# JAVA基础语法

## 注释

### 单行注释

//

### 多行注释

/\*\*/

### 文档注释

/\*\* \*/ 用于生成Java帮助文档

## 关键字

|  |  |
| --- | --- |
| **关键字** | **含义** |
| abstract | 表明类或者成员方法具有抽象属性 |
| assert | 断言，用来进行程序调试 |
| boolean | 基本数据类型之一，声明布尔类型的关键字 |
| break | 提前跳出一个块 |
| byte | 基本数据类型之一，字节类型 |
| case | 用在switch语句之中，表示其中的一个分支 |
| catch | 用在异常处理中，用来捕捉异常 |
| char | 基本数据类型之一，字符类型 |
| class | 声明一个类 |
| const | 保留关键字，没有具体含义 |
| continue | 回到一个块的开始处 |
| default | 默认，例如，用在switch语句中，表明一个默认的分支。Java8 中也作用于声明接口函数的默认实现 |
| do | 用在do-while循环结构中 |
| double | 基本数据类型之一，双精度浮点数类型 |
| else | 用在条件语句中，表明当条件不成立时的分支 |
| enum | 枚举 |
| extends | 表明一个类型是另一个类型的子类型。对于类，可以是另一个类或者抽象类；对于接口，可以是另一个接口 |
| final | 用来说明最终属性，表明一个类不能派生出子类，或者成员方法不能被覆盖，或者成员域的值不能被改变，用来定义常量 |
| finally | 用于处理异常情况，用来声明一个基本肯定会被执行到的语句块 |
| float | 基本数据类型之一，单精度浮点数类型 |
| for | 一种循环结构的引导词 |
| goto | 保留关键字，没有具体含义 |
| if | 条件语句的引导词 |
| implements | 表明一个类实现了给定的接口 |
| import | 表明要访问指定的类或包 |
| instanceof | 用来测试一个对象是否是指定类型的实例对象 |
| int | 基本数据类型之一，整数类型 |
| interface | 接口 |
| long | 基本数据类型之一，长整数类型 |
| native | 用来声明一个方法是由与计算机相关的语言（如C/C++/FORTRAN语言）实现的 |
| new | 用来创建新实例对象 |
| package | 包 |
| private | 一种访问控制方式：私用模式 |
| protected | 一种访问控制方式：保护模式 |
| public | 一种访问控制方式：共用模式 |
| return | 从成员方法中返回数据 |
| short | 基本数据类型之一,短整数类型 |
| static | 表明具有静态属性 |
| strictfp | 用来声明FP\_strict（单精度或双精度浮点数）表达式遵循[IEEE 754](https://baike.baidu.com/item/IEEE%20754)算术规范 |
| super | 表明当前对象的父类型的引用或者父类型的构造方法 |
| switch | 分支语句结构的引导词 |
| synchronized | 表明一段代码需要同步执行 |
| this | 指向当前实例对象的引用 |
| throw | 抛出一个异常 |
| throws | 声明在当前定义的成员方法中所有需要抛出的异常 |
| transient | 声明不用序列化的成员域 |
| try | 尝试一个可能抛出异常的程序块 |
| void | 声明当前成员方法没有返回值 |
| volatile | 表明两个或者多个变量必须同步地发生变化 |
| while | 用在循环结构中 |

## 数据类型

### 基础数据类型

**byte：**

* byte 数据类型是8位、有符号的，以二进制补码表示的整数；
* 最小值是 -128（-2^7）；
* 最大值是 127（2^7-1）；
* 默认值是 0；
* byte 类型用在大型数组中节约空间，主要代替整数，因为 byte 变量占用的空间只有 int 类型的四分之一；
* 例子：byte a = 100，byte b = -50。

**short：**

* short 数据类型是 16 位、有符号的以二进制补码表示的整数
* 最小值是 -32768（-2^15）；
* 最大值是 32767（2^15 - 1）；
* Short 数据类型也可以像 byte 那样节省空间。一个short变量是int型变量所占空间的二分之一；
* 默认值是 0；
* 例子：short s = 1000，short r = -20000。

**int：**

* int 数据类型是32位、有符号的以二进制补码表示的整数；
* 最小值是 -2,147,483,648（-2^31）；
* 最大值是 2,147,483,647（2^31 - 1）；
* 一般地整型变量默认为 int 类型；
* 默认值是 0 ；
* 例子：int a = 100000, int b = -200000。

**long：**

* long 数据类型是 64 位、有符号的以二进制补码表示的整数；
* 最小值是 -9,223,372,036,854,775,808（-2^63）；
* 最大值是 9,223,372,036,854,775,807（2^63 -1）；
* 这种类型主要使用在需要比较大整数的系统上；
* 默认值是 0L；
* 例子： long a = 100000L，Long b = -200000L。  
  "L"理论上不分大小写，但是若写成"l"容易与数字"1"混淆，不容易分辩。所以最好大写。

**float：**

* float 数据类型是单精度、32位、符合IEEE 754标准的浮点数；
* float 在储存大型浮点数组的时候可节省内存空间；
* 默认值是 0.0f；
* 浮点数不能用来表示精确的值，如货币；
* 例子：float f1 = 234.5f。

**double：**

* double 数据类型是双精度、64 位、符合 IEEE 754 标准的浮点数；
* 浮点数的默认类型为 double 类型；
* double类型同样不能表示精确的值，如货币；
* 默认值是 0.0d；
* 例子：
* double d1 = 7D ;  
  double d2 = 7.;   
  double d3 = 8.0;   
  double d4 = 8.D;   
  double d5 = 12.9867;
* 7 是一个 int 字面量，而 7D，7. 和 8.0 是 double 字面量。

**boolean：**

* boolean数据类型表示一位的信息；
* 只有两个取值：true 和 false；
* 这种类型只作为一种标志来记录 true/false 情况；
* 默认值是 false；
* 例子：boolean one = true。

**char：**

char类型是一个单一的 16 位 Unicode 字符；

最小值是 \u0000（即为 0）；

最大值是 \uffff（即为 65535）；

char 数据类型可以储存任何字符；

例子：char letter = 'A';。

### 引用数据类型

类 接口 数组

### 类型转换

#### 基本类型 转换原则

1、类型转换主要在在 赋值、方法调用、算术运算 三种情况下发生。  
 a、赋值和方法调用 转换规则：从低位类型到高位类型自动转换；从高位类型到低位类型需要强制类型转换：  
 （1）布尔型和其它基本数据类型之间不能相互转换；   
 （2）byte型可以转换为short、int、、long、float和double；   
 （3）short可转换为int、long、float和double；   
 （4）char可转换为int、long、float和double；   
 （5）int可转换为long、float和double；   
 （6）long可转换为float和double；   
 （7）float可转换为double；

b、算术运算 中的类型转换：1 基本就是先转换为高位数据类型，再参加运算，结果也是最高位的数据类型；2 byte short char运算会转换为Int；

（1）如操作数之一为double，则另一个操作数先被转化为double，再参与算术运算。

（2）如两操作数均不为double，当操作数之一为float，则另一操作数先被转换为float，再参与运算。

（3）如两操作数均不为double或float，当操作数之一为long，、则另一操作数先被转换为long，再参与算术运算。

（4）如两操作数均不为double、float或long，则两操作数先被转换为int，再参与运算。

特殊：

（1）如采用+=、\*=等缩略形式的运算符，系统会自动强制将运算结果转换为目标变量的类型。

(2) 当运算符为自动递增运算符（++）或自动递减运算符（--）时，如果操作数为byte，short或char类型不发生改变；

#### 引用类型 转换原则

1、基本类型 与 对应包装类 可自动转换，这是自动装箱和折箱的原理；

2、两个引用类型间转换：

(1)子类能直接转换为父类 或 接口类型；

(2)父类转换为子类要 强制类型转换；且在运行时若实际不是对应的对象，会抛出ClassCastException运行时异常；

## 数组

### 概念

数组是指一组相关类型的变量集合，并且这些变量可以按照统一的方式进行操作。数组本身属于引用对象，实际牵扯到内存分配。

### 定义语法

初始化数组

有两种方式进行初始化，一是直接new，二是直接赋值

数据类型 [] 数组名称 = new 数据类型[长度];//初始化，元素为数据类型默认值

数据类型 [] 数组名称 = {元素1,元素2,元素3};//直接赋值

数据类型 [] 数组名称 = new 数据类型[]{元素1,元素2,元素3};//直接赋值

### 数组的访问

（1）获取数组长度：调用数组的length属性即可获取数组长度。

（2）数组的访问：通过下标访问数组元素。下标从0开始，依次+1

（3）遍历数组元素：以数组元素下标为递增变量，循环输出即可遍历。

## 方法

### 概念

（1）解决一类问题的有序步骤组合

（2）存在于对象或者类中

（3）在程序中被创建，在其他地方被引用

### 方法命名规则

（1）方法的名字的第一个单词应以小写字母作为开头，后面的单词则用大写字母开头写，不使用连接符。

（2）下划线可能出现在 JUnit 测试方法名称中用以分隔名称的逻辑组件。一个典型的模式是：test\_，例如 testPop\_emptyStack。

### 定义

修饰符 返回值类型 方法名(参数类型 参数名){ ... 方法体 ... return 返回值; }

方法包含一个方法头和一个方法体。下面是一个方法的所有部分：

* **修饰符：**修饰符，这是可选的，告诉编译器如何调用该方法。定义了该方法的访问类型。
* **返回值类型 ：**方法可能会返回值。returnValueType 是方法返回值的数据类型。有些方法执行所需的操作，但没有返回值。在这种情况下，returnValueType 是关键字**void**。
* **方法名：**是方法的实际名称。方法名和参数表共同构成方法签名。
* **参数类型：**参数像是一个占位符。当方法被调用时，传递值给参数。这个值被称为实参或变量。参数列表是指方法的参数类型、顺序和参数的个数。参数是可选的，方法可以不包含任何参数。
* **方法体：**方法体包含具体的语句，定义该方法的功能。

### 方法调用

类名.方法

对象.方法

类名::方法名

### 方法重载

顾名思义，重新装载。方法名相同，返回类型相同的情况下，参数列表不同

### 可变长参数

#### 定义

使用…表示可变长参数，例如

print(String... args){  
...  
}

在具有可变长参数的方法中可以把参数当成数组使用，例如可以循环输出所有的参数值。

print(String... args){  
 for(String temp:args)  
 System.out.println(temp);  
}

调用的时候可以给出任意多个参数也可不给参数，例如：

print();  
print("hello");  
print("hello","lisi");  
print("hello","张三", "alexia")

#### 使用规则

（1）在调用方法的时候，如果能够和固定参数的方法匹配，也能够与可变长参数的方法匹配，则选择固定参数的方法。

（2）如果要调用的方法可以和两个可变参数匹配，则出现错误。

（3）一个方法只能有一个可变长参数，并且这个可变长参数必须是该方法的最后一个参数

（4）避免带有可变长参数的方法重载；别让null值和空值威胁到变长方法。

## 流程控制

### 复合语句

Java语言的复合语句是以整个块区为单位的语句，又称块语句。复合语句由“{”开始，“}”结束。对于复合语句，我们只需要知道，复合语句为局部变量创建了一个作用域，该作用域为程序的一部分，在该作用域中某个变量被创建并能够被使用，如果在某个变量的作用域外使用该变量，则会发生错误。并且复合语句中可以嵌套复合语句。

### 条件语句

条件语句可根据不同的条件执行不同的语句。包括if条件语句与switch多分支语句。

#### if...else

一个 if 语句包含一个布尔表达式和一条或多条语句。

if(布尔表达式) { //如果布尔表达式为true将执行的语句 }

#### if...else if...else 语句

if(布尔表达式 1){ //如果布尔表达式 1的值为true执行代码 }

else if(布尔表达式 2){ //如果布尔表达式 2的值为true执行代码 }

else if(布尔表达式 3){ //如果布尔表达式 3的值为true执行代码 }

else { //如果以上布尔表达式都不为true执行代码 }

#### 嵌套的 if…else 语句

if(布尔表达式 1){

////如果布尔表达式 1的值为true执行代码

if(布尔表达式 2){

////如果布尔表达式 2的值为true执行代码

}

}

#### switch case

switch(expression){

case value : //语句

break; //可选

case value : //语句

break; //可选

//你可以有任意数量的case语句

default : //可选

//语句

}

switch case 语句有如下规则：

* switch 语句中的变量类型可以是： byte、short、int 或者 char。从 Java SE 7 开始，switch 支持字符串 String 类型了，同时 case 标签必须为字符串常量或字面量。
* switch 语句可以拥有多个 case 语句。每个 case 后面跟一个要比较的值和冒号。
* case 语句中的值的数据类型必须与变量的数据类型相同，而且只能是常量或者字面常量。
* 当变量的值与 case 语句的值相等时，那么 case 语句之后的语句开始执行，直到 break 语句出现才会跳出 switch 语句。
* 当遇到 break 语句时，switch 语句终止。程序跳转到 switch 语句后面的语句执行。case 语句不必须要包含 break 语句。如果没有 break 语句出现，程序会继续执行下一条 case 语句，直到出现 break 语句。
* switch 语句可以包含一个 default 分支，该分支一般是 switch 语句的最后一个分支（可以在任何位置，但建议在最后一个）。default 在没有 case 语句的值和变量值相等的时候执行。default 分支不需要 break 语句。

**switch case 执行时，一定会先进行匹配，匹配成功返回当前 case 的值，再根据是否有 break，判断是否继续输出，或是跳出判断。**

### 循环结构

顺序结构的程序语句只能被执行一次。如果您想要同样的操作执行多次,，就需要使用循环结构。

Java中有三种主要的循环结构：

* **while** 循环
* **do…while** 循环
* **for** 循环

在Java5中引入了一种主要用于数组的增强型for循环。

#### while 循环

while是最基本的循环，它的结构为：

while( 布尔表达式 ) { //循环内容 }

只要布尔表达式为 true，循环就会一直执行下去。

#### do…while 循环

对于 while 语句而言，如果不满足条件，则不能进入循环。但有时候我们需要即使不满足条件，也至少执行一次。

do…while 循环和 while 循环相似，不同的是，do…while 循环至少会执行一次。

#### for循环

虽然所有循环结构都可以用 while 或者 do...while表示，但 Java 提供了另一种语句 —— for 循环，使一些循环结构变得更加简单。

for循环执行的次数是在执行前就确定的。语法格式如下：

for(初始化; 布尔表达式; 更新) { //代码语句 }

关于 for 循环有以下几点说明：

* 最先执行初始化步骤。可以声明一种类型，但可初始化一个或多个循环控制变量，也可以是空语句。
* 然后，检测布尔表达式的值。如果为 true，循环体被执行。如果为false，循环终止，开始执行循环体后面的语句。
* 执行一次循环后，更新循环控制变量。
* 再次检测布尔表达式。循环执行上面的过程。

#### Java 增强 for 循环

Java5 引入了一种主要用于数组的增强型 for 循环。

Java 增强 for 循环语法格式如下:

for(声明语句 : 表达式) { //代码句子 }

**声明语句：**声明新的局部变量，该变量的类型必须和数组元素的类型匹配。其作用域限定在循环语句块，其值与此时数组元素的值相等。

**表达式：**表达式是要访问的数组名，或者是返回值为数组的方法。

#### break 关键字

break 主要用在循环语句或者 switch 语句中，用来跳出整个语句块。

break 跳出最里层的循环，并且继续执行该循环下面的语句。

##### 语法

break 的用法很简单，就是循环结构中的一条语句：跳出当前循环

#### continue 关键字

continue 适用于任何循环控制结构中。作用是让程序立刻跳转到下一次循环的迭代。

在 for 循环中，continue 语句使程序立即跳转到更新语句。

在 while 或者 do…while 循环中，程序立即跳转到布尔表达式的判断语句。

##### 语法

continue 就是循环体中一条简单的语句：跳过当前循环

## 面向对象

### 类与对象

类：类是一个模板，他描述一类对象的行为和状态。一个类包含以下类型变量：

* 局部变量：在方法、构造方法或者语句块中定义的变量被称为局部变量。变量声明和初始化都是在方法中，方法结束后，变量就会自动销毁。
* 成员变量：成员变量是定义在类中，方法体之外的变量。这种变量在创建对象的时候实例化。成员变量可以被类中方法、构造方法和特定类的语句块访问。
* 类变量：类变量也声明在类中，方法体之外，但必须声明为 static 类型

对象：对象是一个类的实例，有状态和行为。

### 构造方法

每个类都有构造方法。如果没有显式地为类定义构造方法，Java 编译器将会为该类提供一个默认构造方法。

在创建一个对象的时候，至少要调用一个构造方法。构造方法的名称必须与类同名，一个类可以有多个构造方法。

### New对象

对象是根据类创建的。在Java中，使用关键字 new 来创建一个新的对象。创建对象需要以下三步：

* + 声明：声明一个对象，包括对象名称和对象类型。
  + 实例化：使用关键字 new 来创建一个对象。
  + 初始化：使用 new 创建对象时，会调用构造方法初始化对象。

### 访问实例变量和方法

通过已经创建的对象来访问成员变量和方法。

### 源文件声明原则

* + 一个源文件中只能有一个 public 类
  + 一个源文件可以有多个非 public 类
  + 源文件的名称应该和 public 类的类名保持一致。例如：源文件中 public 类的类名是 Employee，那么源文件应该命名为Employee.java。
  + 如果一个类定义在某个包中，那么 package 语句应该在源文件的首行。
  + 如果源文件包含 import 语句，那么应该放在 package 语句和类定义之间。如果没有 package 语句，那么 import 语句应该在源文件中最前面。
  + import 语句和 package 语句对源文件中定义的所有类都有效。在同一源文件中，不能给不同的类不同的包声明。

### 封装

在面向对象程式设计方法中，封装（英语：Encapsulation）是指一种将抽象性函式接口的实现细节部分包装、隐藏起来的方法。

封装可以被认为是一个保护屏障，防止该类的代码和数据被外部类定义的代码随机访问。

要访问该类的代码和数据，必须通过严格的接口控制。

封装最主要的功能在于我们能修改自己的实现代码，而不用修改那些调用我们代码的程序片段。

适当的封装可以让程式码更容易理解与维护，也加强了程式码的安全性。

#### 封装的优点：

1. 良好的封装能够减少耦合。

2. 类内部的结构可以自由修改。

3. 可以对成员变量进行更精确的控制。

4. 隐藏信息，实现细节。

#### 实现步骤

1. 修改属性的可见性来限制对属性的访问（一般限制为private）
2. 对每个值属性提供对外的公共方法访问，也就是创建一对赋取值方法，用于对私有属性的访问

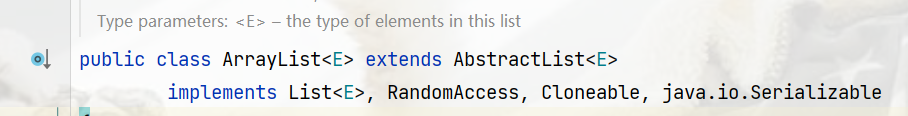
### 继承

继承是java面向对象编程技术的一块基石，因为它允许创建分等级层次的类。

继承就是子类继承父类的特征和行为，使得子类对象（实例）具有父类的实例域和方法，或子类从父类继承方法，使得子类具有父类相同的行为。

#### 类的继承格式

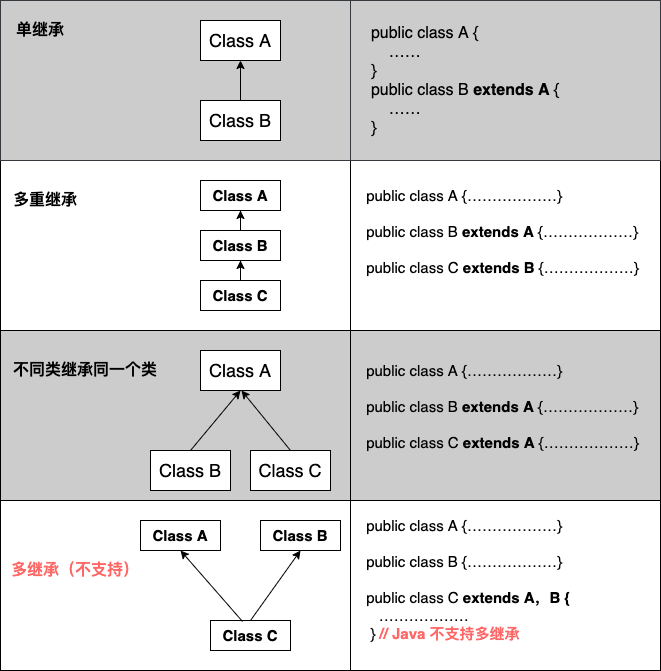
在 Java 中通过 extends 关键字可以申明一个类是从另外一个类继承而来的。以集合类ArrayList为例，



