匿名内部类)

### 输入和输出

#### 流

有能力产出数据的数据源对象

有能力接收数据的数据源对象

**Java输入:** InputStream (字节) 或 Reader (**Unicode**字符) 派生 read()

**Java输出:** OutputStream (字节) 或 Writer (**Unicode**字符) 派生 writer()

产生“流” (装饰器模式) : 通过叠合多个对象来提供期望的功能

### 装饰器模式 (Decorator Pattern, Wrapper)

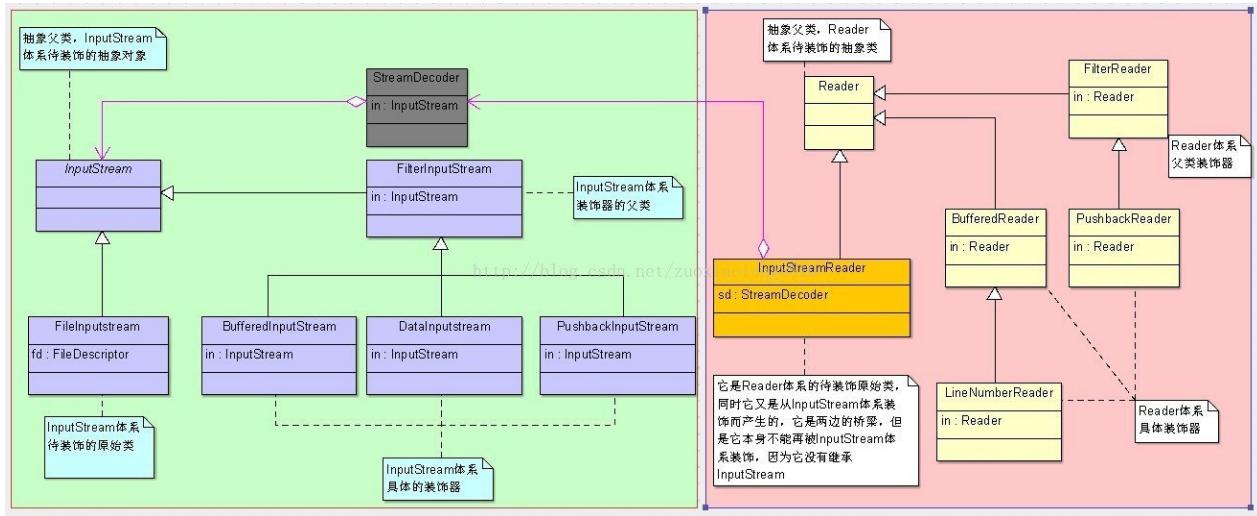
<http://www.cnblogs.com/zuoxiaolong/p/pattern11.html>

定义: 装饰模式是在不必改变原类文件和使用继承的情况下, 动态的扩展一个对象的功能。它是通过创建一个包装对象, 也就是装饰来包裹真实的对象。

1. 不改变原类文件
2. 不使用继承
3. 动态扩展

优点: 灵活, 动态扩展, 用组合的方式取代了继承

缺点: 增加了代码复杂性



## Java 1.0

### InputStream (字节输入的超级父类，抽象类)

InputStream的实体子类！！！，特别注意FilterInputStream（装饰器父类）

FilterInputStream	InputStream体系的装饰器父类
ByteArrayInputStream	内存缓冲区
StringBufferInputStream (Deprecated)	字符串转换成InputStream
FileInputStream	从文件中读取信息
PipedInputStream	多线程中数据源
SequenceInputStream	多个InputStream转换成一个InputStream

FilterInputStream的子类（InputStream的所有装饰器）

DataInputStream (装饰器)	按照可移植方式从流读取 基本数据类型 和 String 对象
BufferedInputStream (装饰器)	使用缓冲区读取数据
LineNumberInputStream(Deprecated)	跟踪输入流的行号
PushbackInputStream (装饰器)	弹出最后一个字节的缓冲区，可以将最后一个读到字符回退

### OutputStream (字节输出的超级父类)

OutputStream的实体子类！！！，特别注意FilterOutputStream（装饰器父类）

<b>FilterOutputStream</b>	<b>OutputStream</b> 体系的装饰器父类
ByteArrayOutputStream	内存缓冲区
FileOututStream	写入文件
PipedOutputStream	写入其中数据都会作为PipedInputStream输入

FilterOutputStream的子类 (**OutputStream**的所有装饰器)

<b>DataOutputStream</b> (装饰器)	按照可移植方式向流写入 <u>基本数据类型</u>
BufferedOutputStream (装饰器)	使用缓冲区写数据
PrintStream (装饰器)	产生格式化输出 (DataOutputStream处理数据存储, PrintStream处理显示)

---

Java 1.1, 新增**Reader**和**Writer**体系, 新类库也比旧类库更快

InputStream -> **InputStreamReader** (适配器) -> Reader

OutputStream -> **OutputStreamWriter** (适配器) -> Writer

以下在Java 1.1没有对应的类:

- DataOutputStream
  - File
  - RandomAccessFile
  - SequenceInputStream
- 

## Java 1.1

### Reader (字符输入的超级父类)

Reader的子类

<b>FilterReader</b>	<b>Reader</b> 体系的装饰器父类（抽象类）
InputStreamReader	InputStream转换为Reader的适配器
StringReader（装饰器）	注意：这里没继承装饰器父类FilterReader！！！
BufferedReader（装饰器）	注意：这里没继承装饰器父类FilterReader！！！只有在需要 <b>readLine</b> 的时候使用，此外优先使用 <b>DataInputStream</b>
CharArrayReader（装饰器）	注意：这里没继承装饰器父类FilterReader！！！
PipedReader（装饰器）	注意：这里没继承装饰器父类FilterReader！！！

FilterReader的子类（**Reader**装饰器）

PushbackReader（装饰器）	注意：这和FilterInputStream很不一样
---------------------	----------------------------

其他装饰器：

FileReader（装饰器）（继承自**InputStreamReader**（适配器））

LineNumberReader（装饰器）（继承自**BufferedReader**（装饰器））

## Writer（字符输入的超级父类）

Writer的子类

<b>FilterWriter</b>	<b>Writer</b> 体系的装饰器父类（抽象类），没有任何子类！！！！！！
OutputStreamReader	OutputStream转换为Writer的适配器
StringWriter（装饰器）	注意：这里没继承装饰器父类FilterWriter！！！
BufferedWriter（装饰器）	注意：这里没继承装饰器父类FilterWriter！！！
CharArrayWriter（装饰器）	注意：这里没继承装饰器父类FilterWriter！！！
PipedWriter（装饰器）	注意：这里没继承装饰器父类FilterWriter！！！
<b>PrintWriter</b> （装饰器，适配器）	1.为了从 <b>PrintStream</b> 过渡，提供了接收PrintStream对象接口 2.默认使用 <b>BufferedWriter</b> ，构造器可以提供一个Boolean是否自动 <b>flush</b>

其他装饰器：

FileWriter（装饰器）（继承自**OutputStreamWriter**（适配器））

# 自我独立的RandomAccessFile

适用于大小已知的记录组成的文件

可以在一个文件内向前向后移动，类似于把DataInputStream和DataOutputStream组合使用（因为使用了相同的接口：DataInput和DataOutput）

JDK 1.4后大多数功能被NIO中的内存映射文件替代

## 标准I/O

概念来自：Unix中“程序所使用的单一信息流”

System.out（包装好的PrintStream对象）

System.err（包装好的PrintStream对象）

System.in（未包装的InputStream对象）：读取前必须进行包装

```
BufferedReader bufferedReader = new BufferedReader(new  
InputStreamReader(System.in));
```

## 标准I/O重定向（重定向的是@字节流@）

System的静态方法：

**setIn(InputStream)**

**setOut(PrintStream)**

**setErr(PrintStream)**

## 进程控制

使用java.lang.ProcessBuilder执行命令，然后返回一个Process

获取命令执行时的标准输出流：Process.getInputStream()

注意：这里不是获取OutputStream，因为**InputStream**是唯一能够从里面获取数据的

获取命令执行时的标准错误流：Process.getErrorStream()

如果要在InputStream里获取标准错误流，先调用**Process.redirectErrorStream(true)**再去获取InputStream

## 新I/O（NIO，同步非阻塞）

Java NIO 是 java 1.4, 之后新出的一套IO接口NIO中的N可以理解为Non-blocking，不单纯是New。

速度的提高来自于更接近操作系统执行I/O的方法：通道和缓冲区

## 通道分类

- 网络读写（SelectableChannel）

- 文件操作 (**FileChannel**)

唯一直接与通道交互的缓冲器就是**ByteBuffer**

NIO的特性/NIO与IO区别:

- 1)IO是面向流的, NIO是面向缓冲区 (**Buffer**) 的
- 2)IO流是阻塞的, NIO流是不阻塞的
- 3)NIO有选择器, 而IO没有。

NIO核心组件简单介绍

- **Channels**
- **Buffers**
- **Selectors** (方法选择集, 用于以原始的字节形式或基本数据类型输出和读取数据, 无法读取或输出对象)

读数据和写数据方式: (通道像煤矿, 缓冲器是卡车)

- 从通道进行数据读取 : 创建一个缓冲区, 然后请求通道读取数据。
- 从通道进行数据写入 : 创建一个缓冲区, 填充数据, 并要求通道写入数据。

旧I/O类修改以产生**FileChannel** (注意: 都是面向字节)

- **FileInputStream**
- **FileOutputStream**
- **RandomAccessFile** (即读又写)

Reader和Writer这种字符操作不能产生**Channel**, 但java.nio.channel.Channels提供了在通道中产生Reader和Writer的方法 new Reader, new Writer

```
FileChannel fc = new FileOutputStream("data.txt").getChannel();
fc.write(ByteBuffer.wrap("some text".getBytes())); //这里是把ByteBuffer里的值写入通道
fc.close();
fc = new RandomAccessFile("data.txt", "rw").getChannel();
fc.position(fc.size()); //移动到最后
fc.write(ByteBuffer.wrap("some more".getBytes()));
fc.close();
fc = new FileInputStream("data.txt").getChannel();
ByteBuffer buff = ByteBuffer.allocate(1024); //对于只读访问, 必须静态分配ByteBuffer
fc.read(buff); //把通道里的值读到ByteBuffer里
buff.flip(); //准备缓冲器, 以便从中读取信息, 每次read后都要调用flip
while(buff.hasRemaining()){
    System.out.println((char)buff.get());
}
// 输出
//some text some more
```

转换数据

缓冲器容纳的是普通字节，为了转换成有意义字符，要么在输入缓冲器时编码，要么在缓冲期输出时解码

```
String encoding = System.getProperty("file.encoding"); //获取系统编码
System.out.println(CharSet.forName(encoding).decode(buff)); //用系统编码去解码
buff直接输出
fc.write(ByteBuffer.wrap("some text").getBytes("UTF-16BE")); //写入buff的时候编码
fc.clear();
fc.read(buff);
fc.flip();
System.out.println(buff.asCharBuffer()); //直接转换成CharBuffer输出，调用了
CharBuffer的toString()，由于写入fc时进行了编码，所以读出时直接转换成CharBuffer输出也不会
乱码
```

## 视图缓冲器

ByteBuffer只能保存字节数据，但是可以用不同的基本类型视图去查看底层 ByteBuffer

通过 ByteBuffer 的不同视图，可以通过 put() 方法为其添加不同的基本类型数据，但添加 short 数据时必须强制类型转换

## 字节存放顺序

**big endian**: 高位字节放在低地址存储单元（默认）

**little endian**: 高位字节放在高存储单元

## 缓冲器细节

### 18.10.5 缓冲器的细节

Buffer 由数据和可以高效地访问及操纵这些数据的四个索引组成，这四个索引是：mark（标记），position（位置），limit（界限）和capacity（容量）。下面是用于设置和复位索引以及查询它们的值的方法。get 和 put 都会增加 position 指针的位置

capacity()	返回缓冲区容量
clear()	清空缓冲区，将 position 设置为 0， limit 设置为 容量。我们可以调用此方法覆写缓冲区
flip()	将 limit 设置为 position， position 设置为 0。此方法用于准备从缓冲区读取已经写入的数据
limit()	返回 limit 值
limit(int lim)	设置 limit 值
mark()	将 mark 设置为 position
position()	返回 position 值
position(int pos)	设置 position 值
remaining()	返回 (limit - position)
hasRemaining()	若有介于 position 和 limit 之间的元素，则返回 true

## 内存映射文件

允许创建和修改因为太大而不能放入内存的文件

FileChannel.map(FileChannel.MapMode.READ\_WRITE, 0, length)

产生的 MapByteBuffer (特殊类型的直接缓冲器，继承自 ByteBuffer，具有其所有方法)

建立映射的开销很大，但是整体收益比I/O流好

## 文件加锁（只能是通道上的，不能是缓冲器）

JDK1.4引入的文件加锁，允许同步访问某个作为共享资源的文件。

文件锁对于其他的操作系统进程是可见的，Java的文件锁映射到了本地操作系统的加锁工具

**FileLock fl = FileChannel.lock()**: 阻塞

**FileLock fl = FileChannel.tryLock()**: 非阻塞

使用共享锁必须由操作系统底层支持，不支持则默认独占锁

可以对内存映射文件部分加锁，例如，数据库就是这样，所以可以多个用户同时访问

JVM会自动释放锁，或者关闭加锁的通道，也可以显示的关闭 FileLock.release()

## 压缩

Java I/O中类库支持压缩的类从InputStream和OutputStream继承而来

压缩按字节方式而不是字符方式

**GZIP**: 对单个数据流进行压缩

**Zip**: 保存多个文件，用了Checksum类（Adler32，快，CRC32，准）来计算校验文件的校验和

**Java档案文件（JAR）**: 包含一组压缩文件，和描述这些文件的文件清单

## 对象序列化

产生背景：在程序不运行的情况下仍能保存对象信息，再重新运行程序的时候就可以恢复到之前运行状态。

主要为了支持两种特性：RMI，Java Beans

**Java 默认的对象序列化**: 实现了**Serializable**接口的对象转换成一个字节序列（跨平台）

**Serializable**接口是一个标记接口，没有任何方法声明

要从字节序列恢复一个对象，必须保证JVM能找到对应的.class文件

如下，写入序列化对象，和恢复序列化对象的方法

- **ObjectOutputStream.writeObject()**
- **ObjectInputStream.readObject()**

```
ObjectOutputStream out = new ObjectOutputStream(new  
FileOutputStream("worm.out"));  
out.writeObject(new Worm(6, 'a'))； //将对象自动序列化之后的字节序列写入文件  
ObjectInputStream in = new ObjectInputStream(new FileInputStream("worm.out"));  
Worm w = (Worm)in.readObject();
```

注意：对一个Serializable对象进行还原的时候，是直接从字节序列中还原的，没有调用任何构造器，包括默认无参构造器

# 序列化的控制

实现**Externalizable**接口（继承自**Serializable**接口，必须要有无参构造器）

新增了两个方法，会在序列化和反序列化过程自动调用

- **public void writeExternal(ObjectOutput out) throws IOException**
- **public void readExternal(ObjectInput in) throws IOException**

和**Serializable**区别：

- 会调用字段定义时初始化（在调用默认构造器之前调用）
- 调用所有普通默认构造器
- 没有在writeExternal和readExternal保存恢复的域就会用默认构造器里的初始化值或字段初始化定义值（如果默认构造器里没有初始化对应字段值）

**transient关键字（用于Serializable对象）**

表示此字段不需要被序列化时保存，也不会在反序列化时被尝试恢复（注意：仅对于自动序列化机制）

但是如果在添加的**readObject**方法和**writeObject**方法中显示保存和恢复**transient**字段是可以的

注意：也可以继承**Externalizable**接口，那么没有东西可以自动序列化

**static**字段也不能被序列化

**Externalizable**的替代实现（但并不会像其一样调用默认构造器）

在**Serializable**接口的实现类中添加（并不是覆盖或实现）

- **private void writeObject(ObjectOutputStream stream) throws IOException**
- **private void readObject(ObjectInputStream stream) throws IOException**

注意这里和 **ObjectOutputStream**，和 **ObjectInputStream**的方法同名了，这里其实是一个混乱的设计，在序列化的过程中会反射调用添加的这两个方法而不是去使用默认序列化机制

## 如何正确序列化Class类

**Class**类是**Serializable**的，但是想要通过序列化**Class**对象保存**static**值，必须手动显示的去**writeObject**和**readObject**，不然会有问题

## 常见的序列化协议有哪些

- COM主要用于Windows平台，并没有真正实现跨平台，另外COM的序列化的原理利用了编译器中虚表，使得其学习成本巨大。
- CORBA是早期比较好的实现了跨平台，跨语言的序列化协议。CORBA的主要问题是参与方过多带来的版本过多，版本之间兼容性较差，以及使用复杂晦涩。
- XML & SOAP
  - XML是一种常用的序列化和反序列化协议，具有跨机器，跨语言等优点。
  - SOAP (Simple Object Access protocol) 是一种被广泛应用的，基于XML为序列化和反序列化协议的结构化消息传递协议。SOAP具有安全、可扩展、跨语言、跨平台并支持多种传输层协议。

- JSON (JavaScript Object Notation)
  - 这种Associative array格式非常符合工程师对对象的理解。
  - 它保持了XML的人眼可读 (Human-readable) 的优点。
  - 相对于XML而言，序列化后的数据更加简洁。
  - 它具备javascript的先天性支持，所以被广泛应用于Web browser的应用常景中，是Ajax的事实标准协议。
  - 与XML相比，其协议比较简单，解析速度比较快。
  - 松散的Associative array使得其具有良好的可扩展性和兼容性。
- Thrift是Facebook开源提供的一个高性能，轻量级RPC服务框架，其产生正是为了满足当前大数据量、分布式、跨语言、跨平台数据通讯的需求。
- Avro的产生解决了JSON的冗长和没有IDL的问题，Avro属于Apache Hadoop的一个子项目。Avro提供两种序列化格式：JSON格式或者Binary格式。Binary格式在空间开销和解析性能方面可以和Protobuf媲美，JSON格式方便测试阶段的调试。适合于高性能的序列化服务。

## Preferences API (相比对象序列化，它和对象持久性更密切)

**键值对**

用于小的、受限的数据集合（只能存储基本类型和字符串，每个字符串不超过8K）

存储用户偏好以及程序配置项

利用系统资源存储，如Windows里就是注册表

通常是以类名命名的单一节点

## 第二十一章 并发

---

**并发解决的问题：**

- 速度
- 设计可管理性

多核处理器可以实现真正意义上的并行，但**并发**针对的是单核处理器上的程序性能

**单核处理器的并发：**实际是在切换多个不同任务线程，增加了上下文切换的开销

**并发对于单核处理器的意义：**程序存在阻塞（通常是I/O）

**多线程最基本的难题：**保证共享资源（如内存、I/O）的独立访问

**Java中多线程：**在执行程序的单一进程中实现的多线程，这也保证了并发程序可移植性

**Java线程机制：**抢占式，调度机制会周期性的中断线程，将上下文切换到另一个线程

Java对线程数量的限制依赖于Java版本

**协作多线程：**每个线程都自动地放弃控制权，要求编写某种类型让步语句。其优势在于（1.上下文切换开销比抢占式低 2.理论上不限制线程的数量）

## 基本的线程机制

**Runnable接口**（可以看作是一个独立的任务，实际上Runnable这个名字并不恰当）

每个任务都需要附着到一个线程上来执行

**Thread**是实现了**Runnable**接口的实体类（都位于**java.lang**包）

虽然new Thread().start时并未保存其引用，但它也会开启一个独立的线程执行附着在其上的任务，在其run方法结束前，GC都无法回收该Thread对象

线程的调度机制是非确定性的：在不同机器、不同JDK版本上的执行情况都可能不同

## Executor (执行器, **java.util.concurrent**)

使用执行器，不必显示创建Thread对象

```
ExecutorService exec = Executors.newCachedThreadPool(); //Executors 创建的
ThreadPoolExecutor实例, 在线程可用时, 重用之前构造好的池中线程
exec.execute(new Runnable(){
    @Override
    public void run(){
        System.out.println("Test Executor");
    }
});
exec.shutdown(); //拒绝提交新任务, 待已提交任务执行完后尽快退出
```

## 线程池

**什么是线程池：**`java.util.concurrent.Executors`提供了一个`java.util.concurrent.Executor`接口的实现用于创建线程池

**什么时候使用线程池：**假设一个服务器完成一项任务所需时间为：T1 创建线程时间，T2 在线程中执行任务的时间，T3 销毁线程时间。如果：T1 + T3 远大于 T2，则可以采用线程池，以提高服务器性能。

一个线程池包括以下四个基本组成部分：

- 线程池管理器（`ThreadPool`）：用于创建并管理线程池，包括创建线程池，销毁线程池，添加新任务；
- 工作线程（`PoolWorker`）：线程池中线程，在没有任务时处于等待状态，可以循环的执行任务；
- 任务接口（`Task`）：每个任务必须实现的接口，以供工作线程调度任务的执行，它主要规定了任务的入口，任务执行完后的收尾工作，任务的执行状态等；
- 任务队列（`taskQueue`）：用于存放没有处理的任务。提供一种缓冲机制。

**常见线程池**（线程池都会自动复用，即先看有无空闲线程可以执行任务，没有再新建线程）：

- **newSingleThreadExecutor** 适用于保证顺序执行各个任务，且不会有多线程场景

单个工作线程来执行一个无边界队列，即线程池中每次只有一个线程工作，单线程串行执行任务。

- **newFixedThreadPool(n)** 适用于可以预测线程数量的业务中

固定数量的线程池，复用 **固定数量的线程** 处理一个 **无边界队列**。每提交一个任务就使用一个线程，直到达线程池的最大数量，然后后面进入等待队列直到前面的任务完成才继续执行

- **newCacheThreadPool**（会自动销毁空闲线程，但最大线程数是**Integer.MAX\_VALUE**）

可缓存线程池，当线程池大小超过了处理任务所需的线程，那么就会回收部分空闲（一般是60秒无执行）的线程，当有任务来时，会根据需要，在线程可用时，重用之前构造好的池中线程。这个线程池在执行大量短生命周期的异步任务时（many short-lived asynchronous task），可以显著提高程序性能

- **newScheduleThreadPool**

大小无限制的线程池，支持定时和周期性的执行线程

- **newWorkStealingPool**

创建一个拥有多个任务队列的线程池，通过使用多个队列减少竞争，不传参数，则默认设定为cpu的数量，适用于大耗时的操作，可以并行来执行。

线程池存在的问题：FixedThreadPool 和 CachedThreadPool 两者对高负载的应用都不是特别友好。

无边界队列会导致高延迟甚至OOM，长时间运行会导致CachedThreadPool在创建线程上失控，导致OOM

## 线程池最佳实践

推荐使用 `ThreadPoolExecutor` 手动创建线程池，它提供了很多参数可以进行细粒度的控制。

1. 将任务队列设置成有边界的队列
2. 使用合适的 `RejectionHandler` - 自定义的 `RejectionHandler` 或 JDK 提供的默认 handler。
3. 如果在任务完成前后需要执行某些操作，可以重载

```
beforeExecute(Thread, Runnable)  
afterExecute(Runnable, Throwable)
```

