

# IDEA

**.cast + tab + .var + tab**: 将Object类对象转化为具体类的对象

**ctrl + D**: 复制当前行

**ctrl + alt + m**: 把main方法中的代码抽出来成为一个独立的方法

**new ClassName() + alt + enter**快速创建一个类的对象

**shift + F6**: 变量名后按, 可以修改全部的变量名

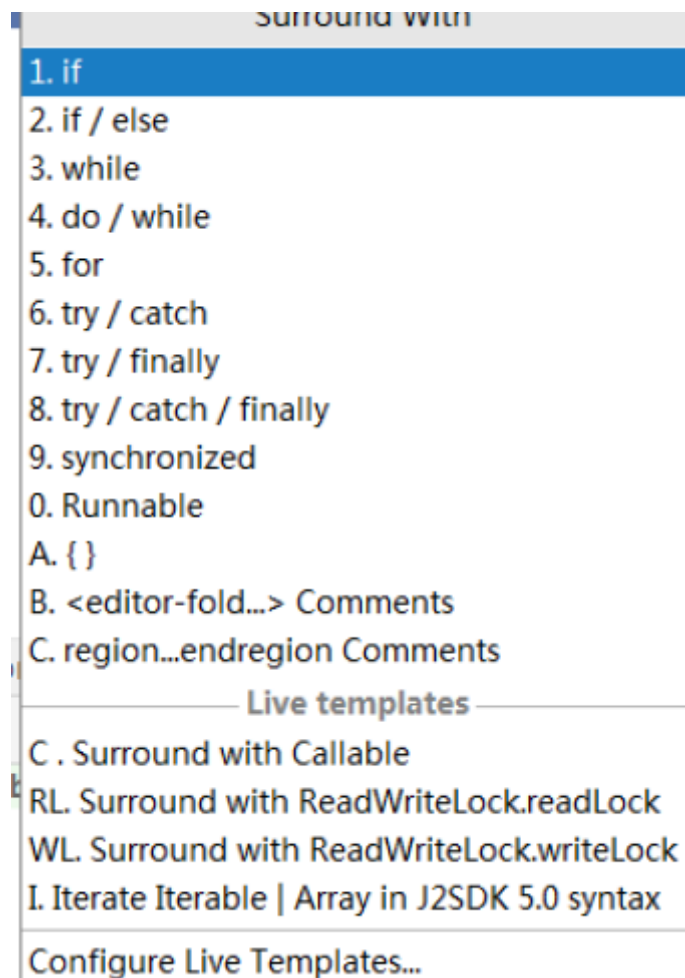
**alt+insert**: 定义各种方法快捷键

**alt + enter**:补全很多东西

如果要看源码: **ctrl + 鼠标左键点击**

如果要找目标方法: 目标类中 **ctrl + f12** 然后键盘搜索要找的方法名

**ctrl + alt + t**



## markdown

- 使用1. + 空格生成有序列表
- 使用> + 空格 来建立引用
- ctrl + k 建立超链接
- + 加上 空格 用来生成无序列表
- ctrl + t 表格
- ctrl + shift + k 代码块
- `` 代码

- ctrl + [ 在列表中升一级
- ctrl + ] 在列表中降一级
- ctrl + 数字 生成层级目录
- == == 高亮
- 4个\* 分隔符
- ctrl + b 加粗

## Week 6

---

### day 3

---

#### 错题集

```
public class Equals{
    public static void add3(Integer i){
        int val = i.intValue();
        val += 3;
        i = new Integer(val);
    }
    public static void main(String args[]){
        Integer i = new Integer(0);
        add3(i);
        System.out.println(i.intValue());
    }
}
```

/\*

方法调用的时候当前线程会从自己的栈中分配栈帧来存储方法中的局部变量  
所以main方法中的i和add3方法中的i是分别存储在两个不同的栈帧中的  
所以add3的i指向了新对象和main中的i没有关系

\*/

```
public class InTheLoop {
    public static final int END = Integer.MAX_VALUE;
    public static final int START = END - 100;
    public static void main(String[] args) {
        int count = 0;
        for (int i = START; i <= END; i++) {
            count++;
            System.out.println(count);
        }
    }
}
```

/\*

Java不会检查数据溢出，所以i <= 2<sup>31</sup>-1这个条件会一直满足  
会无限循环  
不抛出这些异常的一个原因是：如果执行这样的检查，代价会非常高昂

\*/

}

```

class Father {
    int i = 10;
    public Father() {
        System.out.println(getI());
    }
    public int getI() {
        return i; // this.i
    }
}
class Son extends Father {
    int i = 100;
    public Son(int i) {
        this.i = i;
    }
    public int getI() {
        return i; // this.i
    }
}
// in main method: Father father = new Son(1000) 结果是什么:0

```

/\*

如果是首次创建子类对象的话：  
 先加载父类，再加载子类，  
 再初始化父类对象，最后再初始化子类对象  
 初始化父类对象时最后调用父类的构造方法，而因为子类中重写了getI()  
 所以调用到是子类中的getI() 执行的是 return this.i 即子类对象中的成员变量  
 而此时子类对象中的成员变量还未被显式赋值，其值是JVM赋的默认初值

覆盖原因：  
 不管是在子类方法体还是在父类方法体中，调用一个成员方法时，  
 都是先在子类中查找，如果没找到才到父类中查找，  
 所以如果子类中定义了该成员方法，  
 父类的同名成员方法就不可能被调用

\*/

```

public class AnimalFarm{
    public static void main(String[] args){
        final String pig = "length: 10";
        final String dog = "length: " + pig.length();
        System.out. println("Animals are equal: " + pig == dog);
        // +运算符的优先级高于 == 运算符
        // 所以是判断: "Animals are equal: length: 10" == "length: 10"
    }
}

```

```

class Dog {
    public static void bark()
    {

```

```

        System.out.print("woof ");
    }
}
class Basenji extends Dog
{
    public static void bark()
    {
    }
}
public class Bark {
    public static void main(String args[]) {
        Dog woofer = new Dog();
        Dog nipper = new Basenji();
        woofer.bark(); // woof
        nipper.bark(); // woof
    }
}
/*
编译看左边
静态方法不能重写，所以子类对象上调用的还是父类中定义的静态方法
*/
}