继承AbstractList抽象类。继承可以减少重复性代码。

#### 继承类型

需要注意的是，Java不支持多继承，但支持多重继承。



#### 继承的特性

* 子类拥有父类非 private 的属性、方法。
* 子类可以拥有自己的属性和方法，即子类可以对父类进行扩展。
* 子类可以用自己的方式实现父类的方法。
* Java 的继承是单继承，但是可以多重继承，单继承就是一个子类只能继承一个父类，多重继承就是，例如 B 类继承 A 类，C 类继承 B 类，所以按照关系就是 B 类是 C 类的父类，A 类是 B 类的父类，这是 Java 继承区别于 C++ 继承的一个特性。
* 提高了类之间的耦合性（继承的缺点，耦合度高就会造成代码之间的联系越紧密，代码独立性越差）。

#### 继承关键字

继承可以使用 extends 和 implements 这两个关键字来实现继承，而且所有的类都是继承于 java.lang.Object，当一个类没有继承的两个关键字，则默认继承object（这个类在 java.lang 包中，所以不需要 import）祖先类。

#### Extends关键字

在 Java 中，类的继承是单一继承，也就是说，一个子类只能拥有一个父类，所以 extends 只能继承一个类。

#### Implements关键字

使用 implements 关键字可以变相的使java具有多继承的特性，使用范围为类继承接口的情况，可以同时继承多个接口（接口跟接口之间采用逗号分隔）。

#### super 与 this 关键字

super关键字：我们可以通过super关键字来实现对父类成员的访问，用来引用当前对象的父类。

this关键字：指向自己的引用。

#### Final关键字

final 关键字声明类可以把类定义为不能继承的，即最终类；或者用于修饰方法，该方法不能被子类重写：

声明类：

final class 类名 {//类体}

声明方法：

修饰符(public/private/default/protected) final 返回值类型 方法名(){//方法体}

**注**:实例变量也可以被定义为 final，被定义为 final 的变量不能被修改。被声明为 final 类的方法自动地声明为 final，但是实例变量并不是 final

#### 构造器

子类是不继承父类的构造器（构造方法或者构造函数）的，它只是调用（隐式或显式）。如果父类的构造器带有参数，则必须在子类的构造器中显式地通过 super 关键字调用父类的构造器并配以适当的参数列表。

如果父类构造器没有参数，则在子类的构造器中不需要使用 super 关键字调用父类构造器，系统会自动调用父类的无参构造器。

### 多态

多态是同一个行为具有多个不同表现形式或形态的能力。

多态就是同一个接口，使用不同的实例而执行不同操作。

多态性是对象多种表现形式的体现。比如狗叫是“汪汪汪”，猫叫则是“喵喵喵”

同一个事件发生在不同的对象上会产生不同的结果。

#### 多态优点

1. 消除类型之间的耦合关系

2. 可替换性

3. 可扩充性

4. 接口性

5. 灵活性

6. 简化性

#### 多态存在的必要条件

* + 继承
  + 重写
  + 父类引用指向子类对象：Parent p = new Child();

### 修饰符

Java提供了修饰符，主要分为两类

* 访问修饰符
* 非访问修饰符

修饰符用来定义类、方法或者变量，通常放在语句的最前端。

public class ClassName {

// ...

}

private boolean myFlag;

static final double weeks = 9.5;

protected static final int BOXWIDTH = 42;

public static void main(String[] arguments) {

// 方法体

}

#### 访问控制修饰符

Java中，可以使用访问控制符来保护对类、变量、方法和构造方法的访问。Java 支持 4 种不同的访问权限。

* **default** (即默认，什么也不写）: 在同一包内可见，不使用任何修饰符。使用对象：类、接口、变量、方法。
* **private** : 在同一类内可见。使用对象：变量、方法。 **注意：不能修饰类（外部类）**
* **public** : 对所有类可见。使用对象：类、接口、变量、方法
* **protected** : 对同一包内的类和所有子类可见。使用对象：变量、方法。 **注意：不能修饰类（外部类）**。

#### 访问控制和继承

请注意以下方法继承的规则：

* 父类中声明为 public 的方法在子类中也必须为 public。
* 父类中声明为 protected 的方法在子类中要么声明为 protected，要么声明为 public，不能声明为 private。
* 父类中声明为 private 的方法，不能够被继承。

#### 非访问修饰符

#### static 修饰符

* **静态变量：**

static 关键字用来声明独立于对象的静态变量，无论一个类实例化多少对象，它的静态变量只有一份拷贝。 静态变量也被称为类变量。局部变量不能被声明为 static 变量。

* **静态方法：**

static 关键字用来声明独立于对象的静态方法。静态方法不能使用类的非静态变量。静态方法从参数列表得到数据，然后计算这些数据

#### final修饰符

final 表示"最后的、最终的"含义，变量一旦赋值后，不能被重新赋值。被 final 修饰的实例变量必须显式指定初始值。

final 修饰符通常和 static 修饰符一起使用来创建类常量。

#### Abstract修饰符

抽象类：

抽象类不能用来实例化对象，声明抽象类的唯一目的是为了将来对该类进行扩充。

一个类不能同时被 abstract 和 final 修饰。如果一个类包含抽象方法，那么该类一定要声明为抽象类，否则将出现编译错误。

抽象类可以包含抽象方法和非抽象方法。

**抽象方法**

抽象方法是一种没有任何实现的方法，该方法的的具体实现由子类提供。

抽象方法不能被声明成 final 和 static。

任何继承抽象类的子类必须实现父类的所有抽象方法，除非该子类也是抽象类。

如果一个类包含若干个抽象方法，那么该类必须声明为抽象类。抽象类可以不包含抽象方法。

抽象方法的声明以分号结尾，例如：**public abstract sample();**。

#### synchronized 修饰符

synchronized 关键字声明的方法同一时间只能被一个线程访问。synchronized 修饰符可以应用于四个访问修饰符。

#### transient 修饰符

序列化的对象包含被 transient 修饰的实例变量时，java 虚拟机(JVM)跳过该特定的变量。

该修饰符包含在定义变量的语句中，用来预处理类和变量的数据类型。

#### Volatile

volatile 修饰的成员变量在每次被线程访问时，都强制从共享内存中重新读取该成员变量的值。而且，当成员变量发生变化时，会强制线程将变化值回写到共享内存。这样在任何时刻，两个不同的线程总是看到某个成员变量的同一个值。

一个 volatile 对象引用可能是 null。

### 接口

接口（英文：Interface），在JAVA编程语言中是一个抽象类型，是抽象方法的集合，接口通常以interface来声明。一个类通过继承接口的方式，从而来继承接口的抽象方法。

接口并不是类，编写接口的方式和类很相似，但是它们属于不同的概念。类描述对象的属性和方法。接口则包含类要实现的方法。

除非实现接口的类是抽象类，否则该类要定义接口中的所有方法。

接口无法被实例化，但是可以被实现。一个实现接口的类，必须实现接口内所描述的所有方法，否则就必须声明为抽象类。另外，在 Java 中，接口类型可用来声明一个变量，他们可以成为一个空指针，或是被绑定在一个以此接口实现的对象。

JDK1.8中新增：

* default接口，表示子类默认继承此方法，不需要去实现。
* static函数，修饰非抽象方法，有自己的方法体，在接口定义一个静态方法，相当于调用类的静态方法一样。

### 内部类

在一个类中进行其他类的嵌套操作。

class Outer{

private String str ="外部类中的字符串";

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//定义一个内部类

class Inner{

private String inStr= "内部类中的字符串";

//定义一个普通方法

public void print(){

//调用外部类的str属性

System.out.println(str);

}

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//在外部类中定义一个方法，该方法负责产生内部类对象并调用print()方法

public void fun(){

//内部类对象

Inner in = new Inner();

//内部类对象提供的print

in.print();

}

}

由于访问权限问题，所以需要提供一个方法去生成内部类。

#### 内部类优点

1. 内部类与外部类可以方便的访问彼此的私有域（包括私有方法、私有属性）。
2. 内部类是另外一种封装，对外部的其他类隐藏。
3. 内部类可以实现java的单继承局限。

#### 内部类的缺点：

1. 结构复杂

#### 内部类的分类

1. 成员内部类：成员内部类内部不允许存在任何static变量或方法 正如成员方法中不能有任何静态属性 （成员方法与对象相关、静态属性与类有关）。在编译完成后，隐含一个引用指向创建他的外围类。**成员内部类依附于外部类，只有创建了外部类才能创建内部类。**
2. 静态内部类：用关键词static修饰的内部类，由于有static关键词修饰：
   1. 静态内部类的创建不需要依附于外部类，可以当成一个普通的类来使用。
   2. 静态内部类不可以使用任何外部类的非static类（包含属性和方法），但可以存在自己的成员变量。
3. 方法内部类：
   1. 在方法中定义的类叫方法内部类。
   2. 方法内部类不允许使用访问权限修饰符（public、private、protected均不允许）。
   3. 方法内部类如果想要使用方法形参，该形参必须是final声明（1.8中参数变为隐式声明）。这是由于局部变量生命周期与方法内部类对象的生命周期不一致性导致的，方法结束后，局部变量死亡，但是方法内部类对象可能还存在着，这时候方法内部类对象引用了一个死亡的对象，会产生异常。
4. 匿名内部类：一个没有名字的方法内部类，与方法内部类行为完全一致。除此之外，还有以下特点：
   1. 匿名内部类必须继承一个抽象类或者实现一个接口。
   2. 匿名内部类没有类名，因此没有构造方法。

## 网络编程

## 多线程

## I/O流

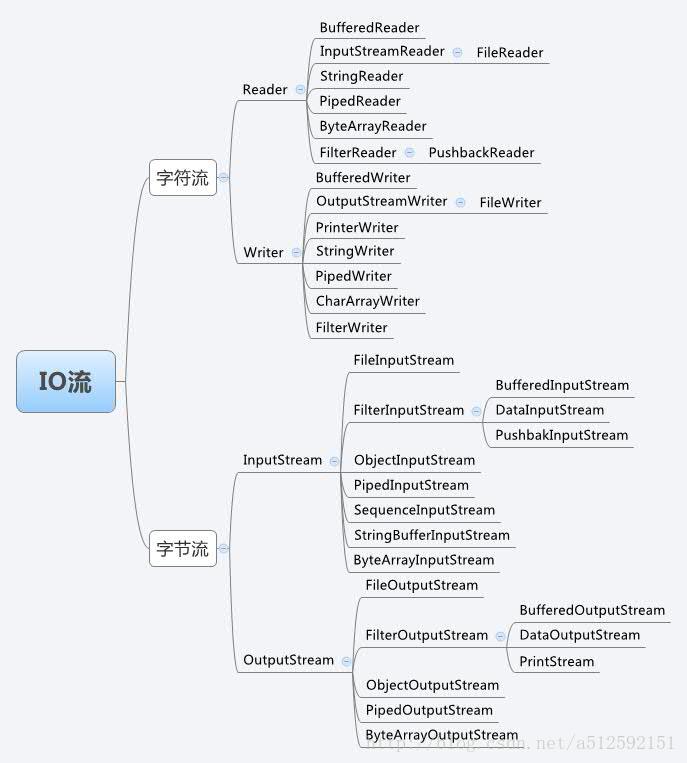
Java的输入输出流，分为I/O和NI/O，NIO就是原来的IO的一个补充，增加了异步处理等API。

### I/O

I/0流分为字符流和字节流：

* 字符流：字符流操作的基本单元为Unicode码元，默认使用缓冲区
* 字节流：字节流操作的基本单元为字节，理论上可以处理任意类型数据。默认不使用缓冲区。

总的接口和API如下所示



#### File对象

在计算机系统中，文件是非常重要的存储方式。Java的标准库java.io提供了File对象来操作文件和目录。要构造一个File对象，需要传入文件路径：

public class Main {

public static void main(String[] args) {

File f = new File("C:\\Windows\\notepad.exe");

System.out.println(f);

}

}

#### InputStream

InputStream就是Java标准库提供的最基本的输入流。它位于java.io这个包里。java.io包提供了所有同步IO的功能。

要特别注意的一点是，InputStream并不是一个接口，而是一个抽象类，它是所有输入流的超类。这个抽象类定义的一个最重要的方法就是int read()，签名如下：

public abstract int read() throws IOException;

这个方法会读取输入流的下一个字节，并返回字节表示的int值（0~255）。如果已读到末尾，返回-1表示不能继续读取了。

FileInputStream是InputStream的一个子类。顾名思义，FileInputStream就是从文件流中读取数据。下面的代码演示了如何完整地读取一个FileInputStream的所有字节：

public void readFile() throws IOException {

// 创建一个FileInputStream对象:

InputStream input = new FileInputStream("src/readme.txt");

for (;;) {

int n = input.read(); // 反复调用read()方法，直到返回-1

if (n == -1) {

break;

}

System.out.println(n); // 打印byte的值

}

input.close(); // 关闭流

}

在计算机中，类似文件、网络端口这些资源，都是由操作系统统一管理的。应用程序在运行的过程中，如果打开了一个文件进行读写，完成后要及时地关闭，以便让操作系统把资源释放掉，否则，应用程序占用的资源会越来越多，不但白白占用内存，还会影响其他应用程序的运行。

InputStream和OutputStream都是通过close()方法来关闭流。关闭流就会释放对应的底层资源。一般在try ... finally来保证InputStream在无论是否发生IO错误的时候都能够正确地关闭：

public void readFile() throws IOException {

InputStream input = null;

try {

input = new FileInputStream("src/readme.txt");

int n;

while ((n = input.read()) != -1) { // 利用while同时读取并判断

System.out.println(n);

}

} finally {

if (input != null) { input.close(); }

}

}

用try ... finally来编写上述代码会感觉比较复杂，更好的写法是利用Java 7引入的新的try(resource)的语法，只需要编写try语句，让编译器自动为我们关闭资源。推荐的写法如下：

public void readFile() throws IOException {

try (InputStream input = new FileInputStream("src/readme.txt")) {

int n;

while ((n = input.read()) != -1) {

System.out.println(n);

}

} // 编译器在此自动为我们写入finally并调用close()

}

实际上，编译器并不会特别地为InputStream加上自动关闭。编译器只看try(resource = ...)中的对象是否实现了java.lang.AutoCloseable接口，如果实现了，就自动加上finally语句并调用close()方法。InputStream和OutputStream都实现了这个接口，因此，都可以用在try(resource)中。

**缓冲**

在读取流的时候，一次读取一个字节并不是最高效的方法。很多流支持一次性读取多个字节到缓冲区，对于文件和网络流来说，利用缓冲区一次性读取多个字节效率往往要高很多。InputStream提供了两个重载方法来支持读取多个字节：

* int read(byte[] b)：读取若干字节并填充到byte[]数组，返回读取的字节数
* int read(byte[] b, int off, int len)：指定byte[]数组的偏移量和最大填充数

利用上述方法一次读取多个字节时，需要先定义一个byte[]数组作为缓冲区，read()方法会尽可能多地读取字节到缓冲区， 但不会超过缓冲区的大小。read()方法的返回值不再是字节的int值，而是返回实际读取了多少个字节。如果返回-1，表示没有更多的数据了。

利用缓冲区一次读取多个字节的代码如下：

public void readFile() throws IOException {

try (InputStream input = new FileInputStream("src/readme.txt")) {

// 定义1000个字节大小的缓冲区:

byte[] buffer = new byte[1000];

int n;

while ((n = input.read(buffer)) != -1) { // 读取到缓冲区

System.out.println("read " + n + " bytes.");

}

}

}

**阻塞**

在调用InputStream的read()方法读取数据时，我们说read()方法是阻塞（Blocking）的。它的意思是，对于下面的代码：

int n;

n = input.read(); // 必须等待read()方法返回才能执行下一行代码

int m = n;

执行到第二行代码时，必须等read()方法返回后才能继续。因为读取IO流相比执行普通代码，速度会慢很多，因此，无法确定read()方法调用到底要花费多长时间。

#### InputStream实现类

用FileInputStream可以从文件获取输入流，这是InputStream常用的一个实现类。此外，ByteArrayInputStream可以在内存中模拟一个InputStream：

public class Main {

public static void main(String[] args) throws IOException {

byte[] data = { 72, 101, 108, 108, 111, 33 };

try (InputStream input = new ByteArrayInputStream(data)) {

int n;

while ((n = input.read()) != -1) {

System.out.println((char)n);

}

}

}

}

#### OutputStream

和InputStream相反，OutputStream是Java标准库提供的最基本的输出流。

和InputStream类似，OutputStream也是抽象类，它是所有输出流的超类。这个抽象类定义的一个最重要的方法就是void write(int b)，签名如下：

public abstract void write(int b) throws IOException;

这个方法会写入一个字节到输出流。要注意的是，虽然传入的是int参数，但只会写入一个字节，即只写入int最低8位表示字节的部分（相当于b & 0xff）。

和InputStream类似，OutputStream也提供了close()方法关闭输出流，以便释放系统资源。要特别注意：OutputStream还提供了一个flush()方法，它的目的是将缓冲区的内容真正输出到目的地

向磁盘、网络写入数据的时候，出于效率的考虑，操作系统并不是输出一个字节就立刻写入到文件或者发送到网络，而是把输出的字节先放到内存的一个缓冲区里（本质上就是一个byte[]数组），等到缓冲区写满了，再一次性写入文件或者网络。对于很多IO设备来说，一次写一个字节和一次写1000个字节，花费的时间几乎是完全一样的，所以OutputStream有个flush()方法，能强制把缓冲区内容输出。通常情况下，我们不需要调用这个flush()方法，因为缓冲区写满了OutputStream会自动调用它，并且，在调用close()方法关闭OutputStream之前，也会自动调用flush()方法。

**阻塞**

和InputStream一样，OutputStream的write()方法也是阻塞的。

#### OutputStream实现类

用FileOutputStream可以从文件获取输出流，这是OutputStream常用的一个实现类，和inputStream相同，考虑到关闭资源，需要用try(resource)来保证OutputStream在无论是否发生IO错误的时候都能够正确地关闭：

public void writeFile() throws IOException {

try (OutputStream output = new FileOutputStream("out/readme.txt")) {

output.write("Hello".getBytes("UTF-8")); // Hello

} // 编译器在此自动为我们写入finally并调用close()

}

#### Filter（或者装饰器模式：Decorator）模式

针对InputStream设计不同的组合功能，比如缓冲、加密。直接使用继承关系的话，很快会出现子类爆炸的情况，无法控制代码复杂度。为了解决依赖继承会导致子类数量失控的问题，JDK首先将InputStream分为两大类：

一类是直接提供数据的基础InputStream，例如：

* FileInputStream
* ByteArrayInputStream
* ServletInputStream
* ...

一类是提供额外附加功能的InputStream，例如：

* BufferedInputStream
* DigestInputStream
* CipherInputStream
* ...

