## 算法

### LRU

我们可以参考<https://blog.csdn.net/crazyhuntsman/article/details/81981014>

写的不错。

### 排序

<https://www.cnblogs.com/onepixel/articles/7674659.html>

## 设计模式

### Factory

### Builder

### Singleton

### Clone

-- 浅克隆， 深克隆

一个类实现了Cloneable接口，就可以实现克隆了。

但是对于引用属性，还需要将这个引用属性也实现Cloneable接口，才能实现深克隆。

### Adapter

我的理解： 一个controller的需求是实验一个功能C，而这个功能可以利用serviceM的A和B函数完成，我们可以在service中写一个C，把A和B穿起来，但是如果在该类中需要多种像C这样的情况，那么都写到一个service中就显得有些多余了，因此，可以把函数C这类的函数全部提取到一个类中，命名为adapter，让controller调用即可。

### Façade

我的理解：

一个controller的需求是实现一个功能C，但是这个功能需要多个不同的serviceM，serviceN，serviceL来组合完成，那么我们就可以写一个类，把这些函数串起来实现功能C，提供给controller调用，简化controller的代码量，使它可以专注于参数校验等逻辑中，这个类，我们叫façade。

### 组合模式

### 装饰着模式

Wrapper就是利用的这个模式，实现我们想要的对某个函数的小处理。

情景可以参考response的wrapper使用情景。

可能你会问，那么我直接修改response不就可以了吗？呵呵，那如果我既需要原先的response，又需要wapper的内容呢？知道了吧。

## 用接口而不是用实现类编程的好处

## 数据结构

参考文章：<https://www.cnblogs.com/skywang12345/p/3245399.html>

**红黑树的特性**:  
**（1）每个节点或者是黑色，或者是红色。**  
**（2）根节点是黑色。**  
**（3）每个叶子节点（NIL）是黑色。 [注意：这里叶子节点，是指为空(NIL或NULL)的叶子节点！]**  
**（4）如果一个节点是红色的，则它的子节点必须是黑色的。**  
**（5）从一个节点到该节点的子孙节点的所有路径上包含相同数目的黑节点。**

**注意**：  
(01) 特性(3)中的叶子节点，是只为空(NIL或null)的节点。  
(02) 特性(5)，确保没有一条路径会比其他路径长出俩倍。因而，红黑树是相对是接近平衡的二叉树。

**1. 左旋**



对x进行左旋，意味着"将x变成一个左节点"。

左旋的伪代码《算法导论》：参考上面的示意图和下面的伪代码，理解“红黑树T的节点x进行左旋”是如何进行的。

LEFT-ROTATE(T, x)

01 y ← right[x] // 前提：这里假设x的右孩子为y。下面开始正式操作

02 right[x] ← left[y] // 将 “y的左孩子” 设为 “x的右孩子”，即 将β设为x的右孩子

03 p[left[y]] ← x // 将 “x” 设为 “y的左孩子的父亲”，即 将β的父亲设为x

04 p[y] ← p[x] // 将 “x的父亲” 设为 “y的父亲”

05 if p[x] = nil[T]

06 then root[T] ← y // 情况1：如果 “x的父亲” 是空节点，则将y设为根节点

07 else if x = left[p[x]]

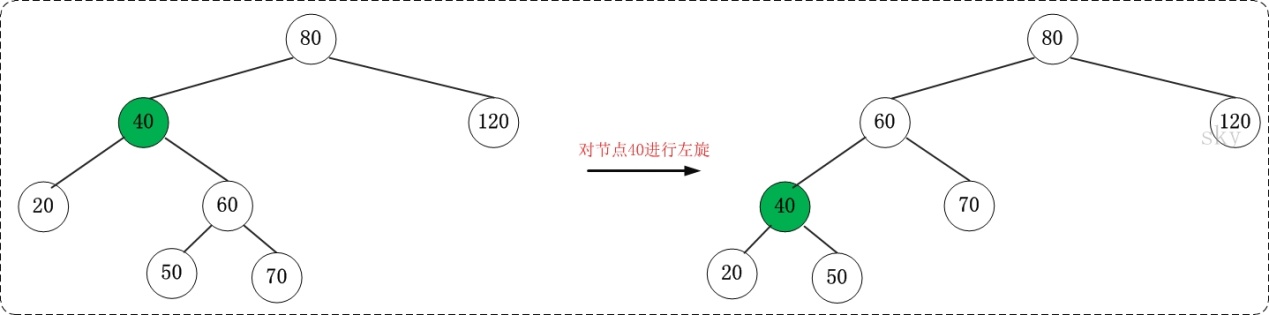
08 then left[p[x]] ← y // 情况2：如果 x是它父节点的左孩子，则将y设为“x的父节点的左孩子”