```

```

public class Test{
    public static void main(String[]args){
        int[]x={0, 1, 2, 3};
        for{int i=0;i<3;i+=2){
            try{
                system.out.println(x[i+2]/x[i]+x[i+1]);
            }catch(ArithmeticException e){
                System.out.println("error1");
            }catch(Exception e){
                System.out.println("error2");
            }
        }
    }
}
/*
catch块的存在就是为了自己处理异常，不向上抛出
i = 0时出现除零异常x[0]，交由第一个catch块处理，
处理完后执行catch块后面的代码，没有代码，进入下一轮循环
i = 2时出现数组索引越界异常x[4]，交由第二个catch块处理，结束本轮循环后
i = 4不满足进入循环的条件，for循环结束
所以输出
error1
error2
*/
}
}

```

**写深度克隆的clone()方法时，该方法体通常都有一行代码，是什么？**

深复制的过程

1. super.clone() 即先调用Object类中的clone方法，先完成浅复制
2. 然后对于引用数据类型的成员变量，再让它们指向相应的复制对象

## sleep和wait方法的比较

1. 所属不同  
sleep定义在Thread类，静态方法  
wait定义在Object类中，非静态方法
2. 唤醒条件不同  
sleep方法是休眠时间到  
wait方法在其他线程中在同一个锁对象上，调用了notify（可能唤醒）或notifyAll方法
3. 使用条件不同：  
sleep 没有任何前提条件  
wait()要求当前线程必须持有锁对象，才能在锁对象上调用wait()
4. **导致线程进入阻塞状态的时候，线程对锁对象的持有情况不同**（最主要最核心）  
线程因为sleep()方法而进入阻塞状态的时候，不会放弃对锁对象的持有  
但是线程因为wait()方法而进入阻塞状态的时候，会放弃对锁对象的持有

- 
1. 课程内容
    - a. 集合
    - b. 基本数据结构
    - c. 数据库
  2. 内容特点  
内容不多, 比较难
  3. 讲述风格  
节奏相对会快一点
  4. 如何学习
    - a. 大家来王道是来学习知识?  
技能
    - b. 学习是一件快乐的事情吗?
- c. 学习技巧
    - a. 学会自己整理笔记
    - b. 一定要多动手敲代码
  - c. 如何解决问题
    1. 定位问题
    2. 尝试自己解决
    3. google, stackoverflow
    4. 请教同学和老师

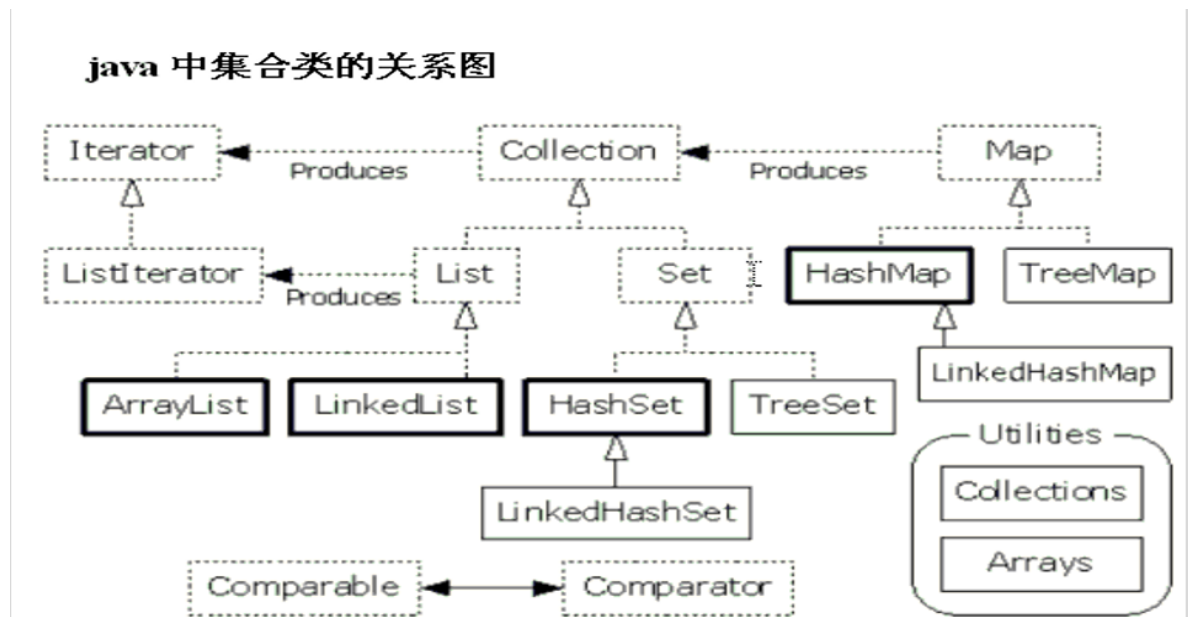
答疑时间：

晚上：8:00 ~ 9:30

一天4节半课，最后一节课半个小时

# 集合

## Collection关系图



## 三个引入的小问题

### 1. 为什么需要集合类

需要对一组对象进行操作，但提前不知道有多少对象（集合容量可变且有丰富的API）

### 2. 集合类的特点

1. 只能存储引用数据类型（对象的地址）
2. 容量可变（可自动扩容：创建新空间，把引用复制过去）

### 3. 数组和集合类都是容器，它们有什么区别

1. 数组可以存储**基本数据类型**的数据，集合不可以
2. 数组容量固定，集合容量可变
3. 数组效率高，集合效率较低
4. 数组没有API，集合有丰富的API

## Collection接口概念

Collection层次结构的根接口，表示一组对象（Collection的元素），一些Collection允许有重复的元素，另一些不允许，一些是有序的，另一些是无序的

## API

### 增：

`boolean add(E e)`

`boolean addAll(Collection c)`

返回false说明：没有对原集合产生影响（如添加了重复的元素，而该集合类型不允许添加重复的元素(如set)，这里的重复是指**对象的内容相同**，而非地址相同）（添加了不允许的元素：应该抛出异常）

**删:**

`void clear()` 清除集合中的所有元素

`boolean remove(Object o)` 删除与目标对象相等的元素

`boolean removeAll(Collection c)` 删除所有与目标集合中的元素相等的元素

`boolean retainAll(Collection c)` 仅保留与目标集合中的元素相等的元素

**查:**

`boolean contains(Object o)` 若含有与目标对象相等的元素，返回true

`boolean containsAll(Collection c)` 若目标集合中的所有元素，都能在本集合中找到相等的元素，返回true

**获取集合属性：**

`boolean isEmpty()`

`int size()`

### 集合类的特点

1. 只能存储引用数据类型
2. 容量可变

---

## day 4

---

### Iterator接口

遍历集合的两种方式

**1.Object[] toArray()** 返回一个Object数组

对Object数组的修改不会影响原集合(把集合中引用指向的对象也复制了一份，即不是视图技术)