当我们需要给一个“基础”InputStream附加各种功能时，我们先确定这个能提供数据源的InputStream，因为我们需要的数据总得来自某个地方，例如，FileInputStream，数据来源自文件：

InputStream file = new FileInputStream("test.gz");

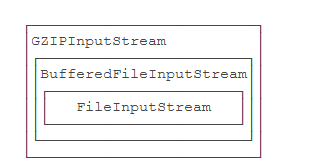
紧接着，我们希望FileInputStream能提供缓冲的功能来提高读取的效率，因此我们用BufferedInputStream包装这个InputStream，得到的包装类型是BufferedInputStream，但它仍然被视为一个InputStream：

InputStream buffered = new BufferedInputStream(file);

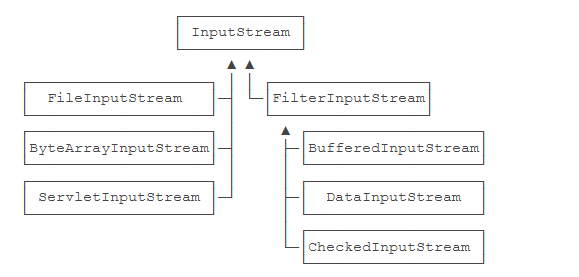
最后，假设该文件已经用gzip压缩了，我们希望直接读取解压缩的内容，就可以再包装一个GZIPInputStream：

InputStream gzip = new GZIPInputStream(buffered);

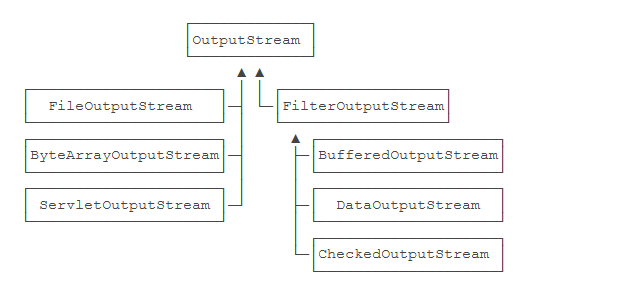
无论我们包装多少次，得到的对象始终是InputStream，我们直接用InputStream来引用它，就可以正常读取：



上述这种通过一个“基础”组件再叠加各种“附加”功能组件的模式，称之为Filter模式（或者装饰器模式：Decorator）。它可以让我们通过少量的类来实现各种功能的组合：



类似的，OutputStream也是以这种模式来提供各种功能：



#### 序列化

#### Reader

#### Writer

#### PrintStream和PrintWriter

#### 使用Files

从Java 7开始，提供了Files和Paths这两个工具类，能极大地方便我们读写文件。

虽然Files和Paths是java.nio包里面的类，但他俩封装了很多读写文件的简单方法，例如，我们要把一个文件的全部内容读取为一个byte[]，可以这么写：

byte[] data = Files.readAllBytes(Paths.get("/path/to/file.txt"));

如果是文本文件，可以把一个文件的全部内容读取为String：

// 默认使用UTF-8编码读取:

String content1 = Files.readString(Paths.get("/path/to/file.txt"));

// 可指定编码:

String content2 = Files.readString(Paths.get("/path/to/file.txt"), StandardCharsets.ISO\_8859\_1);

// 按行读取并返回每行内容:

List<String> lines = Files.readAllLines(Paths.get("/path/to/file.txt"));

写入文件也非常方便：

// 写入二进制文件:

byte[] data = ...

Files.write(Paths.get("/path/to/file.txt"), data);

// 写入文本并指定编码:

Files.writeString(Paths.get("/path/to/file.txt"), "文本内容...", StandardCharsets.ISO\_8859\_1);

// 按行写入文本:

List<String> lines = ...

Files.write(Paths.get("/path/to/file.txt"), lines);

此外，Files工具类还有copy()、delete()、exists()、move()等快捷方法操作文件和目录。

**最后需要特别注意的是，Files提供的读写方法，受内存限制，只能读写小文件，例如配置文件等，不可一次读入几个G的大文件。读写大型文件仍然要使用文件流，每次只读写一部分文件内容。**

### I/O模型

#### 用户空间和内核空间

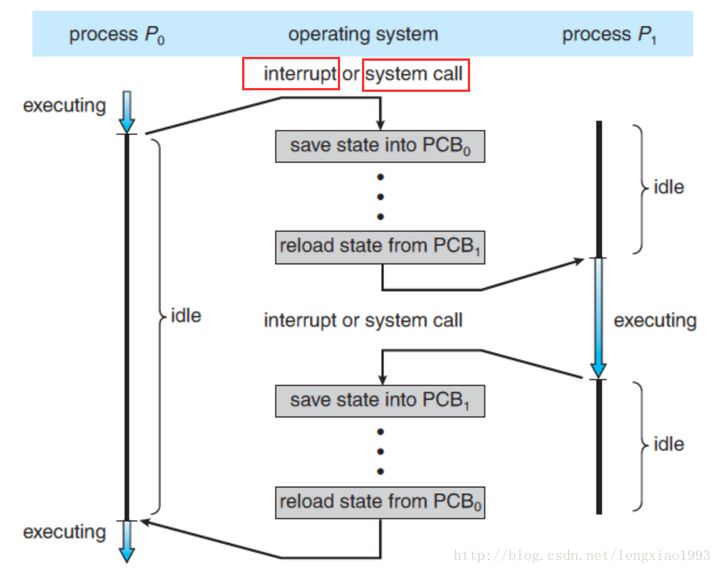
操作系统为了支持多个应用同时运行，需要保证不同进程之间相对独立（一个进程的崩溃不会影响其他的进程 ， 恶意进程不能直接读取和修改其他进程运行时的代码和数据）。 因此操作系统内核**需要拥有高于普通进程的权限**， 以此来调度和管理用户的应用程序。

于是内存空间被划分为两部分，一部分为内核空间，一部分为用户空间，内核空间存储的代码和数据具有更高级别的权限。内存访问的**相关硬件**在程序执行期间会进行访问控制（ Access Control），使得用户空间的程序不能直接读写内核空间的内存。

#### 进程切换

进程切换有如下几个最重要的步骤：

* 当一个程序正在执行的过程中， 中断（interrupt） 或 系统调用（system call） 发生可以使得 CPU 的控制权会从当前进程转移到操作系统内核。
* 操作系统内核负责保存进程 i 在 CPU 中的上下文（程序计数器， 寄存器）到 PCBi （操作系统分配给进程的一个内存块）中。
* 从 PCBj 取出进程 j 的CPU 上下文， 将 CPU 控制权转移给进程 j ， 开始执行进程 j 的指令。



几个底层概念的通俗（不严谨）解释：

* 中断（interrupt）

CPU 微处理器有一个中断信号位， 在每个CPU时钟周期的末尾, CPU会去检测那个中断信号位是否有中断信号到达， 如果有， 则会根据中断优先级决定是否要暂停当前执行的指令， 转而去执行处理中断的指令。 （其实就是 CPU 层级的 while 轮询）

* 时钟中断( Clock Interrupt )

一个硬件时钟会每隔一段（很短）的时间就产生一个中断信号发送给 CPU，CPU 在响应这个中断时， 就会去执行操作系统内核的指令， 继而将 CPU 的控制权转移给了操作系统内核， 可以由操作系统内核决定下一个要被执行的指令。

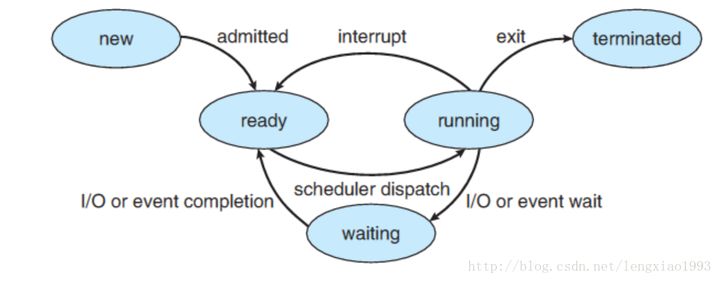
* 系统调用（system call）

system call 是操作系统提供给应用程序的接口。 用户通过调用 systemcall 来完成那些需要操作系统内核进行的操作， 例如硬盘， 网络接口设备的读写等。

从上述描述中， 可以看出来， 操作系统在进行进切换时，需要进行一系列的内存读写操作， 这带来了一定的开销：

* 对于一个运行着 UNIX 系统的现代 PC 来说， 进程切换通常至少需要花费 300 us 的时间

#### 进程阻塞



上图展示了一个进程的不同状态：

* New. 进程正在被创建.
* Running. 进程的指令正在被执行
* Waiting. 进程正在等待一些事件的发生（例如 I/O 的完成或者收到某个信号）
* Ready. 进程在等待被操作系统调度
* Terminated. 进程执行完毕（可能是被强行终止的）

我们所说的 “阻塞”是指进程在发起了一个系统调用（System Call） 后， 由于该系统调用的操作不能立即完成，需要等待一段时间，于是内核将进程挂起为等待 （waiting）状态， 以确保它不会被调度执行， 占用 CPU 资源。友情提示： 在任意时刻， 一个 CPU 核心上（processor）只可能运行一个进程 。

**阻塞**这个词是与系统调用 System Call 紧紧联系在一起的， 因为要让一个进程进入 等待（waiting） 的状态, 要么是它主动调用 wait() 或 sleep() 等挂起自己的操作， 另一种就是它调用 System Call, 而 System Call 因为涉及到了 I/O 操作， 不能立即完成， 于是内核就会先将该进程置为等待状态， 调度其他进程的运行， 等到 它所请求的 I/O 操作完成了以后， 再将其状态更改回 ready 。

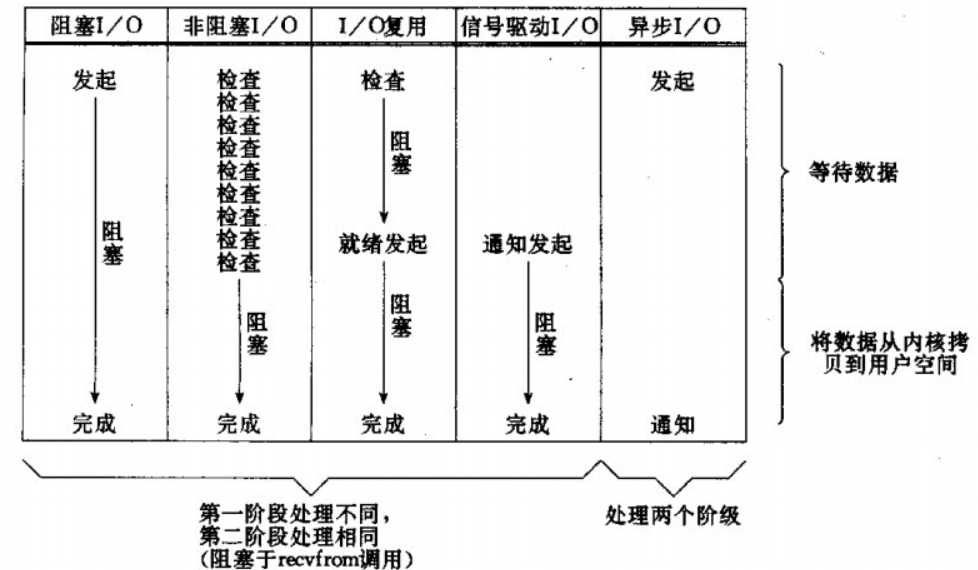
因此，阻塞I/O与非阻塞I/O的区别是：

* 一个**阻塞I/O 系统调用 read()** 操作立即返回的是任何可以立即拿到的数据， 可以是完整的结果， 也可以是不完整的结果， 还可以是一个空值。
* 而**非阻塞I/O系统调用** read（）结果必须是完整的， 但是这个操作完成的通知可以延迟到将来的一个时间点。

阻塞/非阻塞是针对内核空间上，线程与物理设备的概念：

* 阻塞，发起IO请求后，阻塞式的等待IO设备返回结果
* 非阻塞，发起IO请求后，内核空间不会挂起调用程序，而是立即返回一个值，表示有多少bytes数据会被写入/读取。

#### 几种IO模型



IO分两阶段（一旦拿到数据后就变成了数据操作，不再是IO）：

1.数据准备阶段

2.内核空间复制数据到用户进程缓冲区（用户空间）阶段

在操作系统中，程序运行的空间分为内核空间和用户空间。

应用程序都是运行在用户空间的，所以它们能操作的数据也都在用户空间。

阻塞IO和非阻塞IO的区别在于第一步发起IO请求是否会被阻塞：

如果阻塞直到完成那么就是传统的阻塞IO，如果不阻塞，那么就是非阻塞IO。

一般来讲：

阻塞IO模型、非阻塞IO模型、IO复用模型(select/poll/epoll)、信号驱动IO模型都属于同步IO，因为阶段2是阻塞的(尽管时间很短)。

同步IO和异步IO的区别就在于第二个步骤是否阻塞：

如果不阻塞，而是操作系统帮你做完IO操作再将结果返回给你，那么就是异步IO

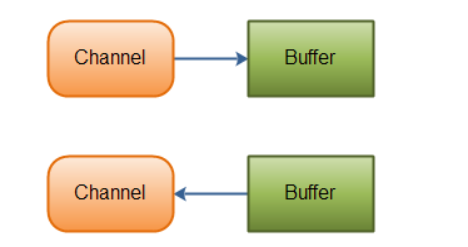
### NI/O框架

Java NIO 由以下几个核心部分组成：

* Channels
* Buffers
* Selectors

**Channel 和 Buffer**

基本上，所有的 IO 在NIO 中都从一个Channel 开始。Channel 有点像流。 数据可以从Channel读到Buffer中，也可以从Buffer 写到Channel中。如图下所示：



Channel和Buffer有好几种类型。下面是JAVA NIO中的一些主要Channel的实现：

* FileChannel
* DatagramChannel
* SocketChannel
* ServerSocketChannel

Buffer的一些主要实现

* ByteBuffer
* CharBuffer
* DoubleBuffer
* FloatBuffer
* IntBuffer
* LongBuffer
* ShortBuffer

#### Selector

Selector选择器是一个单线程文件描述符处理器，处理注册于这选择器的I/O流，由操作系统底层去实现轮询机制。当某一个I/O流的数据 准备好可以处理时，就开始处理这个流上的数据。



要使用Selector，得向Selector注册Channel，然后调用它的select()方法。这个方法会一直阻塞到某个注册的通道有事件就绪。一旦这个方法返回，线程就可以处理这些事件，事件的例子有如新连接进来，数据接收等。

##### 创建

Selector selector = Selector.open();

##### 注册文件流

为了将Channel和Selector配合使用，必须将channel注册到selector上。通过SelectableChannel.register()方法来实现

channel.configureBlocking(false);

SelectionKey key = channel.register(selector, Selectionkey.OP\_READ);

与Selector一起使用时，Channel必须处于非阻塞模式下。这意味着不能将FileChannel与Selector一起使用，因为FileChannel不能切换到非阻塞模式。而套接字通道都可以。

第二个参数，是一个“interest集合”，意思是在通过Selector监听Channel时对什么事件感兴趣。可以监听四种不同类型的事件：

1. Connect
2. Accept
3. Read
4. Write

通道触发了一个事件意思是该事件已经就绪。所以，某个channel成功连接到另一个服务器称为“连接就绪”。一个server socket channel准备好接收新进入的连接称为“接收就绪”。一个有数据可读的通道可以说是“读就绪”。等待写数据的通道可以说是“写就绪”。

这四种事件用SelectionKey的四个常量来表示：

1. SelectionKey.OP\_CONNECT
2. SelectionKey.OP\_ACCEPT
3. SelectionKey.OP\_READ
4. SelectionKey.OP\_WRITE

如果要对多个事件进行处理，则需要用”|”操作符连接多个事件。如下：

int interestSet = SelectionKey.OP\_READ | SelectionKey.OP\_WRITE;

##### SelectionKey

注册channel时，register()方法会返回一个SelectionKey对象。这个对象包含了一些你感兴趣的属性：

1. interest集合

interest集合是你所选择的感兴趣的事件集合。可以通过SelectionKey读写interest集合。

int interestSet = selectionKey.interestOps();

boolean isInterestedInAccept = (interestSet & SelectionKey.OP\_ACCEPT) == SelectionKey.OP\_ACCEPT；

boolean isInterestedInConnect = interestSet & SelectionKey.OP\_CONNECT;

boolean isInterestedInRead = interestSet & SelectionKey.OP\_READ;

boolean isInterestedInWrite = interestSet & SelectionKey.OP\_WRITE;

用“位与”操作interest 集合和给定的SelectionKey常量，可以确定某个确定的事件是否在interest 集合中。

1. ready集合

ready 集合是通道已经准备就绪的操作的集合。在一次选择(Selection)之后，你会首先访问这个ready set。

int readySet = selectionKey.readyOps();

可以用像检测interest集合那样的方法，来检测channel中什么事件或操作已经就绪。但是，也可以使用以下四个方法，它们都会返回一个布尔类型：

selectionKey.isAcceptable();

selectionKey.isConnectable();

selectionKey.isReadable();

selectionKey.isWritable();

1. Channel

Channel channel = selectionKey.channel();

1. Selector

Selector selector = selectionKey.selector();

1. 附加的对象（可选）

可以将一个对象或者更多信息附着到SelectionKey上，这样就能方便的识别某个给定的通道。例如，可以附加 与通道一起使用的Buffer，或是包含聚集数据的某个对象。使用方法如下：

selectionKey.attach(theObject);

Object attachedObj = selectionKey.attachment();

还可以在用register()方法向Selector注册Channel的时候附加对象。如：

SelectionKey key = channel.register(selector, SelectionKey.OP\_READ, theObject);

##### Selector选择通道

一旦向Selector注册了一或多个通道，就可以调用几个重载的select()方法。这些方法返回你所感兴趣的事件（如连接、接受、读或写）已经准备就绪的那些通道。

select()方法：

1. int select()

阻塞到至少有一个通道在你注册的事件上就绪了。

1. int select(long timeout)

和select()一样，除了最长会阻塞timeout毫秒(参数)。

1. int selectNow()

不会阻塞，不管什么通道就绪都立刻返回（译者注：此方法执行非阻塞的选择操作。如果自从前一次选择操作后，没有通道变成可选择的，则此方法直接返回零。）。

select()方法返回的int值表示有多少通道已经就绪。亦即，自上次调用select()方法后有多少通道变成就绪状态。如果调用select()方法，因为有一个通道变成就绪状态，返回了1，若再次调用select()方法，如果另一个通道就绪了，它会再次返回1。如果对第一个就绪的channel没有做任何操作，现在就有两个就绪的通道，但在每次select()方法调用之间，只有一个通道就绪了。

##### selectedKeys()

一旦调用了select()方法，并且返回值表明有一个或更多个通道就绪了，然后可以通过调用selector的selectedKeys()方法，访问“已选择键集（selected key set）”中的就绪通道。如下所示：

Set selectedKeys = selector.selectedKeys();

当向Selector注册Channel时，Channel.register()方法会返回一个SelectionKey 对象。这个对象代表了注册到该Selector的通道。可以通过SelectionKey的selectedKeySet()方法访问这些对象。

可以遍历这个已选择的键集合来访问就绪的通道。如下：

Set selectedKeys = selector.selectedKeys();

Iterator keyIterator = selectedKeys.iterator();

while(keyIterator.hasNext()) {

SelectionKey key = keyIterator.next();

if(key.isAcceptable()) {

// a connection was accepted by a ServerSocketChannel.

} else if (key.isConnectable()) {

// a connection was established with a remote server.