4. 重载 `ThreadFactory`，如果有线程定制化的需求（如设置为后台线程）

5. 在运行时动态控制线程池的大小（[Dynamic Thread Pool]）

## ExecutorService（接口，继承扩展了Executor接口）的各个方法

**submit(Callable task)**: 提交一个线程任务，可以接受回调函数的返回值，适用于需要处理返回值或者异常的业务场景

**execute()**: 执行一个任务，没有返回值

**shutdown()**: 表示不再接受新任务，但不会强行终止已经提交或者正在执行中的任务

**shutdownNow()**: 对于尚未执行的任务全部取消，正在执行的任务全部发出**interrupt()**，停止执行

## Callable接口（Java 1.5 concurrent）和Runnable接口（Java 1.1 lang）

都代表一个需要被其他线程执行的任务，但Callable接口中只有一个call方法声明，并且使用了类型参数规定call方法返回值类型

Callable接口适用需要返回值的任务，并且可以抛出checked exception

运行Callable任务可以拿到一个**Future**对象，表示异步计算的结果。它提供了检查计算是否完成的方法，以等待计算的完成，并检索计算的结果

## Future接口 (concurrent)

提供了三种功能：

- 1) 判断任务是否完成；
- 2) 能够中断任务；
- 3) 能够获取任务执行结果。

```
Future<?> f = exec.submit(new Callable or Runnable);
f.cancel(true);
```

相当于调用了线程的interrupt方法，因为Executor执行任务无法获得线程引用，所以可以用Future来中断线程

## FutureTask (实现了RunnableFuture接口，是Future的唯一实现类)

```
public interface RunnableFuture<V> extends Runnable, Future<V> {
    void run();
}
public class FutureTask<V> implements RunnableFuture<V> {
    /*...*/
}
```

## 异常不能跨线程传播

对Thread.sleep(TimeUnit.Seconds.sleep(1))调用可以抛出InterruptedException，并且必须在run（实现的Runnable接口）方法中处理任务内部产生的异常

在创建该线程的main函数里用try catch块无法捕获线程 run() 方法里抛出的Exception

Thread.UncaughtExceptionHandler：可以实现Thread的这个内部接口，并通过

Thread.setDefaultExceptionHandler(YourHandler)，设置该线程的异常处理逻辑为自己的实现

## 线程优先级（优先级不导致死锁）

线程调度器倾向于优先执行高优先级线程（但并不是意味着低优先级线程不执行），低优先级线程仅仅是执行频率较低

```
Thread.setPriority(Thread.MIN_PRIORITY); // 最低优先级 1, 最高 10, 普通 5
```

## 让步（yield，建议有相同优先级的线程可以运行）

这只是对线程调度器的一个暗示，说明我已经做完一轮迭代要做的工作，现在正式切换时间片的好时机，但没有任何机制保证它会被采纳。

## 后台线程（dameon）

如果所有非后台线程都结束了，那么所有后台线程会立即终止，finally都不会执行

```
Thread.setDameon(true); //设置成后台线程  
Thread.isDameon(); //返回该线程是否为后台线程
```

后台线程创建的所有线程都是后台线程

在构造器中启动线程可能存在的问题：另一个任务在构造器结束前就启动了，那么这个任务可以访问处于不稳定状态的对象

在方法内部创建线程：如果该线程只是辅助操作，那么这比在类的构造器里启动更合适

```
public void runTask(){  
    t = new Thread(){ //匿名内部类继承了Thread类并Override其run方法  
        @Override  
        public void run(){  
            while(true){  
                print(this);  
            }  
        }  
    };  
    t.start();  
}
```

## 一些术语

Java中，Thread类自身不执行任何操作，它只驱动赋予它的任务（Runnable）

Runnable更好的理解是把它当成要执行的一个任务

### join

在一个线程t上调用join()，那么本线程将被挂起并等待t线程结束(t.isAlive() == false)才恢复

## 共享受限资源

解决访问共享资源冲突的方式，就是加锁，因为锁语句产生了一种互相排斥的效果，所以这也种机制也称为互斥量

## 对象锁（monitor，也称监视器）

- 所有对象都含有单一的锁
- 对象的所有synchronized方法共享同一个锁
- 一个任务可以多次获得对象锁：一个任务调用该对象的synchronized方法，此方法又调用了此对象另一个synchronized方法，那么加锁次数会变成2，只有当加锁次数为0的时候，说明该对象被完全释放

## 类锁（属于类的Class对象的一部分）

所以synchronized static可以在类的范围内防止对static数据的并发访问，即把这些临界的static域设置成private，只能通过synchronized static方法访问

## 悲观锁（写多）

总是假设最坏的情况，每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会阻塞直到它拿到锁（共享资源每次只给一个线程使用，其它线程阻塞，用完后再把资源转让给其它线程）。传统的关系型数据库里边就用到了很多这种锁机制，比如行锁，表锁等，读锁，写锁等，都是在做操作之前先上锁。Java中 `synchronized` 和 `ReentrantLock` 等独占锁就是悲观锁思想的实现。

## 乐观锁（读多）

总是假设最好的情况，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，可以使用版本号机制和CAS算法实现。乐观锁适用于多读的应用类型，这样可以提高吞吐量，像数据库提供的类似于 `write_condition` 机制，其实都是提供的乐观锁。在Java中 `java.util.concurrent.atomic` 包下面的原子变量类就是使用了乐观锁的一种实现方式 **CAS** 实现的。

## CAS操作

使用锁时，线程获取锁是一种**悲观锁策略**，即假设每一次执行临界区代码都会产生冲突，所以当前线程获取到锁的时候同时也会阻塞其他线程获取该锁。

CAS操作（又称为无锁操作）是一种**乐观锁策略**，它假设所有线程访问共享资源的时候不会出现冲突，既然不会出现冲突自然而然就不会阻塞其他线程的操作。如果出现冲突了怎么办？无锁操作是使用**CAS(compare and swap)** 又叫做比较交换来鉴别线程是否出现冲突，出现冲突就重试当前操作直到没有冲突为止（自旋）。

CAS(V,O,N)，V 内存地址存放的实际值；O 预期的值（旧值）；N 更新的新值

CAS的实现需要硬件指令集的支撑，在JDK1.5后虚拟机才可以使用处理器提供的**CMPXCHG** 指令实现。

### Synchronized VS CAS

Synchronized(未优化前)最主要的问题是：在存在线程竞争的情况下会出现线程阻塞和唤醒锁带来的性能问题，因为这是一种互斥同步（阻塞同步）

CAS操作失败后会进行一定的尝试，而非进行耗时的挂起唤醒的操作，因此也叫做**非阻塞同步**。这是两者主要的区别。

## CAS存在的问题

### 1. ABA问题

因为CAS会检查旧值有没有变化，这里存在这样一个有意思的问题。比如一个旧值A变为了成B，然后再变成A，刚好在做CAS时检查发现旧值并没有变化依然为A，但是实际上的确发生了变化。解决方案可以沿袭数据库中常用的乐观锁方式，添加一个版本号可以解决。原来的变化路径A->B->A就变成了1A->2B->3C。java这么优秀的语言，当然在java 1.5后的atomic包中提供了 `AtomicStampedReference` 来解决ABA问题，解决思路就是这样的。

### 2. 自旋时间过长

使用CAS时**非阻塞同步**，也就是说不会将线程挂起，会自旋（无非就是一个死循环）进行下一次尝试，如果这里自旋时间过长对性能是很大的消耗。如果JVM能支持处理器提供的pause指令，那么在效率上会有一定的提升。

### 3. 只能保证一个共享变量的原子操作

当对一个共享变量执行操作时CAS能保证其原子性，如果对多个共享变量进行操作,CAS就不能保证其原子性。有一个解决方案是利用对象整合多个共享变量，即一个类中的成员变量就是这几个共享变量。然后将这个对象做CAS操作就可以保证其原子性。atomic中提供了AtomicReference来保证引用对象之间的原子性。

## 并发关键字

### synchronized 关键字（也直接称为同步，JVM提供支持，JVM负责加锁和释放）

不属于方法特征签名的一部分，天生就具有重入性

任务访问synchronized关键字代码片段或方法的时候，会检查当前对象的锁是否可获得，如果可以获得锁，那么才会访问此段代码，在执行结束后释放对象锁。

当一个任务持有某个对象锁的时候，其他访问该对象的任何synchronized代码的线程都会被阻塞，访问非synchronized方法不受影响

#### 原理

通过对对象内部的一个叫做监视器锁（monitor）来实现的。但是监视器锁本质又是依赖于底层的操作系统的Mutex Lock来实现的。

#### 1.Synchronized是如何实现对代码块进行同步

moniterenter和moniterexit字节码指令

#### 2.Synchronized是如何实现对方法进行同步

常量池中多了ACC\_SYNCHRONIZED标志符，JVM就是根据该标志符来实现方法的同步的（其实和代码块同步是一个原理，只不过这里使用隐式的一个标志实现，而不是字节码指令）

wait/notify等方法也依赖于monitor对象，这就是为什么只有在同步的块或者方法中才能调用wait/notify等方法，否则会抛出java.lang.IllegalMonitorStateException的异常的原因。

#### Synchronized为什么低效？

操作系统实现线程之间的切换这就需要从用户态转换到核心态，这个成本非常高，状态之间的转换需要相对比较长的时间，这就是为什么Synchronized效率低的原因。

#### Synchronized的优化

JDK1.6以后，为了减少获得锁和释放锁所带来的性能消耗，提高性能，引入了“轻量级锁”和“偏向锁”。

#### 获取锁的形式没变，但加快了获取锁和释放锁的速度 1.CAS 2.对象头

#### Java对象头

同步块的核心是获取了对象的monitor（即对象锁），对象锁就记录在对象头中

Java对象头里的Mark Word里默认的存放的对象的HashCode,分代年龄和锁标记位。

锁状态	25bit	4bit	1bit 是否是偏向锁	2bit 锁标志位
无锁状态	对象的 hashCode	对象分代年龄	0	01

Java SE 1.6中，锁一共有4种状态，级别从低到高依次是：

- 无锁状态
- 偏向锁状态
- 轻量级锁状态
- 重量级锁状态

锁随着竞争情况升级，但不会降级（目的是为了提高获取和释放锁效率，避免无用CAS自旋）

对象的MarkWord变化如下图

锁状态	25bit		4bit	1bit	2bit	
	23bit	2bit		是否是偏向锁	锁标志位	
轻量级锁	指向栈中锁记录的指针				00	
重量级锁	指向互斥量（重量级锁）的指针				10	
GC 标记	空				11	
偏向锁	线程 ID	Epoch	对象分代年龄	1	01	

## 偏向锁

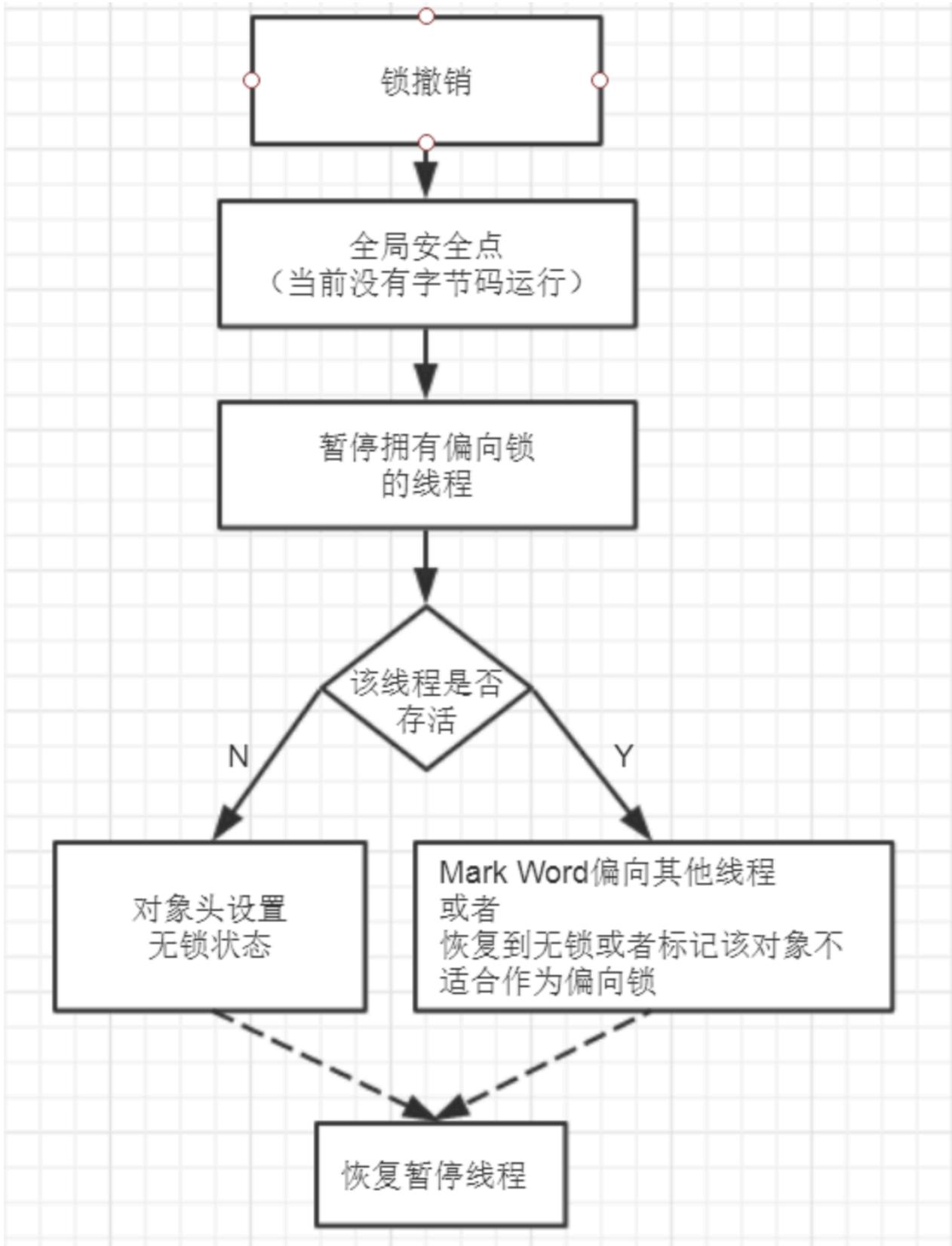
HotSpot的作者经过研究发现，大多数情况下，锁不仅不存在多线程竞争，而且总是由同一线程多次获得，为了让线程获得锁的代价更低而引入了偏向锁。

### 偏向锁的获取

当一个线程访问同步块并获取锁时，会在对象头和栈帧中的锁记录里存储锁偏向的线程ID，以后该线程在进入和退出同步块时不需要进行CAS操作来加锁和解锁，只需简单地测试一下对象头的Mark Word里是否存储着指向当前线程的偏向锁。如果测试成功，表示线程已经获得了锁。如果测试失败，则需要再测试一下Mark Word中偏向锁的标识是否设置成1（表示当前是偏向锁）：如果没有设置，则使用CAS竞争锁；如果设置了，则尝试使用CAS将对象头的偏向锁指向当前线程

### 偏向锁的释放

偏向锁使用了一种等到竞争出现才释放锁的机制，所以当其他线程尝试竞争偏向锁时，持有偏向锁的线程才会释放锁。



偏向锁的撤销，需要等待全局安全点（在这个时间点上没有正在执行的字节码）

栈中的锁记录和对象头的Mark Word

要么重新偏向于其他线程，

要么恢复到无锁

要么标记对象不适合作为偏向锁，最后唤醒暂停的线程。

## 关闭偏向锁

偏向锁在Java 6和Java 7里是默认启用的，但是它在应用程序启动几秒钟之后才激活，如有必要可以使用JVM参数来关闭延迟：**-XX:BiasedLockingStartupDelay=0**。如果你确定应用程序里所有的锁通常情况下处于竞争状态，可以通过JVM参数关闭偏向锁：**-XX:-UseBiasedLocking=false**，那么程序默认会进入轻量级锁状态

## 轻量级锁

### 加锁

线程在执行同步块之前，JVM会先在当前线程的栈桢中创建用于存储锁记录的空间，并将对象头中的**Mark Word**复制到锁记录中，官方称为**Displaced Mark Word**。然后线程尝试使用**CAS**将对象头中的**Mark Word**替换为指向锁记录的指针。如果成功，当前线程获得锁，如果失败，表示其他线程竞争锁，当前线程便尝试使用自旋来获取锁。

### 解锁

轻量级解锁时，会使用原子的**CAS**操作将**Displaced Mark Word**替换回到对象头，如果成功，则表示没有竞争发生。如果失败，表示当前锁存在竞争，锁就会膨胀成重量级锁。

因为自旋会消耗CPU，为了避免无用的自旋（比如获得锁的线程被阻塞住了），一旦锁升级成重量级锁，就不会再恢复到轻量级锁状态。

## 各种锁的比较

锁	优 点	缺 点	适用场景
偏向锁	加锁和解锁不需要额外的消耗，和执行非同步方法相比仅存在纳秒级的差距	如果线程间存在锁竞争，会带来额外的锁撤销的消耗	适用于只有一个线程访问同步块场景
轻量级锁	竞争的线程不会阻塞，提高了程序的响应速度	如果始终得不到锁竞争的线程，使用自旋会消耗CPU	追求响应时间 同步块执行速度非常快
重量级锁	线程竞争不使用自旋，不会消耗CPU	线程阻塞，响应时间缓慢	追求吞吐量 同步块执行速度较长

## synchronized的正确使用

- 所有共享的数据资源都设置为**private**，只能通过方法访问
- 所有访问临界共享资源的方法都必须被同步（synchronized）

## final关键字

final可以修饰变量，方法和类，所修饰的内容一旦赋值之后就不会再被改变

final在多线程中存在的重排序问题也很容易忽略

## final变量

### final成员变量

- 类变量：必须要在静态初始化块中指定初始值或者声明该类变量时指定初始值
- 实例变量：必要要在非静态初始化块，声明该实例变量或者在构造器中指定初始值

### final局部变量

#### final基本数据类型 VS final引用数据类型

基本数据类型保证数据不变

引用类型保证指向地址不变，但指向的对象属性可以改变

### 宏变量

1. 使用final修饰符修饰；
2. 在定义该final变量时就指定了初始值；
3. 该初始值在编译时就能够唯一指定。

## final方法

1. 父类的final方法是不能够被子类重写的
2. final方法是可以被重载的

## final类

不能被继承

## 不变类

- 使用private和final修饰符来修饰该类的成员变量
- 提供带参的构造器用于初始化类的成员变量；
- 仅为该类的成员变量提供getter方法，不提供setter方法，因为普通方法无法修改final修饰的成员变量；
- 如果有必要就重写Object类的hashCode()和equals()方法，应该保证用equals()判断相同的两个对象其HashCode值也是相等的。

String的value就是final

```
/** The value is used for character storage. */
private final char value[];
```

## final 多线程

### 写final域重排序规则

1. JMM禁止编译器把final域的写重排序到构造函数之外；
2. 编译器会在final域写之后，构造函数return之前，插入一个storestore屏障。这个屏障可以禁止处理器把final域的写重排序到构造函数之外。

### 读final域重排序规则

在一个线程中，初次读对象引用和初次读该对象包含的final域，JMM会禁止这两个操作的重排序。实际上，读对象的引用和读该对象的final域存在间接依赖性，一般处理器不会重排序这两个操作。但是有一些处理器会重排序，因此，这条禁止重排序规则就是针对这些处理器而设定的。

### 对final修饰的对象的成员域写操作

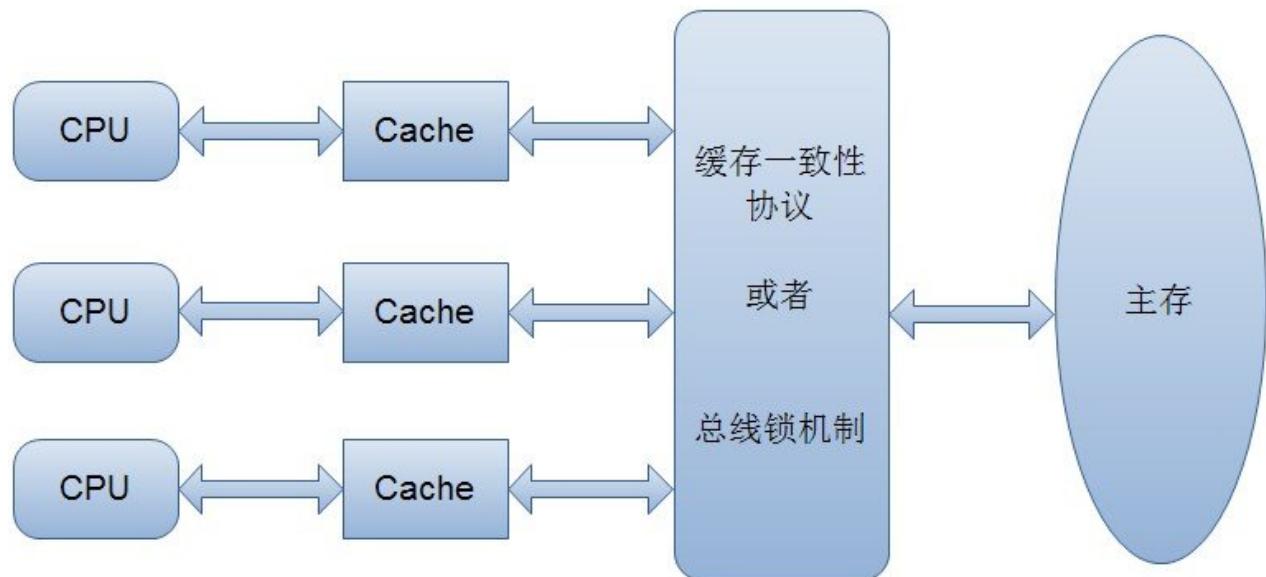
增加了这样的约束：在构造函数内对一个final修饰的对象的成员域的写入，与随后在构造函数之外把这个被构造的对象的引用赋给一个引用变量，这两个操作是不能被重排序的。

## final 的实现原理

以X86处理器为例，X86不会对写-写重排序，所以StoreStore屏障可以省略。由于不会对有间接依赖性的操作重排序，所以在X86处理器中，读final域需要的LoadLoad屏障也会被省略掉。也就是说，以X86为例的话，对final域的读/写的内存屏障都会被省略！具体是否插入还是得看是什么处理器

## volatile关键字解析

### 内存模型



对应到JMM就是，

每个线程的工作内存：其所在CPU的Cache

缓存不一致性解决办法（硬件层面）：

- 通过在总线加LOCK#锁的方式
- 通过缓存一致性协议，Intel 的MESI协议，当CPU写数据时，如果发现操作的变量是共享变量，即在其他CPU中也存在该变量的副本，会发出信号通知其他CPU将该变量的缓存行置为无效状态

### 并发编程的三个概念

#### 原子性

原子性：即一个操作或者多个操作 要么全部执行并且执行的过程不会被任何因素打断，要么就都不执行。

#### 可见性

当多个线程访问同一个变量时，一个线程修改了这个变量的值，其他线程能够立即看得到修改的值

#### 有序性

即程序执行的顺序按照代码的先后顺序执行

指令重排序：程序的实际执行顺序和代码的顺序不一定相同，但处理器保证最后程序结果和代码执行结果是一致的（处理器在重排时会考虑指令之间的依赖性）

指令重排序不会影响单个线程的执行，但是会影响到线程并发执行的正确性

```

//线程1:
context = loadContext();      //语句1
initiated = true;             //语句2

//线程2:
while(!initiated){
    sleep()
}
doSomethingWithConfig(context);

