什么是红黑树

红黑树本质上是一种二叉查找树，但它在二叉查找树的基础上额外添加了一个标记（颜色），同时具有一定的规则。这些规则使红黑树保证了一种平衡，插入、删除、查找的最坏时间复杂度都为 O(logn)。

它的统计性能要好于平衡二叉树（AVL树），因此，红黑树在很多地方都有应用。比如在 Java 集合框架中，很多部分(HashMap, TreeMap, TreeSet 等)都有红黑树的应用，这些集合均提供了很好的性能。

博客

红黑树的五个特征

1. 每个节点要么是红色，要么是黑色；
2. 根节点永远是黑色的；
3. 所有的叶节点都是是黑色的（注意这里说叶子节点其实是上图中的 NIL 节点）；
4. 每个红色节点的两个子节点一定都是黑色；
5. 从任一节点到其子树中每个叶子节点的路径都包含相同数量的黑色节点；

注意：

性质 3 中指定红黑树的每个叶子节点都是空节点，而且并叶子节点都是黑色。但 Java 实现的红黑树将使用 null 来代表空节点，因此遍历红黑树时将看不到黑色的叶子节点，反而看到每个叶子节点都是红色的。

性质 4 的意思是：从每个根到节点的路径上不会有两个连续的红色节点，但黑色节点是可以连续的。因此若给定黑色节点的个数 N，最短路径的情况是连续的 N 个黑色，树的高度为 N - 1;最长路径的情况为节点红黑相间，树的高度为 2(N - 1) 。

性质 5 是成为红黑树最主要的条件，后序的插入、删除操作都是为了遵守这个规定。

红黑树并不是标准平衡二叉树，它以性质 5 作为一种平衡方法，使自己的性能得到了提升。

---

java8引入红黑树

JDK 1.8 以前 HashMap 的实现是 数组+链表，即使哈希函数取得再好，也很难达到元素百分百均匀分布。

当 HashMap 中有大量的元素都存放到同一个桶中时，这个桶下有一条长长的链表，这个时候 HashMap 就相当于一个单链表，假如单链表有 n 个元素，遍历的时间复杂度就是 O(n)，完全失去了它的优势。

针对这种情况，JDK 1.8 中引入了 红黑树（查找时间复杂度为 O(logn)）来优化这个问题。

当链表长度大于等于8时，则会将链表转换为红黑树存储

---

java8移除永久区，加入元空间（MetaSpace），使用物理内存。

Lambda表达式

Lambda是一个**匿名函数**，我们可以把Lambda表达式理解为是一段**可以传递的代码**（将代码像数据一样进行传递）。可以写出更简洁、更灵活的代码。作为一种更紧凑的代码风格，使Java语言表达能力得到了提升。

```
1 /**
```

```
 * 1.Lambda 表达式的基础语法：Java8引入了一个新的操作符 "->" 该操作符称为箭头操作符或Lambda操作符
 *       箭头操作符将Lambda表达式拆分成两部分
 *       左侧：Lambda表达式的参数列表
 *       右侧：Lambda表达式中所需执行的功能，即Lambda体
 *  语法格式一：无参数，无返回值
 *       () -> Sout()
 *
 *  语法格式二：有一个参数，无返回值
 *       (x) -> sout()
 *  语法格式三：若只有一个参数，小括号可以省略不写

 *  语法格式四：两个以上的参数，并且Lambda体中有多条语句，有返回值
 *  语法格式五：若Lambda体中只有一条语句，那么大括号和return可以不写
 *
 *  语法格式六：Lambda表达式的参数列表的数据类型可以省略不写，因为JVM编译器可以通过上下文推断出数据类型，即“类型推断”。
 *
 * 2.Lambda表达式需要"函数式接口"的支持
 *  函数式接口：接口中只有一个抽象方法时，称之为函数式接口。可以使用注解@FunctionalInterface修饰，可以检测这个接口是否是函数式接口
 */


//语法格式一：无参数，无返回值
@Test
public void test1(){
    Runnable r = new Runnable() {
        @Override
        public void run() {
            System.out.println("hello world");
        }
    };
    r.run();
    System.out.println("-------------------");
    Runnable r1 = () -> System.out.println("hello lambda");
    r1.run();
}

//语法格式四：两个以上的参数，并且Lambda体中有多条语句，有返回值
@Test
public void test4(){
    Comparator<Integer> com = (x,y) -> {
        System.out.println("函数式接口");
        return Integer.compare(x,y);
    };
    int compare = com.compare(1, 2);
    System.out.println(compare);
    System.out.println("----------------");

    Comparator<Integer> com1 = (x,y) -> Integer.compare(x,y);
    int compare1 = com1.compare(1, 2);
    System.out.println(compare1);
}
```