} else if (key.isReadable()) {

// a channel is ready for reading

} else if (key.isWritable()) {

// a channel is ready for writing

}

}

这个循环遍历已选择键集中的每个键，并检测各个键所对应的通道的就绪事件。

注意每次迭代末尾的keyIterator.remove()调用。Selector不会自己从已选择键集中移除SelectionKey实例。必须在处理完通道时自己移除。下次该通道变成就绪时，Selector会再次将其放入已选择键集中。

SelectionKey.channel()方法返回的通道需要转型成你要处理的类型，如ServerSocketChannel或SocketChannel等。

##### wakeUp()

某个线程调用select()方法后阻塞了，即使没有通道已经就绪，也有办法让其从select()方法返回。只要让其它线程在第一个线程调用select()方法的那个对象上调用Selector.wakeup()方法即可。阻塞在select()方法上的线程会立马返回。

如果有其它线程调用了wakeup()方法，但当前没有线程阻塞在select()方法上，下个调用select()方法的线程会立即“醒来（wake up）”。

##### close()

用完Selector后调用其close()方法会关闭该Selector，且使注册到该Selector上的所有SelectionKey实例无效。通道本身并不会关闭。

##### 模板代码

打开一个Selector，注册一个通道注册到这个Selector上(通道的初始化过程略去),然后持续监控这个Selector的四种事件（接受，连接，读，写）是否就绪。

ServerSocketChannel ssc = ServerSocketChannel.open();

ssc.socket().bind(new InetSocketAddress("localhost", 8080));

ssc.configureBlocking(false);

Selector selector = Selector.open();

ssc.register(selector, SelectionKey.OP\_ACCEPT);

while(true) {

int readyNum = selector.select();

if (readyNum == 0) {

continue;

}

Set<SelectionKey> selectedKeys = selector.selectedKeys();

Iterator<SelectionKey> it = selectedKeys.iterator();

while(it.hasNext()) {

SelectionKey key = it.next();

if(key.isAcceptable()) {

// 接受连接

} else if (key.isReadable()) {

// 通道可读

} else if (key.isWritable()) {

// 通道可写

}

it.remove();

}

}

#### Channel

Java NIO的通道类似流，但又有些不同：

* 既可以从通道中读取数据，又可以写数据到通道。但流的读写通常是单向的。

通道可以异步地读写。

* 通道中的数据总是要先读到一个Buffer，或者总是要从一个Buffer中写入。



Channel的实现

这些是Java NIO中最重要的通道的实现：

* FileChannel：从文件中读写数据。
* DatagramChannel：能通过UDP读写网络中的数据。
* SocketChannel：能通过TCP读写网络中的数据。
* ServerSocketChannel：可以监听新进来的TCP连接，像Web服务器那样。对每一个新进来的连接都会创建一个SocketChannel。

**基本的 Channel 示例**

RandomAccessFile aFile = new RandomAccessFile("data/nio-data.txt", "rw");

FileChannel inChannel = aFile.getChannel();

ByteBuffer buf = ByteBuffer.allocate(48);

int bytesRead = inChannel.read(buf);

while (bytesRead != -1) {

System.out.println("Read " + bytesRead);

buf.flip();

while(buf.hasRemaining()){

System.out.print((char) buf.get());

}

buf.clear();

bytesRead = inChannel.read(buf);

}

aFile.close();

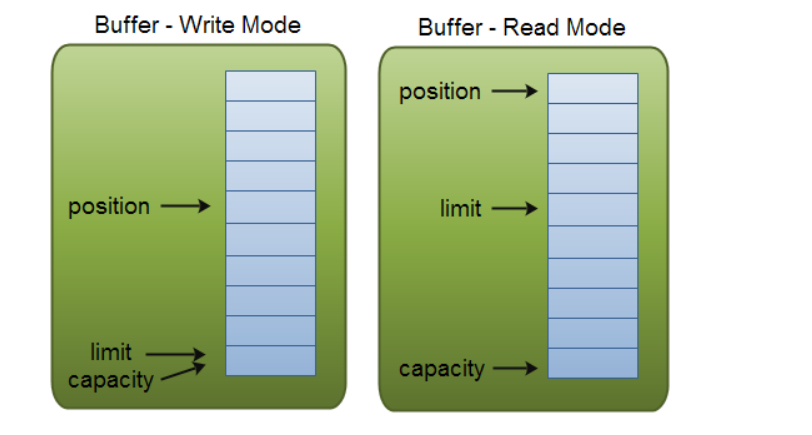
#### buffer

buffer作为I/O过程中的缓冲区，与channel进行直接交互。数据一般从缓冲区流向channel，或者从channel流向缓冲区。使用Buffer读写数据一般遵循以下四个步骤：

1. 写入数据到Buffer
2. 调用flip()方法
3. 从Buffer中读取数据
4. 调用clear()方法或者compact()方法

Buffer中三个属性：

* capacity：作为一个内存块，Buffer有一个固定的大小值，也叫“capacity”.你只能往里写capacity个byte、long，char等类型。一旦Buffer满了，需要将其清空（通过读数据或者清除数据）才能继续写数据往里写数据。
* position：当你写数据到Buffer中时，position表示当前的位置。初始的position值为0.当一个byte、long等数据写到Buffer后， position会向前移动到下一个可插入数据的Buffer单元。position最大可为capacity – 1。当读取数据时，也是从某个特定位置读。当将Buffer从写模式切换到读模式，position会被重置为0. 当从Buffer的position处读取数据时，position向前移动到下一个可读的位置。
* Limit：在写模式下，Buffer的limit表示你最多能往Buffer里写多少数据。 写模式下，limit等于Buffer的capacity。当切换Buffer到读模式时， limit表示你最多能读到多少数据。因此，当切换Buffer到读模式时，limit会被设置成写模式下的position值。换句话说，你能读到之前写入的所有数据（limit被设置成已写数据的数量，这个值在写模式下就是position）



当向buffer写入数据时，buffer会记录下写了多少数据。一旦要读取数据，需要通过flip()方法将Buffer从写模式切换到读模式。在读模式下，可以读取之前写入到buffer的所有数据。

一旦读完了所有的数据，就需要清空缓冲区，让它可以再次被写入。有两种方式能清空缓冲区：调用clear()或compact()方法。clear()方法会清空整个缓冲区。compact()方法只会清除已经读过的数据。任何未读的数据都被移到缓冲区的起始处，新写入的数据将放到缓冲区未读数据的后面。

##### Buffer类型

* ByteBuffer
* MappedByteBuffer
* CharBuffer
* DoubleBuffer
* FloatBuffer
* IntBuffer
* LongBuffer
* ShortBuffer

##### Buffer的分配

要想获得一个Buffer对象首先要进行分配。 每一个Buffer类都有一个allocate方法。下面是一个分配48字节capacity的ByteBuffer的例子。

ByteBuffer buf = ByteBuffer.allocate(48);

这是分配一个可存储1024个字符的CharBuffer：

CharBuffer buf = CharBuffer.allocate(1024);

##### Buffer写数据

写数据到Buffer有两种方式：

1. 从Channel写到Buffer

int bytesRead = inChannel.read(buf); //read into buffer.

1. 通过Buffer的put()方法写到Buffer里。

buf.put(127);

##### flip()方法

flip方法将Buffer从写模式切换到读模式。调用flip()方法会将position设回0，并将limit设置成之前position的值。

##### Buffer读

从Buffer中读取数据有两种方式：

1. 从Buffer读取数据到Channel。

int bytesWritten = inChannel.write(buf);

1. 使用get()方法从Buffer中读取数据。

byte aByte = buf.get();

##### rewind()方法

Buffer.rewind()将position设回0，所以你可以重读Buffer中的所有数据。limit保持不变，仍然表示能从Buffer中读取多少个元素（byte、char等）。

##### clear()与compact()方法

一旦读完Buffer中的数据，需要让Buffer准备好再次被写入。可以通过clear()或compact()方法来完成。

如果调用的是clear()方法，position将被设回0，limit被设置成 capacity的值。换句话说，Buffer 被清空了。Buffer中的数据并未清除，只是这些标记告诉我们可以从哪里开始往Buffer里写数据。

如果Buffer中有一些未读的数据，调用clear()方法，数据将“被遗忘”，意味着不再有任何标记会告诉你哪些数据被读过，哪些还没有。

如果Buffer中仍有未读的数据，且后续还需要这些数据，但是此时想要先先写些数据，那么使用compact()方法。

compact()方法将所有未读的数据拷贝到Buffer起始处。然后将position设到最后一个未读元素正后面。limit属性依然像clear()方法一样，设置成capacity。现在Buffer准备好写数据了，但是不会覆盖未读的数据。

##### mark()与reset()方法

通过调用Buffer.mark()方法，可以标记Buffer中的一个特定position。之后可以通过调用Buffer.reset()方法恢复到这个position。例如：

buffer.mark();

//call buffer.get() a couple of times, e.g. during parsing.

buffer.reset(); //set position back to mark.

##### equals()与compareTo()方法

equals()

当满足下列条件时，表示两个Buffer相等：

* 有相同的类型（byte、char、int等）。
* Buffer中剩余的byte、char等的个数相等。
* Buffer中所有剩余的byte、char等都相同。

equals只是比较Buffer的一部分，不是每一个在它里面的元素都比较。实际上，它只比较Buffer中的剩余元素。

compareTo()方法

compareTo()方法比较两个Buffer的剩余元素(byte、char等)， 如果满足下列条件，则认为一个Buffer“小于”另一个Buffer：

1. 第一个不相等的元素小于另一个Buffer中对应的元素 。
2. 所有元素都相等，但第一个Buffer比另一个先耗尽(第一个Buffer的元素个数比另一个少)。

## 集合框架



Java 集合框架主要包括两种类型的容器，一种是集合（Collection），存储一个元素集合，另一种是图（Map），存储键/值对映射。Collection 接口又有 3 种子类型，List、Set 和 Queue，再下面是一些抽象类，最后是具体实现类，常用的有 ArrayList、LinkedList、HashSet、LinkedHashSet、HashMap、LinkedHashMap 等等。

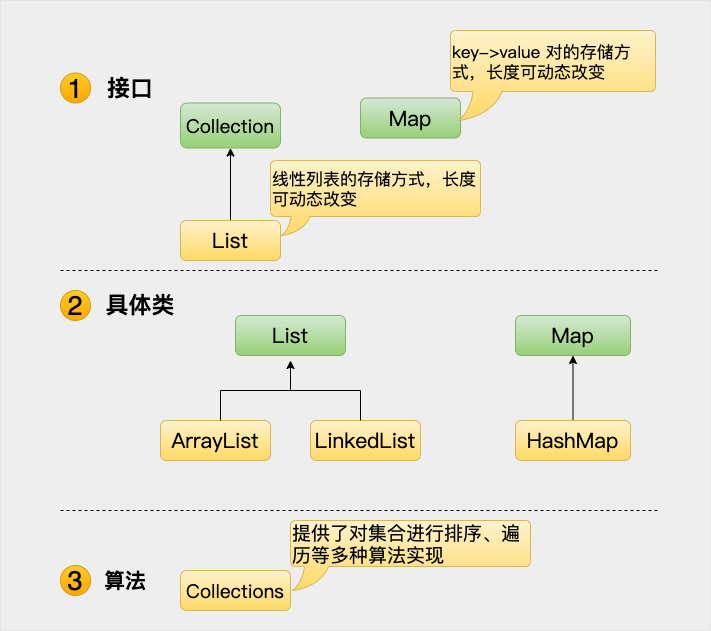
集合框架是一个用来代表和操纵集合的统一架构。所有的集合框架都包含如下内容：接口：是代表集合的抽象数据类型。例如 Collection、List、Set、Map 等。之所以定义多个接口，是为了以不同的方式操作集合对象

实现（类）：是集合接口的具体实现。从本质上讲，它们是可重复使用的数据结构，例如：ArrayList、LinkedList、HashSet、HashMap。

算法：是实现集合接口的对象里的方法执行的一些有用的计算，例如：搜索和排序。这些算法被称为多态，那是因为相同的方法可以在相似的接口上有着不同的实现。

除了集合，该框架也定义了几个 Map 接口和类。Map 里存储的是键/值对。尽管 Map 不是集合，但是它们完全整合在集合中。

### 集合框架体系



### 集合接口

|  |  |
| --- | --- |
| 序号 | 接口描述 |
| 1 | Collection 接口  Collection 是最基本的集合接口，一个 Collection 代表一组 Object，即 Collection 的元素, Java不提供直接继承自Collection的类，只提供继承于的子接口(如List和set)。  Collection 接口存储一组不唯一，无序的对象。 |
| 2 | List 接口  List接口是一个有序的 Collection，使用此接口能够精确的控制每个元素插入的位置，能够通过索引(元素在List中位置，类似于数组的下标)来访问List中的元素，第一个元素的索引为 0，而且允许有相同的元素。  List 接口存储一组不唯一，有序（插入顺序）的对象。 |
| 3 | Set  Set 具有与 Collection 完全一样的接口，只是行为上不同，Set 不保存重复的元素。  Set 接口存储一组唯一，无序的对象。底层实现与map一致，不过value是固定的值，key则是数据本身。 |
| 4 | SortedSet  继承于Set保存有序的集合。 |
| 5 | Map  Map 接口存储一组键值对象，提供key（键）到value（值）的映射。 |
| 6 | Map.Entry  描述在一个Map中的一个元素（键/值对）。是一个 Map 的内部接口。 |
| 7 | SortedMap  继承于 Map，使 Key 保持在升序排列。 |
| 8 | Enumeration  这是一个传统的接口和定义的方法，通过它可以枚举（一次获得一个）对象集合中的元素。这个传统接口已被迭代器取代。 |

### Set和List的区别

* Set 接口实例存储的是无序的，不重复的数据。List 接口实例存储的是有序的，可以重复的元素。
* Set检索效率低下，删除和插入效率高，插入和删除不会引起元素位置改变 **<实现类有HashSet,TreeSet>**。
* List和数组类似，可以动态增长，根据实际存储的数据的长度自动增长List的长度。查找元素效率高，插入删除效率低，因为会引起其他元素位置改变 **<实现类有ArrayList,LinkedList,Vector>** 。

### 集合实现类

Java提供了一套实现了Collection接口的标准集合类。其中一些是具体类，这些类可以直接拿来使用，而另外一些是抽象类，提供了接口的部分实现。

标准集合类汇总于下表：

|  |  |
| --- | --- |
| 序号 | 类描述 |
| 1 | AbstractCollection  实现了大部分的集合接口。 |
| 2 | AbstractList  继承于AbstractCollection 并且实现了大部分List接口。 |
| 3 | AbstractSequentialList  继承于 AbstractList ，提供了对数据元素的链式访问而不是随机访问。 |
| 4 | LinkedList  该类实现了List接口，允许有null（空）元素。主要用于创建链表数据结构，该类没有同步方法 |
| 5 | ArrayList  该类也是实现了List的接口，实现了可变大小的数组，随机访问和遍历元素时，提供更好的性能。该类也是非同步的,在多线程的情况下不要使用。ArrayList 增长当前长度的50%，插入删除效率低。 |
| 6 | AbstractSet  继承于AbstractCollection 并且实现了大部分Set接口。 |
| 7 | HashSet  该类实现了Set接口，不允许出现重复元素，不保证集合中元素的顺序，允许包含值为null的元素，但最多只能一个。 |
| 8 | LinkedHashSet  具有可预知迭代顺序的 Set 接口的哈希表和链接列表实现。 |
| 9 | TreeSet  该类实现了Set接口，可以实现排序等功能。 |
| 10 | AbstractMap  实现了大部分的Map接口。 |
| 11 | HashMap  HashMap 是一个散列表，它存储的内容是键值对(key-value)映射。  该类实现了Map接口，根据键的HashCode值存储数据，具有很快的访问速度，最多允许一条记录的键为null，不支持线程同步。 |
| 12 | TreeMap  继承了AbstractMap，并且使用一颗树。 |
| 13 | WeakHashMap  继承AbstractMap类，使用弱密钥的哈希表。 |
| 14 | LinkedHashMap  继承于HashMap，使用元素的自然顺序对元素进行排序. |
| 15 | IdentityHashMap  继承AbstractMap类，比较文档时使用引用相等。 |

以下部分仅需要了解

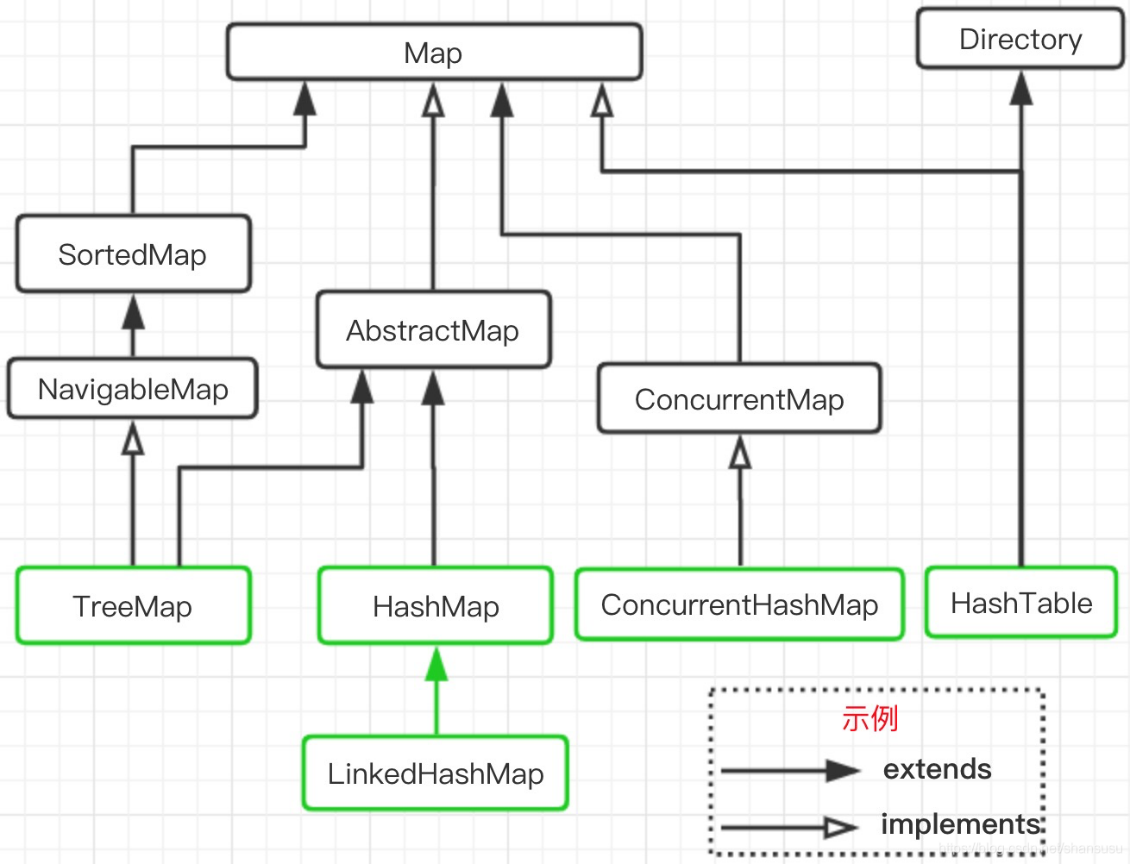
|  |  |
| --- | --- |
| 序号 | 类描述 |
| 1 | Vector  该类和ArrayList非常相似，但是该类是同步的，可以用在多线程的情况，该类允许设置默认的增长长度，默认扩容方式为原来的2倍。 |
| 2 | Stack  栈是Vector的一个子类，它实现了一个标准的后进先出的栈。 |
| 3 | Dictionary  Dictionary 类是一个抽象类，用来存储键/值对，作用和Map类相似。 |
| 4 | Hashtable  Hashtable 是 Dictionary(字典) 类的子类，位于 java.util 包中。 |
| 5 | Properties  Properties 继承于 Hashtable，表示一个持久的属性集，属性列表中每个键及其对应值都是一个字符串。 |
| 6 | BitSet  一个Bitset类创建一种特殊类型的数组来保存位值。BitSet中数组大小会随需要增加。 |

