Java Library

Pionpill [1]

本文档为作者归纳 Java 时的笔记。

2022 年 12 月 6 日

[1]笔名：北岸，电子邮件：673486387@qq.com，Github：https://github.com/Pionpill

前言：

笔者为软件工程系在校本科生，有计算机学科理论基础 (操作系统，数据结构，计算机网络，编译原理等)，本人在撰写此笔记时已有 Java 开发经验，基础知识不再赘述。

本文各部分内容如下:

- Java 基础库: 最基础的 Java 操作的源码解析，主要为 lang 包下几个常用的类。
- Java 容器: util 包下四种容器 (集合) 解析，只讲基本实现逻辑，不涉及具体算法和多线程内容。

本人的编写及开发环境如下：

- Java: Java17
- OS: Windows11

2022 年 12 月 6 日

目录

V 列表，集合与队列

第一部分

Java 基础库

I 基本数据类型相关

1 Object 基类

Object 是所有类的基类，这节将详细说明它的所有成员。

1.1 类与对象

构造函数

```
@IntrinsicCandidate
public Object() {}
```

其中 @IntrinsicCandidate 表示在虚拟机中会有一套高效的实现方式。其他没什么。

反射相关

```
@IntrinsicCandidate
public final native Class<?> getClass();
```

该方法用于获取对象的类型信息。即在 JVM 中获取方法区中对应的类型信息。此外这里有一个 native 关键字，表示对应的方法无法通过 Java 语言实现，而应该让 JVM 调用 JNI(Java Native Interface) 通过更底层的 C/C++ 语言实现。

1.2 常见方法

哈希码[1]

```
@IntrinsicCandidate
public native int hashCode();
```

哈希码与 equals 方法有如下联系:

- 如果两个对象相同，equals 方法一定返回 true，这两个对象的 HashCode 一定相同;
- 两个对象的 HashCode 相同，并不一定表示两个对象就相同，即 equals() 不一定为 true，只能说明这两个对象在一个散列存储结构中。
- 如果对象的 equals 方法被重写，那么对象的 HashCode 也尽量重写。

哈希码的作用主要是快速查找，常常被用在容器 (Collection) 中。哈希算法也称为散列算法，是将数据依特定算法直接指定到一个地址上。这样一来，当集合要添加新的元素时，先调用这个元素的 HashCode 方法，就一下子能定位到它应该放置的物理位置上。

[1]参考: https://blog.csdn.net/SEU_Calvin/article/details/52094115

- 如果这个位置上没有元素，它就可以直接存储在这个位置上；
- 如果这个位置上已经有元素了，就调用它的 equals 方法与新元素进行比较，相同的话就不存了；
- 不相同的话，也就是发生了 Hash key 相同导致冲突的情况，那么就在这个 Hash key 的地方产生一个链表，将所有产生相同 HashCode 的对象放到这个单链表上去，串在一起。这样一来实际调用 equals 方法的次数就大大降低了，几乎只需要一两次。

equals

```
public boolean equals(Object obj) {
    return (this == obj);
}
```

Object 本身的 equals 方法没什么好说的，就是通过 == 比较地址是否相同。但像 String 这样的绝大部分子类都会重写该方法，让其效果为: 比较对象内容是否相等。

比较地址实际上是比较的存在 JVM 栈中的 reference 是否相同。

clone

```
@IntrinsicCandidate
protected native Object clone() throws CloneNotSupportedException;
```

JDK 文档中堆 clone() 方法有如下约定 (前两条必须遵守):

- `x.clone != x;` 克隆对象与原对象不是一个对象。
- `x.clone().getClass() == x.getClass();` 克隆的是同一类型的对象。
- `x.clone().equals(x) == true` 如果 x.equals() 方法定义恰当的话

一般情况下，子类并不需要实现 `clone()` 方法，Object 的 `clone()` 方法将返回一个浅拷贝 (由于是 native 的，非常快)，对于简单对象这就足够了。但如果需要实现深拷贝，需要子类实现 `Cloneable()` 接口，并重写 `clone()` 方法。

toString

```
public String toString() {
    return getClass().getName() + "@" + Integer.toHexString(hashCode());
}
```

返回对象的字符串表达形式，没啥好说的，`print()` 方法会自动调用。

1.3 多线程

唤醒线程

```
@IntrinsicCandidate
public final native void notify();

@IntrinsicCandidate
public final native void notifyAll();
```

这两个就一个不同，`notify()` 随机唤醒一个线程，`notifyAll()` 广播机制，让其他线程抢占资源。

线程等待

```
public final void wait() throws InterruptedException { wait(0L); }

public final native void wait(long timeoutMillis) throws InterruptedException;

public final void wait(long timeoutMillis, int nanos) throws InterruptedException {
    if (timeoutMillis < 0)
        throw new IllegalArgumentException("timeoutMillis value is negative");
    if (nanos < 0 || nanos > 999999)
        throw new IllegalArgumentException( "nanosecond timeout value out of range");
    if (nanos > 0 && timeoutMillis < Long.MAX_VALUE)
        timeoutMillis++;
    wait(timeoutMillis);
}
```

第一个 `wait()` 让线程等待，直到被唤醒，不被唤醒不启动。其余两个，timeoutMillis 表示给定的时间，如果到了给定的时间还没被唤醒，会自动醒来尝试抢占资源 (timeoutMillis 为 0 表示不会自动醒)。其实本质上都是去调用第二个 native 方法。

其中第三个有个 `nanos` 参数，表示格外的时间，单位为纳秒。

2 Wrapper 包装类

String 对八个基本数据类型提供了对应的包装类，根据类型我分为了以下几种:

- 整型: Byte, Short, Integer, Long
- 浮点型: Float, Double
- 布尔: Boolean
- 字符: Character

其中整型方法类似，布尔型没什么好讲的。包装类是装饰器模式的一种应用，部分人也将 Wrapper 等价于装饰器模式。

2.1 Integer 数值型包装类

```
public final class Integer extends Number
    implements Comparable<Integer>, Constable, ConstantDesc {}
```

2.1.1 构造对象

最简单的，使用构造函数，Integer 的构造函数有两个:

```
@Deprecated(since="9", forRemoval = true)
public Integer(int value) {
    this.value = value;
}

@Deprecated(since="9", forRemoval = true)
public Integer(String s) throws NumberFormatException {
    this.value = parseInt(s, 10);
}
```

注意，这两个构造函数虽然可以用，但都被舍弃了，推荐使用静态的 ValueOf 方法。类似的有两类静态方法:

```
@IntrinsicCandidate
public static Integer valueOf(int i) {
    if (i >= IntegerCache.low && i <= IntegerCache.high)
        return IntegerCache.cache[i + (-IntegerCache.low)];
    return new Integer(i);
}

public static Integer valueOf(String s, int radix) throws NumberFormatException {
    return Integer.valueOf(parseInt(s,radix));
}

public static Integer valueOf(String s) throws NumberFormatException {
    return Integer.valueOf(parseInt(s, 10));
}
```

其中传入 int 型的方法做了一些优化，缓存了-128—127 的 Integer 值，如遇到 [-128,127] 范围的值需要转换为 Integer 时会直接从 IntegerCache 中获取[2]。IntegerCache 存储在 JVM 方法区运行时常量池中。另外 @IntrinsicCandidate 注解表示在虚拟机中做了优化。

传入 String 类型的方法，除了可增加一个参数: 基数外，调用了 parseInt 方法。parseInt 的工作很简单：检查字符串是否合法并进行转换。

2.1.2 类型转换

首先是 toString 函数，Integer 提供了各种各样的转换为字符串的方法，这里不一一说明:

```
// 全能款：转为为各种 String 类型
public static String toString(int i, int radix) { ... }
public static String toUnsignedString(int i, int radix) { ... }
// 便携款：相当于制定了 radix
public static String toHexString(int i) { ... }
......
```

其次是 parse 函数，用于将各种字符串或字符序列进行解析，注意返回的是 int 类型的值:

```
// 解析 String
public static int parseInt(String s, int radix) { ... }
public static int parseUnsignedInt(String s, int radix) { ... }
// 解析 序列
public static int parseInt(CharSequence s, int beginIndex, int endIndex, int radix) throws
    NumberFormatException {}
```

最后是 Value 函数，用于转换为其他类型的整型/浮点值:

```
// 本质就是强制类型转换，其他类型一样
public long longValue() { return (long)value; }
```

2.1.3 数学运算

Integer 提供了基本的数学运算函数，本质上和直接使用符号运算效果相同:

```
// 比较运算
public static int compare(int x, int y) {
    return (x < y) ? -1 : ((x == y) ? 0 : 1);
}
// 除法运算
public static int divideUnsigned(int dividend, int divisor) {
    return (int)(toUnsignedLong(dividend) / toUnsignedLong(divisor));
}
// 取余运算
public static int remainderUnsigned(int dividend, int divisor) {
    return (int)(toUnsignedLong(dividend) % toUnsignedLong(divisor));
}
```

[2]IntegerCache 拓展: https://blog.csdn.net/cockroach02/article/details/88857551

另外提供了一些好用位运算方法:

```
// 最高的一位，理解为比数字小的最大 2 的整数倍，例如 17 -> 16 7-> 4
public static int highestOneBit(int i) {
    return i & (MIN_VALUE >>> numberOfLeadingZeros(i));
}
// 最低的一位，解释起来比较麻烦，自行理解
public static int lowestOneBit(int i) {
    return i & -i;
}
// 前置 0 的个数，注意占 32 bit
@IntrinsicCandidate
public static int numberOfLeadingZeros(int i) { ... }
// 后置 0 的个数
@IntrinsicCandidate
public static int numberOfTrailingZeros(int i) { ... }
// 二进制中 1 的个数
@IntrinsicCandidate
public static int bitCount(int i) { ... }
```

2.1.4 Float 浮点型

Float 大部分方法和 Integer 类似，这里只说点特别的:

```
// 判断是否是 NaN (Not a Number)
public static boolean isNaN(float v) {
    return (v != v);
}
// 判断是否无穷
public static boolean isInfinite(float v) { ... }
```

2.2 Character 字符型包装类

2.2.1 常用函数

判断型常用函数:

```
// 判断大小写字符
public static boolean isLowerCase(char ch) { ... }
public static boolean isUpperCase(char ch) { ... }
// 判断数字还是字符
public static boolean isDigit(char ch) { ... }
public static boolean isLetter(char ch) { ... }
public static boolean isLetterOrDigit(char ch) { ... }
// 判断是否为空
public static boolean isWhitespace(char ch) { ... }
```

转换型常用函数

```java
public static char toLowerCase(char ch) { ... }
public static char toUpperCase(char ch) { ... }
```

2.3 自动装箱与拆箱机制

包装类提供的语法糖自然是自动装箱与拆箱机制，下面是编译器自动帮我们完成的事情:

```java
// ========== 自动装箱 ==========
Integer a = 1;
Integer a = Integer.valueOf(1); // 实际过程
// ========== 自动拆箱 ==========
int b = a;
int b = a.intValue();        // 实际过程
int c = a + a;
int c = a.intValue() + a.intValue(); // 实际过程
```

3 String 字符串

3.1 String 类型

String 对象是 Java 中最常用最基础的字符串对象，它的继承关系如下:

```
public final class String
    implements java.io.Serializable, Comparable<String>, CharSequence, Constable, ConstantDesc
```

3.1.1 String 不可变的原因

在 String 中使用 byte 数组存储信息[3]:

```
@Stable
private final byte[] value;
```

在这段代码中，有三个需要注意的地方:

- **@Stable**: 仅允许被修饰的字段被修改至多一次，JVM 会将对应的值看作常量，并对其进行优化:

```
@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Stable { }
```

- **private**: 私有成员，仅能通过公有方法进行修改，但是 String 没有提供修改数据的公有方法，那么还可以通过反射机制跳过安全检查进行修改。
- **final**: 常量，被 final 修饰意味着对象在 JVM 中初始化时，在一般成员赋值之前，该成员的数值就已经被确定，并且随后无法修改，那么反射机制也不能堆它进行改动了。

以上，private 和 final 组合的效果让 value 值无法被二次修改，这也就决定了 String 类型的对象是不可变的。如果要钻牛角尖的话，String 是被 final 修饰的，这也决定了 String 无法被继承，无法通过子类实现可变。