```

线程1中两个指令没有依赖关系，处理器重排，语句2先执行。但线程2中认为初始化已完成！！！

并发程序正确地执行，必须要同时保证原子性、可见性以及有序性。只要有一个没有被保证，就有可能会导致程序运行不正确

## Java内存模型

JVM specification 里定义了一种 (Java Memory Model, JMM) )，JMM 是抽象化了硬件的内存模型（使用了多级缓存导致出现缓存一致性协议），屏蔽了各个 CPU 和操作系统的差异。

**它定义了什么？**

它定义了程序中变量的访问规则，往大一点说是定义了程序执行的次序

**所有的变量都是存在主存当中**（类似于前面说的物理内存），每个线程都有自己的工作内存（类似于前面的高速缓存）。**线程对变量的所有操作都必须在工作内存中进行，而不能直接对主存进行操作**。并且每个线程不能访问其他线程的工作内存。

**存在主存中的：**包括实例字段，静态字段，和构成数据对象的元素，但**不包括局部变量和方法参数**，因为后者是线程私有的

**它没有规定的东西有？**

JMM并没有限制执行引擎使用处理器的寄存器或者高速缓存来提升指令执行速度，也没有限制编译器对指令进行重排序，也就是说Java中同样存在缓存一致性问题和指令重排序的问题

### 1. Java中原子性

在Java中除了long和double之外所有基本类型上的“简单操作”都是原子操作

**简单操作：**只有简单的读取、赋值（而且必须是将数字赋值给某个变量，变量之间的相互赋值不是原子操作，因为需要读取再赋值）才是原子操作。

32位JVM读写64位的long和double时会产生字撕裂，会当成两个分离的32位操作

但如果使用**volatile关键字修饰long或者double**，就会获得简单赋值与返回操作的原子性

**Java保证原子性的手段：**

**synchronized**和**Lock**能够保证任一时刻只有一个线程执行该代码块，那么自然就不存在原子性问题了，从而保证了原子性

**CAS** (compare and swap)，位于java.util.concurrent.atomic原子操作类，如AtomicInteger，利用处理器提供的CMPXCHG指令实现的，而处理器执行CMPXCHG指令是一个原子性操作

## 2.Java中可见性

**Java保证可见性的手段：**volatile关键字、synchronized和Lock

**volatile关键字：**被修改的值会立即被更新到主存，并且使其他CPU的缓存行无效

**volatile的数组只针对数组的引用具有volatile的语义，而不是它的元素**

**synchronized和Lock：**保证同一时刻只有一个线程获取锁执行同步代码，并且在释放锁之前会把修改变量刷新到主存！！！

## 3.Java中的有序性

JMM具备一些“先天的”有序性，即不需要任何手段就可以获得的，也称为 happens-before 原则

如果两个操作次序无法从happens-before原则推断出来，那么虚拟机就可以随意重排

**happens-before 原则：**

- 传递规则：如果操作1在操作2前面，而操作2在操作3前面，则操作1肯定会在操作3前发生。该规则说明了happens-before原则具有传递性
- 锁定规则：一个unlock操作肯定会在后面对同一个锁的lock操作前发生。这个很好理解，锁只有被释放了才会被再次获取
- **volatile变量规则：**对一个被volatile修饰的写操作先发生于后面对该变量的读操作
- 程序次序规则：一个线程内，按照代码顺序执行（但是注意，这保证了单线程的有序性）
- 线程启动规则：Thread对象的start()方法先发生于此线程的其它动作
- 线程终结原则：线程中其它的所有操作先于线程的终止检测 (isAlive() == false or join())
- 线程中断规则：对线程interrupt()方法的调用先发生于对该中断异常的获取 (isInterrupted())
- 对象终结规则：一个对象构造先于它的finalize发生

**Java保证有序性的手段（除了happens-before）：**volatile关键字、synchronized和Lock

**synchronized和Lock：**很显然相当于是让一个线程顺序执行同步代码，保证了有序性（相当于利用了happens before里对单线程提供的程序次序原则）

**volatile（保证可见性，无法保证原子性，一定程度保证有序性）**

- 保证了不同线程对这个变量进行操作时的可见性
- 禁止进行指令重排序

**volatile保证可见性：**

1. 使用volatile关键字会强制将修改的值立即写入主存
2. 如果是写操作，它会导致其他CPU中对应的缓存行无效
3. 由于缓存行无效，所以该线程再次读取变量的值时会去主存读取。

**volatile禁止指令重排：**

volatile关键字时，汇编代码里会多出一个lock前缀指令（CPU指令）

lock前缀指令实际上相当于一个内存屏障（也称内存栅栏），内存屏障会提供3个功能：

1) 它确保指令重排序时不会把其后面的指令排到内存屏障之前的位置，也不会把前面的指令排到内存屏障的后面；即在执行到内存屏障这句指令时，在它前面的操作已经全部完成；

但是栅栏前后部分语句之间的顺序不保证

- 2) 它会强制将对缓存的修改操作立即写入主存（属于保证可见性）
- 3) 如果是写操作，它会导致其他CPU中对应的缓存行无效（属于保证可见性）

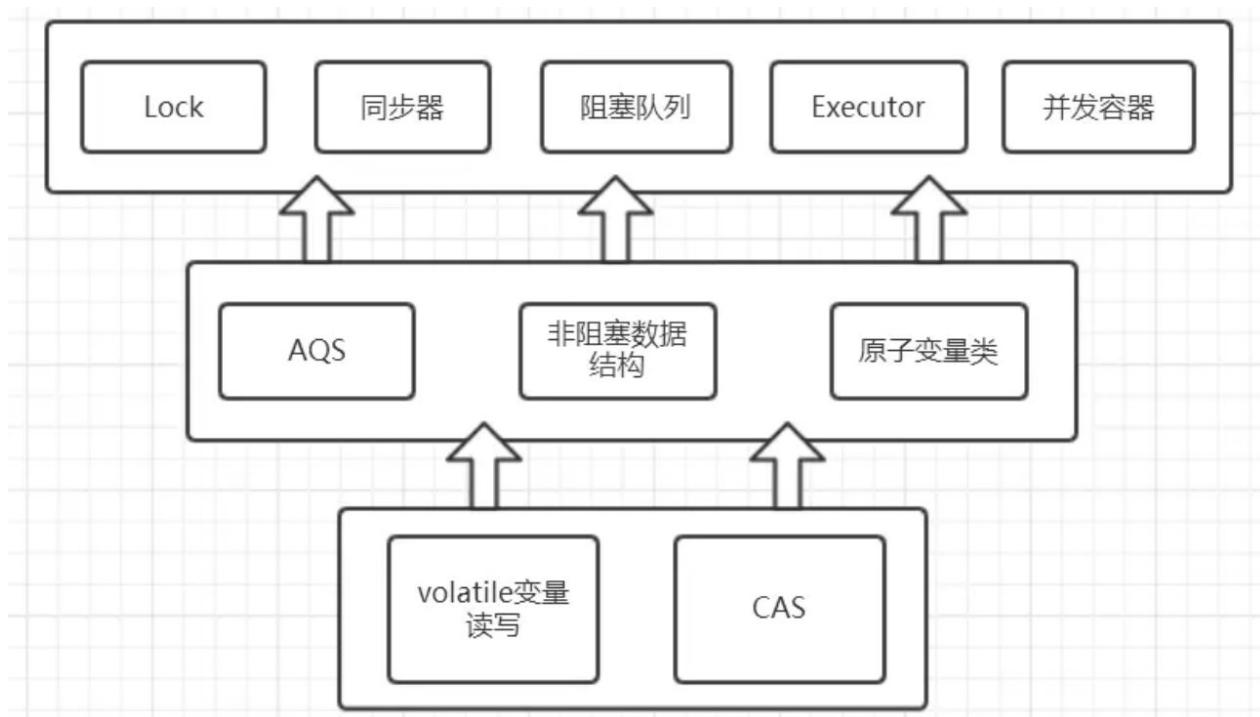
**volatile使用条件：**

- 1) 对变量的写操作不依赖于当前值
- 2) 该变量没有包含在具有其他变量的不变式中

## Lock体系

显示的Lock（也被直接称为加锁，锁，JDK提供支持，必须在编码时注意维护锁的释放）

**concurrent包结构**



## Lock简介

**优点：**如果抛出了异常，有机会去做清理工作，并且提供了更多的功能

**正确使用方式：**对Lock.lock()的调用后面紧跟try-finally，finally块中Lock.release()，确保任何情况下都能释放锁

```

Lock lock = new ReentrantLock();
public int next(){
    lock.lock();
    try{
        //what you are going to do
    }finally{
        lock.unlock();
    }
}
  
```

## ReentrantLock (Lock接口的唯一实现)

基本上所有的方法的实现实际上都是调用了其抽象静态内部类Sync (lock()方法抽象) 中的方法，而Sync类继承了AbstractQueuedSynchronizer (AQS)

- 具有重入性 关键方法 ReentrantLock\$Sync.nonfairTryAcquire(int acquires)
- 在获取ReentrantLock上阻塞的任务可以被中断 (Lock.lockInterruptibly()可中断加锁方式)
- 同时支持公平 (嵌套类FairSync继承自Sync) 与非公平锁 (嵌套类NonfairSync继承自Sync)
- 允许tryLock(), 尝试获取锁失败不阻塞，可以继续做后序工作

### 重入性

synchronized隐式支持重入性

表示能够对共享资源能够重复加锁，即当前线程获取该锁再次获取不会被阻塞。

### 公平锁与非公平锁

是对于获取锁而言的

公平锁 (核心方法 hasQueuedPredecessors)，判断了当前节点在AQS的同步队列中是否有前驱节点)

- 锁的获取顺序就应该符合请求上的绝对时间顺序，满足FIFO
- 保证时间上的绝对顺序，需要频繁的上下文切换

### 非公平锁

- 可能刚释放锁的线程下次继续获取该锁，造成“饿死”现象
- 非公平锁会降低一定的上下文切换，降低性能开销，保证了系统更大的吞吐量

## AbstractQueuedSynchronizer (抽象模板类，AQS，简称同步器)

AbstractQueuedLongSynchronizer 类提供相同的功能但扩展了对同步状态的 64 位的支持。两者都扩展了类

AbstractOwnableSynchronizer (一个帮助记录当前保持独占同步的线程的简单类) .

AQS是构造同步组件的基础框架，有一个int成员变量state表示同步状态，一个FIFO队列构成等待队列。

AQS已经实现的功能：

- 同步状态的管理 (getState, setState以及compareAndSetState)
- 对阻塞线程进行排队
- 等待通知

AQS提供的模板方法：

独占锁

`void acquire(int arg)`：独占式获取同步状态，如果获取失败则插入同步队列进行等待；

`void acquireInterruptibly(int arg)`：与acquire方法相同，但在同步队列中进行等待的时候可以检测中断；

`boolean tryAcquireNanos(int arg, long nanosTimeout)`: 在`acquireInterruptibly`基础上增加了超时等待功能，在超时时间内没有获得同步状态返回false;

`boolean release(int arg)`: 释放同步状态，该方法会唤醒在同步队列中的下一个节点

### 共享锁（带Shared后缀的方法）

`void acquireShared(int arg)`: 共享式获取同步状态，与独占式的区别在于同一时刻有多个线程获取同步状态；

`void acquireSharedInterruptibly(int arg)`: 在`acquireShared`方法基础上增加了能响应中断的功能；

`boolean tryAcquireSharedNanos(int arg, long nanosTimeout)`: 在`acquireSharedInterruptibly`基础上增加了超时等待的功能；

`boolean releaseShared(int arg)`: 共享式释放同步状态

### AQS需要被继承用来实现具体同步语义的方法：

具有独占锁功能的子类：它必须实现`tryAcquire`、`tryRelease`、`isHeldExclusively`等；

共享锁功能的子类：必须实现`tryAcquireShared`和`tryReleaseShared`等方法

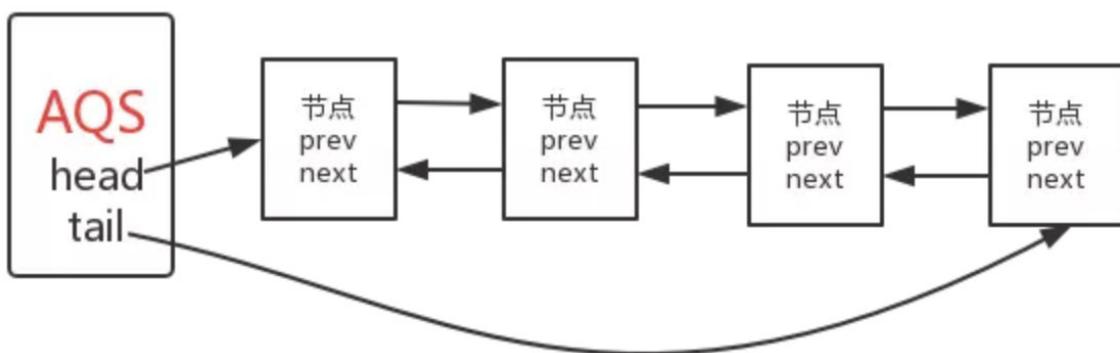
带**Shared**后缀的方法都是支持共享锁加锁的语义。Semaphore、ReentrantReadWriteLock是**共享锁**，ReentrantLock是**独占锁**。

### AQS中的同步队列（存放请求共享资源而阻塞的任务，FIFO）

1. 节点的数据结构是什么样的？
2. 是单向还是双向？
3. 是带头结点的还是不带头节点的？

实现：链式结构的双向队列（通过AQS的static final内部类Node实现）

AQS通过成员变量`head:Node`, `tail:Node`（队列头尾指针，有头节点的队列）来管理



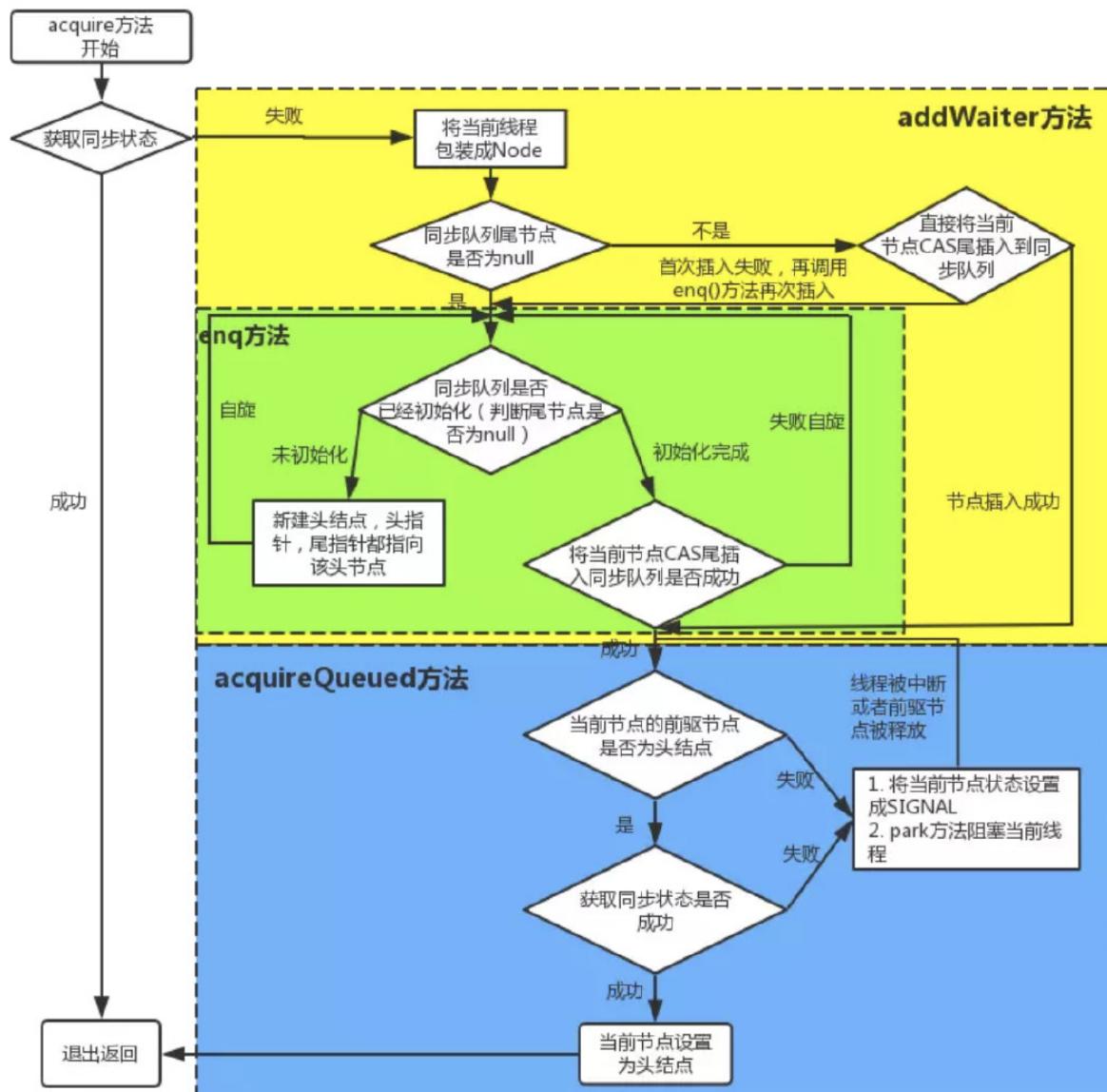
### 1.acquire方法

- 当线程获取独占式锁失败后就会将当前线程加入同步队列，那么加入队列的方式是怎样的？
- 在同步队列中的节点（线程）会做什么事情了来保证自己能够有机会获得独占式锁了？

获取锁失败：入队列，尾插法 `addWaiter(Node node)`，如果tail为null，调用 `enq(node)`，如果是第一个节点 `compareAndSetHead(new Node())`，否则自旋不断尝试CAS `compareAndSetTail` 尾插入节点直至成功为止

在同步队列中竞争锁：排队获取锁 `acquireQueued()` 方法，`acquireQueued()` 在自旋过程中主要完成了两件事情：

1. 如果当前节点的前驱节点是头节点，并且能够获得同步状态的话，当前线程能够获得锁该方法执行结束退出；
2. 获取锁失败的话，先将节点状态由INITIAL设置成SIGNAL，然后调用 `LookSupport.park` 方法使得当前线程阻塞。



## 2.release方法

释放锁：关键方法唤醒后继节点 `unparkSuccessor(Node node)`，每一次锁释放后就会唤醒队列中该节点的后继节点所引用的线程，从而进一步可以佐证获得锁的过程是一个FIFO（先进先出）的过程。

独占锁获取和释放总结：

- 线程获取锁失败，线程被封装成 `Node` 进行入队操作，核心方法在于 `addWaiter()` 和 `enq()`，同时 `enq()` 完成对同步队列的头结点初始化工作以及CAS操作失败的重试；

- 线程获取锁是一个自旋的过程，当且仅当当前节点的前驱节点是头结点并且成功获得同步状态时，节点出队即该节点引用的线程获得锁，否则，当不满足条件时就会调用 `LockSupport.park()` 方法使得线程阻塞
- 释放锁的时候会唤醒后继节点FIFO

**推荐使用：** 使用AQS构造同步组件，用同步组件类的一个静态内部类继承AQS，AQS的子类必须重写AQS的几个方法：

一个 `protected` 修饰的用来改变同步状态的方法

其模版方法设计模式：

将一些方法开放给子类进行重写，而同步器给同步组件所提供模板方法又会重新调用被子类所重写的方法

```
//AQS中需要重写的方法
protected boolean tryAcquire(int arg) {
    throw new UnsupportedOperationException();
}

//而AQS中的模板方法acquire()，这里面定义了调用逻辑：
public final void acquire(int arg) {
    if (!tryAcquire(arg) &&
        acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
        selfInterrupt();
}
```

ReentrantLock的内部类NonfairSync（继承自内部类Sync，Sync继承自AQS）会重写该方法为：

```
protected final boolean tryAcquire(int acquires) {
    return nonfairTryAcquire(acquires);
}
```

AQS中模版方法acquire是供子类同步组件调用的，而其中又调用了专用于子类重写的tryAcquire方法，这样实际

调用的实现就是子类重写过后的办法，但是整体的调用的逻辑又是在AQS中就写好的模板

- 同步组件（这里不仅仅值锁，还包括CountDownLatch等）的实现依赖于同步器AQS，在同步组件实现中，使用**AQS的方式被推荐定义继承AQS的静态内部类**；
- AQS采用模板方法进行设计，AQS的protected修饰的方法需要由继承AQS的子类进行重写实现，当调用AQS的子类的方法时就会调用被重写的方法；
- AQS负责同步状态的管理，线程的排队，等待和唤醒这些底层操作，而Lock等同步组件主要专注于**实现同步语义**；
- 在重写AQS的方式时，使用AQS提供的 `getState()`, `setState()`, `compareAndSetState()` 方法进行修改同步状态

## Lock（同步组件）和同步器(AQS)的关系

锁是面向使用者，它定义了使用者与锁交互的接口，隐藏了实现细节；

同步器是面向锁的实现者，它简化了锁的实现方式，屏蔽了同步状态的管理，线程的排队，等待和唤醒等底层操作。

总的来说，同步组件通过重写AQS的方法实现自己想要表达的同步语义，而AQS只需要同步组件表达的true和false即可，AQS会针对true和false不同的情况做不同的处理

## ReadWriteLock（接口）

只有两个方法，目的是为了把文件的读写分开加锁，为了支持多线程同时读

- Lock readLock(), 获取读文件锁
- Lock writeLock(), 获取写文件锁

## ReentrantReadWriteLock（共享锁，ReadWriteLock唯一实现）

1. 读写锁是怎样实现分别记录读写状态的？
2. 写锁是怎样获取和释放的？
3. 读锁是怎样获取和释放的？

提供了很多丰富的方法，最主要的有两个方法：Lock readLock()和Lock writeLock()用来获取读锁和写锁

内部其实包含了读锁和写锁

### 静态内部类 WriteLock 写锁（独占锁）

获取：重写AQS的tryAcquire方法来实现其语义

其中有个关键方法，exclusiveCount()

释放：重写AQS的tryRelease方法

同步状态的低16位用来表示写锁的获取次数

### 静态内部类 ReadLock 读锁（共享锁）

重写tryAcquireShared方法和tryReleaseShared方法

锁降级

遵循按照获取写锁，获取读锁再释放写锁的次序，写锁能够降级成为读锁

processCachedData()方法，但是在ReentrantReadWriteLock中是被注释的

同步状态的高16位用来表示读锁被获取的次数

- 已经有线程在写时，申请读/写锁都会被阻塞
- 已经有线程在读时，申请写锁会被阻塞
- 已经有线程在读时，申请读锁不阻塞

## Lock和synchronized的选择

- Lock是一个接口（JDK），而synchronized是Java中的关键字（JVM）
- synchronized在发生异常时，会自动释放线程占有的锁，Lock在发生异常时不会自动释放，所以需要finally块
- Lock可以让等待锁而被阻塞的线程响应中断，而synchronized却不行