### ArrayList

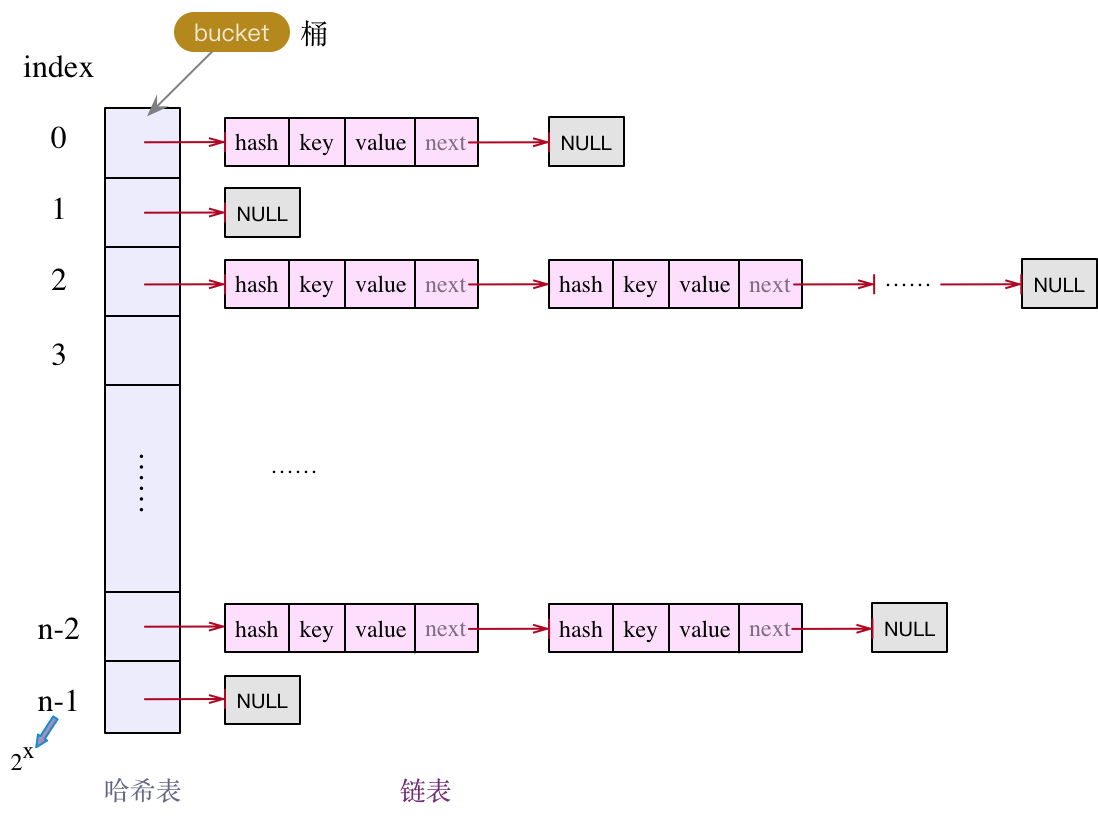
### LinkList

### HashMap

Map家族继承关系如下所示



在1.7中，HashMap如下所示：



这张图囊括了HashMap中最基础的几个点：

Java中HashMap的实现的基础数据结构是数组，每一对key->value的键值对组成Entity类以双向链表的形式存放到这个数组中元素在数组中的位置由key.hashCode()的值决定，如果两个key的哈希值相等，即发生了哈希碰撞，则这两个key对应的Entity将以链表的形式存放在数组中调用HashMap.get()的时候会首先计算key的值，继而在数组中找到key对应的位置，然后遍历该位置上的链表找相应的值。

当然这张图中没有体现出来的有两点：

为了提升整个HashMap的读取效率，当HashMap中存储的元素大小等于桶数组大小乘以负载因子的时候整个HashMap就要扩容，以减小哈希碰撞，在Java 8中如果桶数组的同一个位置上的链表数量超过一个定值，则整个链表有一定概率会转为一棵红黑树。

整体来看，整个HashMap中最重要的点有四个：初始化，数据寻址-hash方法，数据存储-put方法,扩容-resize方法，只要理解了这四个点的原理和调用时机，也就理解了整个HashMap的设计。

**那么问题来了：**

1. HashMap内部的bucket数组长度为什么一直都是2的整数次幂
2. HashMap默认的bucket数组是多大
3. HashMap什么时候开辟bucket数组占用内存
4. HashMap何时扩容？
5. 桶中的元素链表何时转换为红黑树，什么时候转回链表，为什么要这么设计？
6. Java 8中为什么要引进红黑树，是为了解决什么场景的问题？
7. HashMap如何处理key为null的键值对？

#### **初始化**

在JDK 8中，在调用new HashMap()的时候并没有分配数组堆内存，只是做了一些参数校验，初始化了一些常量

public HashMap(int initialCapacity, float loadFactor) {

if (initialCapacity < 0)

throw new IllegalArgumentException("Illegal initial capacity: " +

initialCapacity);

if (initialCapacity > MAXIMUM\_CAPACITY)

initialCapacity = MAXIMUM\_CAPACITY;

if (loadFactor <= 0 || Float.isNaN(loadFactor))

throw new IllegalArgumentException("Illegal load factor: " +

loadFactor);

this.loadFactor = loadFactor;

this.threshold = tableSizeFor(initialCapacity);

}

static final int tableSizeFor(int cap) {

int n = cap - 1;

n |= n >>> 1;

n |= n >>> 2;

n |= n >>> 4;

n |= n >>> 8;

n |= n >>> 16;

return (n < 0) ? 1 : (n >= MAXIMUM\_CAPACITY) ? MAXIMUM\_CAPACITY : n + 1;

}

tableSizeFor的作用是找到大于cap的最小的2的整数幂，我们假设n(注意是n，不是cap哈)对应的二进制为000001xxxxxx，其中x代表的二进制位是0是1我们不关心，

n |= n >>> 1;执行后n的值为：

n = 000001xxxxxx

n >>>1 = 0000001xxxxx

n |= n>>>1 = 0000011xxxxx

可以看到此时n的二进制最高两位已经变成了1（1和0或1异或都是1），再接着执行第二行代码：

n = 0000011xxxxx

n>>>2 = 000001111xxx

n |=n>>>2=000001111xxx

可见n的二进制最高四位已经变成了1，等到执行完代码n |= n >>> 16;之后，n的二进制最低位全都变成了1，也就是n = 2^x - 1其中x和n的值有关，如果没有超过MAXIMUM\_CAPACITY，最后会返回一个2的正整数次幂，因此tableSizeFor的作用就是保证返回一个比入参大的最小的2的正整数次幂。

在JDK 7中初始化的代码大体一致，在HashMap第一次put的时候会调用inflateTable计算桶数组的长度，但其算法并没有变：

// 第一次put时，初始化table

private void inflateTable(int toSize) {

// Find an power of 2 >= toSize

int capacity = roundUpToPowerOf2(toSize);

threshold = (int)Math.min(capacity \* loadFactor, MAXIMUM\_CAPACITY + 1);

table = new Entry(capacity);

initHashSeedAsNeeded(capacity);

}

HashMap什么时候开辟bucket数组占用内存？答案是在HashMap第一次put的时候，无论Java 8还是Java 7都是这样实现的。这里我们可以看到两个版本的实现中，桶数组的大小都是2的正整数幂

#### Hash

在HashMap这个特殊的数据结构中，hash函数承担着寻址定址的作用，其性能对整个HashMap的性能影响巨大，那什么才是一个好的hash函数呢？

* 计算出来的哈希值足够散列，能够有效减少哈希碰撞
* 本身能够快速计算得出，因为HashMap每次调用get和put的时候都会调用hash方法

下面是Java 8中的实现：

static final int hash(Object key) {

int h;

return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);

}

这里比较重要的是(h = key.hashCode()) ^ (h >>> 16)，这个位运算其实是将key.hashCode()计算出来的hash值的高16位与低16位继续异或

hash函数的作用是用来确定key在桶数组中的位置的，在JDK中为了更好的性能，通常会这样写：

index =(table.length - 1) & key.hash();

table.length是一个2的正整数次幂，类似于000100000，这样的值减一就成了000011111，通过位运算可以高效寻址，这也回答了前文中提到的一个问题，HashMap内部的bucket数组长度为什么一直都是2的整数次幂？好处之一就是可以通过构造位运算快速寻址定址。

既然计算出来的哈希值都要与table.length - 1做与运算，那就意味着计算出来的hash值只有低位有效，这样会加大碰撞几率，因此让高16位与低16位做异或，让低位保留部分高位信息，减少哈希碰撞。

Java 7中对hash的实现：

final int hash(Object k) {

int h = hashSeed;

if (0 != h && k instanceof String) {

return sun.misc.Hashing.stringHash32((String) k);

}

h ^= k.hashCode();

// This function ensures that hashCodes that differ only by

// constant multiples at each bit position have a bounded

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

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

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

}

Java 7中为了避免hash值的高位信息丢失，做了更加复杂的异或运算，但是基本出发点都是一样的，都是让哈希值的低位保留部分高位信息，减少哈希碰撞。

#### Put

JAVA 8 中put这个方法的思路分为以下几步：

1. 调用key的hashCode方法计算哈希值，并据此计算出数组下标index
2. 如果发现当前的桶数组为null，则调用resize()方法进行初始化
3. 如果没有发生哈希碰撞，则直接放到对应的桶中
4. 如果发生哈希碰撞，且节点已经存在，就替换掉相应的value
5. 如果发生哈希碰撞，且桶中存放的是树状结构，则挂载到树上
6. 如果碰撞后为链表，添加到链表尾，如果链表超度超过TREEIFY\_THRESHOLD默认是8，则将链表转换为树结构
7. 数据put完成后，如果HashMap的总数超过threshold就要resize

具体代码以及注释如下：

public V put(K key, V value) {

// 调用上文我们已经分析过的hash方法

return putVal(hash(key), key, value, false, true);

}

final V putVal(int hash, K key, V value, boolean onlyIfAbsent,

boolean evict) {

Node<K,V>[] tab; Node<K,V> p; int n, i;

if ((tab = table) == null || (n = tab.length) == 0)

// 第一次put时，会调用resize进行桶数组初始化

n = (tab = resize()).length;

// 根据数组长度和哈希值相与来寻址，原理上文也分析过

if ((p = tab[i = (n - 1) & hash]) == null)

// 如果没有哈希碰撞，直接放到桶中

tab[i] = newNode(hash, key, value, null);

else {

Node<K,V> e; K k;

if (p.hash == hash &&

((k = p.key) == key || (key != null && key.equals(k))))

// 哈希碰撞，且节点已存在，直接替换

e = p;

else if (p instanceof TreeNode)

// 哈希碰撞，树结构

e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);

else {

// 哈希碰撞，链表结构

for (int binCount = 0; ; ++binCount) {

if ((e = p.next) == null) {

p.next = newNode(hash, key, value, null);

// 链表过长，转换为树结构

if (binCount >= TREEIFY\_THRESHOLD - 1) // -1 for 1st

treeifyBin(tab, hash);

break;

}

if (e.hash == hash &&

((k = e.key) == key || (key != null && key.equals(k))))

// 如果节点已存在，则跳出循环

break;

// 否则，指针后移，继续后循环

p = e;

}

}

if (e != null) { // existing mapping for key

// 对应着上文中节点已存在，跳出循环的分支

// 直接替换

V oldValue = e.value;

if (!onlyIfAbsent || oldValue == null)

e.value = value;

afterNodeAccess(e);

return oldValue;

}

}

++modCount;

if (++size > threshold)

// 如果超过阈值，还需要扩容

resize();

afterNodeInsertion(evict);

return null;

}

相比之下Java 7中的put方法就简单不少

public V put(K key, V value) {

// 如果 key 为 null，调用 putForNullKey 方法进行处理

if (key == null)

return putForNullKey(value);

int hash = hash(key.hashCode());

int i = indexFor(hash, table.length);

for (Entry<K, V> e = table[i]; e != null; e = e.next) {

Object k;

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

|| key.equals(k))) {

V oldValue = e.value;

e.value = value;

e.recordAccess(this);

return oldValue;

}

}

modCount++;

addEntry(hash, key, value, i);

return null;

}

void addEntry(int hash, K key, V value, int bucketIndex) {

Entry<K, V> e = table[bucketIndex]; // ①

table[bucketIndex] = new Entry<K, V>(hash, key, value, e);

if (size++ >= threshold)

resize(2 \* table.length); // ②

}

这里有一个小细节，HashMap允许putkey为null的键值对，但是这样的键值对都放到了桶数组的第0个桶中。

#### resize()

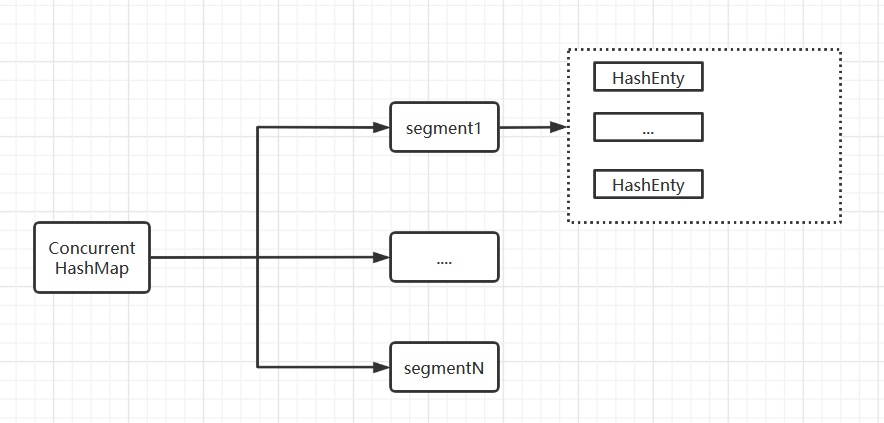
resize是整个HashMap中最复杂的一个模块，如果在put数据之后超过了threshold的值，则需要扩容，扩容意味着桶数组大小变化，我们在前文中分析过，HashMap寻址是通过index =(table.length - 1) & key.hash();来计算的，现在table.length发生了变化，势必会导致部分key的位置也发生了变化，HashMap是如何设计的呢？

这里就涉及到桶数组长度为2的正整数幂的第二个优势了：当桶数组长度为2的正整数幂时，如果桶发生扩容（长度翻倍），则桶中的元素大概只有一半需要切换到新的桶中，另一半留在原先的桶中就可以，并且这个概率可以看做是均等的。

#### 总结

1. HashMap内部的bucket数组长度为什么一直都是2的整数次幂  
   答：这样做有两个好处，第一，可以通过(table.length - 1) & key.hash()这样的位运算快速寻址，第二，在HashMap扩容的时候可以保证同一个桶中的元素均匀的散列到新的桶中，具体一点就是同一个桶中的元素在扩容后一般留在原先的桶中，一般放到了新的桶中。
2. HashMap默认的bucket数组是多大  
   答：默认是16，即时指定的大小不是2的整数次幂，HashMap也会找到一个最近的2的整数次幂来初始化桶数组。
3. HashMap什么时候开辟bucket数组占用内存  
   答：在第一次put的时候调用resize方法
4. HashMap何时扩容？  
   答：当HashMap中的元素熟练超过阈值时，阈值计算方式是capacity \* loadFactor，在HashMap中loadFactor是0.75
5. 桶中的元素链表何时转换为红黑树，什么时候转回链表，为什么要这么设计？  
   答： 当同一个桶中的元素数量大于等于8的时候元素中的链表转换为红黑树，反之，当桶中的元素数量小于等于6的时候又会转为链表，这样做的原因是避免红黑树和链表之间频繁转换，引起性能损耗
6. Java 8中为什么要引进红黑树，是为了解决什么场景的问题？  
   答：引入红黑树是为了避免hash性能急剧下降，引起HashMap的读写性能急剧下降的场景，正常情况下，一般是不会用到红黑树的，在一些极端场景下，假如客户端实现了一个性能拙劣的hashCode方法，可以保证HashMap的读写复杂度不会低于O(lgN)  
   **public** **int** **hashCode**() {  
   **return** 1;  
   }
7. HashMap如何处理key为null的键值对？  
   答：放置在桶数组中下标为0的桶中

### ConcurrentHashMap



对于Java 7来说，这张图已经能完全把ConcurrentHashMap的架构说清楚了：

1. ConcurrentHashMap是一个线程安全的Map实现，其读取不需要加锁，通过引入Segment，可以做到写入的时候加锁力度足够小
2. 由于引入了Segment，ConcurrentHashMap在读取和写入的时候需要需要做两次哈希，但这两次哈希换来的是更小的临界区，也就意味着可以支持更高的并发
3. 每个桶数组中的key-value对仍然以链表的形式存放在桶中，这一点和HashMap是一致的。

关于ConcurrentHashMap有如下几个问题：

1. ConcurrentHashMap的哪些操作需要加锁？
2. ConcurrentHashMap的无锁读是如何实现的？
3. 在多线程的场景下调用size（）方法获取ConcurrentHashMap的大小有什么挑战？ConcurrentHashMap是怎么解决的？
4. 在有Segment存在的前提下，是如何扩容的？

#### 初始化

public ConcurrentHashMap(int initialCapacity, float loadFactor, int concurrencyLevel) {

if (!(loadFactor > 0) || initialCapacity < 0 || concurrencyLevel <= 0)

throw new IllegalArgumentException();

if (concurrencyLevel > MAX\_SEGMENTS)

concurrencyLevel = MAX\_SEGMENTS;

// Find power-of-two sizes best matching arguments

int sshift = 0;

int ssize = 1;

// 保证ssize是大于concurrencyLevel的最小的2的整数次幂

while (ssize < concurrencyLevel) {

++sshift;

ssize <<= 1;

}

// 寻址需要两次哈希，哈希的高位用于确定segment，低位用户确定桶数组中的元素

this.segmentShift = 32 - sshift;

this.segmentMask = ssize - 1;

if (initialCapacity > MAXIMUM\_CAPACITY)

initialCapacity = MAXIMUM\_CAPACITY;

int c = initialCapacity / ssize;

if (c \* ssize < initialCapacity)

++c;

int cap = MIN\_SEGMENT\_TABLE\_CAPACITY;

while (cap < c)

cap <<= 1;

Segment<K,V> s0 = new Segment<K,V>(loadFactor, (int)(cap \* loadFactor), (HashEntry<K,V>[])new HashEntry[cap]);

Segment<K,V>[] ss = (Segment<K,V>[])new Segment[ssize];

UNSAFE.putOrderedObject(ss, SBASE, s0); // ordered write of segments[0]

this.segments = ss;

}

初始化方法中做了三件重要的事：

1. 确定了segments的数组的大小ssize，ssize根据入参concurrencyLevel确定，取大于concurrencyLevel的最小的2的整数次幂
2. 确定哈希寻址时的偏移量，这个偏移量在确定元素在segment数组中的位置时会用到
3. 初始化segment数组中的第一个元素，元素类型为HashEntry的数组，这个数组的长度为initialCapacity / ssize，即初始化大小除以segment数组的大小，segment数组中的其他元素在后续put操作时参考第一个已初始化的实例初始化

static final class HashEntry<K,V> {

final int hash;

final K key;

volatile V value;

volatile HashEntry<K,V> next;

HashEntry(int hash, K key, V value, HashEntry<K,V> next) {

this.hash = hash;

this.key = key;

this.value = value;

this.next = next;

}

final void setNext(HashEntry<K,V> n) {

UNSAFE.putOrderedObject(this, nextOffset, n);

}

}

这里的HashEntry和HashMap中的HashEntry作用是一样的，它是ConcurrentHashMap的数据项，这里要注意两个细节：

**细节一：**

HashEntry的成员变量value和next是被关键字volatile修饰的，也就是说所有线程都可以及时检查到其他线程对这两个变量的改变，因而可以在不加锁的情况下读取到这两个引用的最新值

**细节二：**

HashEntry的setNext方法中调用了UNSAFE.putOrderedObject，这个接口是属于sun安全库中的api，并不是J2SE的一部分，它的作用和volatile恰恰相反，调用这个api设值是使得volatile修饰的变量延迟写入主存，那到底是什么时候写入主存呢？