Java8内置的四大核心函数式接口

```java
/**
 * Java8内置的四大核心函数式接口
 *
 * 1.Consumer<T>:消费型接口
 *      void accept(T t);
 *
 * 2.Supplier<T>:供给型接口
 *      T get();
 *
 * 3.Function<T,R>:函数型接口
 *      R apply(T t);
 *
 * 4.Predicate<T>:断言型接口
 *      boolean test(T t);
 */

//Consumer<T>消费型接口
@Test
public void test1(){
    happy(10000,(x)-> System.out.println("consumer-------"+x));

}
public void happy(double money, Consumer<Double> con){
    con.accept(money);
}

//Supplier<T>供给型接口
@Test
public void test2(){
    List<Integer> numList = getNumList(10, () -> (int)
(Math.random()*100));
    numList.forEach((e) -> System.out.println(e));
}
//需求：产生指定个数的整数，并放入集合中
public List<Integer> getNumList(int num, Supplier<Integer> sup){
    List<Integer> list = new ArrayList<>();
    for (int i = 0; i < num; i++) {
        Integer e = sup.get();
        list.add(e);
    }
    return list;
}

@Test
public void test3(){
    String s = getStr("hello world", (str) -> str.toUpperCase());
    System.out.println(s);
}
//需求：用于处理字符串
//Function<T,R>:函数型接口
public String getStr(String str, Function<String,String> fun){
    String apply = fun.apply(str);
    return apply;
}

@Test
public void test4(){
    List<String> list = Arrays.asList("hello","world","lambda","ok");
```

```
    List<String> list1 = filterStr(list, (s) -> s.length() > 3);
    list1.forEach((s) -> System.out.println(s));
}
//需求：将满足条件的字符串放入集合中
//Predicate<T>:断言型接口,用于判断
public List<String> filterStr(List<String> list, Predicate<String> pre){
    List<String> strList = new ArrayList<>();
    list.forEach((str) -> {
        if (pre.test(str)){
            strList.add(str);
        }
    });
    return strList;
}
```

Lambda方法引用和构造器引用和数组引用

```
/**
     * 1.方法引用：若Lambda体中的内容有方法已经实现了，我们可以使用"方法引用"，（可以理解为方法引用就是Lambda表达式的另一种表现形式）
     *  主要有三种语法格式
     *      对象::实列方法名
     *      类::静态方法名
     *      类::实列方法名
     */
//类::静态方法名
@Test
public void test1(){
    //Comparator<Integer> comparator = (x,y) -> Integer.compare(x,y); //等同于下面语句
    Comparator<Integer> com = Integer::compare;
    int compare = com.compare(1, 2);
    System.out.println(compare);
}

//类::实列方法名
@Test
public void test2(){
    //若Lambda参数列表中的第一个参数是实列方法的调用者，而第二个参数是实列方法的参数时，可以使用ClassName::method
    BiPredicate<String,String> bp = (x,y) -> x.equals(y);
    BiPredicate<String,String> bp1 = String::equals;
    boolean test = bp1.test("a", "a");
    System.out.println(test);
}

/**
     * 2.构造器引用：
     *  ClassName::new
     *    注意：需要调用的构造器的参数列表要与函数式接口中抽象方法的参数列表保持一致
     */
@Test
public void test3(){
    Supplier<Emp> sup = () -> new Emp();
    //构造器引用
    Supplier<Emp> sup1 = Emp::new;
    Emp emp = sup1.get();
```

```
    System.out.println(emp);
}

/**
     * 3.数组引用
     *      Type::new
     */
@Test
public void test4(){
    Function<Integer,String[]> fun = (x) -> new String[x];
    String[] apply = fun.apply(10);
    System.out.println(apply.length);
    //数组引用
    Function<Integer,String[]> fun1 = String[]::new;
    String[] apply1 = fun1.apply(20);
    System.out.println(apply1.length);
}
```

Stream Api

流(Stream)是什么?