- Lock相应方法返回值可以知道有没有成功获取锁，而synchronized却无法办到
- Lock可以提高多个线程进行读操作的效率

## 线程本地存储ThreadLocal (java.lang.ThreadLocal)

和线程同步没关系，其实是线程间数据隔离了

Thread对象中都有一个成员变量**threadLocals**: ThreadLocalMap对象

<ThreadLocal.threadHashCode, 本地线程变量值>K-V对

注意：ThreadLocalMap是属于Thread的，但里面的key是**ThreadLocal.threadHashCode**

每一个**ThreadLocal**对象都包含了一个独一无二的**threadLocalHashCode**值

适用场景：

- ThreadLocal 适用于变量在线程间隔离且在方法间共享的场景（解决 数据库连接、Session管理等）

## 终结任务

### 线程状态

- 新建 (new) : 已经被分配了必须的系统资源，并执行了初始化，有资格获得CPU时间。这个状态很短暂，调度器会把这个线程转变成就绪状态或者阻塞状态
- 就绪 (Runnable) : 只要分配CPU时间片，就可以运行。**不同于死亡和阻塞状态**
- 阻塞 (Blocked) : 线程可以运行，但被某个条件阻止，此状态不会分配CPU时间，知道线程重新进入就绪状态才能执行操作
- 死亡 (dead) : 不可再调度，通常的死亡方式是从run()方法返回，但是**任务的线程还可以被中断！！！**

### 线程阻塞的5种情况

- sleep(milliseconds), 不会释放锁
- wait(), 释放锁，直到得到notify()或notifyAll()（或者是Java 5加入的singal() signalAll()）
- join(), 释放锁，使调用此方法的线程wait()，直到调用此方法的线程对象所在的线程执行完毕后被唤醒。
- 任务在等待某个输入/输出 (I/O) 完成（无法中断（nio除外），但可以通过**close Stream**结束任务）
- 任务调用某个对象的synchronized方法，但是对象锁不可用（无法中断，但获取**ReentrantLock**阻塞可以中断）

### 中断 (Thread.interrupt())

两种情况线程终止的情况

(1) 线程处于阻塞状态，如使用了sleep方法。（但是如果线程不在阻塞，那么这个调用无效，但会使中断标志位为true）

(2) 线程非阻塞状态使用while (! isInterrupted ()) {.....}来判断线程是否被中断。

第一种必须要在部分阻塞状态下才能成功终止线程，并抛出**InterruptedException**，并且置中断标志为**false**

第二种是在非阻塞状态调用**interrupt**后会置中断标志位为**true**

## 如何正确终止线程

1. 使用退出标志，使线程正常退出，也就是当 **run** 方法完成后线程终止

```
public class MyThread implements Runnable {  
    private volatile boolean isCancelled; //要用volatile保证其他线程对标志的修改  
    能立即可见  
    public void run() {  
        while (!isCancelled) {  
            //do something  
        }  
    }  
    public void cancel() { isCancelled=true; }  
}
```

2. 使用 **interrupt** 方法中断线程（用于线程会阻塞的情况）

- 调用中断：**void interrupt()**，如果线程处于被阻塞状态（例如处于 **sleep**, **wait**, **join** 等状态），那么线程将立即退出被阻塞状态，并抛出一个 **InterruptedException** 异常，同时会将中断状态置为**false**

如果线程处于正常活动状态，那么会将该线程的中断标志设置为 **true**。被设置中断标志的线程将继续正常运行

- 查询中断标志：**static boolean interrupted()**，测试当前线程（正在执行这一命令的线程）是否被中断。这一调用会将当前线程的中断状态重置为 **false**！！！
- 查询中断标志：**boolean isInterrupted()**，测试线程是否被终止。不像静态的中断方法，这一调用不改变线程的中断状态

也就是说，interrupt()不能真的终止线程，需要线程自身的配合响应中断！！！

使用静态查询中断标志方法响应中断（静态方法会清除中断状态位）

什么时候如此使用？是为了不结束线程，响应中断后还能够继续执行接下来的任务，所以清除中断状态

```
Thread thread = new Thread(() -> {  
    while (!Thread.interrupted()) {  
        // do more work  
    }  
    // return or throw InterruptedException  
});
```

在阻塞方法中正确响应中断，阻塞方法中的中断会清除中断状态！！！

比较好的做法就是在**catch**块中恢复中断状态

```

public void run() {
    while (!Thread.currentThread().isInterrupted()) { //这里调用的是非静态
        查询中断状态方法
        try {
            System.out.println("test");
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            System.out.println("interrupt");
            // 抛出InterruptedException后中断标志被清除，标准做法是再次调用
            interrupt恢复中断
            Thread.currentThread().interrupt(); //恢复中断状态，让其他线程了解此线程被中断过
        }
    }
    System.out.println("stop");
}

```

## 线程间协作（要协作肯定就要释放锁，等待/通知机制）

**Object类里提供的 等待/通知 机制（和对象锁monitor配合实现，JVM级别）**

忙等待：不释放锁，一直执行空循环

wait(), notify(), notifyAll(), 都是！！！ Object里的native方法！！！，只能在同步块里调用（因为要操作锁）

### wait()

释放锁，可以接受时间参数，也可以无参（一直等待）

正确使用方法：

```

while(感兴趣的条件满足 == false){
    wait();
}

```

这样做是避免wait被唤醒时，等待的条件（或资源）突然不满足了（如被其他线程占有了），这时此线程应该继续回到wait()中

### 错失的信号

```

T1:
synchronized (shareMonitor){
    //setup condition for T2
    sharedMonitor.notify();
}

T2:
while(someCondition == false){
    //Point 1
    synchronized (ssharedMonitor){
        sharedMonitor.wait();
    }
}

```

T1执行到Point1时，someCondition并未满足，所以T1准备wait()，但此时发生了线程调度，切换到了T2，T2设置好someCondition后发出了notify()，但切换到T1时，T1由于已经判断过了someCondition，所以会错误的进行wait()，导致永远无法收到T2发出的notify()，产生死锁。

那么要求就是：**wait()感兴趣的条件的判断必须也放在同步块里**

**notify和notifyAll到底唤醒了什么？**

由于都只能在同步块中使用，所以notify和notifyAll唤醒的是等待此同步块对应的锁的任务

## Lock体系下提供的 等待/通知 机制（和Lock配合实现，JDK级别）

**Condition**（接口，具体实现在AQS中的内部类 `ConditionObject`）简介

1. Condition能够支持不响应中断 `condition.awaitUninterruptibly()`！！，而通过使用Object方式不支持；
2. Condition能够支持多个等待队列（new 多个Condition对象），而Object方式只能支持一个；
3. Condition能够支持超时时间的设置，而Object不支持

参照Object的wait和notify/notifyAll方法，Condition也提供了同样的方法：

### 针对Object的wait方法

1. `void await() throws InterruptedException`: 当前线程进入等待状态，如果其他线程调用condition的signal或者signalAll方法并且当前线程获取Lock从await方法返回，如果在等待状态中被中断会抛出被中断异常；
2. `long awaitNanos(long nanosTimeout)`: 当前线程进入等待状态直到被通知，中断或者超时；
3. `boolean await(long time, TimeUnit unit) throws InterruptedException`: 同第二种，支持自定义时间单位
4. `boolean awaitUntil(Date deadline) throws InterruptedException`: 当前线程进入等待状态直到被通知，中断或者到了某个时间

### 针对Object的notify/notifyAll方法

1. `void signal()`: 唤醒一个等待在condition上的线程，将该线程从等待队列中转移到同步队列中，如果在同步队列中能够竞争到Lock则可以从等待方法中返回。
2. `void signalAll()`: 与1的区别在于能够唤醒所有等待在condition上的线程

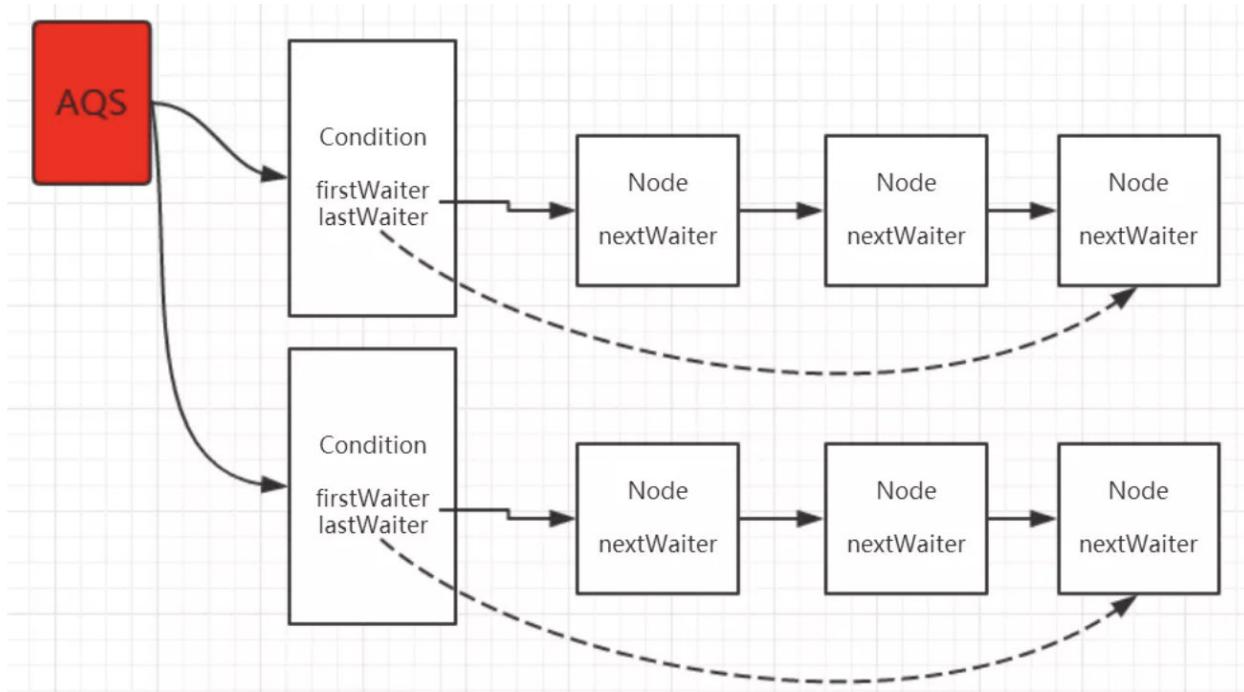
## Condition等待队列

AQS的内部类 ConditionObject 是Condition的实现，其维护等待队列的做法和AQS维护同步队列相似，还复

用了AQS的 Node 来构造单向等待队列（无头节点）！！！，不同于同步队列

可以多次调用lock.newCondition()方法创建多个condition对象，一个lock可以持有多个等待队列

对象Object对象监视器（monitor）上只能拥有一个同步队列和一个等待队列，而并发包中的Lock拥有一个同步队列和多个等待队列。



## await实现原理

```
public final void await() throws InterruptedException {
    if (Thread.interrupted())
        throw new InterruptedException();
    // 1. 将当前线程包装成Node, 尾插入到等待队列中
    Node node = addConditionWaiter();
    // 2. 释放当前线程所占用的lock, 在释放的过程中会唤醒同步队列中的下一个节点
    int savedState = fullyRelease(node);
    int interruptMode = 0;
    while (!isOnSyncQueue(node)) { //判断是否被移到了同步队列中 (signal or
        signalAll), 是则退出循环
        // 3. 当前线程进入到等待状态
        LockSupport.park(this);
        if ((interruptMode = checkInterruptWhileWaiting(node)) != 0)
            break; //如果被中断则退出循环
    }
    // 4. 自旋等待获取到同步状态 (即获取到lock), 获取到lock后才能退出await方法
    if (acquireQueued(node, savedState) && interruptMode != THROW_IE)
        interruptMode = REINTERRUPT;
}
```

```
if (node.nextWaiter != null) // clean up if cancelled
    unlinkCancelledWaiters();
// 5. 处理被中断的情况
if (interruptMode != 0)
    reportInterruptAfterWait(interruptMode);
}
```

1. 是怎样将当前线程添加到等待队列中去的?
2. 释放锁的过程?
3. 怎样才能从await方法退出?

`addConditionWaiter()` 尾插法进入等待队列（同步队列头节点->等待队列）

通过 `fullRelease()` 调用AQS的模板方法 `release`（而模版方法调用的 `tryRelease` 就是由new这个 `ConditionObject`的Lock实现的）释放锁，并且唤醒在同步队列中头节点的后继节点

## 从await中退出

### 1.前提条件（满足以下两种情况中一种）

- 当前线程被中断
- 调用`condition.signal`/`condition.signalAll`（等待队列->同步队列）

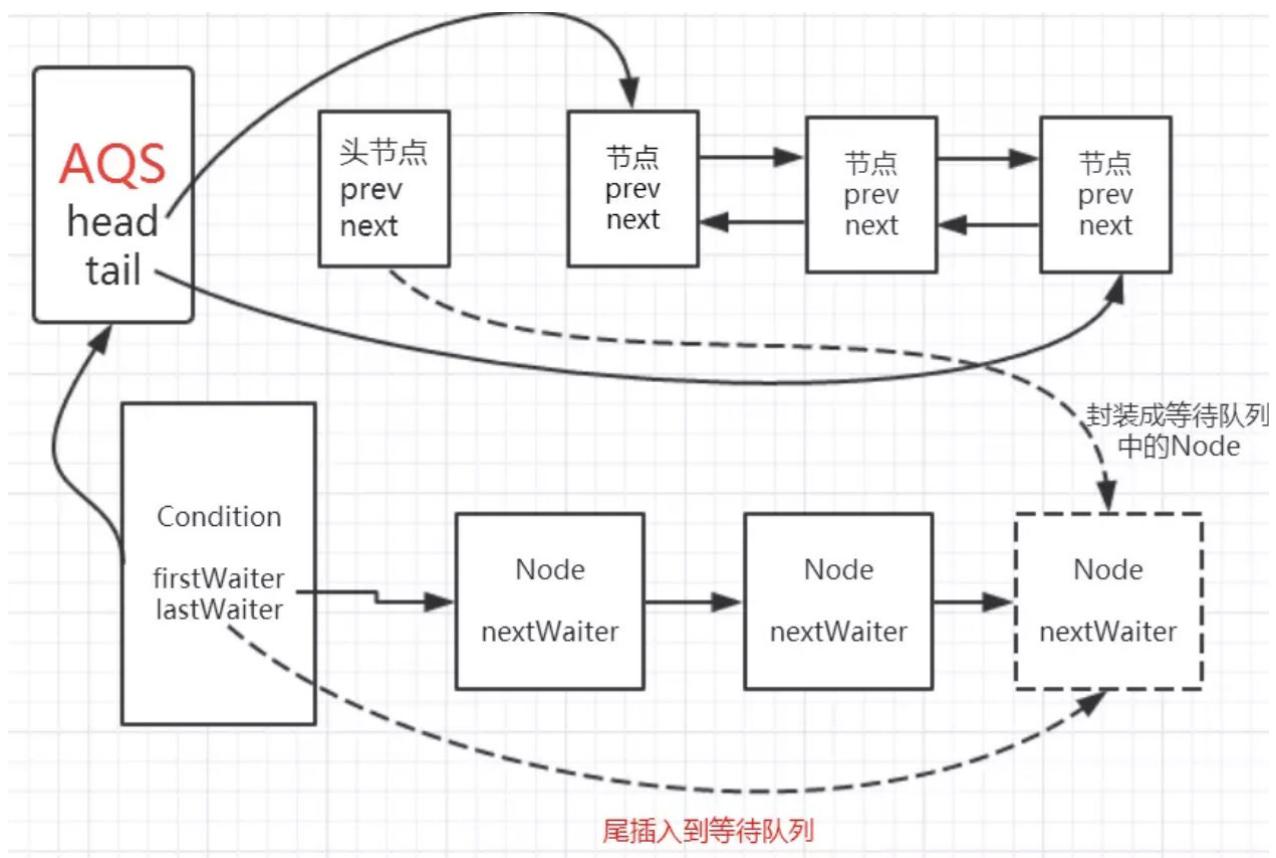
### 2.关键步骤

同步队列中竞争锁 `acquireQueued(node, savedState)`，直至成功（线程获取到lock）这里和AQS中的队列中竞争锁是一个用法。

### 3.成功退出await的标志

已经竞争到了`condition`引用（关联）的lock

下图是await方法的调用过程，调用await的线程必须是获得了lock的线程，也就是同步队列头节点



## signal/signalAll实现原理

调用condition的signal或者signalAll方法可以将等待队列中 等待时间最长 的节点移动到同步队列中

等待队列 (FIFO) : 头节点必定是等待时间最长的

```

public final void signal() {
    //1. 先检测当前线程是否已经获取lock
    if (!isHeldExclusively())
        throw new IllegalMonitorStateException();
    //2. 获取等待队列中第一个节点, 之后的操作都是针对这个节点
    Node first = firstWaiter;
    if (first != null)
        doSignal(first);
}

private void doSignal(Node first) {
    do {
        if ( (firstWaiter = first.nextWaiter) == null)
            lastWaiter = null;
        //1. 将头结点从等待队列中移除
        first.nextWaiter = null;
        //2. while中transferForSignal方法对头结点做真正的处理
    } while (!transferForSignal(first) &&
             (first = firstWaiter) != null);
}

final boolean transferForSignal(Node node) {
    /*
     * If cannot change waitStatus, the node has been cancelled.
    */
}

```

```

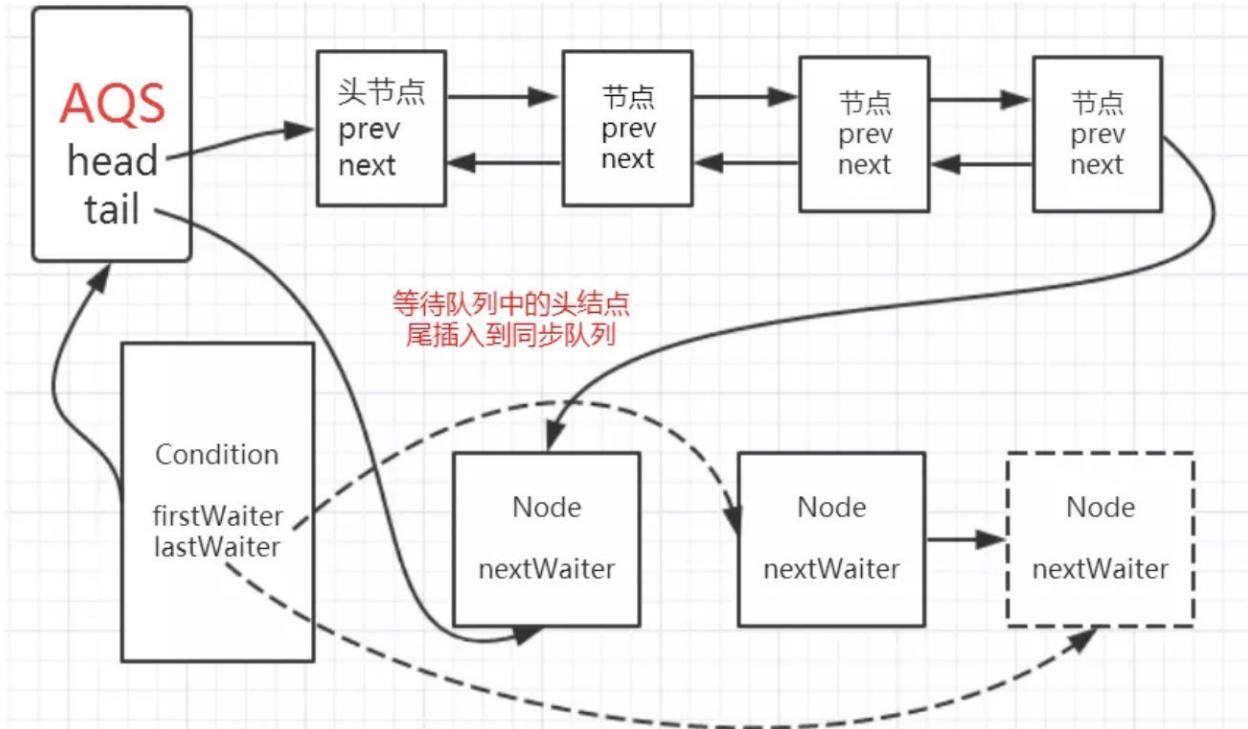
/*
//1. 更新状态为0
if (!compareAndSetWaitStatus(node, Node.CONDITION, 0))
    return false;
/*
 * Splice onto queue and try to set waitStatus of predecessor to
 * indicate that thread is (probably) waiting. If cancelled or
 * attempt to set waitStatus fails, wake up to resync (in which
 * case the waitStatus can be transiently and harmlessly wrong).
*/
//2. 将该节点移入到同步队列中去
Node p = enq(node);
int ws = p.waitStatus;
if (ws > 0 || !compareAndSetWaitStatus(p, ws, Node.SIGNAL)) //node状态变为
signal
    LockSupport.unpark(node.thread);
return true;
}

```

## signal调用总结

调用signal的线程必须是获取了lock的线程

调用效果：等待队列头节点->尾插入同步队列 `enq()` (这样才await的线程才能从 `LockSupport.park(this)` 返回，才能调用 `acquireQueued()` 去竞争锁)



signalAll

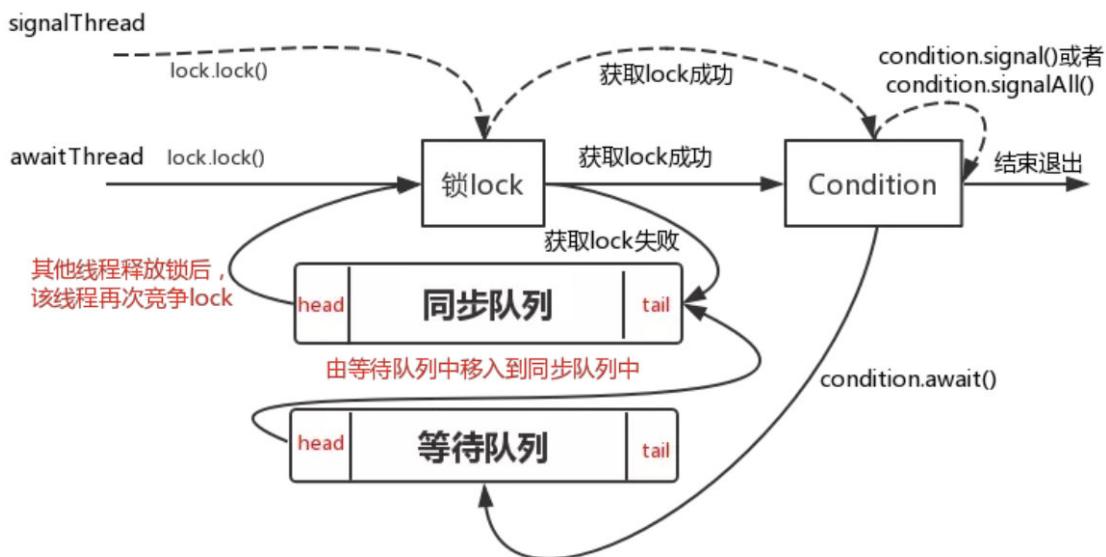
```

private void doSignalAll(Node first) {
    lastWaiter = firstWaiter = null;
    do {
        Node next = first.nextWaiter;
        first.nextWaiter = null;
        transferForSignal(first);
        first = next;
    } while (first != null);
}