**2.Iterator iterator()**：利用迭代器

**Iterable接口:**

可迭代的，某个集合实现了整个接口就表明整个集合是可迭代的：即可**返回一个迭代器**（实现了一个接口：具有了某种功能）

---

**Iterator iterator()**：迭代器，统一遍历的操作（不要把它和它的子接口ListIterator搞混）

**迭代器中的API**

1. `boolean hasNext()` 是否仍有元素可以迭代

2. E next() 越过并返回迭代的下一个元素
3. void remove() 从集合中移除刚刚返回的元素，必须在next() 后使用，且不能连续删除（否则会报异常）  
删除逻辑：找到元素后，判断是否符合条件，再删除

## 并发修改异常

### 注意事项

1. 如果集合的API修改了集合的**结构**（remove/ add），那么所有的迭代器都会失效（光标位置失效了）
2. 如果用某个迭代器修改了集合的**结构**，那么所有**其他的**迭代器都会失效
3. 用迭代器遍历的时候，不要使用while，可以使用for，最好使用foreach（跟局部变量的作用域（或者说生命周期）有关，如果多个迭代器是在一个语句块中的话，只要其中一个迭代器对集合的结构进行了修改，那么其他的迭代器都会失效）

---

## iterator接口设计原理：迭代器设计模型

1. 为什么Iterator要定义成一个接口而不是一个类呢

集合都应该提供遍历的操作（Collection implements Iterable），但是不同的集合底层的数据结构不一样，遍历的方式也不一样，所以需要用到**抽象方法**来提供一个统一的遍历标准，具体实现交给具体子类去做

（所以不能定义成一个普通类(有抽象方法的类一定是抽象类)）

2. Java中什么表示“标准”呢

抽象方法

3. 为什么不能定义成抽象类，而定义成接口呢（继承:is-a 接口: like-a 具有某种功能）

因为集合不是迭代器，但集合都应该具有迭代的功能

4. Iterator it = c.iterator() 获取的是哪个子类的对象：接口的实现子类的对象（即集合中的成员内部类对象）

5. 迭代器必须知道集合底层的数据结构，但集合的数据结构是私有的，**一个类怎么访问另一个类的私有成员呢：内部类**

6. 迭代器应该设计成内部类的形式，内部类有几种

1. 静态内部类
2. 成员内部类
3. 局部内部类
4. 匿名内部类

7. 迭代器应该设计成哪一种形式的内部类：依赖于外部类对象的（静态内部类（静态上下文）只能访问静态变量，而集合中的私有数据是普通成员变量，所以）

成员内部类

---



# List接口

**概述：**列表：数据结构中的线性表 最大特征就是可以**使用索引进行增删改查**

元素之间是有位序关系的集合，此接口的用户可以对象列表中的每个元素插入位置进行精确地控制，用户可以根据元素的整数**索引**访问元素，并搜索列表中的元素

## API

### 增

void add(int index, E element) 将元素插入指定位置

boolean addAll(int index, Collection c) 将整个集合中的元素插入指定位置

### 删

E remove(int index) 移除列表中指定位置的元素

### 改

E set(int index, E element) 用指定元素替换掉指定位置上的元素，并返回被替换的元素

### 查

E get(int index)

int indexOf(Object o) 返回列表中第一次出现的与指定元素**相等**的元素索引（相等 means equals返回true）

int lastIndexOf(Object o) 返回列表中最后一次出现的与指定元素**相等**的元素索引

## 截取子串

List subList(int fromIdx, int toIdx) []

## 视图技术，和list共用同一份数据

即子序列中的引用指向的对象就是原序列中的引用指向的对象，所以通过子序列中的引用对其指向的对象进行操作，也会反映到原序列中

（视图是数据库中的术语：使用场景：只希望子公司能看到母公司数据的一部分，但是修改能反映到母公司这里）

String类：substring不是视图技术：因为字符串是不可变对象（subString操作会产生一个新的对象）

---

## List的遍历

ListIterator listIterator()

ListIterator listIterator(int index)

## list的迭代器（才有光标这一说法）

### List：

ListIterator listIterator()：光标后面的元素索引为0

ListIterator listIterator(int index): 光标后面的元素索引为index

**listIterator**: 继承自Iterator

**概述:**

列表迭代器, 允许程序员按任一方向遍历列表, 并可以在迭代期间修改列表, 获得当前迭代器在列表中的位置 (元素之间)

**API**

**遍历:**

从前往后

boolean hasNext()

E next() 越过并返回一个元素 (移动光标)

从后往前

boolean hasPrevious()

E previous()

**获取当前光标位置**

int nextIndex() 光标移动方向后一个元素位置

int previousIndex() 光标移动方向前面一个元素位置

**修改:**

void add(E e) 插入到光标的位置 (**就是光标的后面位置**) (修改集合**结构**)

void remove() 去除最近返回的元素 (修改集合**结构**)

void set(E e) 替换最近返回的元素 (没有修改集合**结构**)

只要修改集合的**结构** remove方法就会失效: 因为光标失效了

---

## 数组

**概念** (数组就是对象, 也是Object类的子类)

**1.什么是数组, 主要特点是什么**

**固定大小的连续内存空间**, 且这片连续的内存空间又被分割成**等长**的小空间, 它的主要特点是**随机访问**

长度固定, 且只能存储同一种数据类型的元素 (type\_length要一样才能随机访问)

**时间复杂度O(1)**: 时间复杂度描述的是一种资源耗费的增长趋势

操作所需时间不会随着数据规模的增长而增长 ( $y = c$ )

**时间复杂度O(n)**

操作所需时间会随着数据规模的增长而线性增长

注：Java中只有一维数据的内存空间是连续的，多维数组的内存空间不一定连续（多维数组实际上是对象数组）

### 数组如何实现随机访问：

寻址公式： $i\_address = base\_address + i * type\_length$

### 2.为什么数组的索引一般都是从0开始的呢

假设索引从1开始，我们有两种处理方式

1. 修改寻址公式： $i\_address = base\_address + (i - 1) * type\_length$ （多了一步计算）
2. 不变寻址公式，但浪费开头的一个内存空间（初期内存空间很宝贵）

### 3.为什么数组的效率比链表要高

局部性原理

CPU会将一连串相邻的内存空间的数据预先读入高速缓存中（空间局部性），而数组充分利用了这一特性（所以在处理数组时，很多时候CPU是直接从高速缓存读数据而不是从内存中读数据）

