Byte源码分析

位置：java.lang

Byte是原生类型byte的包装类，每个Byte类的对象只包含一个原生类型的字段。Byte类的定义如下：

```
public final class Byte extends Number implements Comparable<Byte>
```

Byte继承了Number，Number类的方法是各种原生类型如byte、double、float、int、long、short之间的转换的抽象方法。当Byte实现了intValue()方法，将必选将必须将Byte类中原生类型byte转换为int原生类型。Number类的抽象方法：

```
//转为int原生类型
public abstract int intValue();
//转为long类型
public abstract long longValue();
//转为float类型
public abstract float floatValue();
//转为double类型
public abstract double doubleValue();
//转为byte类型
public byte byteValue() {
        return (byte)intValue();
}
//转为short类型
public short shortValue() {
        return (short)intValue();
}
```

Byte被final修饰符修饰，是不可变的类，当final修饰类，类不可变且不能被继承；当final修饰方法，方法不能被子类重写；当final修饰属性，属性不能改变。Byte有如下属性：

```
//最小值
public static final byte    MIN_VALUE = -128;
//最大值
public static final byte    MAX_VALUE = 127;
//class类型
public static final Class<Byte>      TYPE = (Class<Byte>)
Class.getPrimitiveClass("byte");
//当Byte被表示为二进制时的位数数量
public static final int SIZE = 8;
//表示为二进制时的字节数
public static final int BYTES = SIZE / Byte.SIZE;
//保存原生类型的值
private final byte value;
```

MIN_VALUE、MAX_VALUE、TYPE、SIZE、BYTES属性同时被static和final关键字修饰，都是不可变的静态类属性。value属性只有final修饰，是不可改变的属性，保存Byte类的原生类型的值。MIN_VALUE是Byte的原生类型的最小值，MAX_VALUE是Byte的原生类型的最大值。TYPE是Byte类的class类型。SIZE和BYTES是当Byte类的原生类型的值表示为二进制时的位数数量和字节数。SIZE大小为8，表示Byte类的原生类型value表示为二进制数时，位数大小为8，字节为1。

ByteCache是Byte类的内部静态类，定义如下：

```
private static class ByteCache {
     private ByteCache(){}

     static final Byte cache[] = new Byte[-(-128) + 127 + 1];

      static {
          for(int i = 0; i < cache.length; i++)
              cache[i] = new Byte((byte)(i - 128));
       }
}
```

ByteCache类的主要作用是为Byte类缓存范围为-128到127之间的数，这些数存在Byte类型cache数组中，cache数组大小为256，ByteCache类中使用static模块为Byte类型的cache数组赋值。利用cache数组缓存这些数值，为了减少创建Byte的性能消耗，提高效率。

Byte类没有无参构造器，只有有参构造器，参数分别为byte和String类型，可以将byte和String类型的参数赋值给Byte类的属性value，构造器如下：

```
public Byte(byte value) {
        this.value = value;
}
public Byte(String s) throws NumberFormatException {
        this.value = parseByte(s, 10);
}
```

除了继承父类的方法外，Byte类的方法大多是静态方法，如下两个比较重要的静态方法：

```
//将byte类型的值包装成Byte类型
public static Byte valueOf(byte b) {
    final int offset = 128;
    return ByteCache.cache[(int)b + offset];
}
//将字符串转为byte类型
public static byte parseByte(String s, int radix)
     throws NumberFormatException {
     int i = Integer.parseInt(s, radix);
    //范围判断，byte类型的值在-128到127之间
     if (i < MIN_VALUE || i > MAX_VALUE)
          throw new NumberFormatException(
             "Value out of range. Value:\"" + s + "\" Radix:" + radix);
    //将int强转为byte
      return (byte)i;
}
```

这两个静态方法是比较重要的，Byte类的其他的静态方法大部分都是在这两个方法上进行封装的。valueOf方法是将byte类型的值包装为Byte，直接在ByteCache的缓存数组cache中获取，所以在开发过程中可以直接用Byte类的静态方法valueOf将byte类型包装为Byte，避免创建Byte的性能消耗。parseByte方法是将字符串转为byte类型，第一参数是String，第二参数是int类型的radix，表示将字符串要转为radix进制的byte类型，parseByte方法直接用Integer.parseInt方法先将字符串转为int类型的值，然后将int类型的值强转为byte。

除了静态方法外，Byte类还继承了Number类的方法，将Byte类的原生类型byte转为其他的基础类型，如下：

```
//返回Byte的原生类型byte
public byte byteValue() {
      return value;
}

//将Byte类的原生类型byte强转为short返回
public short shortValue() {
      return (short)value;
}
//将Byte类的原生类型byte强转为int返回
public int intValue() {
     return (int)value;
}

//将Byte类的原生类型byte强转为int返回
public long longValue() {
     return (long)value;
}

//将Byte类的原生类型byte强转为float返回
public float floatValue() {
       return (float)value;
}
//将Byte类的原生类型byte强转为double返回
public double doubleValue() {
      return (double)value;
}
```

最后来看看hashCode方法和equals方法：

```
public boolean equals(Object obj) {
    if (obj instanceof Byte) {
         return value == ((Byte)obj).byteValue();
     }
     return false;
}

public static int hashCode(byte value) {
        return (int)value;
}
```

equals方法首先判断传入的参数类型是否是Byte类型，如果是，就比较Byte类的原生类型value值与传入参数的value值是否相等，否则直接返回false。也就是不管创建多少Byte类的对象，只要他们的原生类型value值相等，那么这些对象就是相等的。hashCode方法是直接返回Byte类型的原生类型value值，Byte类的value的范围为-128到127，所以只会返回值范围内的数，也最多只能只有256个不同的Byte类的对象。

Byte类的源码分析到此，没有什么难点，其他的方法有兴趣的可以自行阅读源代码，Byte类最重要的就是用一个静态内部类ByteCache缓存Byte所有不同的对象，在使用的时候直接通过数组获取。