```

等待队列所有节点都移入到同步队列中，即“通知”当前调用condition.await()方法的每一个线程。

## await与signal/signalAll的结合思考



如图，线程awaitThread先通过lock.lock()方法获取锁成功后调用了condition.await方法进入等待队列，而另一个线程signalThread通过lock.lock()方法获取锁成功后调用了condition.signal或者signalAll方法，使得线程awaitThread能够有机会移入到同步队列中，当其他线程释放lock后使得线程awaitThread能够有机会获取lock，从而使得线程awaitThread能够从await方法中退出执行后续操作。如果awaitThread获取lock失败会直接进入到同步队列。

## await和signal比较

### 相同点

调用await和signal方法的线程必须都是同步队列的头节点，因为必须是拥有锁的

### 区别

- await是对调用者线程起作用，将当前线程尾插入等待队列，释放锁，并等待singal或者中断
- singal是对等待队列的线程起作用，不释放当前线程锁，将等待队列头节点尾插入同步队列，给其一个竞争锁的机会

## LockSupport

LockSupport是线程的阻塞原语，用来阻塞线程和唤醒线程。

AQS的底层都是基于LockSupport实现，LockSupport不可重入

### 阻塞线程

1. `void park()`: 阻塞当前线程，如果调用unpark方法或者当前线程被中断，从能从park()方法中返回
2. `void park(Object blocker)`: 功能同方法1，入参增加一个Object对象，用来记录导致线程阻塞的阻塞对象，方便进行问题排查；
3. `void parkNanos(long nanos)`: 阻塞当前线程，最长不超过nanos纳秒，增加了超时返回的特性；
4. `void parkNanos(Object blocker, long nanos)`: 功能同方法3，入参增加一个Object对象，用来记录导致线程阻塞的阻塞对象，方便进行问题排查；
5. `void parkUntil(long deadline)`: 阻塞当前线程，直到deadline；
6. `void parkUntil(Object blocker, long deadline)`: 功能同方法5，入参增加一个Object对象，用来记录导致线程阻塞的阻塞对象，方便进行问题排查；

### 唤醒线程

`void unpark(Thread thread)`: 唤醒处于阻塞状态的指定线程

实际上LockSupport阻塞和唤醒线程的功能是依赖于sun.misc.Unsafe，这是一个很底层的类，直接修改内存，所以叫Unsafe，比如park()方法的功能实现则是靠unsafe.park()方法。

**synchronized**致使线程阻塞，线程会进入到BLOCKED状态，而调用**LockSupport**方法阻塞线程会致使线程进入到WAITING状态。

```
public class LockSupportDemo {  
    public static void main(String[] args) {  
        Thread thread = new Thread(() -> {  
            LockSupport.park();  
            System.out.println(Thread.currentThread().getName() + "被唤醒");  
        });  
        thread.start();  
        try {  
            Thread.sleep(3000);  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
        LockSupport.unpark(thread);  
    }  
}
```

## 并发容器

### ConcurrentHashMap

JDK 1.7版本关键要素（将桶分段上锁）：

1. segment继承了ReentrantLock充当锁的角色，为每一个segment提供了线程安全的保障；

2. segment维护了哈希散列表的若干个桶，每个桶由HashEntry构成的链表。

JDK 1.8的ConcurrentHashMap就有了很大的变化（**Synchronized**和**CAS**，提升了性能）

1. 舍弃了**segment**，并且大量使用了**synchronized**，以及**CAS**无锁操作以保证 ConcurrentHashMap操作的线程安全性
2. synchronized做了很多的优化，包括偏向锁，轻量级锁，重量级锁，可以依次向上升级锁状态，但不能降级
3. 底层数据结构改变为采用数组+链表+红黑树的数据形式。

### ConcurrentHashMap的关键属性

#### table (Hash表)

```
transient volatile Node<K,V>[] table;
```

装载Node的数组，采用**懒加载**的方式，直到第一次插入数据的时候才会进行初始化操作，数组的大小总是为2的幂次方。(毫无疑问，这里和HashMap是一样的)

#### nextTable (扩容用)

```
/**  
 * The next table to use; non-null only while resizing.  
 */  
private transient volatile Node<K,V>[] nextTable;
```

这是一个连接表，用于哈希表扩容，扩容完成后会被重置为null。

#### baseCount

```
private transient volatile long baseCount;
```

该属性保存着整个哈希表中存储的所有结点的个数总和，有点类似于HashMap的size属性。

#### sizeCtl (类似HashMap中Threshold)

```
private transient volatile int sizeCtl;
```

这是一个重要的属性，无论是初始化哈希表，还是扩容 rehash 的过程，都是需要依赖这个关键属性的。该属性有以下几种取值：！！！ **sizeCtl** 如果处于扩容状态的话！！！

前 16 位是数据校验标识，后 16 位是当前正在扩容的线程总数

- 0: 默认值
- -1: 代表哈希表正在进行初始化（在initTable里由负责初始化的线程通过CAS操作赋值为-1）
- 大于0: 相当于HashMap中的**threshold**，表示阈值（在构造函数里赋值的）
- 小于-1: -N代表有N-1个线程正在进行扩容

```
private static final sun.misc.Unsafe U;
```

## Unsafe U

大量的同步组件和并发容器的实现中使用**CAS**是通过 `sun.misc.Unsafe` 类实现

该类提供了一些可以**直接操控内存和线程的底层操作**，可以理解为java中的“指针”。该变量获取在静态块中

```
static {
    try {
        U = sun.misc.Unsafe.getUnsafe();
        .....
    } catch (Exception e) {
        throw new Error(e);
    }
}
```

## ConcurrentHashMap中关键内部类

### Node

Node类实现了Map.Entry接口，主要存放key-value对，并且具有next域

```
static class Node<K,V> implements Map.Entry<K,V> {
    final int hash;
    final K key;
    volatile V val;
    volatile Node<K,V> next;
    .....
}
```

### TreeNode

继承于承载数据的Node类。而红黑树的操作是针对TreeBin类的，从该类的注释也可以看出，也就是TreeBin会将TreeNode进行再一次封装

```
**
 * Nodes for use in TreeBins
 */
static final class TreeNode<K,V> extends Node<K,V> {
    TreeNode<K,V> parent; // red-black tree links
    TreeNode<K,V> left;
    TreeNode<K,V> right;
    TreeNode<K,V> prev; // needed to unlink next upon deletion
    boolean red;
    .....
}
```

### TreeBin (独特的，不同于HashMap)

这个类并不负责包装用户的key、value信息，而是包装的很多TreeNode节点。实际的ConcurrentHashMap“数组”中，存放的是TreeBin对象，而不是TreeNode对象。

```
static final class TreeBin<K,V> extends Node<K,V> {
    TreeNode<K,V> root;
    volatile TreeNode<K,V> first;
    volatile Thread waiter;
    volatile int lockState;
    // values for lockState
    static final int WRITER = 1; // set while holding write lock
    static final int WAITER = 2; // set when waiting for write lock
    static final int READER = 4; // increment value for setting read lock
    .....
}
```

## CAS关键操作

几个常用的利用CAS算法来保障线程安全的操作。

### tabAt

该方法用来获取table数组中索引为i的Node元素。

```
static final <K,V> Node<K,V> tabAt(Node<K,V>[] tab, int i) {
    return (Node<K,V>)U.getObjectVolatile(tab, ((long)i << ASHIFT) + ABASE);
}
```

### casTabAt

利用CAS算法设置i位置上的Node节点。之所以能实现并发是因为他指定了原来这个节点的值是多少 在CAS算法中，会比较内存中的值与你指定的这个值是否相等，如果相等才接受你的修改，否则拒绝你的修改 因此当前线程中的值并不是最新的值，这种修改可能会覆盖掉其他线程的修改结果，ABA问题

```
static final <K,V> boolean casTabAt(Node<K,V>[] tab, int i,
                                       Node<K,V> c, Node<K,V> v) {
    return U.compareAndSwapObject(tab, ((long)i << ASHIFT) + ABASE, c, v);
}
```

### setTabAt

利用volatile方法设置数组table中位置为i的node，其实只在同步块内调用，目前编码为volatile是保守做法

```
static final <K,V> void setTabAt(Node<K,V>[] tab, int i, Node<K,V> v) {
    U.putObjectVolatile(tab, ((long)i << ASHIFT) + ABASE, v);
}
```

## ConcurrentHashMap构造器

```

// 1. 构造一个空的map, 即table数组还未初始化, 初始化放在第一次插入数据时, 默认大小为16
ConcurrentHashMap()
// 2. 给定map的大小
ConcurrentHashMap(int initialCapacity)
// 3. 给定一个map
ConcurrentHashMap(Map<? extends K, ? extends V> m)
// 4. 给定map的大小以及加载因子
ConcurrentHashMap(int initialCapacity, float loadFactor)
// 5. 给定map大小, 加载因子以及并发度 (预计同时操作数据的线程)
ConcurrentHashMap(int initialCapacity, float loadFactor, int concurrencyLevel)

```

注意最后一个, 可以设置并发速度的构造器, **ConcurrentHashMap**独有

```

public ConcurrentHashMap(int initialCapacity) {
    //1. 小于0直接抛异常
    if (initialCapacity < 0)
        throw new IllegalArgumentException();
    //2. 判断是否超过了允许的最大值, 超过了话则取最大值, 否则再对该值进一步处理
    int cap = ((initialCapacity >= (MAXIMUM_CAPACITY >>> 1)) ?
               MAXIMUM_CAPACITY :
               tableSizeFor(initialCapacity + (initialCapacity >>> 1) + 1));
    //3. 赋值给sizeCtl
    this.sizeCtl = cap;
}

```

最后将cap赋值给sizeCtl, 关于sizeCtl的说明请看上面的说明, 当调用构造器方法之后, sizeCtl的大小应该就代表了**ConcurrentHashMap**的大小, 即table数组长度。

**initTable**方法 (第一次put时调用, 通过判断sizeCtl < 0只允许一个线程初始化表)

```

private final Node<K,V>[] initTable() {
    Node<K,V>[] tab; int sc;
    //如果表为空才进行初始化操作
    while ((tab = table) == null || tab.length == 0) {
        //当前线程应该放弃 CPU 的使用
        if ((sc = sizeCtl) < 0)
            // 1. 保证只有一个线程正在进行初始化操作
            Thread.yield(); // lost initialization race; just spin
        //否则说明还未有线程对表进行初始化, 那么本线程就来做这个工作
        else if (U.compareAndSwapInt(this, SIZECTL, sc, -1)) {
            try {
                if ((tab = table) == null || tab.length == 0) {
                    //sc 大于零说明容量已经初始化了, 否则使用默认容量
                    int n = (sc > 0) ? sc : DEFAULT_CAPACITY;
                    @SuppressWarnings("unchecked")
                    // 3. 这里才真正的初始化数组
                    Node<K,V>[] nt = (Node<K,V>[])new Node<?,?>[n];

```

```

        table = tab = nt;
        // 4. 计算数组中可用的大小: 实际大小n*0.75 (加载因子)
        sc = n - (n >>> 2);
    }
} finally {
    //设置阈值
    sizeCtl = sc;
}
break;
}
}
return tab;
}

```

这里乘以0.75是怎么算的，0.75为四分之三，这里`n - (n >>> 2)`，如果选择是无参的构造器的话，这里在new Node数组的时候会使用默认大小为`DEFAULT_CAPACITY` (16)

## put方法（实际调用putVal，和HashMap相似）

整体流程：

- 首先对于每一个放入的值，首先利用spread方法对key的hashcode进行一次hash计算，由此来确定这个值在table中的位置；
- 如果当前table数组还未初始化，先将table数组进行初始化操作；
- 如果这个位置是null的，那么使用CAS操作直接放入；
- 如果这个位置存在结点，说明发生了hash碰撞，首先判断这个节点的类型。如果该节点`fh==MOVED`(代表forwardingNode,代表当前桶已经完成所有数据迁移，但是还有线程正在进行扩容)的话，说明正在进行扩容；
- 如果是链表节点(`fh>0`)，则得到的结点就是hash值相同的节点组成的链表的头节点。需要依次向后遍历确定这个新加入的值所在位置。如果遇到key相同的节点，则只需要覆盖该结点的value值即可。否则依次向后遍历，直到链表尾插入这个结点（Synchronized块）；
- 如果这个节点的类型是TreeBin的话，直接调用红黑树的插入方法进行插入新的节点（Synchronized块）；
- 插入完节点之后再次检查链表长度，如果长度大于8，把这个链表转换成红黑树；
- 对当前容量大小进行检查，如果超过了临界值（实际大小\*负载因子）就需要扩容。

```

/** Implementation for put and putIfAbsent */
final V putVal(K key, V value, boolean onlyIfAbsent) {
    //对传入的参数进行合法性判断
    if (key == null || value == null) throw new NullPointerException();
    //1. 计算key的hash值
    int hash = spread(key.hashCode());
    int binCount = 0;
    for (Node<K,V>[] tab = table;;) {
        Node<K,V> f; int n, i, fh;
        //如果哈希表还未初始化，那么调用initTable方法初始化它
        if (tab == null || (n = tab.length) == 0)
            tab = initTable();
        //根据键的 hash 值找到哈希数组相应的索引位置

```

```

//如果为空，那么以cas无锁式向该位置添加一个节点
else if ((f = tabAt(tab, i = (n - 1) & hash)) == null) {
    if (casTabAt(tab, i, null,
        new Node<K,V>(hash, key, value, null)))
        break; // no lock when adding to empty bin
}

//检测到桶结点是 ForwardingNode 类型，协助扩容
else if ((fh = f.hash) == MOVED)
    tab = helpTransfer(tab, f);
//桶结点是普通的结点，锁住该桶头结点并试图在该链表的尾部添加一个节点
else {
    V oldVal = null;
    synchronized (f) {
        if (tabAt(tab, i) == f) {
            //向普通的链表中添加元素，无需赘述
            if (fh >= 0) {
                binCount = 1;
                for (Node<K,V> e = f;; ++binCount) {
                    K ek;
                    if (e.hash == hash &&
                        ((ek = e.key) == key ||
                        (ek != null && key.equals(ek)))) {
                        oldVal = e.val;
                        if (!onlyIfAbsent)
                            e.val = value;
                        break;
                    }
                    Node<K,V> pred = e;
                    if ((e = e.next) == null) {
                        pred.next = new Node<K,V>(hash, key,
                            value, null);
                        break;
                    }
                }
            }
            //向红黑树中添加元素，TreeBin 结点的hash值为TREEBIN (-2)
            else if (f instanceof TreeBin) {
                Node<K,V> p;
                binCount = 2;
                if ((p = ((TreeBin<K,V>)f).putTreeVal(hash, key,
                    value)) != null) {
                    oldVal = p.val;
                    if (!onlyIfAbsent)
                        p.val = value;
                }
            }
        }
    }
    //binCount != 0 说明向链表或者红黑树中添加或修改一个节点成功
}

```

```

//binCount == 0 说明 put 操作将一个新节点添加成为某个桶的首节点
if (binCount != 0) {
    //链表深度超过 8 转换为红黑树
    if (binCount >= TREEIFY_THRESHOLD)
        treeifyBin(tab, i);
    //oldVal != null 说明此次操作是修改操作(修改操作在这里就返回了)
    //直接返回旧值即可, 无需做下面的扩容边界检查
    if (oldVal != null)
        return oldVal;
    break;
}
}

//CAS 式更新baseCount, 并判断是否需要扩容
addCount(1L, binCount);
//程序走到这一步说明此次 put 操作是一个添加操作, 否则早就 return 返回了
return null;
}

```

注意ConcurrentHashMap的扰动函数为 spread()，与HashMap的 hash 不同，但扰动方式不一样，**ConcurrentHashMap扰动函数中多了一步按位与！！！**

1.spread()重哈希，以减小Hash冲突

```

static final int spread(int h) {
    return (h ^ (h >>> 16)) & HASH_BITS;
}
static final int HASH_BITS = 0x7fffffff; // usable bits of normal node hash

```

2.初始化table

紧接着到第2步，会判断当前table数组是否初始化了，没有的话就调用initTable进行初始化

3.能否直接将新值插入到table数组中

**tabAt()**方法获取该位置上的元素，如果当前**Node f**为**null**的话，直接用casTabAt方法将新值插入。插入**null**桶的时候是**CAS无锁操作**

4.当前是否正在扩容

当前节点为特殊节点 (forwardingNode) , (fh = f.hash) == MOVED, 代表正在扩容

```

static final int MOVED      = -1; // hash for forwarding nodes
static final int TREEBIN    = -2; // hash for roots of trees
static final int RESERVED   = -3; // hash for transient reservations
static final int HASH_BITS = 0x7fffffff; // usable bits of normal node hash

```

5.当table[i]为链表的头结点，在链表中插入新值

在table[i]不为null并且不为forwardingNode时，并且当前Node f的hash值大于0，说明f为当前桶的所有节点组成的链表的头结点

插入新值的话就是向这个链表插入新值，**synchronized (f)**以保证线程安全性！！！！

```
if (fh >= 0) {
    binCount = 1;
    for (Node<K,V> e = f;; ++binCount) {
        K ek;
        // 找到hash值相同的key, 覆盖旧值即可
        if (e.hash == hash &&
            ((ek = e.key) == key ||
             (ek != null && key.equals(ek)))) {
            oldVal = e.val;
            if (!onlyIfAbsent)
                e.val = value;
            break;
        }
        Node<K,V> pred = e;
        if ((e = e.next) == null) {
            //如果到链表末尾仍未找到, 则直接将新值插入到链表末尾即可
            pred.next = new Node<K,V>(hash, key,
                                         value, null);
            break;
        }
    }
}
```

两种情况：

1. 在链表中如果找到了与待插入的键值对的key相同的节点，就直接覆盖即可；
2. 如果直到找到了链表的末尾都没有找到的话，就直接将待插入的键值对追加到链表的末尾即可

6.当table[i]为红黑树的根节点，在红黑树中插入新值

```
if (f instanceof TreeBin) {
    Node<K,V> p;
    binCount = 2;
    if ((p = ((TreeBin<K,V>)f).putTreeVal(hash, key,
                                                value)) != null) {
        oldVal = p.val;
        if (!onlyIfAbsent)
            p.val = value;
    }
}
```

如果在红黑树中存在于待插入键值对的Key相同（hash值相等并且equals方法判断为true）的节点的话，就覆盖旧值，否则就向红黑树追加新节点。

7.根据当前节点个数进行调整

当完成数据新节点插入之后，会进一步对当前链表大小进行调整，这部分代码为：

```

if (binCount != 0) {
    if (binCount >= TREEIFY_THRESHOLD)
        treeifyBin(tab, i);
    if (oldVal != null)
        return oldVal;
    break;
}

```

`treeifyBin`方法将`tab[i]`（第*i*个散列桶）拉链转换成红黑树。

## ForwardingNode节点类型

这个节点内部保存了一 `nextTable` 引用，它指向一张 `hash` 表。在扩容操作中，我们需要对每个桶中的结点进行分离和转移，如果某个桶结点中所有节点都已经迁移完成了（已经被转移到新表 `nextTable` 中了），那么会在原 `table` 表的该位置挂上一个 `ForwardingNode` 结点，说明此桶已经完成迁移

```

static final class ForwardingNode<K,V> extends Node<K,V> {
    final Node<K,V>[] nextTable;
    ForwardingNode(Node<K,V>[] tab) {
        //注意这里
        super(MOVED, null, null, null);
        this.nextTable = tab;
    }
    //省略其 find 方法
}

```

## helpTransfer方法

```

final Node<K,V>[] helpTransfer(Node<K,V>[] tab, Node<K,V> f) {
    Node<K,V>[] nextTab; int sc;
    if (tab != null && (f instanceof ForwardingNode) &&
        (nextTab = ((ForwardingNode<K,V>)f).nextTable) != null) {
        //返回一个 16 位长度的扩容校验标识
        int rs = resizeStamp(tab.length);
        while (nextTab == nextTable && table == tab &&
               (sc = sizeCtl) < 0) {
            //sizeCtl 如果处于扩容状态的话
            //前 16 位是数据校验标识, 后 16 位是当前正在扩容的线程总数
            //这里判断校验标识是否相等, 如果校验符不等或者扩容操作已经完成了, 直接退出循环, 不用协助它们扩容
            if (((sc >>> RESIZE_STAMP_SHIFT) != rs || sc == rs + 1 ||
                 sc == rs + MAX_RESIZERS || transferIndex <= 0)
                break;
            //否则调用 transfer 帮助它们进行扩容
            //sc + 1 标识增加了一个线程进行扩容
            if (U.compareAndSwapInt(this, SIZECTL, sc, sc + 1)) {
                transfer(tab, nextTab);
                break;
            }
        }
    }
}

```

```

        }
    }
    return nextTab;
}
return table;
}

```

## get方法（没有锁）

看完了put方法再来看get方法就很容易了，用逆向思维去看就好，这样存的话我反过来这么取就好了。  
get方法源码为：

```

public V get(Object key) {
    Node<K,V>[] tab; Node<K,V> e, p; int n, eh; K ek;
    // 1. 重hash
    int h = spread(key.hashCode());
    if ((tab = table) != null && (n = tab.length) > 0 &&
        (e = tabAt(tab, (n - 1) & h)) != null) {
        // 2. table[i]桶节点的key与查找的key相同，则直接返回
        if ((eh = e.hash) == h) {
            if ((ek = e.key) == key || (ek != null && key.equals(ek)))
                return e.val;
        }
        // 3. 当前节点hash小于0说明为树节点，在红黑树中查找即可
        else if (eh < 0)
            return (p = e.find(h, key)) != null ? p.val : null;
        while ((e = e.next) != null) {
            // 4. 从链表中查找，查找到则返回该节点的value，否则就返回null即可
            if (e.hash == h &&
                ((ek = e.key) == key || (ek != null && key.equals(ek))))
                return e.val;
        }
    }
    return null;
}