09 else right[p[x]] ← y // 情况3：(x是它父节点的右孩子) 将y设为“x的父节点的右孩子”

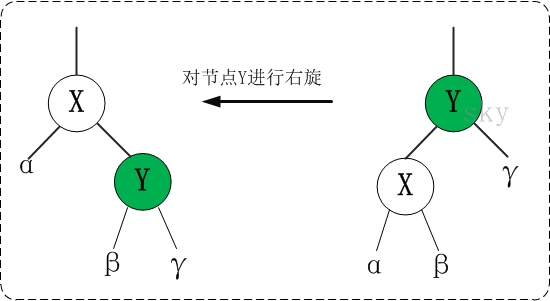
10 left[y] ← x // 将 “x” 设为 “y的左孩子”

11 p[x] ← y // 将 “x的父节点” 设为 “y”

理解左旋之后，看看下面一个更鲜明的例子。你可以先不看右边的结果，自己尝试一下。



**2. 右旋**



对x进行左旋，意味着"将x变成一个左节点"。

右旋的伪代码《算法导论》：参考上面的示意图和下面的伪代码，理解“红黑树T的节点y进行右旋”是如何进行的。

RIGHT-ROTATE(T, y)

01 x ← left[y] // 前提：这里假设y的左孩子为x。下面开始正式操作

02 left[y] ← right[x] // 将 “x的右孩子” 设为 “y的左孩子”，即 将β设为y的左孩子

03 p[right[x]] ← y // 将 “y” 设为 “x的右孩子的父亲”，即 将β的父亲设为y

04 p[x] ← p[y] // 将 “y的父亲” 设为 “x的父亲”

05 if p[y] = nil[T]

06 then root[T] ← x // 情况1：如果 “y的父亲” 是空节点，则将x设为根节点

07 else if y = right[p[y]]

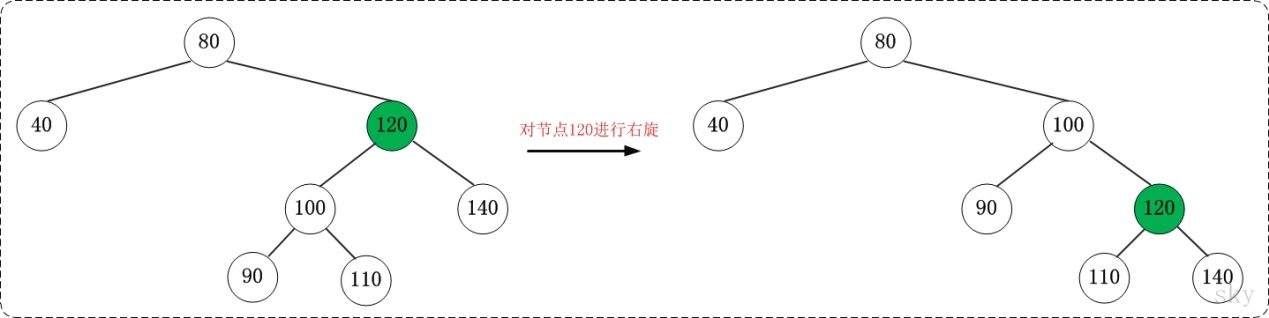
08 then right[p[y]] ← x // 情况2：如果 y是它父节点的右孩子，则将x设为“y的父节点的左孩子”