实际上，如果不考虑安全性，可以通过反射拿到 byte[] 对象，通过下标修改值，String 不可变指的仅是拿到的数组地址不可变，不是数组中的值不可变。

除了比较重要的 `value` 成员，还有一个 `coder` 下面会用到，它是用来确定编码方式的: UTF-16 或 LATIN1。

String 的构造方法有很多，总的来说可以根据这些数据构造 String 对象:

- **String 对象**: 包括字面量，常规 String 对象。
- **数组**: char, int, byte 数组都可。
- **StringBuilder, StringBuffer**: 本质上是调用对应的 toString 方法。

[3]在 Java9 之前是使用 char[] 存储，主要原因是 byte 比 char 可以节省一半空间

3.1.2 常用方法

基础的常用方法有:

```
// 获得长度
public int length() {
    return value.length >> coder();
}
// 判空
public boolean isEmpty() {
    return value.length == 0;
}
// 获取字符
public char charAt(int index) {
    if (isLatin1())
        return StringLatin1.charAt(value, index);
    else
        return StringUTF16.charAt(value, index);
}
```

比较相关的常用类型

```
// equal(Java8)
public boolean equals(Object anObject) {
    if (this == anObject)
        return true;

    if (anObject instanceof String) {
        String anotherString = (String)anObject;
        int n = value.length;
        if (n == anotherString.value.length) {
            char v1[] = value;
            char v2[] = anotherString.value;

            int i = 0;
            while (n-- != 0) {
                if (v1[i] != v2[i])
                    return false;
                i++;
            }
            return true;
        }
    }
    return false;
}
// contentEquals (用于和 StringBuilder, StringBuffer 比较)
public boolean contentEquals(CharSequence cs) { ... }
// 忽略大小写比较 (还有这种好东西)
public boolean equalsIgnoreCase(String anotherString) { ... }
```

处理字符串特有的方法

```
// 判断首部
```

```
public boolean startsWith(String prefix, int toffset) { ... }
// 判断尾部
public boolean endsWith(String suffix) { ... }
// 查找第一个字符/子串
public int indexOf(int ch, int fromIndex) { ... }
public int indexOf(String str, int fromIndex) { ... }
// 查找最后一个字符/子串
public int lastIndexOf(int ch, int fromIndex) { ... }
public int lastIndexOf(String str, int fromIndex) { ... }
// 获取字串/子序列
public String substring(int beginIndex, int endIndex) { ... }
public String subSequence(int beginIndex, int endIndex) { ... }
// 合并字符串
public String concat(String str) { ... }
public static String join(CharSequence delimiter, CharSequence... elements) { ... }
// 拆分字符串
public String[] split(String regex, int limit) { ... }
// 替换字符串
public String replace(char oldChar, char newChar) { ... }
public String replaceFirst(String regex, String replacement) { ... }
public String replaceAll(String regex, String replacement) { ... }
// 匹配
public boolean matches(String regex) { ... }
public boolean contains(CharSequence s) { ... }
// 大小写转换
public String toLowerCase(Locale locale) { ... }
public String toUpperCase(Locale locale) { ... }
// 去除空格
public String trim() { ... } // 删除空格，tab，换行
public String strip() { ... } // 删除更多的空白，包括全角半角，更全面
public String stripLeading() { ... } // 删除首部空格
public String stripTrailing() { ... } // 删除尾部空格
public boolean isBlank() { ... }    // 判断是否是空白字符串
// 转换
public char[] toCharArray() {...}
// 格式化，本质上是调用 Formatter
public String formatted(Object... args) {
    return new Formatter().format(this, args).toString();
}
```

常用函数

```
public static String join(CharSequence delimiter, Iterable<? extends CharSequence> elements)
    { ... }
// 所有的 valueOf 本质上就是调用构造函数或对应的 toString 方法
public static String valueOf(char data[]) { ... }
```

3.2 StringBuilder 与 StringBuffer

StringBuilder 和 StringBuffer 都是可变的字符串序列。唯一的大区别是 StringBuffer 是线程安全的，大部分方法使用了 synchronized 修饰，也因此效率略低一点。下文以 StringBuilder 为例。

```
public final class StringBuilder extends AbstractStringBuilder
    implements java.io.Serializable, Comparable<StringBuilder>, CharSequence
```

3.2.1 常用方法

构造函数，非常有意思的是，StringBuilder 的构造函数大部分使用了 @IntrinsicCandidate 注解，代表会在 JVM 中高效处理，而 String 却只有少部分构造函数拥有这个注解。

```
@IntrinsicCandidate
public StringBuilder() { super(16); }
@IntrinsicCandidate
public StringBuilder(int capacity) { super(capacity); }
@IntrinsicCandidate
public StringBuilder(String str) { super(str); }
// 这个没有被修饰
public StringBuilder(CharSequence seq) { super(seq); }
```

常用方法主要是增删改方法:

```
// 各种各样的 增加
public StringBuilder append(String str) { ... }
// 各种各样的 删除
public StringBuilder delete(int start, int end) { ... }
// 各种各样的 插入
public StringBuilder insert(int offset, char c) { ... }
```

3.2.2 可变的原因

StringBuilder 的几乎所有方法都调用了 super 方法，所以想知道具体实现细节，得看 AbstractStringBuilder 的实现。

和 String 一样，AbstractStringBuilder 也是用 byte 数组保存数据，但是它没有被 private final 修饰。下面看一下 AbstractStringBuilder 的 append 实现机制的相关源码:

```
byte[] value;

public AbstractStringBuilder append(String str) {
    if (str == null)
        return appendNull();
    int len = str.length();
    ensureCapacityInternal(count + len);
    str.getChars(0, len, value, count);
    count += len;
```

```
    return this;
}

private void ensureCapacityInternal(int minimumCapacity) {
    if (minimumCapacity - value.length > 0) {
        value = Arrays.copyOf(value, newCapacity(minimumCapacity));
    }
}
```

所以，本质上来说，StringBuilder 数据每次被修改后，都会通过 `Arrays.copyOf` 申请新的空间，这就是它可变的本质。

3.3 编译器对 String 的优化

3.3.1 String 的加法运算

String 的加法运算是 Java 唯一的运算符重载。它本质上就是 StringBuilder!

```
// 字面量运算
String sql = "";
sql = sql + "aa";
// 实际上
sql = new StringBuilder("").append("aa").toString();
```

这样似乎没什么问题，但如果在循环中使用 + 运算:

```
String ans = "";
for (int i = 0; i<100; i++)
    ans += "Hello World! ";
```

那么就会不断地 newStringBuilder(target).append().toString()。但是!我们不需要中间的 new 与 toString 过程，我们可以只 new 一个 StringBuilder 对象，并且删除中间的所有 toString 过程。

```
String ans = "";
StringBuilder sb = new StringBuilder(ans);
for (int i = 0; i<100; i++)
    sb.append("Hello World! ");
ans = sb.toString();
```

这样会快很多，所以，绝对不要在循环中使用 + 运算，并且尽可能地使用 StringBuilder。

3.3.2 常量折叠

在进入下一节之前，有必要讲一下基础的常量折叠，这是编译器阶段的一种优化，对所有常量都适用。

常量折叠是 Java 在编译期做的一个优化，简单的来说，在编译期就把一些表达式计算好，

不需要在运行时进行计算。例如:

```
// 编译前
String a = "hello " + "world!";
int b = 1+1;
int c = 100*100;
// 编程 class 文件后反编译
String a = "hello world!";
int b = 2;
int c = 10000;
```

但请注意，String 的常量折叠仅在字面量加法运算时存在，如下语句无法进行常量折叠；

```
String a = "hello ";
String b = a + "world!!!";
```

仅有如下类型的常量可以进行常量折叠:

- 8 种基本数据类型 + 字符串常量。
- final 修饰的基本数据类型和字符串变量。
- 字符串通过 “+” 拼接得到的字符串、基本数据类型之间算数运算（加减乘除）、基本数据类型的位运算。

3.3.3 JVM 对 String 的优化

正如 Integer 存在缓存池一样，String 在 JVM 中有一个字符串常量池 (Java8 及之后在堆中)。对于编译期可以确定值的字符串，也就是常量字符串，jvm 会将其存入字符串常量池。并且，字符串常量拼接得到的字符串常量在编译阶段就已经被存放字符串常量池，这个得益于编译器的优化。

如果在字符串常量池中已经存在了一个字符串常量，其他所有相同的字符串常量引用都会指向这个常量数据。这时候并不用担心数据被篡改的影响，前面已经说过 String 的数据是无论如何都不能修改的，只具备可读性，这是这样优化的前提。

```
String a = "hello2";
final String b = "hello";
String d = "hello";
String c = b + 2;
String e = d + 2;
String f ="hello"+"2";
System.out.println(a == c); // true
System.out.println(a == e); // false
System.out.println(a == f); // true
```

为什么 e 不一样呢，e 是通过 StringBuilder new 出来的新对象，凡是 new 出来的新对象，都不指向字符串常量池中已有的对象。相反的，a c，f 在编译阶段，通过常量折叠都是”hello2”，并且指向字符串常量池同一数据。

4 高精度运算

4.1 BigInteger

```
public class BigInteger extends Number implements Comparable<BigInteger>
```

BigInteger 有两个基础属性:

```
final int signum;   // 存储符号位 1: 整 0: 0 -1: 负
final int[] mag;    // 存储数据
```

注意这两个属性都是 final 修饰的，这决定了 `BigInteger` 是不可变对象。

总的来说，BigInteger 的构造函数有两类:

```
// 通过数组构造
public BigInteger(byte[] val, int off, int len)
public BigInteger(int signum, byte[] magnitude, int off, int len) // 指定符号位
// 通过字符串构造
public BigInteger(String val, int radix)
```

同时还提供了一个静态方法:

```
public static BigInteger valueOf(long val)
```

首先看一下加法运算，首先通过 signum 进行了优化，如果符号位不同，则转换为减法运算再处理结果:

```
public BigInteger add(BigInteger val) {
    if (val.signum == 0)
        return this;
    if (signum == 0)
        return val;
    if (val.signum == signum)
        return new BigInteger(add(mag, val.mag), signum);

    int cmp = compareMagnitude(val); // 忽略符号位进行比较
    if (cmp == 0)
        return ZERO;
    int[] resultMag = (cmp > 0 ? subtract(mag, val.mag)
                       : subtract(val.mag, mag));
    resultMag = trustedStripLeadingZeroInts(resultMag);
    return new BigInteger(resultMag, cmp == signum ? 1 : -1);
}
```

其次是减法运算，运算逻辑类似:

```
public BigInteger subtract(BigInteger val) {
    if (val.signum == 0)
        return this;
    if (signum == 0)
        return val.negate();
```

```
    if (val.signum != signum)
        return new BigInteger(add(mag, val.mag), signum);

    int cmp = compareMagnitude(val);
    if (cmp == 0)
        return ZERO;
    int[] resultMag = (cmp > 0 ? subtract(mag, val.mag)
                       : subtract(val.mag, mag));
    resultMag = trustedStripLeadingZeroInts(resultMag);
    return new BigInteger(resultMag, cmp == signum ? 1 : -1);
}
```

乘除法运算，在这部分的算法就比较复杂了，只给出扩展文献连接:

```
public BigInteger multiply(BigInteger val)
public BigInteger divide(BigInteger val)
```

- 乘法运算: https://zhuanlan.zhihu.com/p/391716853

类似的，还提供了一些数字处理的常用方法，没什么好讲的。

4.2 BigDecimal

```
public class BigDecimal extends Number implements Comparable<BigDecimal>
```

BigDecimal 有几个关键属性:

```
private final BigInteger intVal; // 实际存储的数据
private final int scale;         // 标度，表示小数位数
private transient int precision; // 精度
```

注意这里有个 transient 关键字，表示该属性不会被序列化，大概意思就是，该属性不会进入存储和传输过程 (这两个过程需要序列化)。

在构造过程中，会将浮点数膨胀成为整数，并记录标度和精度。标度和精度有什么区别呢？标度是不会改变的，表示实际的小数点位数，辅助 intVal。而精度会在计算过程中改变，例如 1.01 + 1.001 实际上是 101(2) + 1001(3)，这时候就需要改为 1010(3) + 1001(3)，第一个数的精度由此改变了，精度只是一个辅助运算值，对于读取对象记录的大整数具体值没有影响，所以他被 transient 修饰，标度才是真爱。