```

## transfer方法（扩容方法，且并未加锁）

这个方法的基本思想跟HashMap是很像的，但要复杂的多。原因是它支持多线程进行扩容操作，而并没有加锁。

我想这样做的目的不仅仅是为了满足concurrent的要求，而是希望利用并发处理去减少扩容带来的时间影响。

每个新参加进来扩容的线程必然先进while循环的最后一个判断条件中去领取自己需要迁移的桶的区间。然

后 i 指向区间的最后一个位置，表示迁移操作从后往前的做。

```
private final void transfer(Node<K,V>[ ] tab, Node<K,V>[ ] nextTab) {
    int n = tab.length, stride;
    //计算单个线程允许处理的最少table桶首节点个数，不能小于 16
    if ((stride = (NCPU > 1) ? (n >>> 3) / NCPU : n) < MIN_TRANSFER_STRIDE)
        stride = MIN_TRANSFER_STRIDE; // subdivide range
    //刚开始扩容，初始化 nextTab，容量为之前的两倍
    if (nextTab == null) {           // initiating
        try {
            @SuppressWarnings("unchecked")
            Node<K,V>[ ] nt = (Node<K,V>[ ])new Node<?,?>[n << 1];
            nextTab = nt;
        } catch (Throwable ex) {      // try to cope with OOME
            sizeCtl = Integer.MAX_VALUE;
            return;
        }
        nextTable = nextTab;
        //transferIndex 指向最后一个桶，方便从后向前遍历
        transferIndex = n;
    }
    int nextn = nextTab.length;
    //定义 ForwardingNode 用于标记迁移完成的桶
    ForwardingNode<K,V> fwd = new ForwardingNode<K,V>(nextTab);
    boolean advance = true;
    boolean finishing = false; // to ensure sweep before committing nextTab
    //i 指向当前桶，bound 指向当前线程需要处理的桶结点的区间下限
    for (int i = 0, bound = 0;;) {
        Node<K,V> f; int fh;
        //这个 while 循环的目的就是通过 --i 遍历当前线程所分配到的桶结点
        //一个桶一个桶的处理
        while (advance) {
            int nextIndex, nextBound;
            if (--i >= bound || finishing)
                advance = false;
            //transferIndex <= 0 说明已经没有需要迁移的桶了
            else if ((nextIndex = transferIndex) <= 0) {
                i = -1;
                advance = false;
            }
            //更新 transferIndex
            //为当前线程分配任务，处理的桶结点区间为 (nextBound,nextIndex)
            else if (U.compareAndSwapInt
                (this, TRANSFERINDEX, nextIndex,
                 nextBound = (nextIndex > stride ?
                             nextIndex - stride : 0))) {
                bound = nextBound;
                i = nextIndex - 1;
                advance = false;
            }
        }
    }
}
```

```

        }
    }

//当前线程所有任务完成
if (i < 0 || i >= n || i + n >= nextn) {
    int sc;
    if (finishing) {
        nextTable = null;
        table = nextTab;
        sizeCtl = (n << 1) - (n >>> 1);
        return;
    }
    if (U.compareAndSwapInt(this, SIZECTL, sc = sizeCtl, sc - 1)) {
        if ((sc - 2) != resizeStamp(n) << RESIZE_STAMP_SHIFT)
            return;
        finishing = advance = true;
        i = n; // recheck before commit
    }
}

//待迁移桶为空，那么在此位置 CAS 添加 ForwardingNode 结点标识该桶已经被处理过了
else if ((f = tabAt(tab, i)) == null)
    advance = casTabAt(tab, i, null, fwd);
//如果扫描到 ForwardingNode，说明此桶已经被处理过了，跳过即可
else if ((fh = f.hash) == MOVED)
    advance = true; // already processed
else {
    synchronized (f) {
        if (tabAt(tab, i) == f) {
            Node<K,V> ln, hn;
            //链表的迁移操作
            if (fh >= 0) {
                //4.3 处理当前节点为链表的头结点的情况，构造两个链表，一个是原链

```

表 另一个原链表的反序排列

```

                int runBit = fh & n;
                Node<K,V> lastRun = f;
                //整个 for 循环为了找到整个桶中最后连续的 fh & n 不变的结点
                for (Node<K,V> p = f.next; p != null; p = p.next) {
                    int b = p.hash & n;
                    if (b != runBit) {
                        runBit = b;
                        lastRun = p;
                    }
                }
                if (runBit == 0) {
                    ln = lastRun;
                    hn = null;
                }
                else {
                    hn = lastRun;
                    ln = null;

```

```

    }
    //如果fh&n不变的链表的runbit都是0，则nextTab[i]内元素ln前逆
    //序，ln及其之后顺序
    //否则，nextTab[i+n]内元素全部相对原table逆序
    //这是通过一个节点一个节点的往nextTab添加
    for (Node<K,V> p = f; p != lastRun; p = p.next) {
        int ph = p.hash; K pk = p.key; V pv = p.val;
        if ((ph & n) == 0)
            ln = new Node<K,V>(ph, pk, pv, ln);
        else
            hn = new Node<K,V>(ph, pk, pv, hn);
    }
    //把两条链表整体迁移到nextTab中
    setTabAt(nextTab, i, ln);
    setTabAt(nextTab, i + n, hn);
    //将原桶标识位已经处理
    setTabAt(tab, i, fwd);
    //设置advance为true 返回到上面的while循环中 就可以执行i--操作
    advance = true;
}
//红黑树的复制算法，不再赘述
else if (f instanceof TreeBin) {
    TreeBin<K,V> t = (TreeBin<K,V>)f;
    TreeNode<K,V> lo = null, loTail = null;
    TreeNode<K,V> hi = null, hiTail = null;
    int lc = 0, hc = 0;
    for (Node<K,V> e = t.first; e != null; e = e.next) {
        int h = e.hash;
        TreeNode<K,V> p = new TreeNode<K,V>
            (h, e.key, e.val, null, null);
        if ((h & n) == 0) {
            if ((p.prev = loTail) == null)
                lo = p;
            else
                loTail.next = p;
            loTail = p;
            ++lc;
        }
        else {
            if ((p.prev = hiTail) == null)
                hi = p;
            else
                hiTail.next = p;
            hiTail = p;
            ++hc;
        }
    }
    ln = (lc <= UNTREEIFY_THRESHOLD) ? untreeify(lo) :
        (hc != 0) ? new TreeBin<K,V>(lo) : t;
}

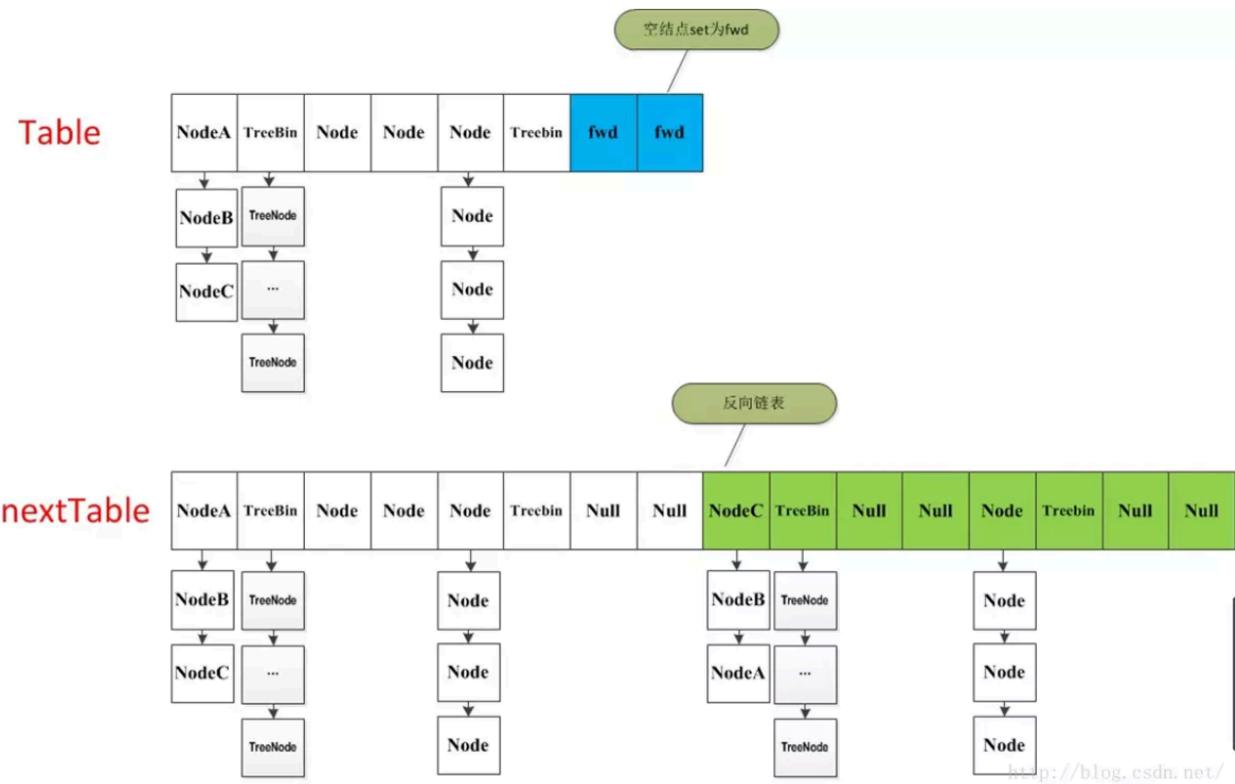
```

```
        hn = (hc <= UNTREEIFY_THRESHOLD) ? untreeify(hi) :  
              (lc != 0) ? new TreeBin<K,V>(hi) : t;  
        setTabAt(nextTab, i, ln);  
        setTabAt(nextTab, i + n, hn);  
        setTabAt(tab, i, fwd);  
        advance = true;  
    }  
}  
}  
}  
}
```

第一部分是构建一个nextTable,它的容量是原来的两倍,这个操作是单线程完成的。新建table数组的代码为: `Node<K,V>[ ] nt = (Node<K,V>[ ])new Node<?,?>[n << 1]`,在原容量大小的基础上右移一位。

第二个部分就是将原来table中的元素复制到nextTable中，主要是遍历复制的过程。根据运算得到当前遍历的数组的位置i，然后利用tabAt方法获得i位置的元素再进行判断：

1. 如果这个位置为空，就在原table中的i位置放入forwardNode节点，这个也是触发并发扩容的关键点；
  2. 如果这个位置是Node节点 ( $fh \geq 0$ )，如果它是一个链表的头节点，就构造一个反序链表，把它们分别放在nextTable的i和*i+n*的位置上
  3. 如果这个位置是TreeBin节点 ( $fh < 0$ )，也做一个反序处理，并且判断是否需要untreefi，把处理的结果分别放在nextTable的i和*i+n*的位置上
  4. 遍历过所有的节点以后就完成了复制工作，这时让nextTable作为新的table，并且更新sizeCtl为新容量的0.75倍，完成扩容。设置为新容量的0.75倍代码为 `sizeCtl = (n << 1) - (n >>> 1)`，仔细体会下是不是很巧妙， $n << 1$ 相当于n右移一位表示n的两倍即 $2n$ , $n >>> 1$ 左右一位相当于n除以2即 $0.5n$ ,然后两者相减为 $2n - 0.5n = 1.5n$ ,是不是刚好等于新容量的0.75倍即 $2n * 0.75 = 1.5n$ 。最后用一个示意图来进行总结（图片摘自网络）：



## remove方法

先定位再删除的复合。

首先遍历整张表的桶结点，如果表还未初始化或者无法根据参数的 hash 值定位到桶结点，那么将返回 null。

如果定位到的桶结点类型是 ForwardingNode 结点，调用 **helpTransfer** 协助扩容。

否则就老老实实的给桶加锁，删除一个节点。

最后会调用 **addCount** 方法 CAS 更新 **baseCount** 的值。

## 与Size相关函数

对于 ConcurrentHashMap 来说，这个 table 里到底装了多少东西其实是个不确定的数量，因为不可能在调用 **size()** 方法的时候像 GC 的“stop the world”一样让其他线程都停下来让你去统计，因此只能说这个数量是个估计值。对于这个估计值，ConcurrentHashMap 也是大费周章才计算出来的。

为了统计元素个数，ConcurrentHashMap 定义了一些变量和一个内部类

```
/**
 * A padded cell for distributing counts. Adapted from LongAdder
 * and Striped64. See their internal docs for explanation.
 */
@sun.misc.Contended static final class CounterCell {
    volatile long value;
    CounterCell(long x) { value = x; }
}
/*****************/
/**
```

```

* 实际上保存的是hashmap中的元素个数 利用CAS锁进行更新
* 但它并不用返回当前hashmap的元素个数
*/
private transient volatile long baseCount;
/**
 * Spinlock (locked via CAS) used when resizing and/or creating CounterCells.
*/
private transient volatile int cellsBusy;
/**
 * Table of counter cells. When non-null, size is a power of 2.
*/
private transient volatile CounterCell[] counterCells;

```

## mappingCount与size方法

**mappingCount**与**size**方法的类似 从给出的注释来看，应该使用mappingCount代替size方法 两个方法都没有直接返回basecount 而是统计一次这个值，而这个值其实也是一个大概的数值，因此可能在统计的时候有其他线程正在执行插入或删除操作。

```

public int size() {
    long n = sumCount();
    return ((n < 0L) ? 0 :
        (n > (long)Integer.MAX_VALUE) ? Integer.MAX_VALUE :
        (int)n);
}

/**
 * Returns the number of mappings. This method should be used
 * instead of {@link #size} because a ConcurrentHashMap may
 * contain more mappings than can be represented as an int. The
 * value returned is an estimate; the actual count may differ if
 * there are concurrent insertions or removals.
 *
 * @return the number of mappings
 * @since 1.8
 */

public long mappingCount() {
    long n = sumCount();
    return (n < 0L) ? 0L : n; // ignore transient negative values
}

final long sumCount() {
    CounterCell[] as = counterCells; CounterCell a;
    long sum = baseCount;
    if (as != null) {
        for (int i = 0; i < as.length; ++i) {
            if ((a = as[i]) != null)
                sum += a.value;//所有counter的值求和
        }
    }
}

```

```
    return sum;
}
```

## addCount方法

在put和remove方法结尾处都调用了addCount方法，把当前ConcurrentHashMap的元素个数+1这个方法一共做了两件事

- 更新baseCount的值,
- 检测是否进行扩容。

```
private final void addCount(long x, int check) {
    CounterCell[] as; long b, s;
    //如果更新失败才会进入的 if 的主体代码中
    //s = b + x 其中 x 等于 1
    //利用CAS方法更新baseCount的值
    if ((as = counterCells) != null ||
        !U.compareAndSwapLong(this, BASECOUNT, b = baseCount, s = b + x)) {
        CounterCell a; long v; int m;
        boolean uncontended = true;
        //高并发下 CAS 失败会执行 fullAddCount 方法
        if (as == null || (m = as.length - 1) < 0 ||
            (a = as[ThreadLocalRandom.getProbe() & m]) == null ||
            !(uncontended =
                U.compareAndSwapLong(a, CELLVALUE, v = a.value, v + x))) {
            fullAddCount(x, uncontended);
            return;
        }
        if (check <= 1)
            return;
        s = sumCount();
    }
    //如果check值大于等于0 则需要检验是否需要进行扩容操作
    if (check >= 0) {
        Node<K,V>[] tab, nt; int n, sc;
        while (s >= (long)(sc = sizeCtl) && (tab = table) != null &&
               (n = tab.length) < MAXIMUM_CAPACITY) {
            int rs = resizeStamp(n);
            //
            if (sc < 0) {
                if (((sc >>> RESIZE_STAMP_SHIFT) != rs || sc == rs + 1 ||
                    sc == rs + MAX_RESIZERS || (nt = nextTable) == null ||
                    transferIndex <= 0)
                    break;
            }
            //如果已经有其他线程在执行扩容操作
            if (U.compareAndSwapInt(this, SIZECTL, sc, sc + 1))
                transfer(tab, nt);
        }
        //当前线程是唯一的或是第一个发起扩容的线程 此时nextTable=null
    }
}
```

```

        else if (U.compareAndSwapInt(this, SIZECTL, sc,
                                      (rs << RESIZE_STAMP_SHIFT) + 2))
            transfer(tab, null);
        s = sumCount();
    }
}
}

```

**ConcurrentHashMap 中的 baseCount 属性不就是记录的所有键值对的总数吗？直接返回它不就行了吗？**

addCount 方法用于 CAS 更新 baseCount，但很有可能在高并发的情况下，更新失败，那么这些节点虽然已经被添加到哈希表中了，但是数量却没有被统计。

还好，addCount 方法在更新 baseCount 失败的时候，会调用 fullAddCount 将这些失败的结点包装成一个 CounterCell 对象，保存在 CounterCell 数组中。那么整张表实际的 size 其实是 baseCount 加上 CounterCell 数组中元素的个数。

## ConcurrentHashMap的版本变化

JDK6,7中的ConcurrentHashMap主要使用Segment来实现减小锁粒度，分割成若干个Segment，在put的时候需要锁住Segment，get时候不加锁，使用volatile来保证可见性，当要统计全局时（比如size），首先会尝试多次计算modcount来确定，这几次尝试中，是否有其他线程进行了修改操作，如果没有，则直接返回size。如果有，则需要依次锁住所有的Segment来计算。

1.8之前put定位节点时要先定位到具体的segment，然后再在segment中定位到具体的桶。而在1.8的时候摒弃了segment臃肿的设计，直接针对的是Node[] table数组中的每一个桶，进一步减小了锁粒度。并且防止拉链过长导致性能下降，当链表长度大于8的时候采用红黑树的设计。

主要设计上的变化有以下几点：

1. 不采用segment而采用node，锁住node来实现减小锁粒度。
2. 设计了MOVED状态 当resize的中过程中 线程2还在put数据，线程2会帮助resize。
3. 使用3个CAS操作来确保node的一些操作的原子性，这种方式代替了锁。
4. sizeCtl的不同值来代表不同含义，起到了控制的作用。
5. 采用synchronized而不是ReentrantLock

## CopyOnWriteArrayList

### COW思想（牺牲实时性，保证最终一致性）

CopyOnWriteArrayList就是通过Copy-On-Write(COW)，即写时复制的思想来通过延时更新的策略来实现数据的最终一致性，并且能够保证读线程间不阻塞。

### COW vs 读写锁（ReentrantReadWriteLock）

相同点：1. 两者都是通过读写分离的思想实现；2. 读线程间是互不阻塞的

不同点：

读写锁：对读线程而言，为了实现数据实时性，在写锁被获取后，读线程会等待或者当读锁被获取后，写线程会等待，从而解决“脏读”等问题。也就是说如果使用读写锁依然会出现读线程阻塞等待的情况。

**COW**: 完全放开了牺牲数据实时性而保证数据最终一致性，即读线程对数据的更新是延时感知的，因此读线程不会存在等待的情况。

## 缺点

- 内存占用
- 数据实时一致性问题

## 线程安全的队列：阻塞队列 和 非阻塞队列

注：阻塞队列和非阻塞队列如何实现线程安全？

- 阻塞队列可以用一个锁（入队和出队共享一把锁）或者两个锁（入队使用一把锁，出队使用一把锁）来实现线程安全，JDK中典型的实现是 `BlockingQueue`；
- 非阻塞队列可以用循环CAS的方式来保证数据的一致性，来达到线程安全的目的。

## ConcurrentLinkedQueue (线程安全队列，非阻塞队列)

```
private static class Node<E> {
    volatile E item;
    volatile Node<E> next;
    .....
}
```

Node节点主要包含了两个域：一个是数据域item，另一个是next指针，用于指向下一个节点从而构成链式队列。

### 操作Node 的几个CAS操作 (来自Unsafe类，该类是hotspot底层方法)

```
//更改Node中的数据域item
boolean casItem(E cmp, E val) {
    return UNSAFE.compareAndSwapObject(this, itemOffset, cmp, val);
}

//更改Node中的指针域next
void lazySetNext(Node<E> val) {
    UNSAFE.putOrderedObject(this, nextOffset, val);
}

//更改Node中的指针域next
boolean casNext(Node<E> cmp, Node<E> val) {
    return UNSAFE.compareAndSwapObject(this, nextOffset, cmp, val);
}
```

## 入队

tail节点并不一定是指向队列的最后一个节点，它可能指向最后一个节点的前一个节点！！！

假如我们每次就让tail节点作为队尾节点，每次的入队所要做的事情其实就是将入队节点设置成尾节点，代码量确实非常少，而且逻辑非常清楚和易懂，但是这样做有个缺点就是每次入队都需要循环CAS更新tail节点。

Doug Lea并没有让tail节点作为队尾节点，只有**tail节点与队尾节点之间的距离等于1**的时候才需要更新tail节点。但是，这样就可能导致当队列长度越长的时候每次入队定位尾节点的时间就会越长，即便是这样，它仍然可以提高入队效率，因为从本质上来看，volatile变量的写操作的开销要远远大于读操作的。

## 出队

**head节点并不一定是指向队列的第一个有效节点，它可能指向有效节点的前一个节点！！！**

注：这里的有效节点是指从head节点向后遍历可达的节点当中，item不为null的节点。

当然，为什么head节点不总是指向队列的第一个有效节点，其原因跟入队是一样的，这么做的最主要也是减少**CAS更新head节点的次数**，从而提高出队效率。

## BlockingQueue（接口，线程安全阻塞队列）

阻塞队列（BlockingQueue）被广泛使用在“生产者-消费者”问题中，其原因是BlockingQueue提供了可阻塞的插入和移除的方法。**当队列容器已满，生产者线程会被阻塞，直到队列未满；当队列容器为空时，消费者线程会被阻塞，直至队列非空时为止。**

BlockingQueue继承于Queue接口，因此，对数据元素的基本操作有：

### 插入元素

1. `add(E e)`：往队列插入数据，当队列满时，插入元素时会抛出`IllegalStateException`异常；
2. `offer(E e)`：当往队列插入数据时，插入成功返回`true`，否则则返回`false`。当队列满时不会抛出异常；

### 删除元素

1. `remove(Object o)`：从队列中删除数据，成功则返回`true`，否则为`false`
2. `poll()`：删除数据，当队列为空时，返回`null`；

### 查看元素

1. `element()`：获取队头元素，如果队列为空时则抛出`NoSuchElementException`异常；
2. `peek()`：获取队头元素，如果队列为空则抛出`NoSuchElementException`异常

BlockingQueue具有的特殊操作：

### 插入数据：

1. `put`：当阻塞队列容量已经满时，往阻塞队列插入数据的线程**会被阻塞**，直至阻塞队列已经有空余的容量可供使用；
2. `offer(E e, long timeout, TimeUnit unit)`：若阻塞队列已经满时，同样会阻塞插入数据的线程，直至阻塞队列已经有空余的地方，与`put`方法不同的是，该方法会有一个超时时间，若超过当前给定的超时时间，插入数据的线程会退出；

### 删除数据

1. **take()**: 当阻塞队列为空时，获取队头数据的线程会被阻塞；
2. **poll(long timeout, TimeUnit unit)**: 当阻塞队列为空时，获取数据的线程会被阻塞，另外，如果被阻塞的线程超过了给定的时长，该线程会退出

## 常用的BlockingQueue

### 1.ArrayBlockingQueue

**ArrayBlockingQueue**是由数组实现的**有界阻塞队列**。该队列命令元素**FIFO**（先进先出）。因此，对头元素时队列中存在时间最长的数据元素，而对尾数据则是当前队列最新的数据元素。

ArrayBlockingQueue可作为“有界数据缓冲区”，生产者插入数据到队列容器中，并由消费者提取。

ArrayBlockingQueue一旦创建，容量不能改变。

当队列容量满时，尝试将元素放入队列将导致操作阻塞；尝试从一个空队列中取一个元素也会同样阻塞。

ArrayBlockingQueue默认情况下不能保证线程访问队列的公平性，所谓公平性是指严格按照线程等待的绝对时间顺序，即最先等待的线程能够最先访问到ArrayBlockingQueue。而非公平性则是指访问ArrayBlockingQueue的顺序不是遵守严格的时间顺序，有可能存在，一旦ArrayBlockingQueue可以被访问时，长时间阻塞的线程依然无法访问到ArrayBlockingQueue。**如果保证公平性（ReentrantLock实现），通常会降低吞吐量**。如果需要获得公平性的ArrayBlockingQueue，可采用如下代码：

```
private static ArrayBlockingQueue<Integer> blockingQueue = new  
ArrayBlockingQueue<Integer>(10, true);
```