**解决CPU、内存、IO设备的传输数据速率差异的方法：**尽可能避免高速设备直接和低速设备交换数据

#### CPU和内存

1. 高速缓存（预读）
2. 编译器的指令重排序（把相关的代码重新排在一起）

有的东西直接放在寄存器里面，这样就不用从内存中拿数据

```

public void method() {
    int a = 1;
    int b = 1;
    int c = 1;
    int d = 1;

    // 对a进行操作
    // 对b进行操作
    // 对c进行操作
    // 对d进行操作
}

```

```

public void method() {
    int a = 1;
    // 对a进行操作
    int b = 1;
    // 对b进行操作
    int c = 1;
    // 对c进行操作
    int d = 1;
    // 对d进行操作
}

```

编译器的指令重排序

- a. 加快机器指令执行的效率
- b. 保证在单线程的环境下，执行结果和顺序执行结果一致。

## 内存和IO

缓存：将磁盘中的数据缓存在内存中

## CPU和IO

1. 中断技术
2. 通道

# day 5

## 逻辑结构

## 数组的基本操作

**增:**(结论前提：保证操作后元素是连续的，且元素之间的相对顺序没有改变)

最好情况：在末尾插入，不需要移动元素  $O(1)$

最坏情况：在头部插入，需要移动 $n$ 个元素  $O(n)$

平均情况：平均需要移动  $(0 + 1 + \dots + n) / n + 1 = n / 2$   $O(n)$

平均：假设插入各个位置的概率一样

**删:** (结论前提：保证操作后元素是连续的，且元素之间的相对顺序没有改变)

最好情况：在末尾删除，不需要移动元素

最坏情况：在头部删除，需要移动 $n - 1$ 个元素

平均情况：平均需要移动  $(0 + 1 + \dots + n - 1) / n = (n - 1) / 2 \quad O(n)$

扩展：我们可以先将元素标记成删除（逻辑删除），等到某个时间点再一次性删除（物理删除），这样做就可以大大减少移动元素的时间

如：GC中的标记整理算法

查：

1. 按索引查找值  $O(1)$

2. 查找与指定元素相等的元素的索引

大小无序： $O(n)$

大小有序：二分查找  $O(\log n)$

$O(\log n)$ 时间复杂度效率非常高

$O(\log n)$ 的时间复杂度效率非常高。

$n=10$

$O(n): 10$

$O(\log n): 4$

$n=43$ 亿

$O(n): 43$ 亿

$O(\log n): 33$

是

问题： $O(\log n)$ 和 $O(1)$ 的算法哪一种效率高？

不一定， $O(1)$ 只是表示算法所需的时间不会随着数据规模的增长而增长，可能是每次操作的时间都是固定的，但是要做5000次，而 $O(\log n)$ 在数据很多的情况下也只是做几十次

时间复杂度只是表示一种增长趋势（加速度大小不能决定速度大小）

总结：数组增删慢（要移动元素），查找快（随机访问）

---

## 链表

概念：一串链子将结点串联起来

结点：数据域 + 指针域

数据域：数据

指针域：下一个结点的地址

## 分类

1. 单链表
2. 循环链表
3. **双向链表** (工程中常用)
4. 循环双链表

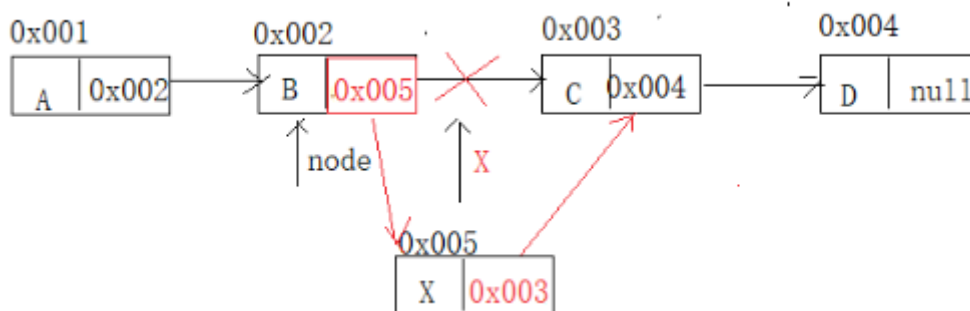
注：循环链表较少使用，只有当处理的数据具有环形结构时，才使用：约瑟夫问题

## 单链表

单链表的基本操作

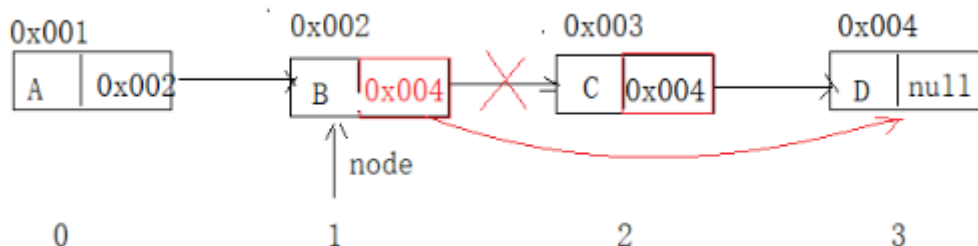
增 (在某个结点后添加)  $O(1)$

```
// 先连后断, 只修改了两个结点的指针域
nodeToAdd.next = node.next;
node.next = nodeToAdd;
```



删 (在某个结点后删除)

```
// 只修改了一个结点的指针域
node.next = node.next.next;
```



## 查

1. 根据索引查找元素  $O(n)$
2. 查找链表中与指定值相等的元素
  1. 元素大小有序  $O(n)$
  2. 元素大小无序  $O(n)$

二分查找的两个前提:

1. 大小有序
2. 可以随机访问（查找到中间元素的时间的是 $O(1)$ ）

总结：链表增删快（修改指针域即可），查找慢（只能顺序访问）

**双向链表**：多了一个指向前驱结点的指针域（必要时可以加空的头结点和尾结点以统一操作）

### 基本操作

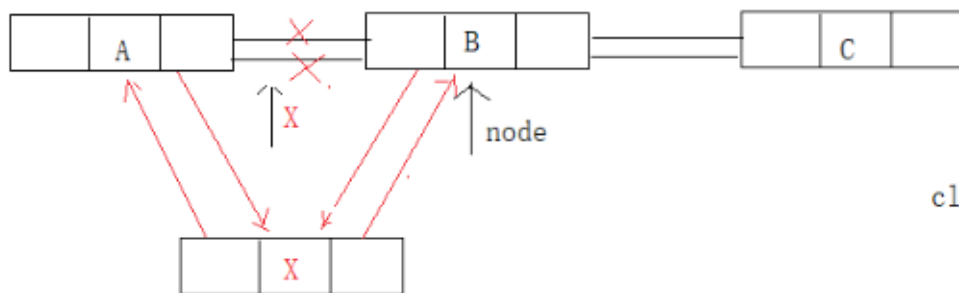
**增**：在某个结点**前面**添加

单链表： $O(n)$ ，需要先从头结点开始遍历，找到该结点的前驱结点

双向链表： $O(1)$

// 先连后断

```
nodeToAdd.next = node;  
nodeToAdd.prev = node.prev;  
node.prev.next = nodeToAdd;  
node.prev = nodeToAdd;
```



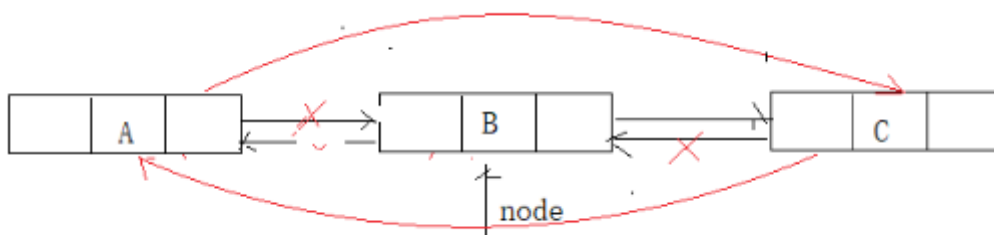
```
class Node {  
    Node prev;  
    char value;  
    Node next;  
}
```

**删**：删除**该**结点

单链表： $O(n)$ ，也是需要先从头结点开始遍历，找到该结点的前驱结点

双向链表： $O(1)$

```
node.prev.next = node.next;  
node.next.prev = node.prev;
```



**查**

### 1. 查找前驱结点

单链表:  $O(n)$  双向链表  $O(1)$

### 2. 根据索引查找值

单链表:  $O(n)$ , 平均查找  $n/2$  个

双向链表:  $O(n)$ , 平均查找  $n/4$  个

### 3. 查找与指定值相等元素的索引

大小无序: 都  $O(n)$

大小有序:

单链表:  $O(n)$ , 平均查找  $n/2$  个

双向链表:  $O(n)$ : 可优化: 保留上一次查找元素, 如果下一次要查找的元素比保留元素大, 就往后找, 如果比保留元素小, 就往前找, 以减少查找元素的个数

总结: 虽然双向链表更占内存空间 (每个结点都多了个指针域), 但是它在某些操作上的性能是优于单链表的

思想: 空间换时间 (常见思想)

---

## 缓存: 空间换时间

内存大小有限, 所以缓存不能无限大, 当缓存满时, 再向缓存中添加数据: 清理出一些内存空间

## 缓存淘汰策略

1. FIFO: first in first out 用队列实现, 不合理: 因为最先加载入内存的很可能是最重要的
2. LFU: least frequently used 用最小堆实现, 不合理: 因为最新加入的数据可能是将要使用的, 但它的使用次数并不多
3. LRU: least recently used 最近最少使用 用链表实现 (实际上是哈希表 + 链表)

## LRU算法

添加 (认为尾结点 (最后一个元素结点) 是最近最少使用的数据)

1. 如果缓存中已经存在该数据: 删除该结点, 并添加到头结点 (因为要保证最近访问的结点在最前面)
2. 如果缓存中不存在该数据
  1. 缓存没满: 添加到头结点: 头插法 (可以利用结点的构造方法, 直接先连) (`new Node(element, head, head.next)`)
  2. 缓存满了: 删除尾结点, 在头结点添加新数据(`end.prev = end.prev.prev; end.prev.next = end;`)

## 题目

### 1. 求链表的中间元素

```
// 快慢指针 time O(n) memory O(1)
public static Node middleElement(Node head) {
```



```

// 处理特殊情况
if (head == null) {
    return null;
}
Node slow = head;
Node fast = head;
// 一定要fast.next在前面, 不然会报空指针异常
// 循环直到fast的后面一个结点为null或者fast的后面的后面一个结点为null
while (fast.next != null && fast.next.next != null) {
    slow = slow.next;
    fast = fast.next.next;
}
// fast到达链表尾部时, slow刚好到达中间元素的位置
return slow;
}

```

## 2.判断链表中是否有环

```

// 迷雾森林: 作标记, 如果再次遇到标记, 就说明又绕回来了 time O(n) memory O(n)
public static boolean hasCircle(Node head) {
    Collection visited = new HashSet();
    Node node = head;
    while (node != null) {
        // 若遇到了相同的元素, 表明绕回来了, 有环
        if (visited.contains(node)) {
            return true;
        }
        visited.add(node);
        node = node.next;
    }
    return false;
}

/*
时间复杂度:
无环: O(n)
有环: 假设环外有a个结点, 环内有r个结点。
    最好情况: O(a)
    最坏情况: O(a+r)
    平均情况: O(a+r/2)
空间复杂度: O(1) 来一百万个结点, 同样也是用两个指针
*/

// 快慢指针 (跑道) time O(n) memory O(1)
public static boolean hasCircle(Node head) {
    Node slow = head;
    Node fast = head;
    // 判断语句中的位置不能调换
    while (fast != null && fast.next != null) {
        slow = slow.next;
        fast = fast.next.next;
        if (slow == fast) {
            return true;
        }
    }
    return false;
}

```

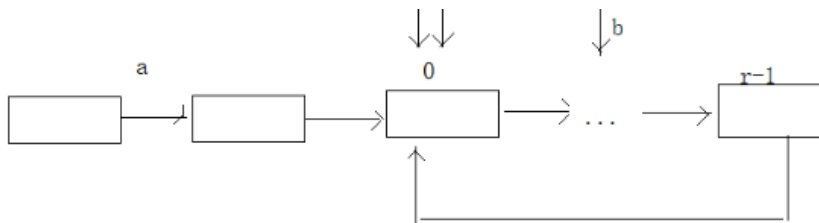
```

}

// 找到链表中环的入口 time O(n) memory O(1)
/*
时间复杂度：
    无环：O(n)
    有环：
        最好情况：O(2a) slow第二次刚走到环入口就相遇了
        最坏情况：O(2a+r) slow在环内绕了一圈才相遇
        平均情况：O(2a+r/2) slow在环内绕了半圈相遇
空间复杂度：O(1)
*/
public static Node findNode(Node head) {
    Node slow = head;
    Node fast = head;
    while (fast != null && fast.next != null) {
        slow = slow.next;
        fast = fast.next.next;
        if (slow == fast) {
            break;
        }
    }
    slow = head;
    while (slow != fast) {
        slow = slow.next;
        fast = fast.next;
    }
    return slow;
}