BigDecimal 精度计算原理比较简单暴力，转换为整数运算再返回。例如 2.00001 会被转换为 BigInteger 的 200001 再计算。由于 BigDecimal 将实际存储值 intVal 与 scale 小数位数分开存储，所以这个过程非常高效，实际上并没有涉及到 BigDecimal 与 BigInteger 的转换。

BigDecimal 提供了两个比较相关函数: equals 和 compareTo:

- **equals**: 比较属性，本质上比较 intVal, scale。
- **compareTo**: 比较值，会比较实际的 BigDecimal 含义值。

什么叫比较实际含义值，例如 1.0 与 1.00 进行比较:

- equals: 两个值的 intVal 和 scale 分别是 (10,1) 和 (100,2) 不同，直接返回 false;
- compareTo: 比较标度 1,2 发现不同，进行标度对其操作: (10,1) → (100,2) 再进行比较，返回 true。

有一个常见的坑，不要使用 double, float 这样精度不确定的值构造 Decimal，应该优先使用 String，具体原因不解释了。

II 注解与反射

5 Annotation 注解

Java 注解它提供了一种安全的类似注释的机制，用来将任何的信息或元数据（metadata）与程序元素（类、方法、成员变量等）进行关联。

Java 注解分为三类:

- 标准注解: Java 自带的一些功能性注解。
- 元注解: 用于定义注解的注解。
- 自定注解: 自己定义的注解 (略)。

5.1 标准注解

@Override

功能: 检查该方法是否是重写方法，如果发现其父类，或者是引用的接口中没有该方法时，会报编译错误。

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.SOURCE)
public @interface Override {}
```

用法: 直接放在要重写的函数上，没有参数。

@Deprecated

功能: 用于标明被修饰的类或类成员、类方法已经废弃、过时，不建议使用。

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(value={CONSTRUCTOR, FIELD, LOCAL_VARIABLE, METHOD, PACKAGE, MODULE, PARAMETER, TYPE})
public @interface Deprecated {
    String since() default "";
    boolean forRemoval() default false;
}
```

参数说明 (两个参数都是 Java9 新增的):

- since: 指定已注解的 API 元素已被弃用的版本。
- forRemoval: 是否在将来的既定版本中会被删除。

```
@Deprecated(since = "1.2", forRemoval = true)
```

@SuppressWarnings

功能: 用于关闭对类、方法、成员编译时产生的特定警告。

```
@Target({TYPE, FIELD, METHOD, PARAMETER, CONSTRUCTOR, LOCAL_VARIABLE, MODULE})
@Retention(RetentionPolicy.SOURCE)
public @interface SuppressWarnings {
    String[] value();
}
```

value 的常见参数如下:

- deprecation: 使用了不赞成使用的类或方法
- unchecked: 执行了未检查的转换
- fallthrough: 当 Switch 程序块直接通往下一种情况而没有 break
- path: 在类路径，源文件路径等中有不存在的路径时的警告
- serial: 当在可序列化的类上缺少 serialVersionUID 定义时的警告
- finally: 任何 finally 子句不能正常完成时的警告
- all: 所有的警告

```
// @SuppressWarnings("unchecked")              // 抑制单类型警告
@SuppressWarnings(value={"unchecked", "rawtypes"}) // 抑制多类型警告
public void addItems(String item){
  @SuppressWarnings("rawtypes")
   List items = new ArrayList();
   items.add(item);
}
```

@FunctionalInterface

用于指示被修饰的接口是函数式接口

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface FunctionalInterface {}
```

函数式接口 (Functional Interface) 就是一个有且仅有一个抽象方法，但是可以有多个非抽象方法的接口。

5.2 元注解

@Retention

定义该注解在哪一个级别可用，在源代码中 (SOURCE)、类文件中 (CLASS) 或者运行时 (RUNTIME)。

```java
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.ANNOTATION_TYPE)
public @interface Retention {
    RetentionPolicy value();
}

public enum RetentionPolicy {
    SOURCE , CLASS , RUNTIME
}
```

value 的常见参数如下:

- RetentionPolicy.SOURCE: 编译时被丢弃，只在源文件中出现
- RetentionPolicy.CLASS: 编译器将注解记录在类文件中，但不会加载到 JVM 中，是默认值
- RetentionPolicy.RUNTIME: 注解信息会保留在源文件、类文件中，在执行的时也加载到 Java 的 JVM 中，因此可以反射性的读取

那怎么来选择合适的注解生命周期呢? [1] 首先要明确生命周期长度 SOURCE < CLASS < RUNTIME ，所以前者能作用的地方后者一定也能作用。一般如果需要在运行时去动态获取注解信息，那只能用 RUNTIME 注解；如果要在编译时进行一些预处理操作，比如生成一些辅助代码（如 ButterKnife），就用 CLASS 注解; 如果只是做一些检查性的操作，比如 @Override 和 @SuppressWarnings，则可选用 SOURCE 注解。

@Documented

作用: 生成文档信息的时候保留注解，对类作辅助说明。

```java
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.ANNOTATION_TYPE)
public @interface Documented {}
```

如果一个注解 @B，被 @Documented 标注，那么被 @B 修饰的类，生成文档时，会显示 @B。如果 @B 没有被 @Documented 标准，最终生成的文档中就不会显示 @B。

@Target

作用: 用于描述注解的使用范围。

```java
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.ANNOTATION_TYPE)
public @interface Target {
    ElementType[] value();
}
```

[1]引用自:https://blog.csdn.net/weixin_42403127/article/details/115999679

ElementType 是一个枚举类型，它定义了被 @Target 修饰的注解可以应用的范围：

- ElementType.Type: 类，接口 (包括注解)，枚举
- ElementType.CONSTRUCTOR: 构造函数
- ElementType.PARAMETER: 方法的参数
- ElementType.FIELD: 字段或属性
- ElementType.METHOD: 方法
- ElementType.PACKAGE: 包
- ElementType.LOCAL_VARIABLE: 局部变量
- ElementType.TYPE_PARAMETER: 类型变量
- ElementType.TYPE_USE: 任何适用类型的语句中

@Inherited

作用: 子类可以继承父类中的该注解, 自动继承注解类型。如果注解类型声明中存在 @Inherited 元注解，则注解所修饰类的所有子类都将会继承此注解。

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.ANNOTATION_TYPE)
public @interface Inherited {}
```

@Repeatable

作用：注解可以重复使用。

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.ANNOTATION_TYPE)
public @interface Repeatable {
    Class<? extends Annotation> value();
}
```

6 Reflect 反射

JAVA 反射机制是在运行状态中，对于任意一个类，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一个方法和属性; 这种动态获取的信息以及动态调用对象的方法的功能称为 java 语言的反射机制[2]。

实际上，我们创建的每一个类也都是对象，即类本身是 java.lang.Class 类的实例对象。这个实例对象称之为类对象，也就是 Class 对象。

放射机制在框架中使用的非常普遍，理解反射机制有助于理解例如 Spring 等框架。

6.1 Class 对象

在 `Object` 源码中有这样一段描述: Class @code Object is the root of the class hierarchy. 既然类本身是 java.lang.Class 类的实例对象，那么类自身也就可以调用 Class 的一些方法。

`Class` 类的声明如下:

```
public final class Class<T> implements java.io.Serializable, GenericDeclaration, Type,
    AnnotatedElement, TypeDescriptor.OfField<Class<?>>, Constable
```

- `Class` 类的实例对象表示正在运行的 Java 应用程序中的类和接口。也就是 jvm 中有很多的实例，每个类都有唯一的 `Class` 对象。
- `Class` 类没有公共构造方法。`Class` 对象是在加载类时由 Java 虚拟机自动构造的。也就是说我们不需要创建，JVM 已经帮我们创建了。
- `Class` 对象用于提供类本身的信息，比如有几种构造方法，有多少属性，有哪些普通方法。

`CLass` 本身是有 “构造函数” 的，只不过我们无法调用，而由 JVM 调用:

```
// Private constructor. Only the Java Virtual Machine creates Class objects.
// This constructor is not used and prevents the default constructor being generated.
private Class(ClassLoader loader, Class<?> arrayComponentType) {
    classLoader = loader;
    componentType = arrayComponentType;
}
```

获取类对象的方式有三种 (假设有一个 Hero 类):

- 静态方法 (常用): `Class.forName("com.xxx.Hero")`
- 公有属性: `Hero.class`
- 对象方法: `new Hero().getClass()`

其中 `forName` 的源码如下:

```
@CallerSensitive
public static Class<?> forName(String className) throws ClassNotFoundException {
```

[2]本节参考: https://blog.csdn.net/lililuni/article/details/83449088

```
    // native 操作
}
```

这里有个注解 @CallerSensitive 代表这些方法非常危险，不希望开发者调用，并会在 jvm 级别进行检查。forName 的具体操作是使用 native 修饰的，我们无从得知。

为了第一种方法危险但还是常用呢? 当然是全靠同行承托，第二种方法需要导入类的包，依赖性太强。第三种方法都已经有了对象还要反射干嘛。不过一般开发者不需要调用这些方法，做框架的人才回去考虑。

反射机制不是很常用，因此这节不一一讲源码，只对常用的操作做一些解释。

6.2 Constructor 与创建新对象

利用反射机制，一般会需要获取对象实例，获取对象实例的方式则是通过构造函数创建对象实例。

```
Class class = Class.forName("com.xxx.Hero"); // 获取 Class 对象
Constructor con = class.getConstructor();   // 通过 Class 对象获取构造函数
```

Class 类中有两种获取构造函数对象的方法，分别为批量获取构造函数对象，获取有参的构造函数对象:

```
// 无参构造函数对象数组
@CallerSensitive
public Constructor<?>[] getConstructors() throws SecurityException
// 有参构造函数对象，注意参数是类型
@CallerSensitive
public Constructor<T> getConstructor(Class<?>... parameterTypes)
    throws NoSuchMethodException, SecurityException
```

同以上两种方法都只能获得公有的构造方法,想获得私有构造方法只需要将 getConstructor 替换为 getDeclaredConstructor。下文也类似，加上 Declared 基本都代表跳过安全检查，获取私有成员。

这样我们就获取了 Constructor 对象。

```
public final class Constructor<T> extends Executable
```

在 Constructor 类中有功能性方法如下:

```
// 暴力访问，如果参数为 true，忽视访问修饰符
@Override
@CallerSensitive
public void setAccessible(boolean flag)
// 创建实例对象
@CallerSensitive
@ForceInline
public T newInstance(Object ... initargs)
```

这两个方法非常重要，第一个方法让反射机制能够访问到私有成员，这使得我们操作对象更加灵活，同时安全性更低。第二个方法可以让我们通过构造器创建对象，相当于直接 new 一个对象。正因为这两个方法，放射机制才如此强大，同时安全性如此低。

同时也有一些常规方法:

```
// 获取对应的 Class 对象
@Override
public Class<T> getDeclaringClass() {
    return clazz;
}
// 获取对应的 Class 对象名
@Override
public String getName() {
    return getDeclaringClass().getName();
}
// 获取参数类型，还有几个类似的，返回值不同
@Override
public Class<?>[] getParameterTypes() {
    return parameterTypes.clone();
}
```

6.3 Field 与成员变量

利用反射，我们可以对成员的属性进行操作

```
Class clazz = Class.forName("com.xxx.Hero");
Field f = clazz.getDeclaredField("fieldName");
f.set(...);
```

和构造器类似，这里通过 getDeclaredField 获得了一个 Field 对象。`getField` 方法和前面讲的 `getConstructor` 方法类似，这里不再赘述。

首先来看一下 `Field` 的声明:

```
public final class Field extends AccessibleObject implements Member
```

`Field` 中的功能性方法如下