是数据渠道,用于操作数据源(集合,数组等)所生成的元素序列."集合讲的是数据,流讲的是计算"

注意:

1. Stream自己不会存储数据
2. Stream不会改变源对象.相反,它们会返回一个持有结果的新Stream.
3. Stream操作是延迟执行的.这意味着它们会等到需要结果的时候才执行.

```
/**
 * Stream的三个步骤
 *  1.创建Stream
 *  2.中间操作
 *  3.终止操作（中端操作）
 */
```

创建流4种方式

```
//1.创建Stream
@Test
public void test(){
    //1.可以通过Collection系列集合提供的stream()或parallelStream()
    List<String> list = new ArrayList<>();
    Stream<String> stream = list.stream();

    //2.通过Arrays中的静态方法stream()获取数组流
    Emp[] emps = new Emp[10];
    Stream<Emp> stream1 = Arrays.stream(emps);

    //3.通过Stream类中的静态方法of()
    Stream<String> stream2 = Stream.of("aa", "bb", "cc");

    //4.创建无限流
    //4.1迭代
    Stream<Integer> stream3 = Stream.iterate(0, (x) -> x + 2);
    stream3.limit(10).forEach(System.out::println);
```

```
    //4.2生成
    Stream.generate(() -> Math.random()).limit(10).forEach(System.out::println);

}
```

中间操作

多个**中间操作**可以连接起来形成一个**流水线**,**除非流水线上触发终止操作**,否则中间操作**不会执行任何的处理**,而在终止操作时**一次性全部处理**,称为**惰性求值**.

```
/**中间操作
     * 筛选与切片
     * filter--接收Lambda,从流中排除某些元素
     * limit--截断流,使其元素不超过给定数量
     * skip(n)--跳过元素,返回一个扔掉了前n个元素的流.若流中元素不足n个,则返回一个空流.与limit互补
     * distinct--筛选,通过流所生成元素的hashCode()和equals()去除重复元素.
     */
@Test
public void test(){
    //内部迭代:迭代由Stream Api完成
    Stream<Emp> empStream = emps.stream()
        .filter((x) -> x.getAge() > 35)
        .limit(3)
        .skip(1);
    //终止操作,一次性执行全部操作,即"惰性求值"
    empStream.forEach(System.out::println);
}
//外部迭代
//自己写的迭代方法
```

```
/**
     * 映射
     * map--接收Lambda,将元素转换成其他形式或提取信息.接收一个函数作为参数,该函数会被应用到每个元素上,并将其映射成一个新的元素.
     * flatMap--接收一个函数作为参数,将流中的每个值都换成另一个流,然后把所有流连接成一个流
     */
@Test
public void test2(){
    List<String> list = Arrays.asList("aa","b","v");
    list.stream()
        .map((str) -> str.toUpperCase())
        .forEach(System.out::println);

    emps.stream()
        .map(Emp::getName)
        .forEach(System.out::println);
}
```

```java
/**
     * 排序
     * sorted()--自然排序
     * sorted(Comparator com)--定制排序
     */
@Test
public void test4(){
    List<String> list = Arrays.asList("aa","b","v");
    list.stream().sorted().forEach(System.out::println);
}
```

终止操作

```java
/**
     * 查找与匹配
     *  allMatch()--检查是否匹配所有元素
     *  anyMatch()--检查是否至少一个匹配
     *  noneMatch--检查是否没有匹配所有元素
     *  findFirst--返回第一个元素
     *  findAny--返回当前流中的任意元素
     *  count--返回流中元素的总个数
     *  max--返回流中的最大值
     *  min--返回流中的最小值
     */
@Test
public void test(){
    boolean free = emps.stream().allMatch((e) ->
e.getSTATUS().equals("FREE"));
    System.out.println(free);

    boolean r = emps.stream().noneMatch((e) -> e.getSTATUS().equals("r"));
    System.out.println(r);
}
```

```java
/**
     * 规约:
     *  reduce(T identity,BinaryOperator) / reduce(BinaryOperator)--可以将流中元素反复结合起来,得到一个值.
     */
@Test
public void test5(){
    List<Integer> list = Arrays.asList(1,2,3,4,5,6,7);
    Integer sum = list.stream().reduce(0, (x, y) -> x + y);
    System.out.println(sum);
    System.out.println("----------------");
    Optional<Double> optional = emps.stream()
        .map(Emp::getSalary)
        .reduce(Double::sum);
    System.out.println(optional.get());
}