### 2.LinkedBlockingQueue

LinkedBlockingQueue是用链表实现的**有界阻塞队列**，同样满足FIFO的特性，与ArrayBlockingQueue相比起来具有更高的**吞吐量**，为了防止LinkedBlockingQueue容量迅速增，损耗大量内存。通常在创建LinkedBlockingQueue对象时，会指定其大小，如果未指定，容量等于Integer.MAX\_VALUE

### 3.PriorityBlockingQueue

PriorityBlockingQueue是一个支持优先级的**无界阻塞队列**。默认情况下元素采用自然顺序进行排序，也可以通过自定义类实现compareTo()方法来指定元素排序规则，或者初始化时通过构造器参数Comparator来指定排序规则。

### 4.SynchronousQueue

SynchronousQueue每个插入操作必须等待另一个线程进行相应的删除操作，因此，SynchronousQueue实际上没有存储任何数据元素，因为只有线程在删除数据时，其他线程才能插入数据，同样的，如果当前有线程在插入数据时，线程才能删除数据。SynchronousQueue也可以通过构造器参数来为其指定公平性。

### 5.LinkedTransferQueue

LinkedTransferQueue是一个由链表数据结构构成的**无界阻塞队列**，由于该队列实现了TransferQueue接口，与其他阻塞队列相比主要有以下不同的方法：

**transfer(E e)** 如果当前有线程（消费者）正在调用take()方法或者可延时的poll()方法进行消费数据时，生产者线程可以调用**transfer**方法将数据传递给消费者线程。如果当前没有消费者线程的话，生产者线程就会将数据插入到队尾，直到有消费者能够进行消费才能退出；

**tryTransfer(E e)** tryTransfer方法如果当前有消费者线程（调用take方法或者具有超时特性的poll方法）正在消费数据的话，该方法可以将数据立即传送给消费者线程，如果当前没有消费者线程消费数据的话，就立即返回 `false`。因此，与transfer方法相比，transfer方法是必须等到有消费者线程消费数据时，生产者线程才能够返回。而tryTransfer方法能够立即返回结果退出。

**tryTransfer(E e,long timeout,imeUnit unit)** 与transfer基本功能一样，只是增加了超时特性，如果数据在规定的超时时间内没有消费者进行消费的话，就返回 `false`。

## 6.LinkedBlockingDeque

LinkedBlockingDeque是基于链表数据结构的**有界阻塞双端队列**，如果在创建对象时未指定大小时，其默认大小为Integer.MAX\_VALUE。与LinkedBlockingQueue相比，主要的不同点在于，LinkedBlockingDeque具有双端队列的特性。

## 7.DelayQueue（配合ScheduledThreadPoolExecutor）

DelayQueue是一个存放实现Delayed接口的数据的**无界阻塞队列**，只有当数据对象的延时时间达到时才能插入到队列进行存储。如果当前所有的数据都还没有达到创建时所指定的延时期，则队列没有队头，并且线程通过poll等方法获取数据元素则返回null。所谓数据延时期满时，则是通过Delayed接口的 `getDelay(TimeUnit.NANOSECONDS)` 来进行判定，如果该方法返回的是小于等于0则说明该数据元素的延时期已满。

# 线程池（Executor体系）

## 线程池实现原理（ThreadPoolExecutor）

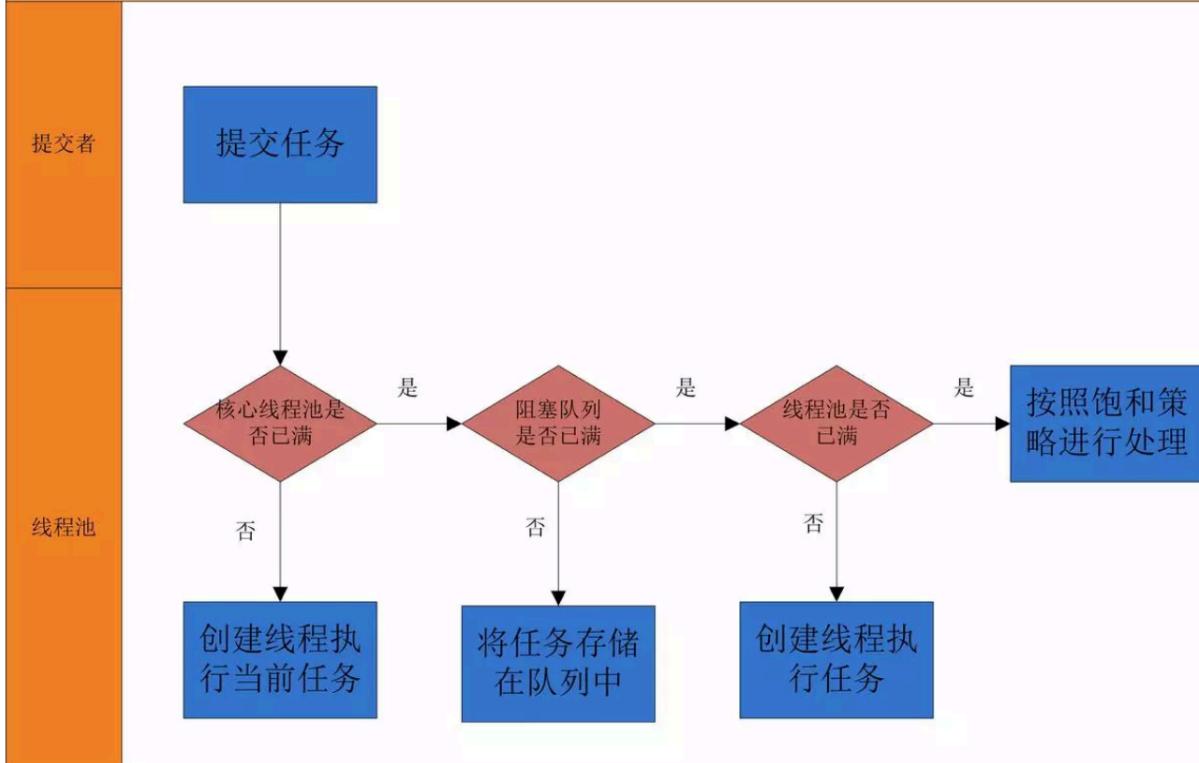
### 为什么要使用线程池

在实际使用中，线程是很占用系统资源的，如果对线程管理不善很容易导致系统问题。因此，在大多数并发框架中都会使用线程池来管理线程，使用线程池管理线程主要有如下好处：

1. **降低资源消耗**。通过复用已存在的线程和降低线程关闭的次数来尽可能降低系统性能损耗；
2. **提升系统响应速度**。通过复用线程，省去创建线程的过程，因此整体上提升了系统的响应速度；
3. **提高线程的可管理性**。线程是稀缺资源，如果无限制的创建，不仅会消耗系统资源，还会降低系统的稳定性，因此，需要使用线程池来管理线程。

### 工作原理

## 线程池执行过程



先判断线程池中核心线程池所有的线程是否都在执行任务。如果不是，则新创建一个线程执行刚提交的任务，否则，核心线程池中所有的线程都在执行任务，则进入第2步；

判断当前阻塞队列是否已满，如果未满，则将提交的任务放置在阻塞队列中；否则，则进入第3步；

判断线程池中所有的线程是否都在执行任务，如果没有，则创建一个新的线程来执行任务，否则，则交给饱和策略进行处理

### 线程池的创建

创建线程池主要是**ThreadPoolExecutor**类来完成，**ThreadPoolExecutor**的有许多重载的构造方法，通过参数最多的构造方法来理解创建线程池有哪些需要配置的参数。**ThreadPoolExecutor**的构造方法为：

```
ThreadPoolExecutor(int corePoolSize,
                    int maximumPoolSize,
                    long keepAliveTime,
                    TimeUnit unit,
                    BlockingQueue<Runnable> workQueue,
                    ThreadFactory threadFactory,
                    RejectedExecutionHandler handler)
```

**corePoolSize**：表示核心线程池的大小。当提交一个任务时，如果当前核心线程池的线程个数没有达到**corePoolSize**，则会创建新的线程来执行所提交的任务，即使当前核心线程池有空闲的线程。如果当前核心线程池的线程个数已经达到了**corePoolSize**，则不再重新创建线程。如果调用了**prestartCoreThread()** 或者 **prestartAllCoreThreads()**，线程池创建的时候所有的核心线程都

会被创建并且启动。

`maximumPoolSize`：表示线程池能创建线程的最大个数。如果当阻塞队列已满时，并且当前线程池线程个数没有超过`maximumPoolSize`的话，就会创建新的线程来执行任务。

`keepAliveTime`：空闲线程存活时间。如果当前线程池的线程个数已经超过了`corePoolSize`，并且线程空闲时间超过了`keepAliveTime`的话，就会将这些空闲线程销毁，这样可以尽可能降低系统资源消耗。

`unit`：时间单位。为`keepAliveTime`指定时间单位。

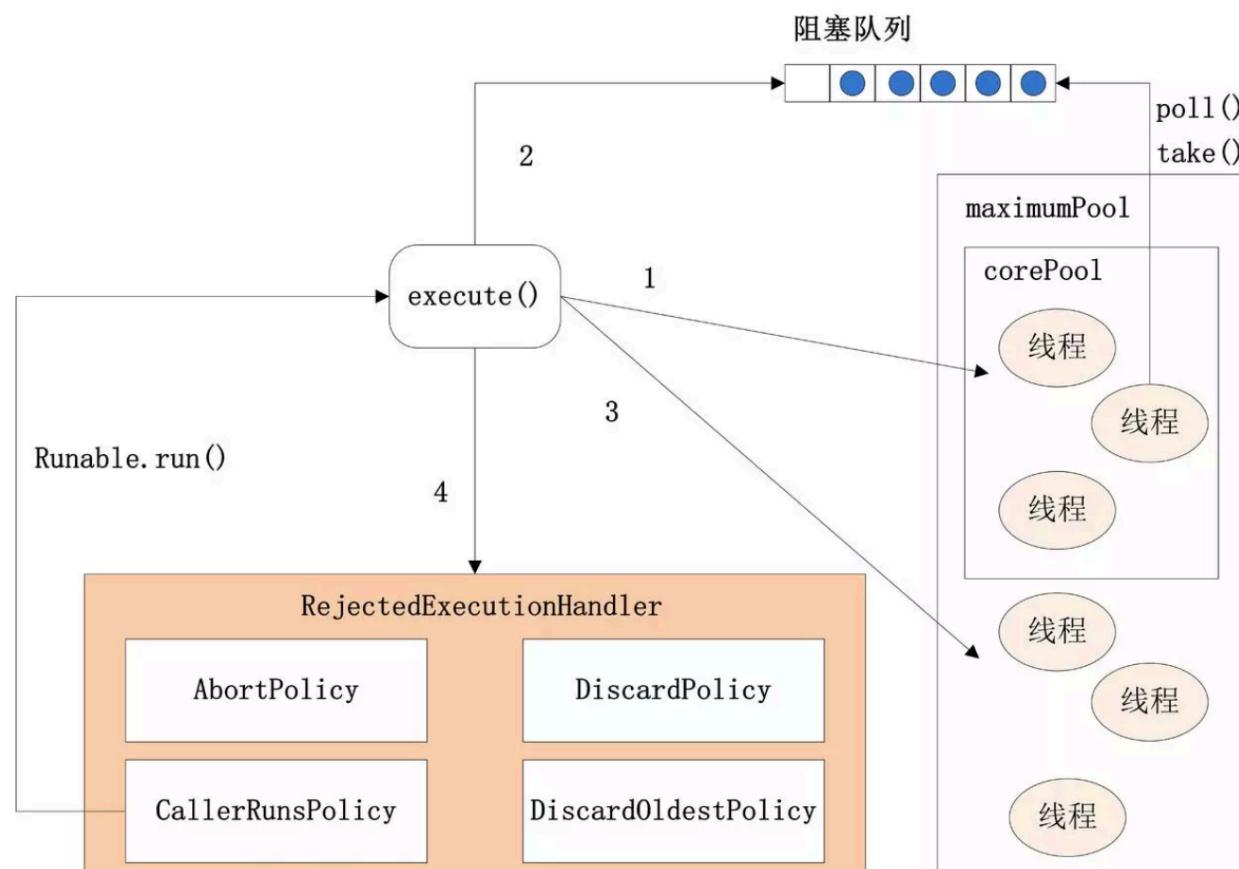
`workQueue`：阻塞队列。用于保存任务的阻塞队列。可以使用`ArrayBlockingQueue`, `LinkedBlockingQueue`, `SynchronousQueue`, `PriorityBlockingQueue`。

`threadFactory`：创建线程的工程类。可以通过指定线程工厂为每个创建出来的线程设置更有意义的名字，如果出现并发问题，也方便查找问题原因。

`handler`：饱和策略。当线程池的阻塞队列已满和指定的线程都已经开启，说明当前线程池已经处于饱和状态了，那么就需要采用一种策略来处理这种情况。采用的策略有这几种：

1. `AbortPolicy`: 直接拒绝所提交的任务，并抛出`RejectedExecutionException`异常；
2. `CallerRunsPolicy`: 只用调用者所在的线程来执行任务；
3. `DiscardPolicy`: 不处理直接丢弃掉任务；
4. `DiscardOldestPolicy`: 丢弃掉阻塞队列中存放时间最久的任务，执行当前任务

## 线程池执行逻辑



`execute`方法执行逻辑有这样几种情况：

1. 如果当前运行的线程少于corePoolSize，则会创建新的线程来执行新的任务；
2. 如果运行的线程个数等于或者大于corePoolSize，则会将提交的任务存放到阻塞队列workQueue中；
3. 如果当前workQueue队列已满的话，则会创建新的线程来执行任务；
4. 如果线程个数已经超过了maximumPoolSize，则会使用饱和策略RejectedExecutionHandler来进行处理。

需要注意的是，线程池的设计思想就是使用了核心线程池**corePoolSize**，阻塞队列**workQueue**和线程池**maximumPoolSize**，这样的缓存策略来处理任务，实际上这样的设计思想在需要框架中都会使用。

## 线程池的关闭

关闭线程池，可以通过 `shutdown` 和 `shutdownNow` 这两个方法。它们的原理都是遍历线程池中所有的线程，然后依次中断线程。`shutdown` 和 `shutdownNow` 还是有不一样的地方：

1. `shutdownNow` 首先将线程池的状态设置为**STOP**，然后尝试停止所有的正在执行和未执行任务的线程，并返回等待执行任务的列表；
2. `shutdown` 只是将线程池的状态设置为**SHUTDOWN**状态，然后中断所有没有正在执行任务的线程

可以看出`shutdown`方法会将正在执行的任务继续执行完，而`shutdownNow`会直接中断正在执行的任务。调用了这两个方法的任意一个，`isShutdown` 方法都会返回true，当所有的线程都关闭成功，才表示线程池成功关闭，这时调用 `isTerminated` 方法才会返回true。

## 如何合理配置线程池参数

要想合理的配置线程池，就必须首先分析任务特性，可以从以下几个角度来进行分析：

1. 任务的性质：CPU密集型任务，IO密集型任务和混合型任务。
2. 任务的优先级：高，中和低。
3. 任务的执行时间：长，中和短。
4. 任务的依赖性：是否依赖其他系统资源，如数据库连接。

**任务性质不同：**的任务可以用不同规模的线程池分开处理。CPU密集型任务配置尽可能少的线程数量，如配置**Ncpu+1**个线程的线程池。IO密集型任务则由于需要等待IO操作，线程并不是一直在执行任务，则配置尽可能多的线程，如**2xNcpu**。混合型的任务，如果可以拆分，则将其拆分成一个CPU密集型任务和一个IO密集型任务，只要这两个任务执行的时间相差不是太大，那么分解后执行的吞吐率要高于串行执行的吞吐率，如果这两个任务执行时间相差太大，则没必要进行分解。我们可以通过

`Runtime.getRuntime().availableProcessors()` 方法获得当前设备的**CPU个数**。

**优先级不同的任务：**可以使用优先级队列**PriorityBlockingQueue**来处理。它可以让优先级高的任务先得到执行，需要注意的是如果一直有优先级高的任务提交到队列里，那么优先级低的任务可能永远不能执行。

**执行时间不同的任务：**可以交给不同规模的线程池来处理，或者也可以使用优先级队列，让执行时间短的任务先执行。

**依赖数据库连接池的任务：**，因为线程提交SQL后需要等待数据库返回结果，如果等待的时间越长CPU空闲时间就越长，那么线程数应该设置越大，这样才能更好的利用CPU。

并且，阻塞队列最好是使用有界队列，如果采用无界队列的话，一旦任务积压在阻塞队列中的话就会占用过多的内存资源，甚至会使得系统崩溃。

## ScheduledThreadPoolExecutor (延时执行、周期执行多个任务，比Timer强大)

可以用来在给定延时后执行异步任务或者周期性执行任务，相对于任务调度的Timer来说，其功能更加强大，Timer只能使用一个后台线程执行任务，而ScheduledThreadPoolExecutor则可以通过构造函数来指定后台线程的个数。

1. ScheduledThreadPoolExecutor继承了 ThreadPoolExecutor，也就是说 ScheduledThreadPoolExecutor拥有execute()和submit()提交异步任务的基础功能，但是， ScheduledThreadPoolExecutor类实现了 ScheduledExecutorService，该接口定义了 ScheduledThreadPoolExecutor能够延时执行任务和周期执行任务的功能；
2. ScheduledThreadPoolExecutor也两个重要的内部类： DelayedWorkQueue 和 ScheduledFutureTask。可以看出DelayedWorkQueue实现了BlockingQueue接口，也就是一个阻塞队列， ScheduledFutureTask则是继承了 FutureTask类，也表示该类用于返回异步任务的结果。这两个关键类，下面会具体详细来看。

### ScheduledFutureTask

ScheduledThreadPoolExecutor实现了 ScheduledExecutorService 接口，该接口定义了可延时执行异步任务和可周期执行异步任务的特有功能，相应的方法分别为：

```
//达到给定的延时时间后，执行任务。这里传入的是实现Runnable接口的任务，  
//因此通过ScheduledFuture.get()获取结果为null  
public ScheduledFuture<?> schedule(Runnable command,  
                                     long delay, TimeUnit unit);  
  
//达到给定的延时时间后，执行任务。这里传入的是实现Callable接口的任务，  
//因此，返回的是任务的最终计算结果  
public <V> ScheduledFuture<V> schedule(Callable<V> callable,  
                                         long delay, TimeUnit unit);  
  
//是以上一个任务开始的时间计时，period时间过去后，  
//检测上一个任务是否执行完毕，如果上一个任务执行完毕，  
//则当前任务立即执行，如果上一个任务没有执行完毕，则需要等上一个任务执行完毕后立即执行  
public ScheduledFuture<?> scheduleAtFixedRate(Runnable command,  
                                              long initialDelay,  
                                              long period,  
                                              TimeUnit unit);  
  
//当达到延时时间initialDelay后，任务开始执行。上一个任务执行结束后到下一次  
//任务执行，中间延时时间间隔为delay。以这种方式，周期性执行任务。  
public ScheduledFuture<?> scheduleWithFixedDelay(Runnable command,  
                                                long initialDelay,  
                                                long delay,  
                                                TimeUnit unit);
```

当调用 schedule, scheduleAtFixedRate 和 scheduleWithFixedDelay 方法时，实际上是将提交的任务转换成的 ScheduledFutureTask 类

**ScheduledFutureTask**最主要的功能是根据当前任务是否具有周期性，对异步任务进行进一步封装。如果不是周期性任务（调用**schedule**方法）则直接通过**run()**执行，若是周期性任务，则需要在每一次执行完后，重设下一次执行的时间，然后将下一次任务继续放入到阻塞队列中。

## DelayedWorkQueue

**DelayedWorkQueue**是一个基于堆的数据结构（优先级队列）。类似于**DelayQueue**和**PriorityQueue**。**DelayedWorkQueue**的工作就是按照执行时间的升序来排列，执行时间距离当前时间越近的任务在队列的前面。

为什么要使用DelayedWorkQueue呢？

定时任务执行时需要取出最近要执行的任务

**DelayedWorkQueue**是基于堆的数据结构，按照时间顺序将每个任务进行排序，将待执行时间越近的任务放在在队列的队头位置，以便于最先进行执行。

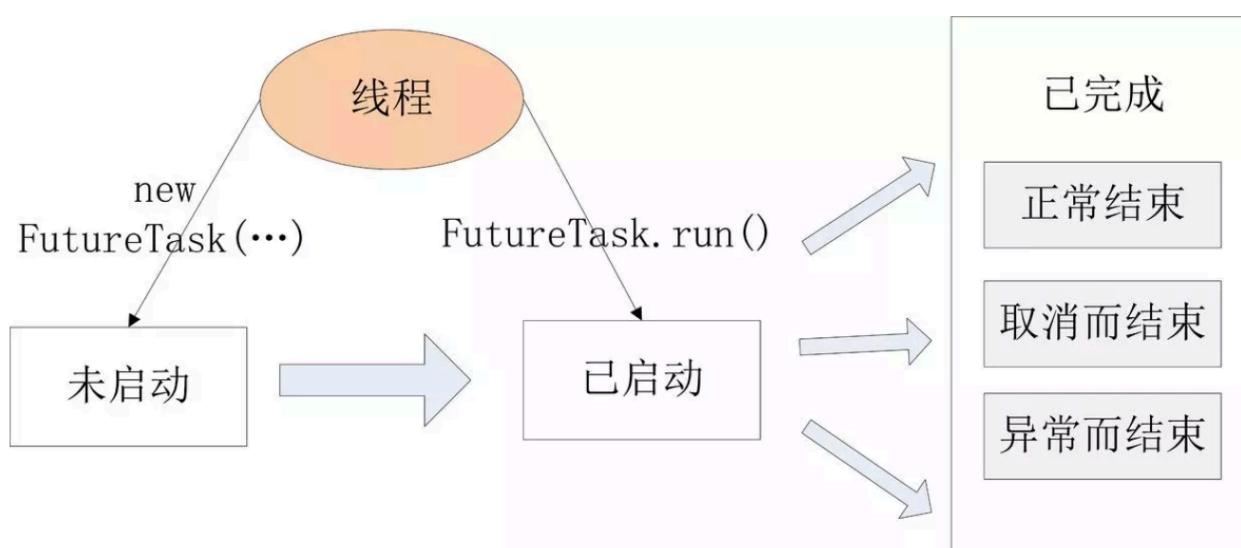
## FutureTask（可获取结果的异步任务）

FutureTask用来表示可获取结果的异步任务。FutureTask实现了Future接口，FutureTask提供了启动和取消异步任务，查询异步任务是否计算结束以及获取最终的异步任务的结果的一些常用的方法。通过**get()**方法来获取异步任务的结果，但是会阻塞当前线程直至异步任务执行结束。一旦任务执行结束，任务不能重新启动或取消，除非调用**runAndReset()**方法。

未启动。FutureTask.run()方法还没有被执行之前，FutureTask处于未启动状态。当创建一个FutureTask，还没有执行FutureTask.run()方法之前，FutureTask处于未启动状态。

已启动。FutureTask.run()方法被执行的过程中，FutureTask处于已启动状态。

已完成。FutureTask.run()方法执行结束，或者调用FutureTask.cancel(...)方法取消任务，或者在执行任务期间抛出异常，这些情况都称之为FutureTask的已完成状态。