```
// 暴力访问，设置为 true，可忽略访问修饰符
public void setAccessible(boolean flag)
// get/set 方法
public Object get(Object obj) throws IllegalArgumentException, IllegalAccessException
public void set(Object obj, Object value) throws IllegalArgumentException,
    IllegalAccessException
```

对于 get/set 方法，有两点要说明的:

- `obj`: 为什么要加一个 Object 参数，因为我们通过 `getField` 方法获得的 Field 对象仅存储了作为参数名的 name 和类型 type 两个关键属性。我们并没有将这个 Field 绑定到

某个对象上，即这个 Field 是一个有名字的对象，仅此而已。

- 基本类型：可以看到这里只能设置 Object 类型，基本类型有专门的方法，例如 `getInt`，`setFloat`。

`getDeclaredField` 可以获取本类所有的字段，包括 private 的，但是不能获取继承来的字段。(注：这里只能获取到 private 的字段，但并不能访问该 private 字段的值，除非加上 `setAccessible(true)`)

6.4 Method 与成员方法

利用反射，我们可以对成员的方法进行操作

```
Class clazz = Class.forName("com.xxx.Hero");
Method m = clazz.getDeclaredMethod("setName", String.class); // Hero 有一个 setName 方法
m.invoke();
```

首先看一下 `Method` 的声明:

```
public final class Method extends Executable
```

`Method` 的功能性方法如下:

```
// 暴力访问，设置为 true，可忽略访问修饰符
public void setAccessible(boolean flag)
// 调用函数
@CallerSensitive
@ForceInline
@IntrinsicCandidate
public Object invoke(Object obj, Object... args) throws IllegalAccessException,
    IllegalArgumentException, InvocationTargetException
```

理解起来和前面一样，没啥好嗦的。

III 其他重要库

7 Interface 接口

7.1 Appendable

如果某个类的实例打算接收取自 java.util.Formatter 的格式化输出，那么该类必须实现 Appendable 接口。这个接口在 IO 相关的类中被经常使用。

```
public interface Appendable {
    Appendable append(CharSequence csq) throws IOException;
    Appendable append(CharSequence csq, int start, int end) throws IOException;
    Appendable append(char c) throws IOException;
}
```

7.2 AutoCloseable

AutoCloseable 和 try-with-resource 语句相关，如果要在该语句中声明某个对象，该对象必须实现 AutoCloseable 接口，try-with-resource 语句块会在结束后自动调用 close 方法。

```
public interface AutoCloseable {
    void close() throws Exception;
}
```

7.3 CharSequence

CharSequence 是用于统一字符串操作的，String, StringBuilder, StringBuffer 都实现了 CharSequence。该接口提供了很多方法，部分给予了默认实现:

```
public interface CharSequence {
    int length();
    char charAt(int index);
    default boolean isEmpty() {
        return this.length == 0;
    }
    @NotNull
    CharSequence subSequence(int start, int end);
    @NotNull
    public String toString();
    @Contract(pure=true) @NotNull
    public default IntStream chars() {...}
    @Contract(pure=true) @NotNull
```

```
    public default IntStream codePoints() {...}
    public static int compare(CharSequence cs1, CharSequence cs2) {...}
}
```

7.4 Cloneable

Cloneable 接口是一个标记接口，只有实现这个接口后，然后在类中重写 Object 中的 clone 方法，然后通过类调用 clone 方法才能克隆成功，如果不实现这个接口，则会抛出 CloneNotSupportedException(克隆不被支持) 异常。

默认的 Object 的 clone 方法是浅拷贝，如果我们实现了 Cloneable 并重写 clone 方法，需要尽量实现深拷贝。

```
@IntrinsicCandidate
protected native Object clone() throws CloneNotSupportedException;
```

7.5 Comparable

Comparable 是一个很常见的接口，用于实现比较：

```
public interface Comparable<T> {
    @Contract(pure = true)
    public int compareTo(T o);
}
```

7.6 Iterator

Iterator 为迭代器对象，还有一个 Iterable 为可迭代对象，Iterable 有一个 iterator 方法返回了 Iterator 对象。

首先看一下 Iterator 接口：

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    default void remove() {
        throw new UnsupportedOperationException("remove");
    }
    default void forEachRemaining(Consumer<? super E> action) {
        Objects.requireNonNull(action);
        while (hasNext())
            action.accept(next());
    }
}
```

前两个方法是 Iterator 的主要方法，用于迭代下一个对象，具体实现由子类负责，最后一个方法和 for-each 语句相关。

其次是 Iterable 接口:

```
public interface Iterable<T> {
    Iterable<T> iterator();
    default void forEach(Consumer<? super T> action) {
        Objects.requireNonNull(action);
        for (T t : this) {
            action.accept(t);
        }
    }
    default Spliterator<T> spliterator() {
        return Spliterators.spliteratorUnknownSize(iterator(), 0);
    }
}
```

这里只有一个主要方法 iterator，用于返回一个 Iterable 对象。

在看完这两个接口后，我们看一下 for-each 语法糖的实际处理过程:

```
// 语法糖
for (Integer i:list) {
    System.out.println(i);
}
// 反编译
Integer i;
for (Iterator iterator = list.iterator(); i = (Integer)iterator.next();) {
    System.out.println(i);
}
```

所以，为什么 Iterable 返回 Iterator 却不直接在 Iterable 内部实现 Iterator 方法？原因是有些集合类可能不止一种遍历方式，实现了 Iterable 的类可以再实现多个 Iterator 内部类。

7.7 Readable

如果一个类继承了 readable 接口并实现了 read 方法，我们就可以使用 scanner 类来进行操作。

```
public interface Readable {
    public int read(java.nio.CharBuffer cb) throws IOException;
}
```

7.8 Runnable

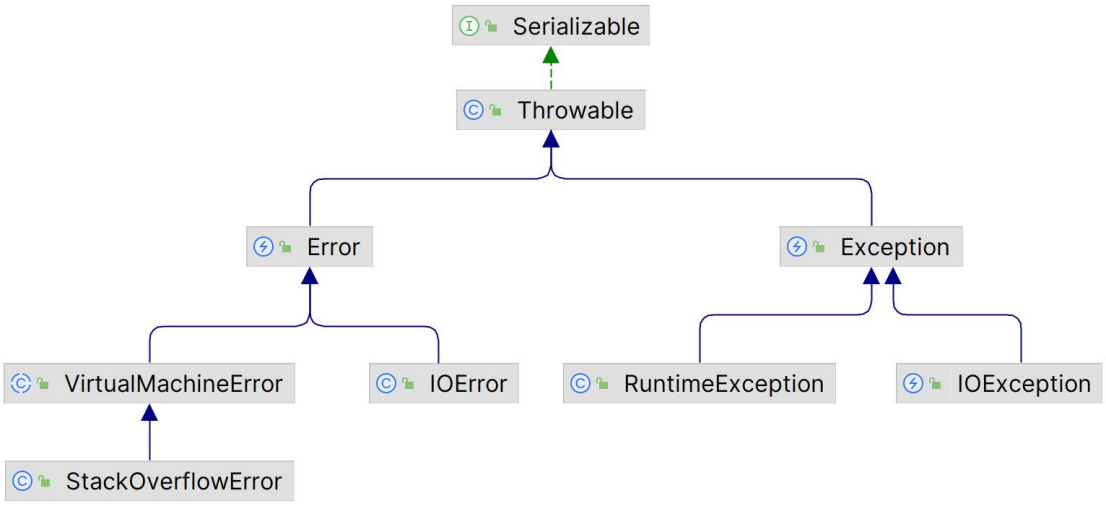
Runnable 是实现函数式接口，他有一个 run 方法，多用在线程中:

```
@FunctionalInterface
```

```
2 public interface Runnable {
3     public abstract void run();
4 }
```

8 Exception 异常

Java 的异常继承结构如下:



Java 异常可以分为两类：检查型异常与非检查型异常:

- **检查型异常**
  - 范围：Error 与 RuntimeException 以外的异常
  - 定义：编译器要求必须处理的异常，编译器会在编译前检查代码，没处理这些异常不能编译。
  - 处理：继续 throws 抛出，或者 try...catch 处理。
- **非检查型异常**
  - 范围：Error 与 RuntimeException
  - 定义：编译器不要求强制处置的异常，虽然你有可能出现错误，但是我不会在编译的时候检查，没必要，也不可能。
  - 处理：捕获，抛出需要程序员预知有这些错误，不会静态提示。或者不处理，让程序出错。

也可以分为运行时异常与非运行时异常:

- 运行时异常：RunTimeException 都是非检查型异常，例如空指针，下标越界，一般是逻辑错误引起的。
- 非运行时异常：RunTimeException 以外的异常，从语法角度将必须在编译前处理。

8.1 Error

Error 的源码非常简单，抛出错误时可以携带消息或者错误原因，其他错误的源码也几乎一样，部分源代码如下:

```
1 public class Error extends Throwable {
2     public Error(String message) {
```

```
        super(message);
    }
    public Error(String message, Throwable cause) {
        super(message, cause);
    }
    public Error(Throwable cause) {
        super(cause);
    }
}
```

第二部分

Java 容器

IV 容器工具

9 容器基础与工具

9.1 Arrays 数组工具类

9.1.1 Array 动态数组类

与这几章大部分类不同的是，Array 类位于 java.lang.reflect 包下，其他都位于 java.util 包下。可能是因为 Array 都是 native 方法。

Array 为数组类，为我们提供了动态创建和访问 Java 数组的方法。它是最高效的，但是其容量是固定的，并且无法动态改变，且存放同一数据类型的数据。与 Arrays 相同，他的所有方法都是静态的，并且全部方法根本上都是 native 的。

Array 只包含三类方法: newInstance 用于创建数组；get, set 方法用于获取与设置数组元素，由于是 native 的不讲了。

newInstance 创建数组

newInstance 方法是唯一不直接被 native 修饰的方法，用于创建一个定长数组:

```
public static Object newInstance(Class<?> componentType, int length)
    throws NegativeArraySizeException {
    return newArray(componentType, length);
}

@IntrinsicCandidate
private static native Object newArray(Class<?> componentType, int length)
    throws NegativeArraySizeException;
```

可以看到，本质上还是用 native 方法实现，这也是 Array 高效的原因。

newInstance 还有一个重载的同名函数，用于创建多维数组。

9.1.2 Arrays 工具类

Arrays 包含各种操作数组的方法 (例如排序和搜索)。该类还包含一个静态工厂，允许将数组视为列表。Arrays 类里的方法都是静态方法, 下面介绍几个常用方法。

sort 排序

排序，默认使用双轴快速排序算法，在排序前会对传入的下标 (如果有) 进行检查。

```java
public static void sort(int[] a, int fromIndex, int toIndex) {
    rangeCheck(a.length, fromIndex, toIndex);
    DualPivotQuicksort.sort(a, 0, fromIndex, toIndex);
}
```

JDK8 提供了另一种排序方案: parallelSort 用于处理大数据。它的实现逻辑和 sort 类似。

```java
public static void parallelSort(int[] a, int fromIndex, int toIndex) {
    rangeCheck(a.length, fromIndex, toIndex);
    DualPivotQuicksort.sort(a, ForkJoinPool.getCommonPoolParallelism(), fromIndex, toIndex);
}
```

可以看出，两种排序方式仅有一个值不同，这两个啊脾虚方法特点如下:

- **sort**: 单线程，适用于处理小数据。
- **parallelSort**: 多线程并行排序，将数据分段调用多核心处理再合并，是用于处理大量数据。

一般的，数组长度大于 10000，采用并行排序，其他采用普通排序即可。

此外，还有一个私有的 legacyMergeSort 方法:

```java
private static void legacyMergeSort(Object[] a) {
    Object[] aux = a.clone();
    mergeSort(aux, a, 0, a.length, 0);
}
```

JDK7 之前用的是归并排序，这个方法是用来兼容旧版的，此外所有归并排序都是私有方法，只有特性情况下会在 sort 中调用该方法。

parallelPrefix 并行相关计算

给出某种运算方法，对数组中下标为 1 的元素开始进行计算:

```java
public static void parallelPrefix(int[] array, IntBinaryOperator op) {
    Objects.requireNonNull(op);
    if (array.length > 0)
        new ArrayPrefixHelpers.IntCumulateTask
                (null, op, array, 0, array.length).invoke();
}
```

其中 IntBinaryOperator(不同类型有对应不同的接口) 是函数式编程接口:

```java
@FunctionalInterface
public interface IntBinaryOperator {
    int applyAsInt(int left, int right);
}
```

一般的，配合 lambda 表达式使用，例如:

```java
int arr[] = {1,2,3,4};
Arrays.parallelPrefix(arr, (left, right) -> left + right);
```

```
// arr: {1,3,6,10}
```