```

```java
/**
     * 收集
```

```
     *  collect--将流转换为其他形式.接收一个Collector接口的实现,用于给Stream中元素汇总的方法
     */
@Test
public void test6(){
    List<String> list = emps.stream()
        .map(Emp::getName)
        .collect(Collectors.toList());
    list.forEach(System.out::println);
    System.out.println("-------------");
    HashSet<String> hashSet = emps.stream()
        .map(Emp::getName)
        .collect(Collectors.toCollection(HashSet::new));
    System.out.println(hashSet);
}
```

```
/**
 * 分组,分区
 */
```

并行流和顺序流

并行流就是把一个内容分成多个数据块，并用不同的线程分别处理每个数据块的流。

Java8中将并行进行了优化，我们可以很容易的对数据进行并行操作。Stream Api可以声明性地通过parallel()与sequential()在并行流与顺序流之间进行切换.

```
@Test
public void test(){
    Instant start = Instant.now();
    LongStream.rangeClosed(0,100000000000L)
        .parallel()
        .reduce(0,Long::sum);
    Instant end = Instant.now();
    System.out.println(Duration.between(start,end).toMillis());
}
```

Optional类

接口中的默认方法和静态方法

Java8允许在接口中实现默认方法,必须用`default`修饰,称为默认方法.

接口默认方法的**"类优先"**原则:若一个接口中定义了一个默认方法,而另外一个父类或接口中又定义了一个同名的方法时,选择**父类中的方法**;

接口冲突:如果一个接口提供一个默认方法,而另一个接口也提供了一个具有相同名称和参数列表的方法,那么必须覆盖该方法来解决冲突.

Java8允许在接口中实现静态方法,使用`static`修饰.

新时间日期Api

新时间日期Api不存在线程安全问题.

```
//1.LocalDate LocalTime LocalDateTime
@Test
public void test(){
    LocalDateTime ldt = LocalDateTime.now();
    System.out.println(ldt);
    LocalDateTime plusYears = ldt.plusYears(2);
    System.out.println(plusYears);
    LocalDateTime minusMonths = ldt.minusMonths(2);
    System.out.println(minusMonths);

    LocalDateTime of = LocalDateTime.of(2015, 1, 1, 1, 1, 1);
    System.out.println(of);
}
```

```
//2.Instant, 时间戳（以Unix元年：1970.01.01 00:00:00 到某个时间之间的毫秒值）
@Test
public void test2(){
    Instant now = Instant.now(); //默认获取UTC时区 2020-05-16T04:02:19.322Z
    System.out.println(now);
    OffsetDateTime offsetDateTime = now.atOffset(ZoneOffset.ofHours(8));
    System.out.println(offsetDateTime);//获取东8区时间

    System.out.println(now.toEpochMilli());//获取时间戳
}
```

```
//3. Duration 计算两个 时间 之间的间隔
//   Period  计算两个 日期  之间的间隔
@Test
public void test3() throws InterruptedException {
    Instant ins1 = Instant.now();
    Thread.sleep(1000);
    Instant ins2 = Instant.now();
    Duration between = Duration.between(ins1, ins2);
    System.out.println(between.toMillis());
    System.out.println(between.getSeconds());

    System.out.println("-------------------");

    LocalDate ld1 = LocalDate.of(2020,1,1);
    LocalDate ld2 = LocalDate.of(2020,5,5);
    Period between1 = Period.between(ld1, ld2);
    System.out.println(between1.getYears());
    System.out.println(between1.getMonths());
    System.out.println(between1.getDays());
}
```

日期的操作

- TemporalAdjuster:时间校正器.有时我们可能需要获取例如:将日期调整到"下个周日"等操作
- TemporalAdjusters: 该类通过静态方法提供了大量的常用TemporalAdjuster的实现.

```java
//DateTimeFormatter : 格式化时间/日期
@Test
public void test5(){
    LocalDateTime ldt = LocalDateTime.now();
    System.out.println(ldt);
    //格式化日期
    DateTimeFormatter dtf = DateTimeFormatter.ofPattern("yyyy/MM/dd hh:mm");
    String format = dtf.format(ldt);
    System.out.println(format);
    //jie'xi
    LocalDate parse = LocalDate.parse(format, dtf);
    System.out.println(parse);
}
```

重复注解和类型注解