09 else left[p[y]] ← x // 情况3：(y是它父节点的左孩子) 将x设为“y的父节点的左孩子”

10 right[x] ← y // 将 “y” 设为 “x的右孩子”

11 p[y] ← x // 将 “y的父节点” 设为 “x”

理解右旋之后，看看下面一个更鲜明的例子。你可以先不看右边的结果，自己尝试一下。



**旋转总结**：

(01) 左旋 和 右旋 是相对的两个概念，原理类似。理解一个也就理解了另一个。

(02) 下面谈谈如何区分 左旋 和 右旋。  
在实际应用中，若没有彻底理解 左旋 和 右旋，可能会将它们混淆。下面谈谈我对如何区分 左旋 和 右旋 的理解。

## 四大域对象

PageContext -- page域（指的就是九大内置对象中的pageContext，不要被page这个单词迷惑了）

HttpRequest --request

HttpSession --session

ServletContext--application

## 九大内置对象

一篇不错的文章 <https://www.cnblogs.com/raozihao/p/7711582.html>

PageContext – pageContext， jsp上下文对象，提供了对jsp页面所有对象以及命名空间的访问。一个jsp一个pageContext，当重定向或者请求转发之后，pageContext都是当前jsp自己了，与上一个不同。pageContext 对象的作用是取得任何范围的参数，通过它可以获取 JSP页面的out、request、reponse、session、application 等另外八个对象。pageContext对象的创建和初始化都是由容器来完成的，在JSP页面中可以直接使用 pageContext对象。

HttpRequest -- request

HttpResponse -- response

HttpSession -- session

Throwable -- exception

JSPWritter -- out

ServletContext – application, 是所有servelt共享数据的地方

ServletConfig -- config

Object(HttpJspPage) – page，this 。 是这个jsp生成的servlet中的this，也就是servlet对象本身。

page对象是对该页面实例的实际引用。可以认为它是表示整个JSP页面的对象。

page对象是this对象的直接同义词。

## 九大对象与四大作用域

|  |  |  |  |
| --- | --- | --- | --- |
| 编号 | 对象 | 所属作用域 | 作用域描述 |
| 1 | **request** | **request** | 在当前**请求**中有效 |
| 2 | **response** | **page** | 在当前**页面**有效 |
| 3 | **out** | **page** | 在当前**页面**有效 |
| 4 | **session** | **session** | 在当前**会话**中有效 |
| 5 | **application** | **application** | 在所有**应用程序**中有效 |
| 6 | **config** | **page** | 在当前**页面**有效 |
| 7 | **pageContext** | **page** | 在当前**页面**有效 |
| 8 | **page** | **page** | 在当前**页面**有效 |
| 9 | **Exception** | **page** | 在当前**页面**有效 |

## 数据库

### 事务的四大基本要素

ACID：分别是 原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation）、持久性（Durability），也就是我们常说的事务ACID，这样才能保证事务（（Transaction）中数据的正确性。

### 事务使用不当会出现的情况

脏读：事务A读到了事务B未提交的数据。

不可重复度：事务A先后两次读取同一条记录，发现先后的值不一样，是由于update操作产生的。

幻读：属于记录数量级别的一种概念，是由于插入或者删除操作产生的。如：事务A第一次查询所有记录数为1，再次查询时居然变成2了。

### 事务隔离级别

#### 读未提交（Read Uncommitted）

读未提交，顾名思义，就是可以读到未提交的内容。

因此，在这种隔离级别下，查询是不会加锁的，也由于查询的不加锁，所以这种隔离级别的一致性是最差的，可能会产生“脏读”、“不可重复读”、“幻读”。

如无特殊情况，基本是不会使用这种隔离级别的。

#### 读提交（Read Committed）

读提交，顾名思义，就是只能读到已经提交了的内容。

这是各种系统中最常用的一种隔离级别，也是SQL Server和Oracle的默认隔离级别。这种隔离级别能够有效的避免脏读，但除非在查询中显示的加锁，如：

select \* from T where ID=2 lock in share mode;

select \* from T where ID=2 for update;