有两点需要注意:

- 运算从第二个元素开始，即下标为 1 的元素。
- 每次运算都会修改元素值。

可以将其看作循环运算的一种简便写法，不过用的人不多。

setAll 迭代计算

和 parallelPrefix 类似，不过 parallelPrefix 会计算前后相关的元素，但 setAll 不会。还有一个并行方法: parallelSetAll

```
public static <T> void setAll(T[] array, IntFunction<? extends T> generator) {
    Objects.requireNonNull(generator);
    for (int i = 0; i < array.length; i++)
        array[i] = generator.apply(i);
}
```

copyOf 复制数组

有两个复制数组的函数如下:

```
public static int[] copyOf(int[] original, int newLength)
public static int[] copyOfRange(int[] original, int from, int to)
```

用于复制一个新的数组，可以指定长度或范围。

copyOf 比较常用，源码如下:

```
@IntrinsicCandidate
public static <T,U> T[] copyOf(U[] original, int newLength, Class<? extends T[]> newType) {
    @SuppressWarnings("unchecked")
    T[] copy = ((Object)newType == (Object)Object[].class)
        ? (T[]) new Object[newLength]
        : (T[]) Array.newInstance(newType.getComponentType(), newLength);
    // 这是个 native 方法
    System.arraycopy(original, 0, copy, 0, Math.min(original.length, newLength));
    return copy;
}
```

其他常用方法

```
// 元素交换
private static void swap(Object[] x, int a, int b)
// 二分查找
public static int binarySearch(int[] a, int fromIndex, int toIndex,int key)
// 等于判断
```

```
public static boolean equals(int[] a, int[] a2)
// 比较判断
public static int compare(int[] a, int[] b)
// 找第一个不相等的元素
public static int mismatch(int[] a, int[] b)
// 填充：改变范围内元素值
public static void fill(int[] a, int fromIndex, int toIndex, int val)
// 计算 hashCode
public static int hashCode(int a[])
// toString
public static String toString(int[] a)
// 流处理，返回 Stream
public static <T> Stream<T> stream(T[] array, int startInclusive, int endExclusive)
// 转换为集合
public static <T> List<T> asList(T... a);
```

并行迭代:在多线程中会应用,参考文章: https://blog.csdn.net/sunboylife/article/details/103307072

```
public static <T> Spliterator<T> spliterator(T[] array, int startInclusive, int endExclusive)
```

9.2 Collections 集合工具类

Collections 不仅对 Collection 接口提供了方法，对 Map 也提供了一些处理方法。

功能性函数

Collection 通用:

```
// 找最小值
public static <T extends Object & Comparable<? super T>> T min(Collection<? extends T> coll)
// 找最大值
public static <T extends Object & Comparable<? super T>> T max(Collection<? extends T> coll)
// 返回视图
public static <E> Collection<E> checkedCollection(Collection<E> c, Class<E> type)
// 获取对象出现次数
public static int frequency(Collection<?> c, Object o)
// 判断结合是否相交
public static boolean disjoint(Collection<?> c1, Collection<?> c2)
// 添加元素
public static <T> boolean addAll(Collection<? super T> c, T... elements)
```

针对 List 类型，Arrays 有的方法几乎都有，此外还添加的函数有:

```
// 洗牌，打乱
public static void shuffle(List<?> list)
// 左移/右移
public static void rotate(List<?> list, int distance)
// 找子列表
```

```
public static int indexOfSubList(List<?> source, List<?> target)
// 找最后一个子列表
public static int lastIndexOfSubList(List<?> source, List<?> target)
```

获取集合对象

Collections 写了很多内置类，方便开发者使用。

```
// 获取不可变的集合对象
public static <T> Collection<T> unmodifiableCollection(Collection<? extends T> c)
// 获取线程安全的集合对象
public static <T> Collection<T> synchronizedCollection(Collection<T> c)
// 返回空且不可变对象
public static final <T> List<T> emptyList()
// 返回特定且不可变对象
public static <T> List<T> singletonList(T o)
// 转换为枚举
public static <T> Enumeration<T> enumeration(final Collection<T> c)
public static <T> ArrayList<T> list(Enumeration<T> e)
```

V 列表，集合与队列

10 Collection 与 Map 接口

10.1 Collection 接口与默认实现

10.1.1 Collection 接口

Java 容器接口有两大接口: Collection 和 Map。Map 用于存放键值对，Collection 用于存放单一元素。Collection 派生的容器有 List, Set, Queue。

Collection 接口定义了最基础的容器方法，这里简单说明:

```
public interface Collection<E> extends Iterable<E>
```

- int size(): 获取容器元素数量。
- boolean isEmpty(): 判断容器是否为空。
- boolean contains(Object o): 判断元素是否存在。
- Iterator<E> iterator(): 获取容器迭代对象。
- Object[] toArray(): 转换为数组。
- boolean add(E e): 添加元素。
- boolean remove(Object o): 删除元素。
- boolean containsAll(Collection<?> c): 判断是否包含容器。
- boolean addAll(Collection<? extends E> c): 添加容器元素。
- boolean removeAll(Collection<?> c): 删除容器元素。
- boolean retainAll(Collection<?> c): 保留容器元素。
- void clear(): 删除本身的元素。

此外还有几个方法给出了默认实现:

removeIf: 删除指定元素

```
default boolean removeIf(Predicate<? super E> filter) {
    Objects.requireNonNull(filter);
    boolean removed = false;
    final Iterator<E> each = iterator();
    while (each.hasNext()) {
        if (filter.test(each.next())) {
            each.remove();
            removed = true;
        }
    }
    return removed;
```

```
}
```

spliterator: 并行迭代

```
@Override
default Spliterator<E> spliterator() {
    return Spliterators.spliterator(this, 0);
}
```

Steam: 流化处理

```
default Stream<E> stream() {
    return StreamSupport.stream(spliterator(), false);
}

default Stream<E> parallelStream() {
    return StreamSupport.stream(spliterator(), true);
}
```

10.1.2 AbstractCollection 默认实现

AbstractCollection 是 Collection 的默认实现，完成了大部分方法的实现，仍有两个方法需要子类实现:

```
public abstract Iterator<E> iterator();
public abstract int size();
```

AbstractCollection 中大部分方法的实现都用到了迭代器。

10.2 Map 接口与默认实现

10.2.1 Map 接口

Map 用于存放键值对，表示一对一的映射关系:

```
public interface Map<K, V>
```

它包含的特有的方法如下:

- boolean containsKey(Object key): 判断键是否存在。
- boolean containsValue(Object value): 判断值是否存在。
- V get(Object key): 通过键获取值。
- V getOrDefault(Object key, V defaultValue): get 的升级版。
- V put(K key, V value): 加入键值对。
- V putIfAbsent(K key, V value): 不存在才加入键值对。
- V remove(Object key): 根据键盘删除键值对。

- V remove(Object key, Object value): 根据键值删除键值对。
- boolean replace(K key, V value): 替换值
- void putAll(Map<? extends K, ? extends V> m): 加入很多键值对。
- Set<K> keySet(): 获取键集合。
- Collection<V> values(): 获取值容器。
- Set<Map.Entry<K, V> entrySet(): 获取键值对集合。
- static <K, V> Map<K, V> of(K k1, V v1): 返回不可修改的 Map 对象。
- static <K, V> Map<K, V> copyOf(Map<? extends K, ? extends V> map): 赋值对象。
- default void forEach(BiConsumer<? super K, ? super V> action): 遍历执行

此外，Map 有一个内部类接口 Entry<K,V> 用于存储单个键值对。

10.2.2 AbstractMap 默认实现

AbstractSet 的 entrySet 方法未实现，其他大部分需要便利的方法都依赖于该方法。

```
1 public abstract Set<Entry<K,V>> entrySet();
```

entrySet 的返回值必须是一个可迭代的 Set 对象。

章节概述

这章会讲的几个容器如下 (不涉及多线程容器):

表 5.1 容器概述

类型	类名	使用频率	多线程	备注
List	ArrayList	★★★	不安全	List 基本只用 ArrayList
	LinkedList	★	不安全	性能不及 ArrayList
	Vector	★	安全	ArrayList 的古早实现
	Stack	★	安全	栈
Map	Hashtable	★	安全	HashMap 的古早实现
	HashMap	★★★	不安全	Map 默认使用 HashMap
	TreeMap	★★	不安全	有序的 Map
	LinkedHashMap	★	不安全	有序的 Map, 按插入顺序排序
Set	HashSet	★★★	不安全	对应 HashMap
	TreeSet	★★	不安全	对应 TreeMap
	LinkedHashSet	★	不安全	对应 LinkedHashMap
Queue	PriorityQueue	★★	不安全	优先队列，需要顺序取值就用
	ArrayDeque	★★★	不安全	Queue 和栈的默认选择

11 List 列表

```
public interface List<E> extends Collection<E>
```

List 接口继承自 Collection 接口，新增了以下方法:

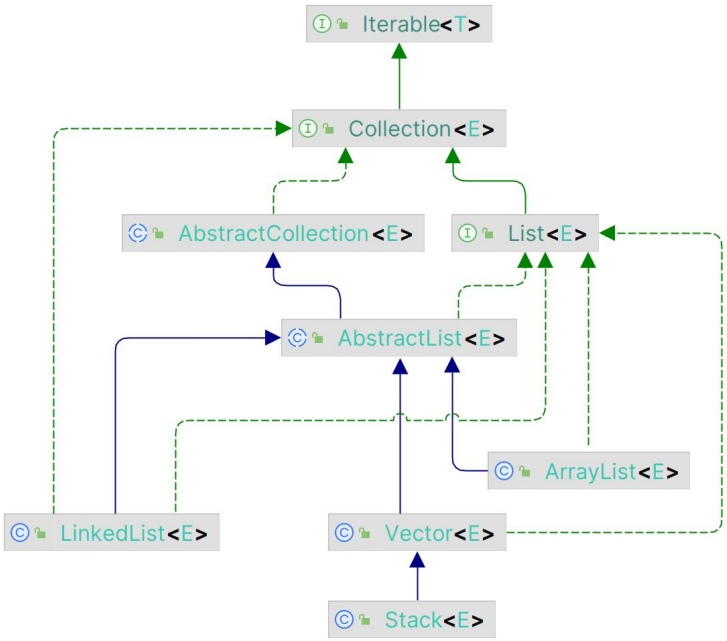
```
// get 与 set 方法
E get(int index);
E set(int index, E element);
// 搜索元素下标
int indexOf(Object o);
int lastIndexOf(Object o);
// 获取子列表
List<E> subList(int fromIndex, int toIndex);
```

此外，还提供了一个静态的 of 方法，用于获取不可变的列表。

```
static <E> List<E> of(E... elements)
```

AbstractList 是 List 的默认实现。

Java 中有四个主要的 List 容器，他们都直接或简介继承/实现了 List 接口与 AbstractList 抽象类，他们的主要继承关系如下:



11.1 ArrayList

ArrayList 是列表中最常用的容器，底层基于数组实现容器大小动态变化，允许 null 存在。一般的，使用列表这种数据结构无脑选 ArrayList。

```
public class ArrayList<E> extends AbstractList<E> implements List<E>, RandomAccess,
    Cloneable, java.io.Serializable
```

其中 RandomAccess 是一个标记接口，表示可以随机访问，ArrayList 底层是数组，自然可以通过下标快速随机访问。

11.1.1 数据存储机制

ArrayList 有一个核心的成员变量 elementData 用于存储数据:

```
transient Object[] elementData;
```

这里有个问题，ArrayList 继承了 Serializable 接口，是可以序列化的，但是存储数据的 elementData 被 transient 修饰，说明 elementData 不想被序列化。

这是因为序列化 ArrayList 的时候，ArrayList 里面的 elementData 未必是满的，比方说 elementData 有 10 的大小，但是我只用了其中的 3 个，就没有必要序列化整个 elementData，具体序列化方式放在了 writeObject 方法中:

```
@java.io.Serial
private void writeObject(java.io.ObjectOutputStream s)
    throws java.io.IOException {
    int expectedModCount = modCount;
    // 序列化非 transient 元素
    s.defaultWriteObject();
    s.writeInt(size);
    for (int i=0; i<size; i++) {
        s.writeObject(elementData[i]);
    }
    if (modCount != expectedModCount) {
        throw new ConcurrentModificationException();
    }
}
```