JMM对此规定：  
对一个变量执行unlock操作之前，必须先把此变量同步到主内存中（执行store和write操作）

#### 哈希

由于引入了segment，因此不管是调用get方法读还是调用put方法写，都需要做两次哈希，还记得在上文我们讲初始化的时候系统做了一件重要的事：

* 确定哈希寻址时的偏移量，这个偏移量在确定元素在segment数组中的位置时会用到

this.segmentShift = 32 - sshift;

这里用32去减是因为int型的长度是32，有了segmentShift，ConcurrentHashMap是如何做第一次哈希的呢？

public V put(K key, V value) {

Segment<K,V> s;

if (value == null)

throw new NullPointerException();

int hash = hash(key);

// 变量j代表着数据项处于segment数组中的第j项

int j = (hash >>> segmentShift) & segmentMask;

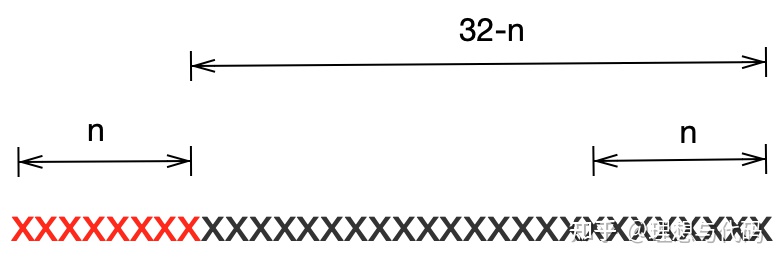
// 如果segment[j]为null,则下面的这个方法负责初始化之

s = ensureSegment(j);

return s.put(key, hash, value, false);

}

我们以put方法为例，变量j代表着数据项处于segment数组中的第j项。如下图所示假如segment数组的大小为2的n次方，则hash >>> segmentShift正好取了key的哈希值的高n位，再与掩码segmentMask相与相当与仍然用key的哈希的高位来确定数据项在segment数组中的位置。



hash方法与非线程安全的HashMap相似

**细节三：**

在延迟初始化Segment数组时，作者采用了CAS避免了加锁，而且CAS可以保证最终的初始化只能被一个线程完成。在最终决定调用CAS进行初始化前又做了两次检查，第一次检查可以避免重复初始化tab数组，而第二次检查则可以避免重复初始化Segment对象，每一行代码作者都有详细的考虑。

private Segment<K,V> ensureSegment(int k) {

final Segment<K,V>[] ss = this.segments;

long u = (k << SSHIFT) + SBASE; // raw offset 实际的字节偏移量

Segment<K,V> seg;

if ((seg = (Segment<K,V>)UNSAFE.getObjectVolatile(ss, u)) == null) {

Segment<K,V> proto = ss[0]; // use segment 0 as prototype

int cap = proto.table.length;

float lf = proto.loadFactor;

int threshold = (int)(cap \* lf);

HashEntry<K,V>[] tab = (HashEntry<K,V>[])new HashEntry[cap];

if ((seg = (Segment<K,V>)UNSAFE.getObjectVolatile(ss, u)) == null) { // recheck 再检查一次是否已经被初始化

Segment<K,V> s = new Segment<K,V>(lf, threshold, tab);

while ((seg = (Segment<K,V>)UNSAFE.getObjectVolatile(ss, u)) == null) {

if (UNSAFE.compareAndSwapObject(ss, u, null, seg = s)) // 使用 CAS 确保只被初始化一次

break;

}

}

}

return seg;

}

#### put方法

final V put(K key, int hash, V value, boolean onlyIfAbsent) {

HashEntry<K,V> node = tryLock() ? null : scanAndLockForPut(key, hash, value);

V oldValue;

try {

HashEntry<K,V>[] tab = table;

int index = (tab.length - 1) & hash;

HashEntry<K,V> first = entryAt(tab, index);

for (HashEntry<K,V> e = first;;) {

if (e != null) {

K k; // 如果找到key相同的数据项，则直接替换

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

oldValue = e.value;

if (!onlyIfAbsent) {

e.value = value;

++modCount;

}

break;

}

e = e.next;

}

else {

if (node != null)

// node不为空说明已经在自旋等待时初始化了，注意调用的是setNext，不是直接操作next

node.setNext(first);

else

// 否则，在这里新建一个HashEntry

node = new HashEntry<K,V>(hash, key, value, first);

int c = count + 1; // 先加1

if (c > threshold && tab.length < MAXIMUM\_CAPACITY)

rehash(node);

else

// 将新节点写入，注意这里调用的方法有门道

setEntryAt(tab, index, node);

++modCount;

count = c;

oldValue = null;

break;

}

}

} finally {

unlock();

}

return oldValue;

}

**细节四：**

CPU的调度是公平的，好不容易轮到的时间片如果因为获取不到锁就将本线程挂起无疑会降低本线程的效率，更何况挂起之后还要重新调度，切换上下文，又是一笔不小的开销。如果可以遇见其他线程占有锁的时间不会很长，采用自旋将会是一个比较好的选择，在这里面也有一个权衡，如果别的线程占有锁的时间过长，反而是挂起阻塞等待性能好一点，我们来看下ConcurrentHashMap的做法：

private HashEntry<K,V> scanAndLockForPut(K key, int hash, V value) {

HashEntry<K,V> first = entryForHash(this, hash);

HashEntry<K,V> e = first;

HashEntry<K,V> node = null;

int retries = -1; // negative while locating node

while (!tryLock()) { // 自旋等待

HashEntry<K,V> f; // to recheck first below

if (retries < 0) {

if (e == null) { // 这个桶中还没有写入k-v项

if (node == null) // speculatively create node 直接创建一个新的节点

node = new HashEntry<K,V>(hash, key, value, null);

retries = 0;

}

// key值相等，直接跳出去尝试获取锁

else if (key.equals(e.key))

retries = 0;

else // 遍历链表

e = e.next;

}

else if (++retries > MAX\_SCAN\_RETRIES) {

// 自旋等待超过一定次数之后只能挂起线程，阻塞等待了

lock();

break;

}

else if ((retries & 1) == 0 && (f = entryForHash(this, hash)) != first) {

// 如果头节点改变了，则重置次数，继续自旋等待

e = first = f;

retries = -1;

}

}

return node;

}

ConcurrentHashMap的策略是自旋MAX\_SCAN\_RETRIES次，如果还没有获取到锁则调用lock挂起阻塞等待，当然如果其他线程采用头插法改变了链表的头结点，则重置自旋等待次数。

**细节五：**

要知道，如果要从编码的角度提升系统的并发度，一个黄金法则就是减少并发临界区的大小。在scanAndLockForPut这个方法的设计上，有个小细节让我眼前一亮，就是在自旋的过程中初始化了一个HashEntry，这样做的好处就是线程在拿到锁之后不用初始化HashEntry了，占有锁的时间相应减小，进而提升并发，可见Doug Lea对并发的理解程度之深，尽可能的优化每一处细节。

**细节六：**

在put方法的开头，有这么一行不起眼的代码：

HashEntry<K,V>[] tab = table;

看起来好像就是简单的临时变量赋值，其实大有来头，我们看一下table的声明：

transient volatile HashEntry<K,V>[] table;

table变量被关键字volatile修饰，CPU在处理volatile修饰的变量的时候采取下面的行为：

嗅探

每个处理器通过嗅探在总线上传播的数据来检查自己缓存的值是不是过期了，当处理器发现自己缓存行对应的内存地址被修改，就会将当前处理器的缓存行设置成无效状态，当处理器对这个数据进行修改操作的时候，会重新从系统内存中把数据读到处理器缓存里

因此直接读取这类变量的读取和写入比普通变量的性能消耗更大，因此在put方法的开头将table变量赋值给一个普通的本地变量目的是为了消除volatile带来的性能损耗。这里就有另外一个问题：那这样做会不会导致table的语义改变，让别的线程读取不到最新的值呢？

**细节七：**

static final <K,V> HashEntry<K,V> entryAt(HashEntry<K,V>[] tab, int i) {

return (tab == null) ? null : (HashEntry<K,V>) UNSAFE.getObjectVolatile(tab, ((long)i << TSHIFT) + TBASE);

}

这个方法的底层会调用UNSAFE.getObjectVolatile，这个方法的目的就是对于普通变量读取也能像volatile修饰的变量那样读取到最新的值，在前文中我们分析过，由于变量tab现在是一个普通的临时变量，如果直接调用tab[i]则大概率是拿不到最新的首节点的。细心的读者读到这里可能会想：Doug Lea是不是糊涂了，兜兜转换不是回到了原点么，为啥不刚开始就操作volatile变量呢，费了这老大劲。我们继续往下看。

**细节八：**

在put方法的实现中，如果链表中没有key值相等的数据项，则会把新的数据项插入到链表头写入到数组中，其中调用的方法是：

static final <K,V> void setEntryAt(HashEntry<K,V>[] tab, int i, HashEntry<K,V> e) {

UNSAFE.putOrderedObject(tab, ((long)i << TSHIFT) + TBASE, e);

}

putOrderedObject这个接口写入的数据不会马上被其他线程获取到，而是在put方法最后调用unclock后才会对其他线程可见，参见前文中对JMM的描述：

*对一个变量执行unlock操作之前，必须先把此变量同步到主内存中（执行store和write操作）*

这样的好处有两个，第一是性能，因为在持有锁的临界区不需要有同步主存的操作，因此持有锁的时间更短。第二是保证了数据的一致性，在put操作的finally语句执行完之前，put新增的数据是不对其他线程展示的，这是ConcurrentHashMap实现无锁读的关键原因。

我们在这里稍微总结一下put方法里面最重要的三个细节，首先将volatile变量转为普通变量提升性能，因为在put中需要读取到最新的数据，因此接下来调用UNSAFE.getObjectVolatile获取到最新的头结点，但是通过调用UNSAFE.putOrderedObject让变量写入主存的时间延迟到put方法的结尾，一来缩小临界区提升性能，而来也能保证其他线程读取到的是完整数据。

**细节九：**

如果put真的需要往链表头插入数据项，那也得注意了，ConcurrentHashMap相应的语句是：

node.setNext(first);

final void setNext(HashEntry<K,V> n) {

UNSAFE.putOrderedObject(this, nextOffset, n);

}

因为next变量是用volatile关键字修饰的，这里调用UNSAFE.putOrderedObject相当于是改变了volatile的语义，这里面的考量有两个，第一个仍然是性能，这样的实现性能明显更高，这一点前文已经详细的分析过，第二点是考虑了语义的一致性，对于put方法来说因为其调用的是UNSAFE.getObjectVolatile，仍然能获取到最新的数据，对于get方法，在put方法未结束之前，是不希望不完整的数据被其他线程通过get方法读取的，这也是合理的。

#### resize扩容

private void rehash(HashEntry<K,V> node) {

HashEntry<K,V>[] oldTable = table;

int oldCapacity = oldTable.length;

int newCapacity = oldCapacity << 1;

threshold = (int)(newCapacity \* loadFactor);

HashEntry<K,V>[] newTable = (HashEntry<K,V>[]) new HashEntry[newCapacity];

int sizeMask = newCapacity - 1;

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

HashEntry<K,V> e = oldTable[i];

if (e != null) {

HashEntry<K,V> next = e.next;

int idx = e.hash & sizeMask;

if (next == null) // Single node on list 只有一个节点，简单处理

newTable[idx] = e;

else {

HashEntry<K,V> lastRun = e;

int lastIdx = idx;

// 保证下文中newTable[k]不会为null

for (HashEntry<K,V> last = next;

last != null;

last = last.next) {

int k = last.hash & sizeMask;

if (k != lastIdx) {

lastIdx = k;

lastRun = last;

}

}

newTable[lastIdx] = lastRun;

// Clone remaining nodes 对标记之前的不能重用的节点进行复制，再重新添加到新数组对应的hash桶中去

for (HashEntry<K,V> p = e; p != lastRun; p = p.next) {

V v = p.value;

int h = p.hash;

int k = h & sizeMask;

HashEntry<K,V> n = newTable[k];

newTable[k] = new HashEntry<K,V>(h, p.key, v, n);

}

}

}

}

int nodeIndex = node.hash & sizeMask; // add the new node 部分的put功能，把新节点添加到链表的最前面

node.setNext(newTable[nodeIndex]);

newTable[nodeIndex] = node;

table = newTable;

}

在整个桶数组长度为2的正整数幂的情况下，扩容前同一个桶中的元素在扩容后只会分布在两个桶中，其中一个桶的下标保持不变，我们称之为旧桶，另一个桶的下标为旧桶下标加上旧的容量，我们称之为新桶，其实第一个for循环的目的就是在一个链表中找到最后一个应该移到新桶的数据项，直接移到新桶中，这样做是为了保证后面调用HashEntry<K,V> n = newTable[k];的时候不会读取到null。第二个for就比较简单了，将所有的数据项移到新的桶数组中，当所有的操作完成之后才将newTable赋值给table。

rehash方法中是没有加锁的，并不是说调用这个方法不需要加锁，作者是在外层加了锁，这一点需要注意。

#### size方法

单线程环境下hashmap使用一个全局的变量解决，同样的方案当然也可以在多线程场景下使用，不过要在多线程环境下读取全局变量又会陷入到无尽的“锁”中。

public int size() {

final Segment<K,V>[] segments = this.segments;

int size;

boolean overflow; // true if size overflows 32 bits

long sum; // sum of modCounts

long last = 0L; // previous sum

int retries = -1; // first iteration isn't retry

try {

for (;;) {

if (retries++ == RETRIES\_BEFORE\_LOCK) {

for (int j = 0; j < segments.length; ++j)

ensureSegment(j).lock(); // force creation

}

sum = 0L;

size = 0;

overflow = false;

for (int j = 0; j < segments.length; ++j) {

Segment<K,V> seg = segmentAt(segments, j);

if (seg != null) {

sum += seg.modCount;

int c = seg.count;

if (c < 0 || (size += c) < 0)

overflow = true;

}

}

if (sum == last)

break;

last = sum;

}

} finally {

if (retries > RETRIES\_BEFORE\_LOCK) {

for (int j = 0; j < segments.length; ++j)

segmentAt(segments, j).unlock();

}

}

return overflow ? Integer.MAX\_VALUE : size;

}

modCount，它的主要作用就是记录整个Segment中写入操作的次数，因为写入操作是会影响整个ConcurrentHashMap的大小的。因为在读取ConcurrentHashMap大小的时候需要保证读到的是最新的值，因此其调用了UNSAFE.getObjectVolatile这个方法，虽然这个方法的性能比普通变量要差，但是比起全局加锁，可好多了。

static final <K,V> Segment<K,V> segmentAt(Segment<K,V>[] ss, int j) {

long u = (j << SSHIFT) + SBASE; // 计算实际的字节偏移量

return ss == null ? null : (Segment<K,V>) UNSAFE.getObjectVolatile(ss, u);

}

**细节十：**

在size方法的设计上，ConcurrentHashMap先尝试无锁的方法，如果两次遍历所有segment数组的时候整个ConcurrentHashMap没有发生写入操作，则直接返回每个segment数组的size()之和，否则重新遍历，如果写入操作频繁，则不得已加锁处理，这里的加锁相当于是一个全局的锁，因为对segment数组的每一个元素都加了锁。那如何判断整个ConcurrentHashMap的写入是否频繁呢？就看无锁重试的次数，当无锁重试的次数超过阈值的话就全局加锁处理。

#### 总结

在看完ConcurrentHashMap中的这些细节之后我们尝试回答一下文章开头提出来的问题：

1. ConcurrentHashMap的哪些操作需要加锁？  
   答：只有写入操作才需要加锁，读取操作不需要加锁
2. ConcurrentHashMap的无锁读是如何实现的？  
   答：首先HashEntry中的value和next都是有volatile修饰的，其次在写入操作的时候通过调用UNSAFE库延迟同步了主存，保证了数据的一致性
3. 在多线程的场景下调用size（）方法获取ConcurrentHashMap的大小有什么挑战？ConcurrentHashMap是怎么解决的？  
   答：size()具有全局的语义，如何能保证在不加全局锁的情况下读取到全局状态的值是一个很大的挑战，ConcurrentHashMap通过查看两次无锁读中间是否发生了写入操作来决定读取到的size()是否可信，如果写入操作频繁，则再退化为全局加锁读取。
4. 在有Segment存在的前提下，是如何扩容的？  
   答：segment数组的大小在一开始初始化的时候就已经决定了，扩容主要扩的是HashEntry数组，基本的思路与HashTable一致，但这是一个线程不安全方法，调用之前需要加锁。

### ConcurrentHashMap 1.8版本

Java 7中ConcurrentHashMap为了提升性能使用了很多的编程技巧，但是引入Segment的设计还是有很大的改进空间的，Java 7中ConcurrrentHashMap的设计有下面这几个可以改进的点：

1. Segment在扩容的时候非扩容线程对本Segment的写操作时都要挂起等待的
2. 对ConcurrentHashMap的读操作需要做两次哈希寻址，在读多写少的情况下其实是有额外的性能损失的
3. 尽管size()方法的实现中先尝试无锁读，但是如果在这个过程中有别的线程做写入操作，那调用size()的这个线程就会给整个ConcurrentHashMap加锁，这是整个ConcurrrentHashMap唯一一个全局锁，这点对底层的组件来说还是有性能隐患的
4. 极端情况下（比如客户端实现了一个性能很差的哈希函数）get()方法的复杂度会退化到O(n)。

针对1和2，在Java 8的设计是废弃了Segment的使用，将悲观锁的粒度降低至桶维度，因此调用get的时候也不需要再做两次哈希了。size()的设计是Java 8版本中最大的亮点，

#### 准备知识

ForwardingNode

static final class ForwardingNode<K,V> extends Node<K,V> {

final Node<K,V>[] nextTable;

ForwardingNode(Node<K,V>[] tab) {

// MOVED = -1，ForwardingNode的哈希值为-1

super(MOVED, null, null, null);

this.nextTable = tab;

}

}

除了普通的Node和TreeNode之外，ConcurrentHashMap还引入了一个新的数据类型ForwardingNode，我们这里只展示他的构造方法，ForwardingNode的作用有两个：

* 在动态扩容的过程中标志某个桶已经被复制到了新的桶数组中
* 如果在动态扩容的时候有get方法的调用，则ForwardingNode将会把请求转发到新的桶数组中，以避免阻塞get方法的调用，ForwardingNode在构造的时候会将扩容后的桶数组nextTable保存下来。

#### UNSAFE.compareAndSwap\*\*\*

这是在Java 8版本的ConcurrentHashMap实现CAS的工具，以int类型为例其方法定义如下：

public final native boolean compareAndSwapInt(Object o, long offset,

int expected,

int x);

相应的语义为：

如果对象o起始地址偏移量为offset的值等于expected，则将该值设为x，并返回true表明更新成功，否则返回false，表明CAS失败

#### 初始化

public ConcurrentHashMap(int initialCapacity, float loadFactor, int concurrencyLevel) {

if (!(loadFactor > 0.0f) || initialCapacity < 0 || concurrencyLevel <= 0) // 检查参数

throw new IllegalArgumentException();

if (initialCapacity < concurrencyLevel)

initialCapacity = concurrencyLevel;

long size = (long)(1.0 + (long)initialCapacity / loadFactor);

int cap = (size >= (long)MAXIMUM\_CAPACITY) ?