不然，普通的查询是不会加锁的。

那为什么“读提交”同“读未提交”一样，都没有查询加锁，但是却能够避免脏读呢？

这就要说道另一个机制“快照（snapshot）”，而这种既能保证一致性又不加锁的读也被称为“快照读（Snapshot Read）”

假设没有“快照读”，那么当一个更新的事务没有提交时，另一个对更新数据进行查询的事务会因为无法查询而被阻塞，这种情况下，并发能力就相当的差。

而“快照读”就可以完成高并发的查询，不过，“读提交”只能避免“脏读”，并不能避免“不可重复读”和“幻读”。

#### 可重复读(Repeated Read)

可重复读，顾名思义，就是专门针对“不可重复读”这种情况而制定的隔离级别，自然，它就可以有效的避免“不可重复读”。而它也是MySql的默认隔离级别。

在这个级别下，普通的查询同样是使用的“快照读”，但是，和“读提交”不同的是，当事务启动时，就不允许进行“修改操作（Update）”了，而“不可重复读”恰恰是因为两次读取之间进行了数据的修改，因此，“可重复读”能够有效的避免“不可重复读”，但却避免不了“幻读”，因为幻读是由于“插入或者删除操作（Insert or Delete）”而产生的。

#### 串行化（Serializable）

这是数据库最高的隔离级别，这种级别下，事务“串行化顺序执行”，也就是一个一个排队执行。

这种级别下，“脏读”、“不可重复读”、“幻读”都可以被避免，但是执行效率奇差，性能开销也最大，所以基本没人会用。

总结一下

**为什么会出现“脏读”？**因为没有“select”操作没有规矩。

**为什么会出现“不可重复读”？**因为“update”操作没有规矩。

**为什么会出现“幻读”？**因为“insert”和“delete”操作没有规矩。

**“读未提（Read Uncommitted）”能预防啥？**啥都预防不了。

**“读提交（Read Committed）”能预防啥？**使用“快照读（Snapshot Read）”，避免“脏读”，但是可能出现“不可重复读”和“幻读”。

**“可重复读（Repeated Red）”能预防啥？**使用“快照读（Snapshot Read）”，锁住被读取记录，避免出现“脏读”、“不可重复读”，但是可能出现“幻读”。

**“串行化（Serializable）”能预防啥？**排排坐，吃果果，有效避免“脏读”、“不可重复读”、“幻读”，不过效果谁用谁知道。

## 异常

### （1）非受检异常

非受检异常是指编译器不要求强制处置的异常。一般是指因设计或实现方式不当而导致的问题。也可以说，是程序员的原因导致的，是本来可以避免发生的情况。

java.lang.RuntimeException类及其子类都是非受检异常。具体如下：

● java.lang.ClassCastException：错误的类型转换异常。

● java.lang.ArrayIndexOutOfBoundsException：组下标越界异常。

● java.lang.NullPointException：空指针访问异常。

● java.lang.ArithmeticException：除零溢出异常。

如果事先检查数组元素下标，保证其不超出数组长度，ArrayIndexOutOfBoundsException异常就不会抛出；再如，先检查并确保一个引用类型变量值不为NULL，然后再访问其属性和方法，那么，NullPointException异常就不会抛出。因此，如果程序设计良好并且正确实现，这类异常就不会发生，所以通常也不会处理这类异常。

### （2）受检异常

受检异常是指编译器要求必须处置的异常，即程序在运行时由于外界因素造成的一般性异常，具体如下：

● java.lang.ClassNotFoundExeption：没有找到具有指定名称的类异常。

● java.lang.FileNotFoundException：访问不存在的文件异常。

● java.lang.IO Exception：操作文件时发生的异常。

● java.sql.SQL Exception：操作数据库时发生的异常。

Java编译器要求Java程序必须捕获或声明所有受检异常。如FileNotFoundException、IO Exception等。因为，对于这类异常来说，如果程序不进行处理，可能会带来意想不到的结果。而非受检异常可以不做处理，因为这类异常很普遍，若全部处理可能会对程序的可读性和运行效率产生影响。