此外，还有几个重要的成员变量:

```
// 记录实际大小，每次增减元素都会改变
private int size;
// 记录被修改的次数
protected transient int modCount = 0;
```

在 ArrayList 数据更改的过程中，核心操作时调用 Arrays.copyOf 方法对数组进行复制，常见的方法如下:

```
// 构造方法。本质上是根据参数给 elementData 加数据
public ArrayList(int initialCapacity) {
    if (initialCapacity > 0) {
        this.elementData = new Object[initialCapacity];
    } else if (initialCapacity == 0) {
        this.elementData = EMPTY_ELEMENTDATA;
    } else {
        throw new IllegalArgumentException("Illegal Capacity: "+ initialCapacity);
    }
}
```

```java
// clone 方法
public Object clone() {
    try {
        ArrayList<?> v = (ArrayList<?>) super.clone();
        v.elementData = Arrays.copyOf(elementData, size);
        v.modCount = 0;
        return v;
    } catch (CloneNotSupportedException e) {
        throw new InternalError(e);
    }
}
```

transient 还有很多类似的用法，其本质都是通过优化序列化过程减轻数据传输与存储压力，下文不再赘述。

11.1.2 扩容机制

> **注意 11.1.** 本人查资料的时候发现，*JDK8* 与 *JDK17* 扩容机制的核心代码有很大不同，本文以 *JDK17* 为准。

ArrayList 的强大之处就在于它实现了数组长度的动态变化，本质上还是 Arrays.copyOf 复制数组。理解扩容机制的切入点是 add 方法:

```java
public boolean add(E e) {
    modCount++;
    add(e, elementData, size);
    return true;
}

private void add(E e, Object[] elementData, int s) {
    if (s == elementData.length)
        elementData = grow();
    elementData[s] = e;
    size = s + 1;
}
```

往下找，可以找到扩容机制的核心方法: grow

```java
private Object[] grow() {
    return grow(size + 1);
}

private Object[] grow(int minCapacity) {
    int oldCapacity = elementData.length;
    // 非空列表扩容
    if (oldCapacity > 0 || elementData != DEFAULTCAPACITY_EMPTY_ELEMENTDATA) {
        // newCapacity 是旧容器的 1.5 倍
        int newCapacity = ArraysSupport.newLength(oldCapacity,
                minCapacity - oldCapacity, /* minimum growth */
                oldCapacity >> 1           /* preferred growth */);
        return elementData = Arrays.copyOf(elementData, newCapacity);
    // 空列表扩容
```

```
    } else {
        // 10 或 要扩的大小
        return elementData = new Object[Math.max(DEFAULT_CAPACITY, minCapacity)];
    }
}
```

因此，我们可以进行总结: 如果是一个空列表，扩容增加 10，如果不是，扩容增加 1.5 倍。这里有两个注意点:

- 对于空列表，ArrayList 有一个 static final 的空数组，每次都会调用它，是个小优化。
- 对于 1.5 倍扩容，采用的是去尾法，15 → 22，但是 1 会扩为 2。

此外，ArrayList 还提供了一个缩容方法:

```
public void trimToSize() {
    modCount++;
    if (size < elementData.length) {
        elementData = (size == 0)
          ? EMPTY_ELEMENTDATA
          : Arrays.copyOf(elementData, size);
    }
}
```

可以手动调用该方法对容器进行缩容。

11.2 LinkedList

LinkedList 使用双向链表实现列表。理论上 ArrayList 更适合查询，LinkedList 更适合插入操作. 但实际运用中，除非是头部或者尾部插入，LinkedList 会有明显的优势 (因为要先查询再插入)，其他方面并没有任何优势，并且 LinkedList 需要维护链表指针有一定的空间开销，所以在要使用列表结构时 LinkedList 很少用。

因为 LinkedList 比较少用，所以这里不做过多讲解。

```
public class LinkedList<E> extends AbstractSequentialList<E> implements List<E>, Deque<E>,
    Cloneable, java.io.Serializable
```

这里可以看到 LinkedList 直接继承自 AbstractSequentialList, AbstractSequentialList 又直接继承自 AbstractList。AbstractSequentialList 提供了一套基于顺序访问的接口。它提供的方法基本上都是通过 ListIterator 实现。

11.2.1 Node 节点

既然是链表结构，拿必然有节点元素，LinkedList 内部有两个只想头部和尾部的节点。

```
transient Node<E> first;
transient Node<E> last;
```

发现又是 transient 修饰的，查看序列方法:

```
@java.io.Serial
private void writeObject(java.io.ObjectOutputStream s)
    throws java.io.IOException {
    s.defaultWriteObject();
    s.writeInt(size);
    for (Node<E> x = first; x != null; x = x.next)
        s.writeObject(x.item);
}
```

发现传的时候只会依次穿节点中的数据, 节点本身被抛弃了。

Node 本身是 LinkedList 的一个内部类:

```
private static class Node<E> {
    E item;
    Node<E> next;
    Node<E> prev;
    Node(Node<E> prev, E element, Node<E> next) {
        this.item = element;
        this.next = next;
        this.prev = prev;
    }
}
```

常用的方法没什么好介绍的，无非是一些常规的链表操作，看一下构造函数和常用方法:

```
public LinkedList() { }
public LinkedList(Collection<? extends E> c) {
    this();
    addAll(c);
}
```

其中 addAll 并没有调用 add 方法，而是迭代插入新节点。原因之一是这样 modCount 只会加一。

此外，clear 操作也进行了优化，将每个节点的引用关系都赋空，主要是方便 GC:

```
public void clear() {
    for (Node<E> x = first; x != null; ) {
        Node<E> next = x.next;
        x.item = null;
        x.next = null;
        x.prev = null;
        x = next;
    }
    first = last = null;
    size = 0;
    modCount++;
}
```

LinkedList 还实现了很多队列操作，这里不做说明。

11.3 Vector

Vector 是 List 的古早实现，现在已经被 ArrayList 代替，但是 Vector 是线程安全的，ArrayList 和 LinkedList 是线程不安全的。当然，可以从 Collections 调用 synchronizedList 方法获取线程安全列表。

```
public class Vector<E> extends AbstractList<E> implements List<E>, RandomAccess, Cloneable,
    java.io.Serializable
```

Vector 的主要成员方法如下:

```
protected Object[] elementData;
protected int elementCount;
protected int capacityIncrement;
```

和 ArrayList 不同，没有对 elementData 添加 transient 关键字。

另外不同的是，Vector 需要在构造时指定扩容大小:

```
public Vector(int initialCapacity, int capacityIncrement)
```

它的扩容机制核心代码如下:

```
private Object[] grow(int minCapacity) {
    int oldCapacity = elementData.length;
    int newCapacity = ArraysSupport.newLength(oldCapacity,
            minCapacity - oldCapacity, /* minimum growth */
            capacityIncrement > 0 ? capacityIncrement : oldCapacity
                                    /* preferred growth */);
    return elementData = Arrays.copyOf(elementData, newCapacity);
}
```

这里只有一句和 ArrayList 不同: capacityIncrement > 0 ? capacityIncrement : oldCapacity

- 如果没有显示指定 capacityIncrement 大小，capacityIncrement 为 0，因此每次扩容大小为初始 Vector 大小。
- 如果显示制定了 capacityIncrement 大小，每次按 capacityIncrement 大小扩容。

此外，Vector 线程安全的原理是加了 synchronized 关键字。具体方法实现和 ArrayList 差别不大。

11.4 Stack

从数据结构上讲，栈应该是独立于列表的一种数据结构，Java 中 Stack 继承自 Vector，因此 Stack 保留了许多和 ArrayList，Vector 类似的性质，且是线程安全的。

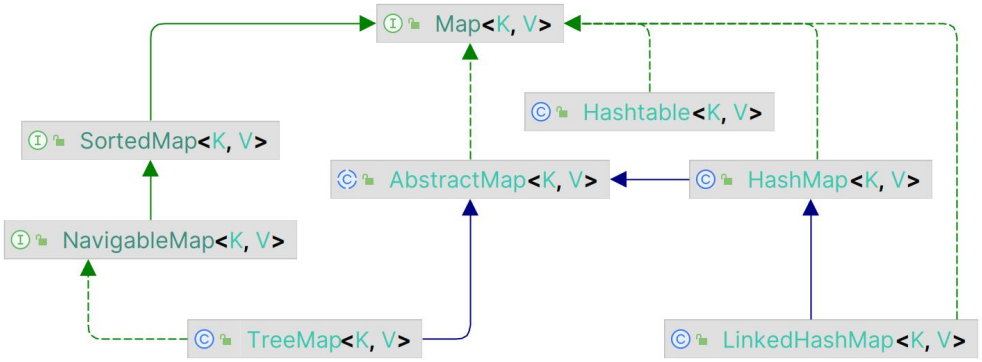
```
public class Stack<E> extends Vector<E>
```

它新增的方法如下:

```java
// 无参构造函数，就这一个构造函数
public Stack()
// 压栈，就调用了一个 synchronized 方法，所以无需 synchronized 关键字
public E push(E item)
// 弹栈
public synchronized E pop()
// 看栈顶元素
public synchronized E peek()
// 判空：和 push 类似，线程安全
public boolean empty()
    return size() == 0;
// 查找，返回的是与栈顶的距离
public synchronized int search(Object o)
```

12 Map 映射

Map 接口已经在前文说明过，不再介绍。Map 的底层实现逻辑是容器中最复杂，但由于 Set 容器需要依赖 Map 实现，所以必须先讲 Map。



12.1 HashTable

Hashtable 是 HashMap 的古早实现，现在已经不推荐使用 Hashtable 而应该使用 HashMap，但 Hashtable 相对较简单，而且和 Vector 与 ArrayList 的关系类似，Hashtable 是线程安全的，这章只利用 Hashtable 作为切入口，简单入门哈希表结构，不做详细说明。

我们知道，底层数据结构的物理存储结构只有两种：顺序存储结构与链式存储结构。List 的 ArrayList 与 LinkedList 底层实现就对应了这两种储存结构。其他所有的数据结构都是基于这两种物理组织形式。

- 顺序存储：直接或间接采用数组实现，需要连续的存储单元存储数据，可以快速查询，但插入删除操作较慢。
- 线性存储：一般采用链表实现，插删操作非常快，查找则需要遍历。

哈希表的底层实现就是这两种存储结构的结合，通过数组快速定位，通过链表解决哈希冲突:

```
1 public class Hashtable<K,V> extends Dictionary<K,V> implements Map<K,V>, Cloneable,
      java.io.Serializable
```

12.1.1 Dictionary 字典

可以看到，Hashtable 直接继承自 Dictionary，熟悉 Python 或者 Json 的读者对字典肯定不陌生，字典是所有实现了键值映射的类的抽象基类，他为我们提供了一些基础方法 (全是抽象的)。

- int size(): 获取大小。
- boolean isEmpty(): 判断是否为空。

- Enumeration<K> keys(): 获取所有键。
- Enumeration<K> elements(): 获取所有值。
- V get(Object key): 通过键获取值。
- V put(K key, V value): 插入键值对。
- V remove(Object key): 通过键删除键值对。

这些方法，Map 接口也基本都定义了，但 Dictionary 并不是 Map，它的抽象程度更高，同时没有 entry，更加 “轻量化”。

比较关键的两个方法 keys(), elements() 返回的是 Enumeration 对象，Enumeration 是一个接口。

```
public interface Enumeration<E>
```

Enumeration 定义了从一个数据结构得到连续数据的手段，它提供了两个主要方法:

- boolean hasMoreElements(): 判断是否还有元素。
- E nextElement(): 获取下一个元素。

还有一个给了默认实现的方法，其实是将上面两个方法封装成迭代器:

```
default Iterator<E> asIterator() {
    return new Iterator<>() {
        @Override public boolean hasNext() {
            return hasMoreElements();
        }
        @Override public E next() {
            return nextElement();
        }
    };
}
```

这些东西全都没实现，留给 Hashtable 完成。

12.1.2 Entry 单向链表

Entry 在 Map 和 Set 中经常使用，不同容器的 Entry 大同小异，下面看一下 Hashtable 的 Entry:

```
private static class Entry<K,V> implements Map.Entry<K,V>
```