MAXIMUM\_CAPACITY : tableSizeFor((int)size); // tableSizeFor，求不小于size的 2^n的算法，jdk1.8的HashMap中说过

this.sizeCtl = cap;

}

即使是最复杂的一个初始化方法代码也是比较简单的，这里我们只需要注意两个点：

* concurrencyLevel在Java 7中是Segment数组的长度，由于在Java 8中已经废弃了Segment，因此concurrencyLevel只是一个保留字段，无实际意义
* sizeCtl这个值第一次出现，这个值如果等于-1则表明系统正在初始化，如果是其他负数则表明系统正在扩容，在扩容时sizeCtl二进制的低十六位等于扩容的线程数加一，高十六位（除符号位之外）包含桶数组的大小信息

#### put方法

public V put(K key, V value) {

return putVal(key, value, false);

}

put方法将调用转发到putVal方法：

final V putVal(K key, V value, boolean onlyIfAbsent) {

if (key == null || value == null) throw new NullPointerException();

int hash = spread(key.hashCode());

int binCount = 0;

for (Node<K,V>[] tab = table;;) {

Node<K,V> f; int n, i, fh;

// 【A】延迟初始化

if (tab == null || (n = tab.length) == 0)

tab = initTable();

// 【B】当前桶是空的，直接更新

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

}

// 【C】如果当前的桶的第一个元素是一个ForwardingNode节点，则该线程尝试加入扩容

else if ((fh = f.hash) == MOVED)

tab = helpTransfer(tab, f);

// 【D】否则遍历桶内的链表或树，并插入

else {

// 暂时折叠起来，后面详细看

}

}

// 【F】流程走到此处，说明已经put成功，map的记录总数加一

addCount(1L, binCount);

return null;

}

从整个代码结构上来看流程还是比较清楚的，我用括号加字母的方式标注了几个非常重要的步骤，put方法依然牵扯出很多的知识点

#### 桶数组的初始化

private final Node<K,V>[] initTable() {

Node<K,V>[] tab; int sc;

while ((tab = table) == null || tab.length == 0) {

if ((sc = sizeCtl) < 0)

// 说明已经有线程在初始化了，本线程开始自旋

Thread.yield(); // lost initialization race; just spin

else if (U.compareAndSwapInt(this, SIZECTL, sc, -1)) {

// CAS保证只有一个线程能走到这个分支

try {

if ((tab = table) == null || tab.length == 0) {

int n = (sc > 0) ? sc : DEFAULT\_CAPACITY;

@SuppressWarnings("unchecked")

Node<K,V>[] nt = (Node<K,V>[])new Node<?,?>[n];

table = tab = nt;

// sc = n - n/4 = 0.75n

sc = n - (n >>> 2);

}

} finally {

// 恢复sizeCtl > 0相当于释放锁

sizeCtl = sc;

}

break;

}

}

return tab;

}

在初始化桶数组的过程中，系统如何保证不会出现并发问题呢，关键点在于自旋锁的使用，当有多个线程都执行initTable方法的时候，CAS可以保证只有一个线程能够进入到真正的初始化分支，其他线程都是自旋等待。这段代码中关注三点即可：

* 依照前文所述，当有线程开始初始化桶数组时，会通过CAS将sizeCtl置为-1，其他线程以此为标志开始自旋等待
* 当桶数组初始化结束后将sizeCtl的值恢复为正数，其值等于0.75倍的桶数组长度，这个值的含义和之前HashMap中的THRESHOLD一致，是系统触发扩容的临界点
* 在finally语句中对sizeCtl的操作并没有使用CAS是因为CAS保证只有一个线程能够执行到这个地方

#### 添加桶数组第一个元素

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);

}

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);

}

put方法的第二个分支会用tabAt判断当前桶是否是空的，如果是则会通过CAS写入，tabAt通过UNSAFE接口会拿到桶中的最新元素，casTabAt通过CAS保证不会有并发问题，如果CAS失败，则通过循环再进入其他分支

#### 判断是否需要新增线程扩容

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) {

int rs = resizeStamp(tab.length);

while (nextTab == nextTable && table == tab &&

(sc = sizeCtl) < 0) {

// RESIZE\_STAMP\_SHIFT = 16

if ((sc >>> RESIZE\_STAMP\_SHIFT) != rs || sc == rs + 1 ||

sc == rs + MAX\_RESIZERS || transferIndex <= 0)

break;

// 这里将sizeCtl的值自增1，表明参与扩容的线程数量+1

if (U.compareAndSwapInt(this, SIZECTL, sc, sc + 1)) {

transfer(tab, nextTab);

break;

}

}

return nextTab;

}

return table;

}

static final int resizeStamp(int n) {

// RESIZE\_STAMP\_BITS = 16

return Integer.numberOfLeadingZeros(n) | (1 << (RESIZE\_STAMP\_BITS - 1));

}

因为入参n是一个int类型的值，所有Integer.numberOfLeadingZeros(n)的返回值介于0到32之间，如果转换成二进制

Integer.numberOfLeadingZeros(n)的最大值是：00000000 00000000 00000000 00100000

Integer.numberOfLeadingZeros(n)的最小值是：00000000 00000000 00000000 00000000

因此resizeStampd的返回值也就介于00000000 00000000 10000000 00000000到00000000 00000000 10000000 00100000之间，从这个返回值的范围可以看出来resizeStamp的返回值高16位全都是0，是不包含任何信息的。因此在ConcurrrentHashMap中，会把resizeStamp的返回值左移16位拼到sizeCtl中，这就是为什么sizeCtl的高16位包含整个Map大小的原理。有了这个分析，这段代码中比较长的if判断也就能看懂了

if ((sc >>> RESIZE\_STAMP\_SHIFT) != rs || sc == rs + 1 ||

sc == rs + MAX\_RESIZERS || transferIndex <= 0)

break;

* (sc >>> RESIZE\_STAMP\_SHIFT) != rs保证所有线程要基于同一个旧的桶数组扩容
* transferIndex <= 0已经有线程完成扩容任务了

至于sc == rs + 1 || sc == rs + MAX\_RESIZERS这两个判断条件如果是细心的同学一定会觉得难以理解，这个地方确实是JDK的一个BUG，这个BUG已经在JDK 12中修复，详细情况可以参考一下Oracle的官网：https://bugs.java.com/bugdatabase/view\_bug.do?bug\_id=JDK-8214427，这两个判断条件应该写成这样：sc == (rs << RESIZE\_STAMP\_SHIFT) + 1 || sc == (rs << RESIZE\_STAMP\_SHIFT) + MAX\_RESIZERS,因为直接比较rs和sc是没有意义的，必须要有移位操作。它表达的含义是

sc == (rs << RESIZE\_STAMP\_SHIFT) + 1当前扩容的线程数为0，即已经扩容完成了，就不需要再新增线程扩容sc == (rs << RESIZE\_STAMP\_SHIFT) + MAX\_RESIZERS参与扩容的线程数已经到了最大，就不需要再新增线程扩容

真正扩容的逻辑在transfer方法中，我们后面会详细看，不过有个小细节可以提前注意，如果nextTable已经初始化了，transfer会返回nextTable的的引用，后续可以直接操作新的桶数组。

#### 插入新值

如果桶数组已经初始化好了，该扩容的也扩容了，并且根据哈希定位到的桶中已经有元素了，那流程就跟普通的HashMap一样了，唯一一点不同的就是，这时候要给当前的桶加锁，且看代码：

final V putVal(K key, V value, boolean onlyIfAbsent) {

if (key == null || value == null) throw new NullPointerException();

int hash = spread(key.hashCode());

int binCount = 0;

for (Node<K,V>[] tab = table;;) {

Node<K,V> f; int n, i, fh;

if (tab == null || (n = tab.length) == 0)// 折叠

else if ((f = tabAt(tab, i = (n - 1) & hash)) == null) {// 折叠}

else if ((fh = f.hash) == MOVED)// 折叠

else {

V oldVal = null;

synchronized (f) {

// 要注意这里这个不起眼的判断条件

if (tabAt(tab, i) == f) {

if (fh >= 0) { // fh>=0的节点是链表，否则是树节点或者ForwardingNode

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; // 如果链表中有值了，直接更新

if (!onlyIfAbsent)

e.val = value;

break;

}

Node<K,V> pred = e;

if ((e = e.next) == null) {

// 如果流程走到这里，则说明链表中还没值，直接连接到链表尾部

pred.next = new Node<K,V>(hash, key, value, null);

break;

}

}

}

// 红黑树的操作先略过

}

}

}

}

// put成功，map的元素个数+1

addCount(1L, binCount);

return null;

}

这段代码中要特备注意一个不起眼的判断条件（上下文在源码上已经标注出来了）：tabAt(tab, i) == f，这个判断的目的是为了处理调用put方法的线程和扩容线程的竞争。因为synchronized是阻塞锁，如果调用put方法的线程恰好和扩容线程同时操作同一个桶，且调用put方法的线程竞争锁失败，等到该线程重新获取到锁的时候，当前桶中的元素就会变成一个ForwardingNode，那就会出现tabAt(tab, i) != f的情况。

#### 多线程动态扩容

private final void transfer(Node<K,V>[] tab, Node<K,V>[] nextTab) {

int n = tab.length, stride;

if ((stride = (NCPU > 1) ? (n >>> 3) / NCPU : n) < MIN\_TRANSFER\_STRIDE)

stride = MIN\_TRANSFER\_STRIDE; // subdivide range

if (nextTab == null) { // 初始化新的桶数组

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 = n;

}

int nextn = nextTab.length;

ForwardingNode<K,V> fwd = new ForwardingNode<K,V>(nextTab);

boolean advance = true;

boolean finishing = false; // to ensure sweep before committing nextTab

for (int i = 0, bound = 0;;) {

Node<K,V> f; int fh;

while (advance) {

int nextIndex, nextBound;

if (--i >= bound || finishing)

advance = false;

else if ((nextIndex = transferIndex) <= 0) {

i = -1;

advance = false;

}

else if (U.compareAndSwapInt

(this, TRANSFERINDEX, nextIndex,

nextBound = (nextIndex > stride ?

nextIndex - stride : 0))) {

bound = nextBound;

i = nextIndex - 1;

advance = false;

}

}

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)) {

// 判断是会否是最后一个扩容线程

if ((sc - 2) != resizeStamp(n) << RESIZE\_STAMP\_SHIFT)

return;

finishing = advance = true;

i = n; // recheck before commit

}

}

else if ((f = tabAt(tab, i)) == null)

advance = casTabAt(tab, i, null, fwd);

else if ((fh = f.hash) == MOVED) // 只有最后一个扩容线程才有机会执行这个分支

advance = true; // already processed

else { // 复制过程与HashMap类似，这里不再赘述

synchronized (f) {

// 折叠

}

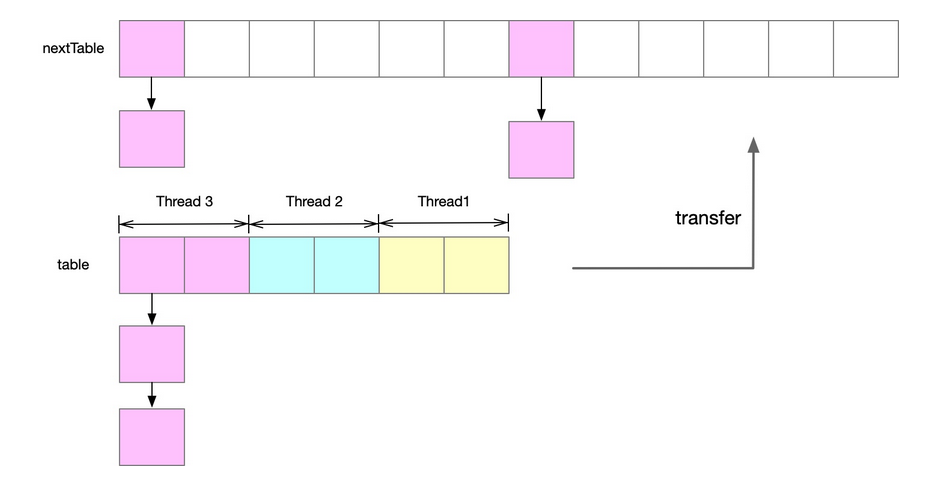
}

}

}

在Java 8中ConcurrentHashMap扩容的几个特点：

* 新的桶数组nextTable是原先桶数组长度的2倍，这与之前HashMap一致
* 参与扩容的线程也是分段将table中的元素复制到新的桶数组nextTable中
* 桶一个桶数组中的元素在新的桶数组中均匀的分布在两个桶中，桶下标相差n(旧的桶数组的长度)，这一点依然与HashMap保持一致



#### 各个线程的合作

一个关键的变量transferIndex，这是一个被volatile修饰的变量，这一点可以保证所有线程读到的一定是最新的值。

private transient volatile int transferIndex;

这个值会被第一个参与扩容的线程初始化，因为只有第一个参与扩容的线程才满足条件nextTab == null

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 = n;

}

while (advance) {

int nextIndex, nextBound;

// 当bound <= i <= transferIndex的时候i自减跳出这个循环继续干活

if (--i >= bound || finishing)

advance = false;

// 扩容的所有任务已经被认领完毕，本线程结束干活

else if ((nextIndex = transferIndex) <= 0) {

i = -1;

advance = false;

}

// 否则认领新的一段复制任务，并通过`CAS`更新transferIndex的值

else if (U.compareAndSwapInt

(this, TRANSFERINDEX, nextIndex,

nextBound = (nextIndex > stride ?

nextIndex - stride : 0))) {

bound = nextBound;

i = nextIndex - 1;

advance = false;

}

}

transferIndex就像是一个游标，每个线程认领一段复制任务的时候都会通过CAS将其更新为transferIndex - stride， CAS可以保证transferIndex可以按照stride这个步长降到0。

对于每一个扩容线程，for循环的变量i代表要复制的桶的在桶数组中的下标，这个值的上限和下限通过游标transferIndex和步长stride计算得来，当i减小为负数，则说明当前扩容线程完成了扩容任务，这时候流程会走到这个分支：

// i >= n || i + n >= nextn现在看来取不到

if (i < 0 || i >= n || i + n >= nextn) {

int sc;

if (finishing) { // 【A】完成整个扩容过程

nextTable = null;

table = nextTab;

sizeCtl = (n << 1) - (n >>> 1);

return;

}

// 【B】判断是否是最后一个扩容线程，如果是，则需要重新扫描一遍桶数组，做二次确认

if (U.compareAndSwapInt(this, SIZECTL, sc = sizeCtl, sc - 1)) {

// (sc - 2) == resizeStamp(n) << RESIZE\_STAMP\_SHIFT 说明是最后一个扩容线程

if ((sc - 2) != resizeStamp(n) << RESIZE\_STAMP\_SHIFT)

return;

// 重新扫描一遍桶数组，做二次确认

finishing = advance = true;

i = n; // recheck before commit

}

}

因为变量finishing被初始化为false，所以当线程第一次进入这个if分支的话，会先执行注释为【B】的这个分支，同时因为sizeCtl的低16位被初始化为参与扩容的线程数加一，因此，当条件(sc - 2) != resizeStamp(n) << RESIZE\_STAMP\_SHIFT满足时，就能证明当前线程就是最后一个扩容线程了，这这时候将i置为n重新扫描一遍桶数组，并且将finishing置为true保证当桶数组被扫描结束后能够进入注释为【A】的分支结束扩容。

这里就有一个问题，按照我们前面的分析，扩容线程能够通力协作，保证各自负责的桶数组的分段不重不漏，这里为什么还需要做二次确认么？有一个开发者在concurrency-interest这个邮件列表中也关于这件事咨询了Doug Lea(地址：http://cs.oswego.edu/pipermail/concurrency-interest/2020-July/017171.html)，他给出的回复是：

Yes, this is a valid point; thanks. The post-scan was needed in a previous version, and could be removed. It does not trigger often enough to matter though, so is for now another minor tweak that might be included next time CHM is updated.

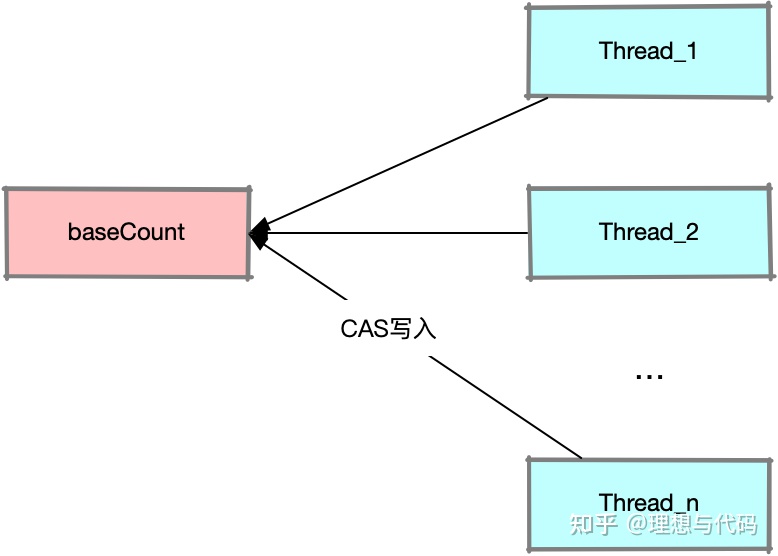
虽然Doug在邮件中的措辞用了could be, not often enough等，但也确认了最后一个扩容线程的二次检查是没有必要的。

#### size()方法

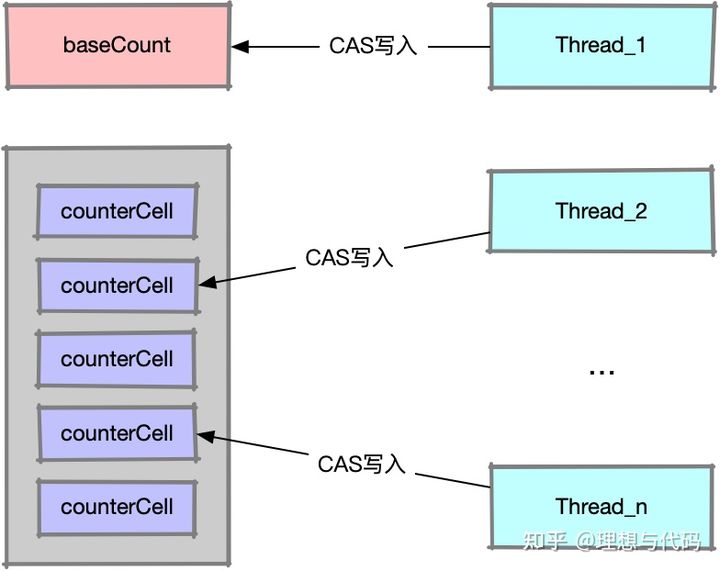
addCount()方法

private transient volatile long baseCount;

在put方法的最后，有一个addCount方法，因为putVal执行到此处说明已经成功新增了一个元素，所以addCount方法的作用就是维护当前ConcurrentHashMap的元素总数，在ConcurrentHashMap中有一个变量baseCount用来记录map中元素的个数，如下图所示，如果同一时刻有n个线程通过CAS同时操作baseCount变量，有且仅有一个线程会成功，其他线程都会陷入无休止的自旋当中，那一定会带来性能瓶颈。



为了避免大量线程都在自旋等待写入baseCount，ConcurrentHashMap引入了一个辅助队列，如下图所示，现在操作baseCount的线程可以分散到这个辅助队列中去了，调用size()的时候只需要将baseCount和辅助队列中的数值相加即可，这样就实现了调用size()无需加锁。



辅助队列是一个类型为CounterCell的数组：

@sun.misc.Contended static final class CounterCell {

volatile long value;

CounterCell(long x) { value = x; }

}

可以简单理解为只是包装了一个long型的变量value，还需要解决一个问题是，对于某个具体的线程它是如何知道操作辅助队列中的哪个值呢？

static final int getProbe() {

return UNSAFE.getInt(Thread.currentThread(), PROBE);

}

getProbe方法会返回当前线程的一个唯一身份码，这个值是不会变的，因此可以将getProbe的返回值与辅助队列的长度作求余运算得到具体的下标，它的返回值可能是0，如果返回0则需要调用ThreadLocalRandom.localInit()初始化。addCount方法中有两个细节需要注意

private final void addCount(long x, int check) {

CounterCell[] as; long b, s;