```

$a = mr - b$ 的意思是从相遇点再走 $mr - b$ 步 就等于从链表头走  $a$ 步， 两者就能相遇，且相遇的地方为环的入口



假设环外有 $a$ 个结点，环内有 $r$ 个结点。假设快指针转了 $m$ 圈之后，与慢指针在环内 $b$ 的位置相遇。

$$2 * (a + b) = a + mr + b$$

$$a = mr - b$$

结论：快慢指针相遇之后，将快指针移动到头结点，然后快慢指针每次只走一步，当它们再次相遇的时候，相遇的结点就是入环的第一个结点。

*Proof of second step:*

Distance traveled by tortoise while meeting =  $x + y$

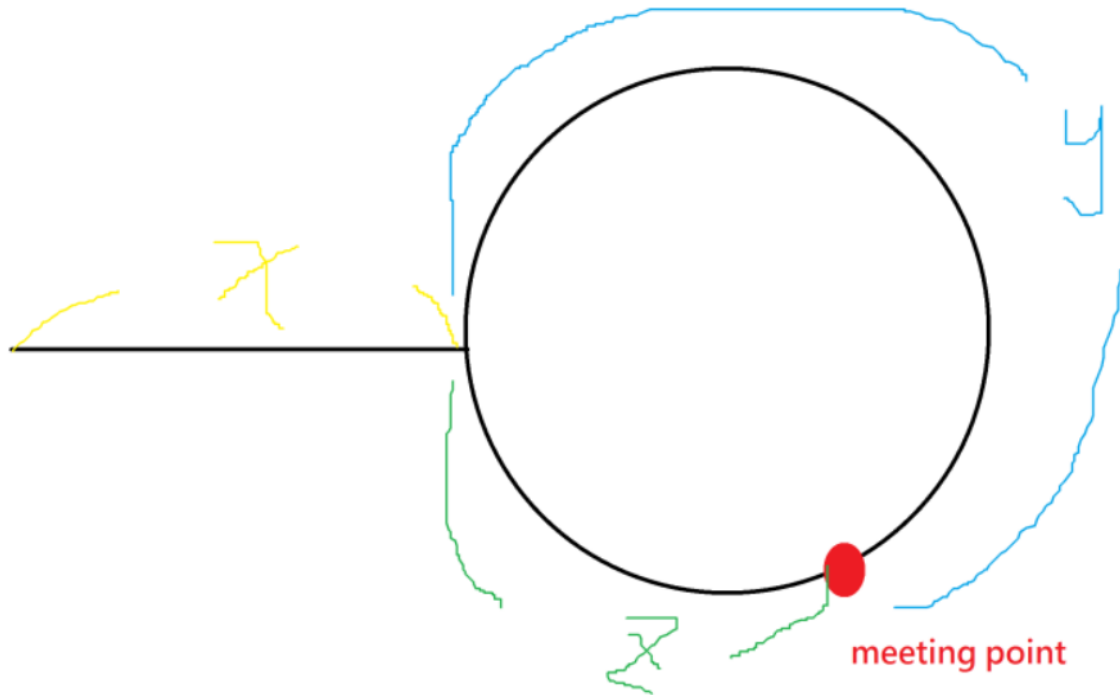
Distance traveled by hare while meeting =  $(x + y + z) + y = x + 2y + z$

Since hare travels with double the speed of tortoise,

so  $2(x+y) = x+2y+z \Rightarrow x+2y+z = 2x+2y \Rightarrow x=z$

Hence by moving tortoise to start of linked list, and making both animals to move one node at a time, they both have same distance to cover .

They will reach at the point where the loop starts in the linked list



### 3.反转单链表

```
// 原地使用头插法
// iteratively
public static Node reverse(Node head) {
    Node prev = null;
    Node curr = head;
    while (curr != null) {
        Node next = curr.next;
        curr.next = prev;
        prev = curr;
        curr = next;
    }
    return prev;
}

// recursively 伪递归
public static Node reverse(Node head) {
    return helper(head, null);
}

private static Node helper(Node curr, Node prev) {
    // 递归出口: curr == null说明最后一个结点已经逆转, 返回新的头结点即可
    if (curr == null) {
        return prev;
    }
    Node next = curr.next;
    curr.next = prev;
}
```

```

        // 用参数传递完成赋值操作
        return helper(next, curr);
    }

    public static Node reverse(Node head) {
        // 递归边界
        if (head == null || head.next == null) {
            return head;
        }
        Node newHead = reverse(head.next); // 就先考虑两个结点的情况就好
        // 递归式
        head.next.next = head;
        head.next = null;

        return newHead;
    }
}