Hashtable.Entry 实现了 Map.Entry 接口，Map.Entry 接口定义了对 Entry 的基本操作，包含一些 get,set,compare 方法。Hashtable.Entry 的方法也类似，但它的数据存储结构非常重要，它的重要成员与构造函数如下。

```
final int hash;
final K key;
V value;
Entry<K,V> next;
protected Entry(int hash, K key, V value, Entry<K,V> next) {
```

```
    ...
}
```

可以看到，它包含了三个基本成员，hash 对应哈希值，key,value 为一组键值对，如果 hash 值相同 (出现了哈希碰撞)，则需要通过比较 key 进一步判断 (不比较值肯定是因为值不好比较)。

此外，Entry 只有一个 Entry<K,V> next 用于获取下一个 Entry 对象，这说明 Entry 集合是单向链表的。

12.1.3 哈希表存储原理

理解哈希表实现原理之前先看一下最关键的构造函数，put[1] 与 get 方法。

```
public Hashtable() {
    this(11, 0.75f);
}

public Hashtable(int initialCapacity, float loadFactor) {
    if (initialCapacity < 0)
        throw new IllegalArgumentException("Illegal Capacity: "+
                                           initialCapacity);
    if (loadFactor <= 0 || Float.isNaN(loadFactor))
        throw new IllegalArgumentException("Illegal Load: "+loadFactor);
    if (initialCapacity==0)
        initialCapacity = 1;
    this.loadFactor = loadFactor;
    table = new Entry<?,?>[initialCapacity];
    threshold = (int)Math.min(initialCapacity * loadFactor, MAX_ARRAY_SIZE + 1);
}
```

其中 loadFactor 是负载系数，用于降低哈希碰撞，具体作用在 HashMap 节中说明，initialCapacity 为初始化数组大小。

构造函数本质上初始化了三个值:

```
private float loadFactor; // 负载系数
private transient Entry<?,?>[] table; // 哈希表，用于存储核心数据
private int threshold;   // 用于判断是否要重新哈希
```

最核心的，根据参数 (默认为 11)，创建了一个指定大小的 Entry 数组。然后来看一下加入键值对后会发生的事:

```
public synchronized V put(K key, V value) {
    if (value == null) {
        throw new NullPointerException();
    }
    Entry<?,?> tab[] = table;
    int hash = key.hashCode();
```

[1]为什么不叫 add，因为 Hashtable 不是顺序加入，且需要添加 K-V。

```
    int index = (hash & 0x7FFFFFFF) % tab.length;
    @SuppressWarnings("unchecked")
    Entry<K,V> entry = (Entry<K,V>)tab[index];
    for(; entry != null ; entry = entry.next) {
        if ((entry.hash == hash) && entry.key.equals(key)) {
            V old = entry.value;
            entry.value = value;
            return old;
        }
    }
    addEntry(hash, key, value, index);
    return null;
}

private void addEntry(int hash, K key, V value, int index) {
    Entry<?,?> tab[] = table;
    if (count >= threshold) {
        // Rehash the table if the threshold is exceeded
        rehash();

        tab = table;
        hash = key.hashCode();
        index = (hash & 0x7FFFFFFF) % tab.length;
    }
    // Creates the new entry.
    @SuppressWarnings("unchecked")
    Entry<K,V> e = (Entry<K,V>) tab[index];
    tab[index] = new Entry<>(hash, key, value, e);
    count++;
    modCount++;
}
```

其中 0x7FFFFFFF 是 32 位最大整数，hash 值与他进行与运算能保证结果必为正数，再取余计算出 index 值。然后加入到 table 数组对应的位置中，如果该位置已经有元素，比较 key 值，没有则放在 Entry 单向链表的尾部，有则改变。

为什么要单独 Entry<?,?> tab[] = table 拿一个 table 引用，多线程！

比较重要的，如果哈希表中元素过多，会对哈希表进行 rehash 操作，本质上是扩大 table 数组。

最后看一下 get 方法，逻辑和 put 类似，找到对应 index 和 key 值，取 value:

```
public synchronized V get(Object key) {
    Entry<?,?> tab[] = table;
    int hash = key.hashCode();
    int index = (hash & 0x7FFFFFFF) % tab.length;
    for (Entry<?,?> e = tab[index] ; e != null ; e = e.next) {
        if ((e.hash == hash) && e.key.equals(key)) {
            return (V)e.value;
        }
    }
    return null;
```

```
11 }
```

最后用一张图解释哈希表的存储结构:

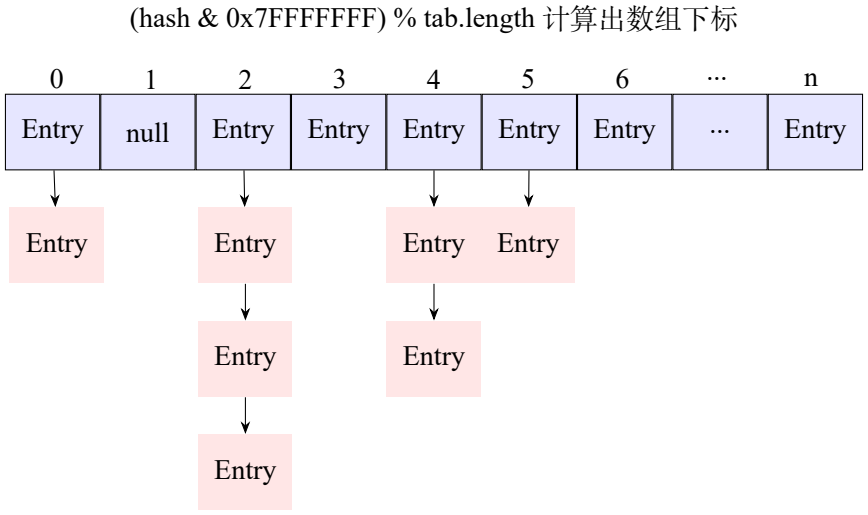


图 12.1 HashTable 存储结构

12.2 HashMap

12.2.1 红黑树

> **注意 12.1.** 这节主要介绍原理，不对具体算法做过多研究。红黑树是一个高度综合的数据结构，涉及到二叉树的平衡问题，染色问题。虽然不是看不懂的数据结构，但也属于看得懂中较难的层次，如果扩展开来从头讲，这一节将成为最长的一节。因此，本节只将最基础的理论，以及 *Java* 的具体实现。

Hashtable 在查找时还存在一个问题：Entry 单链表很长怎么办，无论数组多长，只需要计算一次就可以得到下标值。但单链表不同，他需要每次取 key 值进行比较，如果单链表过长，查找的速度仍然很慢。HashMap 为了解决这个问题加入了新的数据结构: 红黑树。

红黑树是一种自平衡的二叉查找树，二叉查找树时满足以下三个特性的二叉树:

- 若左子树非空，则左子树上所有节点的值都小于根节点的值;
- 若其右子树非空，则右子树上所有节点的值都大于根节点的值;
- 其左右子树都是一棵二叉查找树。

讲人话，如果画的标准，将所有节点映射向下到平面上，节点的值是从大到小排列的。

二叉查找树可以进行十分快速的插入，删除，查找操作，但是它存在一种极端情况，插入的数据是顺序递增或者递减的，这样二叉查找树会退化为单链表。为了解决这个问题，需要平衡二叉查找树。

平衡就是保持每个节点左子树和右子树高度差不超过 1。但如果要满足平衡二叉树，在频繁插入和删除操作过程中，由于二叉树需要旋转保持平衡，非常消耗性能。权衡利弊，Java

选择了红黑树，而红黑树是一种接近平衡的二叉树。

红黑树是一个二叉搜索树，它在每个节点增加了一个存储位记录节点的颜色，可以是 RED, 也可以是 BLACK(实际上用 boolean 标记，true 为 REA，false 为 BLACK)。它具有以下性质:

- 根节点是 BLACK;
- 节点被标记为 RED 或 BLACK;
- 叶子节点都是 null 且为 BLACK，null 节点的父节点在红黑树里不将其看作叶子节点;
- RED 节点的子节点和父节点都是 BLACK 节点 (从根节点到叶子节点的所有路径上不能有 2 个连续的红色节点)。
- 从任一节点到叶子节点 (null 节点) 的所有路径都包含相同数目的黑色节点。

在 Java 中红黑树节点的关键属性如下:

```
static final class TreeNode<K,V> extends LinkedHashMap.Entry<K,V> {
    TreeNode<K,V> parent;
    TreeNode<K,V> left;
    TreeNode<K,V> right;
    TreeNode<K,V> prev;
    boolean red;
}
```

此外，还封装了如旋转，平衡，查找等操作，实际开发过程中，我们并不会用到这些操作。

12.2.2 底层实现

在 JDK8 之前，HashMap 采用的是链表哈希，和前面 Hashtable 原理一致。

首先看一下构造方法:

```
public HashMap() {
    this.loadFactor = DEFAULT_LOAD_FACTOR; // all other fields defaulted
}
```

默认构造函数只指定了一个负载系数，这玩意在 HashMap 中实际起作用只有两句 (resize 和 putMapEntries 方法中) 同样的话:

```
float ft = ((float)s / loadFactor) + 1.0F;
int t = ((ft < (float)MAXIMUM_CAPACITY) ? (int)ft : MAXIMUM_CAPACITY);
if (t > threshold) threshold = tableSizeFor(t);
```

其作用是决定哈希表数组的大小，loadFactor 越大，数组长度越小，空间利用率越高，哈希冲突发生可能性越高。一般的使用默认的 0.75 即可。

接下来看 put 方法:

```
public V put(K key, V value) {
    return putVal(hash(key), key, value, false, true);
```

```
}
```

这个 putVal 方法很长，先看 hash 方法:

```
static final int hash(Object key) {
    int h;
    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
}
```

不同于 Hashtale 直接用 key.hashCode() 作为哈希值，HashMap 加了一步运算，这样的好处如下:

- 右移 16 位是为了让高 16 位也参与运算，可以更好的均匀散列，减少碰撞。
- 异或运算是为了更好保留两组 32 位二进制数中各自的特征。

总而言之，这样一算，就更难发生哈希碰撞了。

下面看 putVal 方法 (非常恐怖):

```
final V putVal(int hash, K key, V value, boolean onlyIfAbsent, boolean evict) {
    Node<K,V>[] tab; Node<K,V> p; int n, i;
    // 判断 table 表是否为空，为空则 resize 整一个
    if ((tab = table) == null || (n = tab.length) == 0)
        n = (tab = resize()).length;
    // 判断 table 表指定位置中元素是否为 null
    if ((p = tab[i = (n - 1) & hash]) == null)
        tab[i] = newNode(hash, key, value, null); // null 则直接放 Node
    else { // 不是 null 就麻烦了
        Node<K,V> e; K k;
        // 通过 hash 和 key 判断相同
        if (p.hash == hash && ((k = p.key) == key || (key != null && key.equals(k))))
            e = p;
        // 如果这个元素是 TreeNode 类型，放入红黑树
        else if (p instanceof TreeNode)
            e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);
        // 向单链表添加数据
        else {
            for (int binCount = 0; ; ++binCount) {
                if ((e = p.next) == null) {
                    p.next = newNode(hash, key, value, null);
                    // 添加过程中发现单链表长度大于等于 8，转换为 红黑树
                    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))))
                    break;
                p = e;
            }
        }
        // null 特殊处理
        if (e != null) {
```

```
            V oldValue = e.value;
            if (!onlyIfAbsent || oldValue == null)
                e.value = value;
            afterNodeAccess(e);
            return oldValue;
        }
    }
    ++modCount;
    // 判断是否要扩容
    if (++size > threshold)
        resize();
    afterNodeInsertion(evict);
    return null;
}
```

对比 Hashtable，put 方法变复杂的主要原因有如下:

- 红黑树: 需要判断是插入链表还是插入红黑树，还是插入到链表后需要转换为红黑树。
- null: 不同于 Hashtable，HashMap 可以添加键为 null 的一个 Entry，以及值为 null 的多个节点。