// 注意这里的判断条件，是有技巧的

if ((as = counterCells) != null ||

!U.compareAndSwapLong(this, BASECOUNT, b = baseCount, s = b + x)) {

CounterCell a; long v; int m;

boolean uncontended = true;

if (as == null || (m = as.length - 1) < 0 ||

(a = as[ThreadLocalRandom.getProbe() & m]) == null ||

// 变量uncontended记录着这个CAS操作是否成功

!(uncontended =

U.compareAndSwapLong(a, CELLVALUE, v = a.value, v + x))) {

fullAddCount(x, uncontended);

return;

}

if (check <= 1)

return;

s = sumCount();

}

if (check >= 0) {

// 检查是否需要扩容，后面再详细看

}

}

细节一：

首先我们要注意方法中刚进来的if判断条件：

if ((as = counterCells) != null ||

!U.compareAndSwapLong(this, BASECOUNT, b = baseCount, s = b + x)) {

}

作者在这里巧妙的运用了逻辑短路，如果(as = counterCells) != null则后面的CAS是不会执行的，为什么要这么设置呢？作者有两点考虑：

* 原因在于如果(as = counterCells) != null，则说明辅助队列已经初始化好了，相比于所有的线程都自旋等待baseCount这一个变量，让线程通过CAS去操作队列中的值有更大的可能性成功，因为辅助队列的最大长度为大于当前处理器个数的2的正整数幂，可以支持更大的并发
* 如果辅助队列还没有初始化好，直到有必要的时候再去创建队列，如何判断“必要性”呢？就看对baseCount的CAS操作能否成功，如果失败，就说明当前系统的并发已经比较高了，需要队列的辅助，否则直接操作baseCount

细节二：

只有当辅助队列已存在，且由ThreadLocalRandom.getProbe()在辅助队列中确定的位置不为null时，才对其做CAS操作，这本来是一个正常的防御性判断，但是uncontended记录了CAS是否成功，如果失败，则会在fullAddCount中调用ThreadLocalRandom.advanceProbe换一个身份码调整下当前线程在辅助队列的位置，避免所有线程都在辅助队列的同一个坑位自旋等待。

fullAddCount()方法

private final void fullAddCount(long x, boolean wasUncontended) {

int h;

if ((h = ThreadLocalRandom.getProbe()) == 0) {

ThreadLocalRandom.localInit(); // force initialization

h = ThreadLocalRandom.getProbe();

wasUncontended = true;

}

boolean collide = false; // True if last slot nonempty

for (;;) {

CounterCell[] as; CounterCell a; int n; long v;

// 【A】如果辅助队列已经创建，则直接操作辅助队列

if ((as = counterCells) != null && (n = as.length) > 0) {

if ((a = as[(n - 1) & h]) == null) {

if (cellsBusy == 0) { // Try to attach new Cell

CounterCell r = new CounterCell(x); // Optimistic create

if (cellsBusy == 0 &&

U.compareAndSwapInt(this, CELLSBUSY, 0, 1)) {

boolean created = false;

try { // Recheck under lock

CounterCell[] rs; int m, j;

if ((rs = counterCells) != null &&

(m = rs.length) > 0 &&

rs[j = (m - 1) & h] == null) {

rs[j] = r;

created = true;

}

} finally {

cellsBusy = 0;

}

if (created)

break;

continue; // Slot is now non-empty

}

}

collide = false;

}

else if (!wasUncontended) // 如果调用方CAS失败了，本轮空跑，下一个循环换下标继续操作

wasUncontended = true; // Continue after rehash

else if (U.compareAndSwapLong(a, CELLVALUE, v = a.value, v + x))

break;

else if (counterCells != as || n >= NCPU)

// 如果辅助队列长度已经超过了CPU个数，本轮空跑，下一个循环换下标继续操作

collide = false; // At max size or stale

else if (!collide) // 如果上一次操作失败了(CAS失败或者新建CounterCell失败)，本轮空跑，下一个循环换下标继续操作

collide = true;

else if (cellsBusy == 0 && // 如果连续两次操作辅助队列失败，则考虑扩容

U.compareAndSwapInt(this, CELLSBUSY, 0, 1)) {

try {

if (counterCells == as) {// Expand table unless stale

CounterCell[] rs = new CounterCell[n << 1];

for (int i = 0; i < n; ++i)

rs[i] = as[i];

counterCells = rs;

}

} finally {

cellsBusy = 0;

}

collide = false;

continue; // Retry with expanded table

}

// 如果上一次操作失败或者调用方CAS失败，都会走到这里，变换要操作的辅助队列下标

h = ThreadLocalRandom.advanceProbe(h);

}

// 【B】如果辅助队列还未创建，则加锁创建

else if (cellsBusy == 0 && counterCells == as &&

U.compareAndSwapInt(this, CELLSBUSY, 0, 1)) {

boolean init = false;

try { // Initialize table

if (counterCells == as) {

CounterCell[] rs = new CounterCell[2];

rs[h & 1] = new CounterCell(x);

counterCells = rs;

init = true;

}

} finally {

cellsBusy = 0;

}

if (init)

break;

}

// 【C】如果辅助队列创建失败(拿锁失败)，则尝试直接操作`baseCount`

else if (U.compareAndSwapLong(this, BASECOUNT, v = baseCount, v + x))

break; // Fall back on using base

}

}

因为counterCells是一个普通的数组，因此对其的写操作，包括初始化，扩容以及元素的写都需要加锁，加锁的方式是对全局变量cellsBusy的自旋锁。先看最外层的三个分支：

* 【B】如果辅助队列还没有创建，则加锁创建
* 【C】如果因为拿锁失败导致辅助队列创建失败，则尝试自旋写入变量baseCount，万一真的成功了呢
* 【A】如果辅助队列已经创建了，则直接去操作辅助队列相应的元素

注释中标注【A】的这个分支代码较多，其主要思路是如果通过CAS或者加锁操作辅助队列中的某个元素失败，则首先通过调用ThreadLocalRandom.advanceProbe(h)换一个队列中的元素继续操作，这次操作是否成功会记录在临时变量collide中。如果下一次操作还是失败，则说明此时的并发量比较大需要扩容了。如果辅助队列的长度已经超过了CPU的个数，那就不再扩容，继续换一个元素操作，因为同一时间能运行的线程数最大不会超过计算机的CPU个数。

注释中标注【A】的这个分支代码较多，其主要思路是如果通过CAS或者加锁操作辅助队列中的某个元素失败，则首先通过调用ThreadLocalRandom.advanceProbe(h)换一个队列中的元素继续操作，这次操作是否成功会记录在临时变量collide中。如果下一次操作还是失败，则说明此时的并发量比较大需要扩容了。如果辅助队列的长度已经超过了CPU的个数，那就不再扩容，继续换一个元素操作，因为同一时间能运行的线程数最大不会超过计算机的CPU个数。

在这个过程中有四个细节仍然需要注意：

细节一：

counterCells只是一个普通的数组，因此并不是线程安全的，所以对其写操作需要加锁保证并发安全

细节二：

加锁的时候，作者做了一个double-check的动作，我看有的文章将其解读为“类似于单例模式的double-check”，这个是不对的，作者这样做的原因我们在上一篇文章中有讲过，首先第一个检查cellsBusy == 0是流程往下走的基础，如果cellsBusy == 1则直接拿锁失败退出，调用h = ThreadLocalRandom.advanceProbe(h);更新h后重试，如果cellsBusy == 0校验通过，则调用CounterCell r = new CounterCell(x);初始化一个CounterCell，这样做是为了减少自旋锁的临界区的大小，以此来提升并发性能

细节三：

在加锁的时候先判断下cellsBusy是否为0，如果为1那直接宣告拿锁失败，为什么这么做呢？因为相比于调用UNSAFE的CAS操作，直接读取volatile的消耗更少，如果直接读取cellsBusy已经能判断出拿锁失败，那就没必要再调用耗时更多的CAS了

细节四：

对cellsBusy从0到1的更改调用了CAS但是从1置为0却只用了赋值操作，这是因为CAS可以保证能走到这条语句的只有一个线程，因此可以用赋值操作来更改cellsBusy的值。

sumCount

前面两个方法主要是把ConcurrentHashMap中的元素个数分散的记录到baseCount和辅助队列中，调用size()方法的时候只需要把这些值相加即可。

public int size() {

long n = sumCount();

return ((n < 0L) ? 0 :

(n > (long)Integer.MAX\_VALUE) ? Integer.MAX\_VALUE :

(int)n);

}

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;

}

}

return sum;

}

## 注解与反射

### 注解（Annotation）

Annotation是JDK5.0开始引入的技术。Annotation的作用：

* 不是程序本身，可以对程序作出解释
* 可以被其它程序（比如编译器）读取。

#### 注解格式

注解是以"@注释名"在代码中存在，还可以添加一些参数值，例如：@SuppressWarnings(value="unchecked").

#### 内置注解

内置注解就是我们的jdk所带的一些注解。常用的三个注解：

* @Override

这个应该都不陌生，修辞方法，表示打算重写超类中的方法声明。

* @Deprecated

这个注解我们应该也不会陌生，我们可能看不到这个注解，但是我们肯定在使用一些方法时会出现横线。表示废弃，这个注释可以修辞方法，属性，类，表示不鼓励程序员使用这样的元素，通常是因为他很危险或有更好的选择。

* @SuperWarnings

这个注解主要是用来抑制警告信息的，我们在写程序时，可能会报很多黄线的警告，但是不影响运行，我们就可以用这个注解来抑制隐藏它。与前俩个注解不同的是我们必须给注解参数才能正确使用他。

|  |  |
| --- | --- |
| 参数 | 说明 |
| deprecation | 使用了过时的类或方法的警告 |
| unchecked | 执行了未检查的转换时的警告 如：使用集合时未指定泛型 |
| fallthrough | 当在switch语句使用时发生case穿透 |
| path | 在类路径、源文件路径中有不存在路径的警告 |
| serial | 当在序列化的类上缺少serialVersionUID定义时的警告 |
| finally | 任何finally子句不能完成时的警告 |
| all | 关于以上所有的警告 |

上表中就是@SuperWarnings注解的一些参数，按需使用即可。

@SuperWarnings(“finally”)

@SuperWarnings(value={“unchecked”,“path”})

#### 注解位置

被注解的位置有哪些，这些都在一个枚举类：ElementType当中：

* TYPE：类、接口、注解、枚举
* FIELD：字段
* METHOD：方法
* PARAMETER：参数
* CONSTRUCTOR：构造方法
* LOCAL\_VARIABLE：本地变量
* ANNOTATION\_TYPE：注解
* PACKAGE：包
* TYPE\_PARAMETER：类型参数
* TYPE\_USE：类型使用

注解位置配合@Target使用，当只有一个位置时可以这么使用：

@Target(ElementType.ANNOTATION\_TYPE)

当指定多个位置时，使用方法如下：

@Target(value={CONSTRUCTOR, FIELD, LOCAL\_VARIABLE, METHOD, PACKAGE, PARAMETER, TYPE})

#### 注解生命周期

注解也是有相应的声明周期的，也是封装在一个枚举类：RetentionPolicy中：

* SOURCE：源代码期间，在编译时会去除，所以这都是给编译器使用的
* CLASS：会保留在类文件中，但是运行时JVM不需要保存，默认的生命周期
* RUNTIME：会持续保存到JVM运行时，可以通过反射来获取

声明周期配合@Retention来使用，使用方法如下：

@Retention(RetentionPolicy.RUNTIME)

#### 自定义注解

注解和接口其实很相似，接口里面的方法定义了行为，注解的方法定义了属性，下面先给一个例子：

public @interface MyAnnotation {

//声明属性,可以使用如下类型

String name();

String password() default "123";

int age() default 12;

TimeUnit gender() default TimeUnit.SECONDS;

Class<?> clazz();

int[] arr() default {1,2,3};

//为了嵌套配置

Override my2();

}

简单总结下：

* 属性是以方法的形式定义的，属性名即为方法名。
* 可以设置默认值，那么当使用该注解的时候，如果不指定该属性则使用默认这
* 没有默认值的，使用时必须赋值
* 属性类型可以为：String、基本类型、Class类型、枚举、注解、以上的一维数组
* 若只有一种属性，且名为value，则赋值时可以不指定属性名
* 同样的只有一个一维数组value[]，则赋值时可以不指定属性名

#### 注解使用

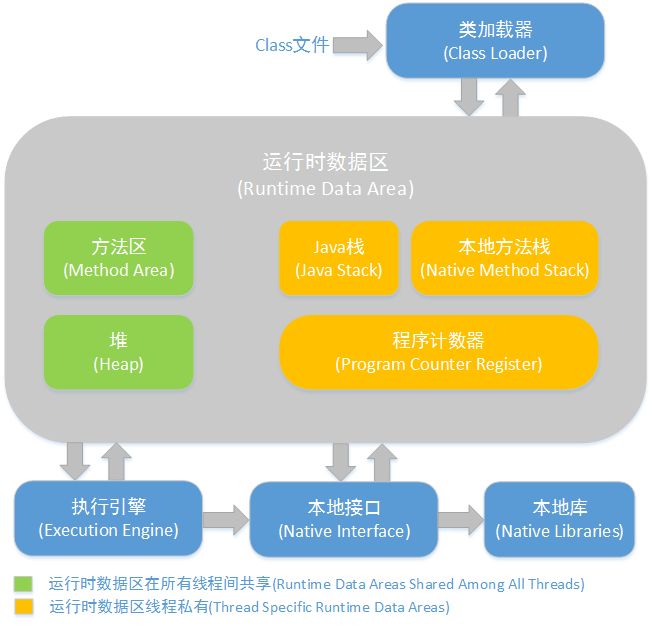
当你提供一个注解供别人使用时，那么对方可能将注解应用于允许的位置，并有可能赋值，而我们无法知道注解具体的位置，这时候只能通过反射加遍历的方式来获得注解的位置。

通过调用 getAnnotation/ getAnnotations 方法来获取注解

### 反射

#### 基础概念

Java源码编译后生成.class文件，程序启动，Java虚拟机加载.class文件，类被加载到JVM内存中，流程如下：



Java反射机制是在JVM运行期间，对于任何一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意方法和属性；这种动态获取信息以及动态调用对象方法的功能称为 Java 语言的反射机制。

Java 反射机制主要提供了以下功能，这些功能都位于java.lang.reflect包。

* 在运行时判断任意一个对象所属的类。
* 在运行时构造任意一个类的对象。
* 在运行时判断任意一个类所具有的成员变量和方法。
* 在运行时调用任意一个对象的方法。
* 生成动态代理。

**通过反射方法， 能够拿到改xxx类相关的所有资源：**

* Class类：代表一个类。
* Field 类：代表类的成员变量（成员变量也称为类的属性）。
* Method类：代表类的方法。
* Constructor 类：代表类的构造方法。
* Array类：提供了动态创建数组，以及访问数组的元素的静态方法。

#### 常用API介绍

|  |  |  |
| --- | --- | --- |
| 序号 | 方法名 | 作用 |
| 1 | Class Class.forName(String packageNameAndClassName) throws ClassNotFoundException; | 通过完整的包名.类名获取对应的class类对象 |
| 2 | Class 类对象.getClass(); | 通过类名获取当前类对象对应的class类对象 |
| 3 | Class 类名.class; | 通过类名获取 |

其他API在其他部分查看

#### 访问构造方法

|  |  |  |
| --- | --- | --- |
| 序号 | 方法 | 作用 |
| 1 | constructor[] getConstructors(); | 通过class类对象获取对应的Constructor构造方法类对象 |
| 2 | Constructors[] getDeclaredConstructors(); | 暴力获取所有构造方法类对象数组，包括私有化构造方法。 |
| 3 | Constructor getConstructor(Class... parameterTypes); | 根据指定的参数类型获取构造方法（非私有化） |
| 4 | Constructor getDeclaredConstructor(Class... parameterTypes); | 暴力获取指定数据类型的构造方法（包括私有化） |
| 6 | 获取类上注解 |  |
| 5 | Object newInstance(Object... parameters); | 通过Constructor类对象，执行对应的构造方法，创建对应的类对象 |

#### 访问方法

|  |  |  |
| --- | --- | --- |
| 序号 | 方法 | 作用 |
| 1 | Method[] getDeclaredMethods() | 获取所有方法 |
| 2 | Method getDeclaredMethod(String methodName); | 获取给定名称方法 |
| 3 | void setAccessible(Boolean b) | 设置访问许可 |
| 4 | void invoke() | 调用 |

#### 访问成员变量

|  |  |  |
| --- | --- | --- |
| 序号 | 方法 | 作用 |
| 1 | Field [] getDeclaredFields () | 获取所有属性 |
| 2 | Field getDeclaredField() (String methodName); | 获取给定名称方法 |
| 3 | void setAccessible(Boolean b) | 设置访问许可 |
| 4 | Field#set(Object o,Object value); | 赋值 |

#### 动态代理中的反射使用

动态代理：同一个服务由不同的实例帮我们去实现方法

JDK实现步骤：

* 继承InvocationHandler接口，invoke方法进行调用、增强、转发实现业务。
* 实例化上个步骤中继承InvocationHandler接口的类，注入增强对象object。
* 调用方法public static Object newProxyInstance(ClassLoader loader,

Class<?>[] interfaces,

InvocationHandler h)

生成被代理的对象reflectionObject，方法参数依次为：代理接口的所属类加载器，代理接口，代理实现类对象object。

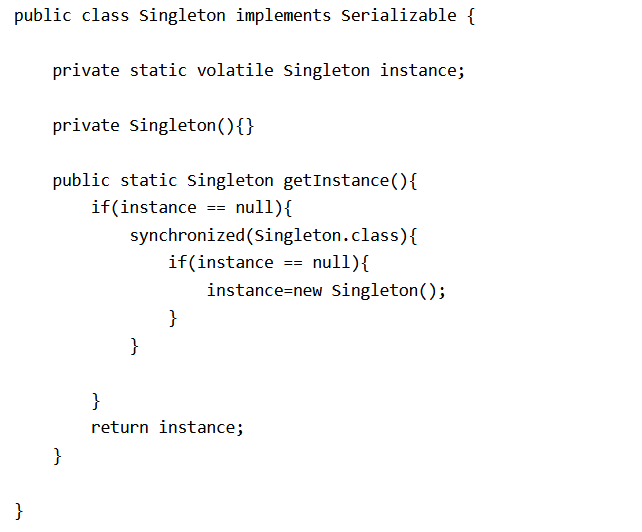
* reflectionObject调用方法即可实现增强

### 单例模式的探究

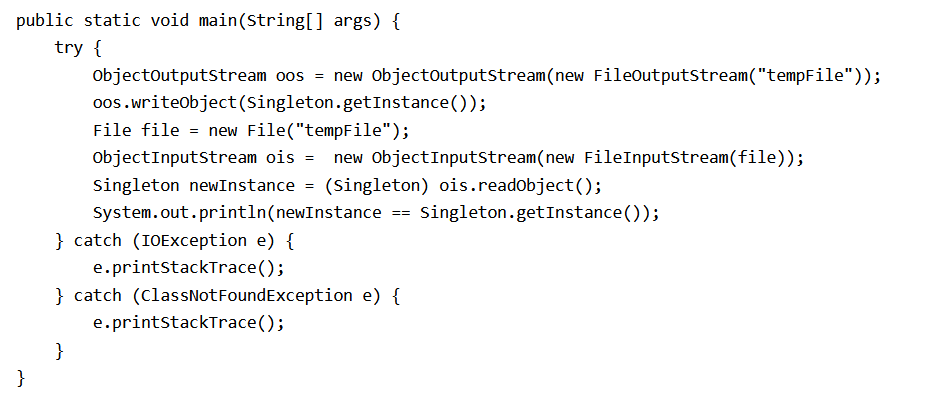
单例模式，是设计模式中最简单的一种。通过单例模式可以保证系统中一个类只有一个实例且该实例易于外界访问，从而方便对实例个数的控制节约资源。

但是可以通过反射，序列化和反序列化绕过单例模式。

使用双重校验锁方式编写一个单例模式：



测试类：



明显输出结果为false，