```

## 递归：

### 1. 什么情况下可以考虑用递归

递（原问题可以分解成子问题）+ 归（把子问题的解归结为原问题的解）

原问题可以分解成若干个子问题（递归式）

子问题和原问题的求解方式一样，只是数据的规模不同

最终可以分解成基本问题，基本问题可以直接求解（递归边界）

最后子问题的解可以合并成原问题的解

### 2. 递归的基本要素

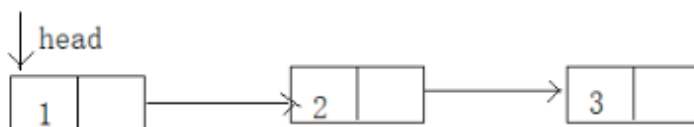
递归式

递归边界

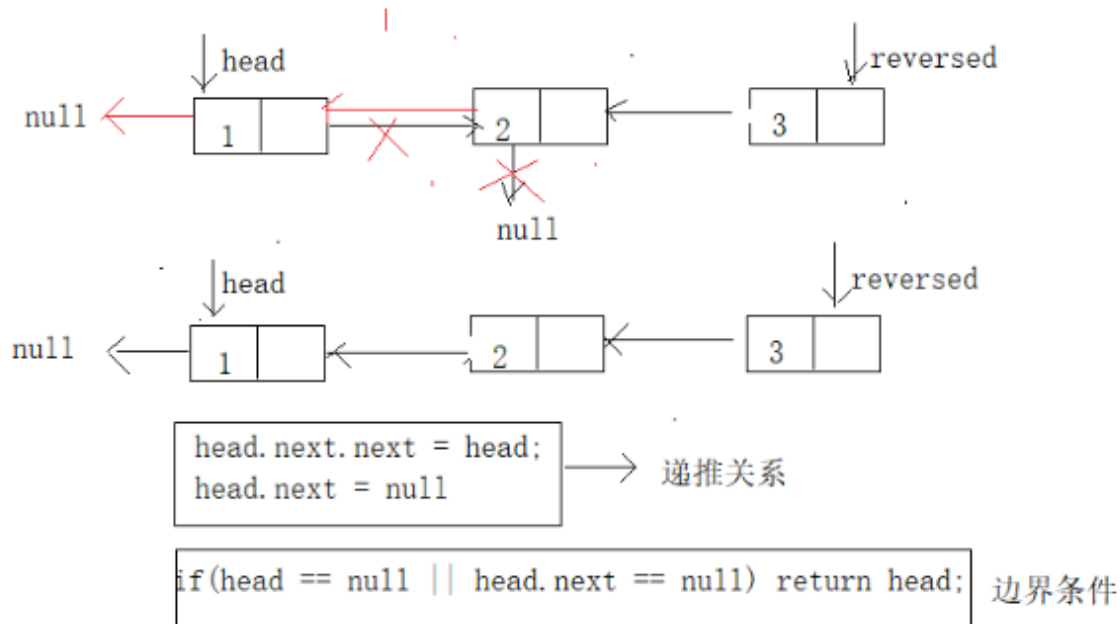
### 3. 递归的注意事项

警惕重复计算

## 反转单链表



1. 假设head.next已经反转了, 如何反转head结点?



## 数组和链表的比较

时间复杂度	数组	链表
插入/删除	$O(n)$	$O(1)$
随机访问 (按索引访问)	$O(1)$	$O(n)$

- 内存模型**: 数组使用的是连续的内存空间, 可以利用CPU的高速缓存预读数据, 而链表的内存空间不是连续的, 不能有效预读数据。但是如果所需创建的数组过大, 而系统没有足够的连续内存空间, 会抛出OOM
- 灵活性**: 数组的缺点是大小固定, 没法动态调整大小。要存储一些对象时, 如果数组太大, 则浪费内存空间, 如果数组太小, 又会需要重新申请一个更大的数组, 并将数据拷贝过去 (目前见到的只有subList是视图技术)
- 性能**: 如果业务对内存的使用非常苛刻, 则数组更适合, 因为链表结点有指针域, 更消耗内存。而且对链表的频繁插入和删除, 会导致结点对象的频繁创建和销毁, 有可能会导导致频繁的GC活动

## day 6

# List的子类

arratlist + vector + linkedList

## ArrayList

### 概述

1. 底层数据结构是数组，查询快，增删慢
2. 不同步，线程不安全，效率高
3. 允许null元素
4. 提供操作底层数组大小的方法

---

### 构造方法

ArrayList() 默认容量为10

ArrayList(int initialCapacity) 指定初始容量大小 (因为频繁自动扩容会影响性能)

ArrayList(Collection c) 把C中的对象也复制了一份 不是视图技术

---

### API

void ensureCapacity(int minCapacity)

手动扩容，避免频繁扩容

void trimToSize()

将数组缩小到size大小：扩容会自动扩到原来的1.5倍，10W元素 + 1，容量变为15W，浪费了很多内存空间

注意事项：当确定不再添加元素的时候，我们可以使用这个方法

**实现Cloneable接口的类才能重写clone**

---

## Vector

### 概述

1. 底层数据结构是数组，查询快，增删慢
2. 同步，线程安全，效率低
3. 允许null元素
4. JDK1.0提供

---

### API

Vector类	List接口
void addElement(E obj)	void add(E e)
E elementAt(int index)	E get(int index)
void insertElementAt(E obj, int index)	void add(int index, E e)
void removeAllElements()	void clear()
boolean removeElement(Object obj)	boolean remove(Object obj)
void removeElementAt(int index)	E remove(int index)
void setElementAt(E obj, int index)	E set(int index, E e)
Enumeration elements()	Iterator iterator()

E firstElement()

E lastElement()

int indexOf(Object o, int index)

**int lastIndexOf(Object o, int index)** 从index~~往前面找~~的第一个与o相等的对象的索引

int capacity()

**void setSize(int newSize)**

**void copyInto(Object[] anArray)**

---

**Stack**: Vector的子类

Vector

| -- Stack

Deque接口中提供了更完整更一致的~~栈~~的操作，应该优先使用**Deque**，而非使用Stack（Deque接口引用指向LinkedList对象）

原因：

1. Stack继承了Vector类，拥有Vector的所有方法，可以对中间或者是栈底元素进行操作，使用起来不安全
2. 同步，效率低

---

**LinkedList**

**概述**

1. 底层数据结构是**链表**，查询慢，增删快
2. 不同步，线程不安全，效率高

- 3. 允许null元素（线性表其实都可以存null）
- 4. **实现了Deque接口**，可以当成栈、队列、双端队列

---

### 构造方法

LinkedList()

LinkedList(Collection c) 按照c的迭代顺序添加

---

### API

```
for (int i = list.size() - 1; i >= 0; i--) {  
    list.get(i);  
}  
// list底层是数组时：O(n)；底层是链表时O(n^2),因为没有随机访问特性
```

Iterator descendingIterator() 逆向遍历链表（双向链表应该是维护了一个尾结点指针）

boolean removeFirstOccurence(Object o)

boolean removeLastOccurence(Object o)

查找元素和删除元素在一次遍历中就完成了（链表本来就可以）

用list的方法做：int index = lastIndexOf(o); list.remove(index); 查找元素的索引，和根据索引删除元素在两次遍历中完成

---

## Deque接口

概述：只能在线性表的**两端**操作(不能直接操作中间的元素)（添加，删除，查找），可以被用作栈、队列、双端队列

都是接口

Collection

|-- Queue

|-- Deque : double ended queue

栈：

入栈	出栈	查看栈顶元素
void push(E e)	E pop()	E peek()

队列：用Queue接口 + LinkedList就行 方法用 offer poll peek



队尾添加	队头删除	看队头元素
void add(E e)	E remove()	E element()
void offer(E e)	E poll()	E peek()