对应的，remove 操作也变得十分复杂，但逻辑都类似。

12.3 TreeMap

TreeMap 是只是用红黑树存储的一种映射，他没有使用 hashCode 进行排序，默认使用自然序列 (整数的升序，字符串的字母表排序) 排列，可以通过 Comparator 接口指定排列方式。

```
public class TreeMap<K,V> extends AbstractMap<K,V> implements NavigableMap<K,V>, Cloneable,
    java.io.Serializable
```

```
public TreeMap() {
    comparator = null;
}
```

它实现了 NavigableMap 接口，该接口字面意思是可导航的，而 NavigableMap 接口又继承自 SortedMap 可以理解为是有序映射。

```
public interface NavigableMap<K,V> extends SortedMap<K,V>
public interface SortedMap<K,V> extends Map<K,V>
```

SortedMap 包含以下几个常用方法:

- SortedMap<K,V> subMap(K fromKey, K toKey): 获取键范围子映射。
- SortedMap<K,V> headMap(K toKey): 获取头部子映射，相当于 fromKey 为首个 key。
- SortedMap<K,V> tailMap(K fromKey): 获取尾部子映射，相当于 toKey 为最后一个 key。
- K firstKey() / lastKey(): 获取第一/最后一个 key。

NavigableMap 在 SortedMap 基础上增加了很多类似的新的方法，可以更灵活的返回 K，

V，Entry 对象。

12.4 LinkedHashMap

如果说 TreeMap 是 HashMap - 数组, 那么 LinkedHashMap 就是 HashMap + 双向链表。LinkedHashMap 在不对 HashMap 做任何改变的基础上，给 HashMap 的任意两个节点间加了两条连线 (before 指针和 after 指针)，使这些节点形成一个双向链表。

```
public class LinkedHashMap<K,V> extends HashMap<K,V> implements Map<K,V>
```

在 LinkedHashMapMap 中，所有 put 进来的 Entry 都保存在 HashMap 中，但由于它又额外定义了一个以 head 为头结点的空的双向链表，因此对于每次 put 进来 Entry 还会将其插入到双向链表的尾部。

```
public LinkedHashMap() {
    super();
    accessOrder = false;
}
```

构造方法增添了一个 accessOrder，默认为 false，表示按插入顺序遍历，设置为 true 表示按访问顺序遍历。按插入顺序遍历，这是和 TreeMap 在功能上最大的不同。

为了实现双向链表，LinkedHashMap 对 Node 增添了两个引用:

```
static class Entry<K,V> extends HashMap.Node<K,V> {
    Entry<K,V> before, after;
    Entry(int hash, K key, V value, Node<K,V> next) {
        super(hash, key, value, next);
    }
}
```

类似 LinkedList，LinkedHashMap 也添加了头部和尾部节点:

```
transient LinkedHashMap.Entry<K,V> head;
transient LinkedHashMap.Entry<K,V> tail;
```

具体实现逻辑不讲了，参考 LinkedList。

13 Set 集合

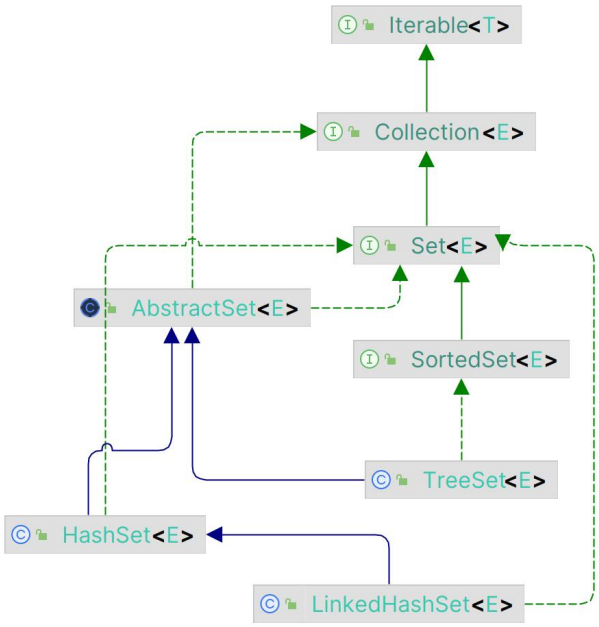
```
public interface Set<E> extends Collection<E>
```

Set 接口继承自 Collection 接口，新增的方法和 List 类似。

Set 的默认实现类 AbstractSet 实现了三个方法:

```
// 判断相等，需要调用 containsAll 方法
public boolean equals(Object o)
// 获取哈希码
public int hashCode()
// 移除所有元素，需要调用 remove 方法
public boolean removeAll(Collection<?> c)
```

Set 的主要子类继承关系如下:



可以发现，Set 的几个实现类和 Map 存在对应关系，因为 Set 的底层就是用 Map 进行存储。会了 Map 基本就会了 Set。这节只讲个 HashSet，其他两个自行理解。

13.1 HashSet

HashSet 底层使用 HashMap 存储数据，本质上是对 HashMap 的封装以及 Set 功能的实现:

```
public class HashSet<E> extends AbstractSet<E> implements Set<E>, Cloneable,
    java.io.Serializable
```

HashSet 有两个关键的成员:

```
private transient HashMap<E,Object> map;
private static final Object PRESENT = new Object();
```

这个 PRESENT 是干什么的呢，我们知道，Map.Entry 必须具有四个属性: hashCode, key, value, next。其中 KV 用于存储键值对，但是 SET 是单个元素的容器，因此，KV 只能留一个，而要留也只能留 K，所以我们必须找个东西填充 V，这个东西就是 PRESENT。因此前面加了 static final 进行优化。

这样我们就能很好地理解 HashSet 实现 SET 功能的原理了: K-V 对中，K 必须是唯一的，V 全是相同的一个对象。到这里，HashSet 就讲完了，它的所有特性都和 HashMap 一致。

```
public boolean add(E e) {
    return map.put(e, PRESENT)==null;
}
public boolean remove(Object o) {
    return map.remove(o)==PRESENT;
}
```

14 Queue 队列

Queue 继承自 Collection 接口，表示单端队列，遵循 FIFO(先进先出) 规则。Deque 继承自 Queue，表示双端队列，两端都可以插删元素:

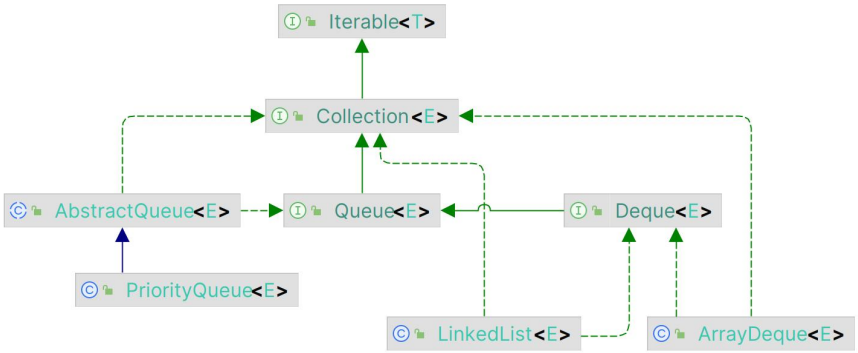
```
public interface Queue<E> extends Collection<E>
public interface Deque<E> extends Queue<E>
```

这两个接口定义的方法比较少，根据因为容量问题而导致操作失败后处理方式的不同可以分为两类方法: 一种在操作失败后会抛出异常，另一种则会返回特殊值。

表 5.2 队列接口

接口	方法	抛异常	返回值
Queue	插入队尾	add(E e)	offer(E e)
	删除队首	remove()	poll()
	查询队首元素	element()	peek()
Deque	插入队首	addFirst(E e)	offerFirst(E e)
	插入队尾	addLast(E e)	offerLast(E e)
	删除队首	removeFirst()	pollFirst()
	删除队尾	removeLast()	pollLast()
	查询队首元素	getFirst()	peekFirst()
	查询队尾元素	getLast()	peekLast()

Deque 还有 push 和 pop 方法，模拟栈操作。



Queue 有一个抽象类 AbstractQueue，看一下它的方法实现就知道，抛异常和返回值函数之间的关系了。

```
public boolean add(E e) {
    if (offer(e))
        return true;
    else
        throw new IllegalStateException("Queue full");
}
```

可以发现，抛异常函数本质是对返回值函数的进一步封装。

14.1 PriorityQueue

PriorityQueue 是优先队列，优先队列的作用是能保证每次取出的元素都是队列中权值最小的 (默认为元素本身的自然顺序)。

```
public class PriorityQueue<E> extends AbstractQueue<E> implements java.io.Serializable
```

在底层，PriorityQueue 使用数组 + 小根堆[2]存储数据。小根堆的实现还是比较简单的，建议读者先了解一下。

```
transient Object[] queue;
public PriorityQueue() {
    this(DEFAULT_INITIAL_CAPACITY, null);
}
```

让我们看一下常规操作的实现，增加元素，小根堆增加元素默认增加在队尾:

```
public boolean add(E e) {
    return offer(e);
}

public boolean offer(E e) {
    if (e == null)
        throw new NullPointerException();
    modCount++;
    int i = size;
    if (i >= queue.length)
        grow(i + 1);
    siftUp(i, e);
    size = i + 1;
    return true;
}
```

我们发现，add 和 offer 没有严格按照抛异常和返回值区分开来，其他几个方法也类似。

这里有两个重要的方法，shiftUp 是元素插入数组尾部，小根堆自底向上递归重构的函数，涉及具体算法，不讲。grow 是数组扩容，注释写的非常明了：容量小于 64，扩容到两倍，否则扩容到 1.5 倍。

```
private void grow(int minCapacity) {
    int oldCapacity = queue.length;
    // Double size if small; else grow by 50%
    int newCapacity = ArraysSupport.newLength(oldCapacity,
            minCapacity - oldCapacity, /* minimum growth */
            oldCapacity < 64 ? oldCapacity + 2 : oldCapacity >> 1
                                       /* preferred growth */);
    queue = Arrays.copyOf(queue, newCapacity);
}
```

其余几个操作如下:

[2]参考文章:https://blog.csdn.net/Zj_boring/article/details/105157360

- boolean remove(Object o): 删除指定元素，没用无参数的函数重载。
- E poll(): 删除队首元素并返回。
- E peek(): return (E) queue[0];
- E element(): 调用 peek，没有则抛错。

14.2 ArrayDeque

ArrayDeque 顾名思义,是使用数组实现的双向队列,是 Queue 的首选实现 (其次是 LinkedList),同时也是栈结构的首选实现 (Java 不推荐使用 Stack)。

```
public class ArrayDeque<E> extends AbstractCollection<E> implements Deque<E>, Cloneable,
    Serializable
```

ArrayDeque 底层采用数组存储数据，同时维护两个指向头部和尾部元素的引用:

```
transient Object[] elements;
transient int head;
transient int tail;
```

如何判断头部和尾部引用呢，头部前一个元素为空，尾部后一个元素为空。因此必须有一个空元素，默认构造函数就预留了一单位的空间:

```
public ArrayDeque() {
    elements = new Object[16 + 1];
}
```

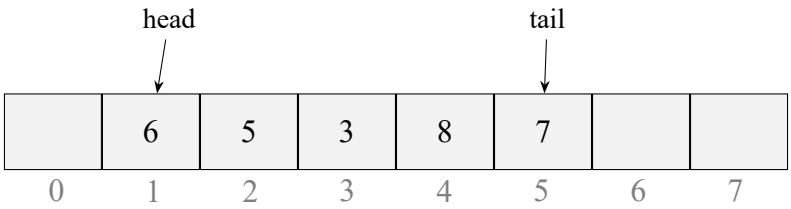


图 14.1 ArrayDeque 内部结构

删除和查找操作没什么好说的，唯一注意的是指针可能会在头部和尾部之间跳转，在插入操作中一并说明，看一下增加操作。

```
public void addFirst(E e) {
    if (e == null)
        throw new NullPointerException();
    final Object[] es = elements;
    es[head = dec(head, es.length)] = e;
    if (head == tail)
        grow(1);
}

static final int dec(int i, int modulus) {
```

```
    if (--i < 0) i = modulus - 1;
    return i;
}
```

扩容操作就不说了，小则两倍，大则增加 50%。这里有个 dec 函数，是为了防止头部指向下标为 0 的区域，如果不是，则直接在 head-1 位置插入元素，否则，在尾部插入元素。

其他就没什么好讲的了，扩容是通过 native 的 copyOf 方法。