由于FutureTask具有这三种状态，因此执行FutureTask的get方法和cancel方法，当前处于不同的状态对应的结果也是大不相同。这里对get方法和cancel方法做个总结：

get方法

当FutureTask处于未启动或已启动状态时，执行FutureTask.get()方法将导致调用线程阻塞。如果FutureTask处于已完成状态，调用FutureTask.get()方法将导致调用线程立即返回结果或者抛出异常

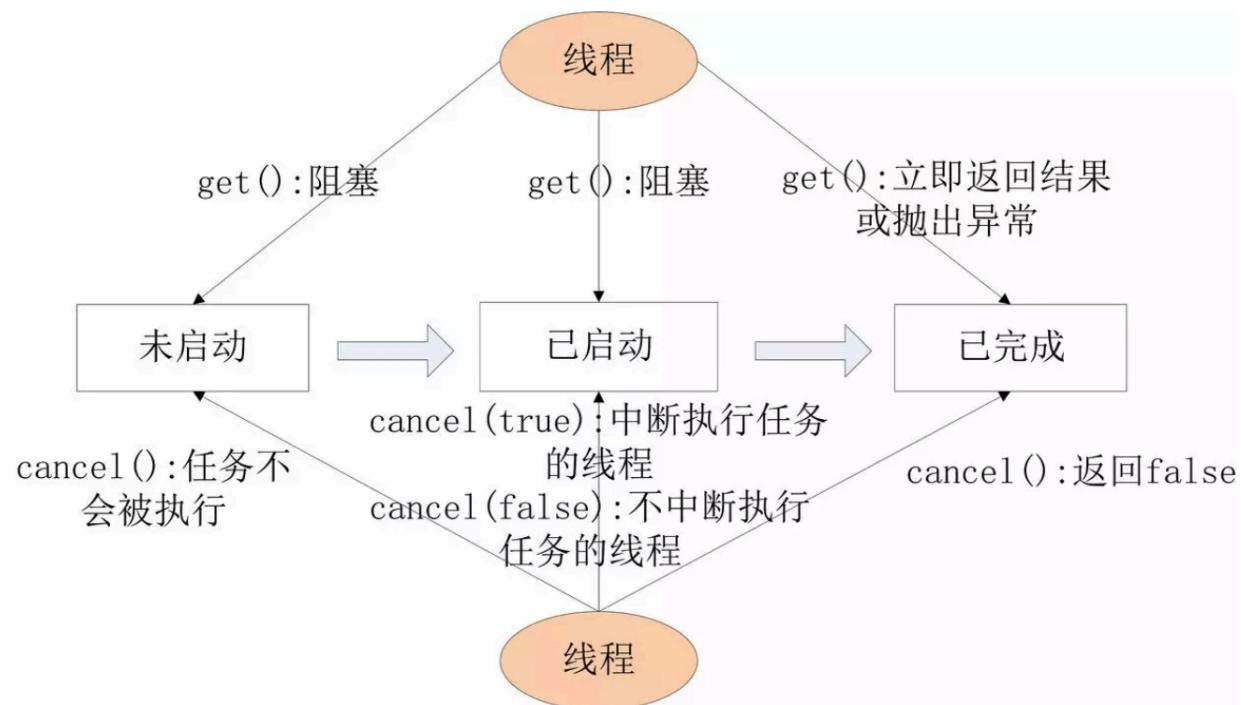
#### cancel方法

当FutureTask处于未启动状态时，执行FutureTask.cancel()方法将此任务永远不会执行；

当FutureTask处于已启动状态时，执行FutureTask.cancel(true)方法将以中断线程的方式来阻止任务继续进行，如果执行FutureTask.cancel(false)将不会对正在执行任务的线程有任何影响；

当**FutureTask**处于已完成状态时，执行FutureTask.cancel(...)方法将返回false。

对Future的get()方法和cancel()方法用下图进行总结



## Future基本使用

FutureTask除了实现Future接口外，还实现了**Runnable**接口。因此，FutureTask可以交给Executor执行，也可以由调用的线程直接执行（FutureTask.run()）。另外，FutureTask的获取也可以通过ExecutorService.submit()方法返回一个FutureTask对象，然后在通过FutureTask.get()或者FutureTask.cancel方法。

**应用场景：**当一个线程需要等待另一个线程把某个任务执行完后它才能继续执行，此时可以使用FutureTask。假设有两个线程执行若干任务，每个任务最多只能被执行一次。当多个线程试图执行同一个任务时，只允许一个线程执行任务，其他线程需要等待这个任务执行完后才能继续执行。

## 原子操作类

### 原子更新基本类型

atomic包提高原子更新基本类型的工具类，主要有这些：

1. AtomicBoolean：以原子更新的方式更新boolean；
2. AtomicInteger：以原子更新的方式更新Integer；
3. AtomicLong：以原子更新的方式更新Long；

这几个类的用法基本一致，这里以AtomicInteger为例总结常用的方法

1. addAndGet(int delta)：以原子方式将输入的数值与实例中原本的值相加，并返回最后的结果；
2. incrementAndGet()：以原子的方式将实例中的原值进行加1操作，并返回最终相加后的结果；
3. getAndSet(int newValue)：将实例中的值更新为新值，并返回旧值；
4. getAndIncrement()：以原子的方式将实例中的原值加1，返回的是自增前的旧值；

```
public final int getAndIncrement() {  
    return unsafe.getAndAddInt(this, valueOffset, 1);  
}
```

可以看出，该方法实际上是调用了**unsafe**实例的**getAndAddInt**方法，**unsafe**实例的获取时通过**UnSafe**类的静态方法**getUnsafe**获取：

**Unsafe**类提供了一些底层操作，**atomic**包下的原子操作类的主要是通过**Unsafe**类提供的**compareAndSwapInt**, **compareAndSwapLong**等一系列提供**CAS**操作的方法来进行实现。

**boolean**变量的更新类**AtomicBoolean**类是怎样实现更新的呢？核心方法是**compareAndSet**方法，其源码如下：

```
public final boolean compareAndSet(boolean expect, boolean update) {  
    int e = expect ? 1 : 0;  
    int u = update ? 1 : 0;  
    return unsafe.compareAndSwapInt(this, valueOffset, e, u);  
}
```

可以看出**atomic**包中只提供了对**boolean,int ,long**这三种基本类型的原子更新的方法，参考对**boolean**更新的方式，原子更新**char,doule,float**也可以采用类似的思路进行实现。

## 原子更新数组类型

**atomic**包下提供能原子更新数组中元素的类有：

1. **AtomicIntegerArray**：原子更新整型数组中的元素；
2. **AtomicLongArray**：原子更新长整型数组中的元素；
3. **AtomicReferenceArray**：原子更新引用类型数组中的元素

这几个类的用法一致，就以**AtomicIntegerArray**来总结下常用的方法：

1. addAndGet(int i, int delta)：以原子更新的方式将数组中索引为i的元素与输入值相加；
2. getAndIncrement(int i)：以原子更新的方式将数组中索引为i的元素自增加1；
3. compareAndSet(int i, int expect, int update)：将数组中索引为i的位置的元素进行更新

```
public class AtomicDemo {  
    //    private static AtomicInteger atomicInteger = new AtomicInteger(1);  
    private static int[] value = new int[]{1, 2, 3};  
    private static AtomicIntegerArray integerArray = new  
    AtomicIntegerArray(value);  
  
    public static void main(String[] args) {
```

```

//对数组中索引为1的位置的元素加5
int result = integerArray.getAndAdd(1, 5);
System.out.println(integerArray.get(1));
System.out.println(result);
}
}

输出结果:
7
2

```

## 原子更新引用类型

如果需要原子更新引用类型变量的话，为了保证线程安全，atomic也提供了相关的类：

1. AtomicReference：原子更新引用类型；
2. AtomicReferenceFieldUpdater：原子更新引用类型里的字段；
3. AtomicMarkableReference：原子更新带有标记位的引用类型；

这几个类的使用方法也是基本一样的，以AtomicReference为例，来说明这些类的基本用法。下面是一个demo

```

public class AtomicDemo {

    private static AtomicReference<User> reference = new AtomicReference<>();

    public static void main(String[] args) {
        User user1 = new User("a", 1);
        reference.set(user1);
        User user2 = new User("b", 2);
        User user = reference.getAndSet(user2);
        System.out.println(user);
        System.out.println(reference.get());
    }

    static class User {
        private String userName;
        private int age;

        public User(String userName, int age) {
            this.userName = userName;
            this.age = age;
        }

        @Override
        public String toString() {
            return "User{" +
                "userName='" + userName + '\'' +
                ", age=" + age +
                '}';
        }
    }
}
```

```
    }
}
```

输出结果：

```
User{userName='a', age=1}
User{userName='b', age=2}
```

## 原子更新字段类型

如果需要更新对象的某个字段，并在多线程的情况下，能够保证线程安全，atomic同样也提供了相应的原子操作类：

1. AtomicIntegerFieldUpdater：原子更新整型字段类；
2. AtomicLongFieldUpdater：原子更新长整型字段类；
3. AtomicStampedReference：原子更新引用类型，这种更新方式会带有版本号。而为什么在更新的时候会带有版本号，是为了解决**CAS的ABA问题**；

要想使用原子更新字段需要两步操作：

1. 原子更新字段类都是抽象类，只能通过静态方法 `newUpdater` 来创建一个更新器，并且需要设置想要更新的类和属性；
2. 更新类的属性必须使用 `public volatile` 进行修饰；

这几个类提供的方法基本一致，以AtomicIntegerFieldUpdater为例来看看具体的使用：

```
public class AtomicDemo {

    private static AtomicIntegerFieldUpdater updater =
    AtomicIntegerFieldUpdater.newUpdater(User.class, "age");

    public static void main(String[] args) {
        User user = new User("a", 1);
        int oldValue = updater.getAndAdd(user, 5);
        System.out.println(oldValue);
        System.out.println(updater.get(user));
    }

    static class User {
        private String userName;
        public volatile int age;

        public User(String userName, int age) {
            this.userName = userName;
            this.age = age;
        }

        @Override
        public String toString() {
            return "User{" +
                "userName='" + userName + '\'' +
                ", age=" + age +
                '}';
        }
    }
}
```

```
        '}';
    }
}
```

输出结果：

```
1  
6
```

## 并发工具

### CountDownLatch (倒计时器)

**应用场景：**有时候需要等待其他多个线程完成任务之后，主线程才能继续往下执行业务功能。使用join和线程间通信机制也能实现。但倒计时也很方便。

CountDownLatch的方法不是很多，将它们一个个列举出来：

1. await() throws InterruptedException：调用该方法的线程等到构造方法传入的N减到0的时候，才能继续往下执行；
2. await(long timeout, TimeUnit unit)：与上面的await方法功能一致，只不过这里有了时间限制，调用该方法的线程等到指定的timeout时间后，不管N是否减至为0，都会继续往下执行；
3. countDown()：使CountDownLatch初始值N减1；
4. long getCount()：获取当前CountDownLatch维护的值；

```
public class CountDownLatchDemo {  
    private static CountDownLatch startSignal = new CountDownLatch(1);  
    //用来表示裁判员需要维护的是6个运动员  
    private static CountDownLatch endSignal = new CountDownLatch(6);  
  
    public static void main(String[] args) throws InterruptedException {  
        ExecutorService executorService = Executors.newFixedThreadPool(6);  
        for (int i = 0; i < 6; i++) {  
            executorService.execute(() -> {  
                try {  
                    System.out.println(Thread.currentThread().getName() + " 运动员等  
待裁判员响哨！！！");  
                    startSignal.await();  
                    System.out.println(Thread.currentThread().getName() + " 正在全力  
冲刺");  
                    endSignal.countDown();  
                    System.out.println(Thread.currentThread().getName() + " 到达终  
点");  
                } catch (InterruptedException e) {  
                    e.printStackTrace();  
                }  
            });  
        }  
        System.out.println("裁判员发号施令啦！！！");  
    }  
}
```

```
        startSignal.countDown();
        endSignal.await();
        System.out.println("所有运动员到达终点，比赛结束！");
        executorService.shutdown();
    }
}
```

输出结果：

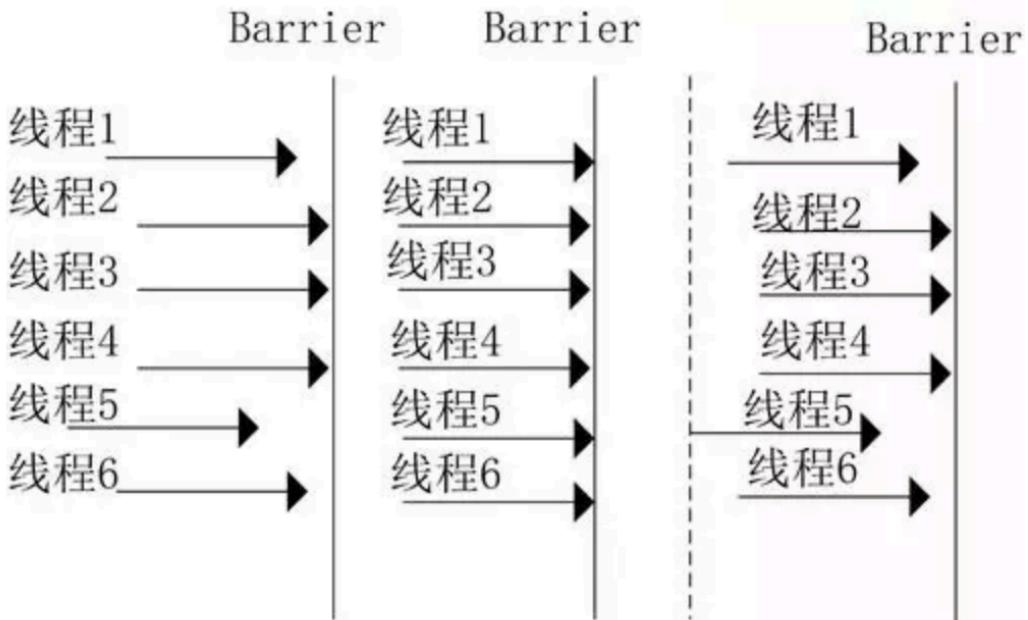
```
pool-1-thread-2 运动员等待裁判员响哨！！！
pool-1-thread-3 运动员等待裁判员响哨！！！
pool-1-thread-1 运动员等待裁判员响哨！！！
pool-1-thread-4 运动员等待裁判员响哨！！！
pool-1-thread-5 运动员等待裁判员响哨！！！
pool-1-thread-6 运动员等待裁判员响哨！！！
裁判员发号施令啦！！！
pool-1-thread-2 正在全力冲刺
pool-1-thread-2 到达终点
pool-1-thread-3 正在全力冲刺
pool-1-thread-3 到达终点
pool-1-thread-1 正在全力冲刺
pool-1-thread-1 到达终点
pool-1-thread-4 正在全力冲刺
pool-1-thread-4 到达终点
pool-1-thread-5 正在全力冲刺
pool-1-thread-5 到达终点
pool-1-thread-6 正在全力冲刺
pool-1-thread-6 到达终点
所有运动员到达终点，比赛结束！
```

需要注意的是，当调用**CountDownLatch**的**countDown**方法时，当前线程是会被阻塞，会继续往下执行，比如在该例中会继续输出 pool-1-thread-4 到达终点。

## CyclicBarrier（循环栅栏）

CyclicBarrier也是一种多线程并发控制的实用工具，和CountDownLatch一样具有等待计数的功能，但是相比于CountDownLatch功能更加强大。

## 下一次计数



相当于栅栏在「临界点」凑齐6个线程才会允许一波执行，并且栅栏接下来继续有效

。CyclicBarrier在使用一次后，下面依然有效，可以继续当做计数器使用，这是与CountDownLatch的区别之一。

```
//等到所有的线程都到达指定的临界点
await() throws InterruptedException, BrokenBarrierException
//与上面的await方法功能基本一致，只不过这里有超时限制，阻塞等待直至到达超时时间为止
await(long timeout, TimeUnit unit) throws InterruptedException,
BrokenBarrierException, TimeoutException
//获取当前有多少个线程阻塞等待在临界点上
int getNumberWaiting()
//用于查询阻塞等待的线程是否被中断
boolean isBroken()
//将屏障重置为初始状态。如果当前有线程正在临界点等待的话，将抛出BrokenBarrierException。
void reset()
```

另外需要注意的是，CyclicBarrier提供了这样的构造方法：

```
public CyclicBarrier(int parties, Runnable barrierAction)
```

可以用来，当指定的线程都到达了指定的临界点的时，接下来执行的操作可以由barrierAction传入即可。

相当于在栅栏开放时，先执行这个传入的线程barrierAction

```
public class CyclicBarrierDemo {
    //指定必须有6个运动员到达才行
    private static CyclicBarrier barrier = new CyclicBarrier(6, () -> {
        System.out.println("所有运动员入场，裁判员一声令下！！！！！");
    })
```

```

    });
    public static void main(String[] args) {
        System.out.println("运动员准备进场，全场欢呼.....");

        ExecutorService service = Executors.newFixedThreadPool(6);
        for (int i = 0; i < 6; i++) {
            service.execute(() -> {
                try {
                    System.out.println(Thread.currentThread().getName() + " 运动员，进场");
                    barrier.await();
                    System.out.println(Thread.currentThread().getName() + " 运动员出发");
                } catch (InterruptedException e) {
                    e.printStackTrace();
                } catch (BrokenBarrierException e) {
                    e.printStackTrace();
                }
            });
        }
    }
}

```

输出结果：

```

运动员准备进场，全场欢呼.....
pool-1-thread-2 运动员，进场
pool-1-thread-1 运动员，进场
pool-1-thread-3 运动员，进场
pool-1-thread-4 运动员，进场
pool-1-thread-5 运动员，进场
pool-1-thread-6 运动员，进场
所有运动员入场，裁判员一声令下!!!!!
pool-1-thread-6 运动员出发
pool-1-thread-1 运动员出发
pool-1-thread-5 运动员出发
pool-1-thread-4 运动员出发
pool-1-thread-3 运动员出发
pool-1-thread-2 运动员出发

```

## CountDownLatch和CyclicBarrier比较

都可以理解成维护的就是一个计数器，但是这两者还是各有不同侧重点的：

CountDownLatch一个等多个

CyclicBarrier多个线程等一个状态，再携手共进

CyclicBarrier提供的方法更多，比如能够通过getNumberWaiting(), isBroken()这些方法获取当前多个线程的状态，并且CyclicBarrier的构造方法可以传入**barrierAction**，指定当所有线程都到达时执行的业务功能；

CountDownLatch是不能复用的，而CyclicBarrier是可以复用的

## 控制资源并发访问Semaphore（信号量）

Semaphore可以用于做流量控制，特别是公共资源有限的应用场景，比如数据库连接。假如有多个线程读取数据后，需要将数据保存在数据库中，而可用的最大数据库连接只有10个，这时候就需要使用Semaphore来控制能够并发访问到数据库连接资源的线程个数最多只有10个。在限制资源使用的应用场景下，Semaphore是特别合适

```
//获取许可，如果无法获取到，则阻塞等待直至能够获取为止
void acquire() throws InterruptedException

//同acquire方法功能基本一样，只不过该方法可以一次获取多个许可
void acquire(int permits) throws InterruptedException

//释放许可
void release()

//释放指定个数的许可
void release(int permits)

//尝试获取许可，如果能够获取成功则立即返回true，否则，则返回false
boolean tryAcquire()

//与tryAcquire方法一致，只不过这里可以指定获取多个许可
boolean tryAcquire(int permits)

//尝试获取许可，如果能够立即获取到或者在指定时间内能够获取到，则返回true，否则返回false
boolean tryAcquire(long timeout, TimeUnit unit) throws InterruptedException

//与上一个方法一致，只不过这里能够获取多个许可
boolean tryAcquire(int permits, long timeout, TimeUnit unit)

//返回当前可用的许可证个数
int availablePermits()

//返回正在等待获取许可证的线程数
int getQueueLength()

//是否有线程正在等待获取许可证
boolean hasQueuedThreads()

//获取所有正在等待许可的线程集合
Collection<Thread> getQueuedThreads()
```

另外，在Semaphore的构造方法中还支持指定是否具有公平性，默认的是非公平性，这样也是为了保证吞吐量。

Semaphore用来做特殊资源的并发访问控制是相当合适的，如果有业务场景需要进行流量控制，可以优先考虑Semaphore。

## 线程间交换数据工具Exchanger

Exchanger是一个用于线程间协作的工具类，用于两个线程间能够交换。它提供了一个交换的同步点，在这个同步点两个线程能够交换数据。具体交换数据是通过exchange方法来实现的，如果一个线程先执行exchange方法，那么它会同步等待另一个线程也执行exchange方法，这个时候两个线程就都达到了同步点，两个线程就可以交换数据。

```
//当一个线程执行该方法的时候，会等待另一个线程也执行该方法，因此两个线程就都达到了同步点  
//将数据交给另一个线程，同时返回获取的数据  
V exchange(V x) throws InterruptedException  
  
//同上一个方法功能基本一样，只不过这个方法同步等待的时候，增加了超时时间  
V exchange(V x, long timeout, TimeUnit unit)  
throws InterruptedException, TimeoutException
```

## 生产者-消费者问题

这个共享数据区域中应该具备这样的线程间并发协作的功能：

1. 如果共享数据区已满的话，阻塞生产者继续生产数据放置入内；
2. 如果共享数据区为空的话，阻塞消费者继续消费数据；

在实现生产者消费者问题时，可以采用三种方式：

1. 使用Object的wait/notify的消息通知机制；
2. 使用Lock的Condition的await/signal的消息通知机制；
3. 使用BlockingQueue实现。本文主要将这三种实现方式进行总结归纳。

### wait/notify消息通知潜在的一些问题

#### 1. 通知遗漏

总结：在使用线程的等待/通知机制时，一般都要配合一个 boolean 变量值（或者其他能够判断真假的条件），在 **notify** 之前改变该 boolean 变量的值，让 **wait** 返回后能够退出 **while** 循环（一般都要在 **wait** 方法外围加一层 **while** 循环，以防止早期通知），或在通知被遗漏后，不会被阻塞在 **wait** 方法处。这样便保证了程序的正确性。

#### 2. 等待的**wait**条件发生变化，即被唤醒后那一瞬间条件又不满足了

总结：在使用线程的等待/通知机制时，一般都要在 **while** 循环中调用 **wait()** 方法，因此需要配合使用一个 boolean 变量，满足 **while** 循环的条件时，进入 **while** 循环，执行 **wait()** 方法，不满足 **while** 循环的条件时，跳出循环，执行后面的代码。

#### 3. “假死”状态

现象：如果是多消费者和多生产者情况，如果使用notify方法可能会出现“假死”的情况，即唤醒的是同类线程。

原因分析：假设当前多个生产者线程会调用wait方法阻塞等待，当其中的生产者线程获取到对象锁之后使用notify通知处于WAITTING状态的线程，如果唤醒的仍然是生产者线程，就会造成所有的生产者线程都处于等待状态。

解决办法：将**notify**方法替换成**notifyAll**方法，如果使用的是lock的话，就将**signal**方法替换成**signalAll**方法。

在**Object**提供的消息通知机制应该遵循如下这些条件：

1. 永远在**while**循环中对条件进行判断而不是if语句中进行**wait**条件的判断；
2. 使用**notifyAll**而不是使用**notify**。