就用抛出异常的：add + remove + element

或者：offer poll peek

**双端队列：**

添加（两端）	删除（两端）	查看（两端）
void addFirst(E e)	E removeFirst()	E getFirst()
void addLast(E e)	E removeLast()	E getLast()
void offerFirst(E e)	E pollFirst()	E peekFirst()
void offerLast(E e)	E pollLast()	E peekLast()

**操作失败时：**

add remove get是抛出异常

offer poll peek特殊值：null/false

---

## 栈和队列

### 栈

**概念：**操作受限的线性表，表现只能在一端（栈顶）插入和删除

#### API

压栈：void push(E e)

出栈：E pop()

查看栈顶元素：E peek()

判空：boolean isEmpty()

---

### 队列

**概念：**操作受限的线性表，只能在一端（队尾）添加元素，在另一端（队头）删除元素

#### API

入队：void enqueue(E e)

出队: E dequeue()

查看队头元素: E peek()

判空: boolean isEmpty()

## Deque的使用

当做栈

```
Deque stack = new LinkedList();
stack.push("a");
stack.push("b");
stack.push("c");
System.out.println(stack.peek()); // c
while (!stack.isEmpty()) {
    String s = (String) stack.pop();
    System.out.println(s); // c b a
}
```

当做队列

```
Deque queue = new LinkedList();
queue.offer("a");
queue.offer("b");
queue.offer("c");
System.out.println(queue.peek()) // a
while (!queue.isEmpty()) {
    String s = (String) queue.poll();
    System.out.println(s); // a b c
}
```

题目: arraylist实现栈 (写)

```
public class MyStack {
    private ArrayList list;

    public MyStack() {
        list = new ArrayList();
    }

    public MyStack(int initialCapacity) {
        list = new ArrayList(initialCapacity);
    }

    public void push(Object o) {
        list.add(o);
    }

    public Object pop() {
        if (isEmpty()) {
            return new EmptyStackException();
        }
        return list.remove(list.size() - 1); // o(1)
    }
}
```

```
public Object peek() {
    if (isEmpty()) {
        return new EmptyStackException();
    }
    return list.get(list.size() - 1);
}

public boolean isEmpty() {
    return list.isEmpty();
}
}
```

---

## 组合

组合：

概念

1. 持有一个类的对象，就能够拥有这个类的功能（public method），而且可以选择拥有哪些功能（与继承不同）
2. 可以对这些功能进行加强（加控制逻辑）
3. 可以持有多个类的对象

问题：

1. 有哪些技术可以加强方法：继承（重写），组合
2. 继承vs组合哪个好

设计原则：组合优于继承

有的新的编程语言甚至没有了继承特性：Go语言

3. 什么时候使用继承：类与类之间是is-a关系

## Week 7

---

### day 1

---

## 泛型

学习要求

1. 能够用泛型操作集合
2. 能够看懂简单的泛型代码

泛型的好处

1. 提供了程序的安全性

2. 将运行期遇到的问题转移到了**编译期**
3. 省去了（自己）类型强转的麻烦

### 设计原则：及早失败原则

1. 节省计算机资源
2. 可以更好地排查问题

---

### 泛型应用：泛型类、泛型方法、泛型接口

#### 泛型类

泛型定义在类上(类比于形式参数)，作用域：这个类

格式：public class 类名<泛型类型1, ...>

注意：参数化类型必须是**引用类型**（因为擦除后就是一个Object）

```
public class Tool<T> {  
    private T obj; // 用于成员变量  
  
    public T getObj() { // 用于方法返回值类型  
        return obj;  
    }  
  
    public void setObj(T obj) { // 用于方法参数  
        this.obj = obj;  
    }  
}
```

#### 泛型的命名规则：满足标识符规则即可

业界规范：大写字母

T: type

E: element

K: key

V: value

#### 注意事项：

1. 参数化类型必须是引用数据类型：`List<String> list = new ArrayList<String>();`

右边的String就是参数化类型，类比于实参（因为泛型的具体类型是在创建对象时确定的，所以类中的static成员不能使用泛型，只能用Object）

2. JDK7提供了菱形操作符<>，可以利用类型推断机制（即右边可以不写String）
-

## 泛型方法

泛型定义（就是那个）在方法上，作用域：方法签名（这里指返回类型和参数列表）和方法内

**格式：**public <泛型类型> 返回类型 方法名（参数列表...）

```
public class Tool {  
    public <T> T echo(T obj) {  
        return obj;  
    }  
}
```

## 问题

1. 为什么在返回值类型前面定义泛型：因为**返回值类型**也可以使用泛型，泛型必须先定义后使用
2. 有泛型方法 的类一定是泛型类吗：不一定（类比：有抽象方法的类一定是抽象类，因为无法实例化）

## Collection中的泛型方法

Object[] toArray()

T[] toArray(T[] a) 泛型方法

如果实参中的数组能容纳集合中的元素，那么就会把元素赋值到该数组中，

否则就会创建一个和集合大小一样的数组，再讲元素赋值到新数组中（并发环境下是参数最好是new ElementType[0])

---

## 泛型接口

泛型定义在接口上，作用域：整个接口

格式：public interface 接口名<泛型类型1..>

```
public interface Auto<T> {  
    void run(T t);  
}
```

**实现类：**普通类或者泛型类

1.

```
public class Bus implements Auto<String> // 在这里确定类型 {  
    @Override  
    public void run(String s) {  
        // do something  
    }  
}
```

2.(常用)

```
// Car<T> 定义泛型，Auto<T>使用泛型
public class Car<T> /*在这里确定类型*/ implements Auto<T>{
    @Override
    public void run(T t) {
        T tmp;
    }
}
// in main method: 类型参数的传递:对象 -> Car -> Auto
Car<String> car = new Car<>();
```

## 引入问题:

1. String是Object的子类吗? Y (原来: 父类引用可以指向子类对象)
2. String[]是Object[]的子类吗? N (但是父类型的数组的引用仍然可以指向子类型数组的对象)

数组是一种可协变类型: JVM对数组类型进行了特殊处理, 可以让Object[] 类型的引用, 指向String[]类型的对象, 这样是为了简化开发, 但也带来的新的问题

```
String[] strs = {"hello", "world", "java"};
Object[] objs = strs;
objs[0] = new Date();
// 运行时才报错ArrayStoreException
```

## 泛型通配符

引入目的: 集合不是一种可协变类型, 但是我们提供类似数组"协变"的功能, 但是不引入数组中可能出现的问题

### 1. ? 任意类型

不能添加

### 2. ? extends E 向下限定, E及其子类

不能添加

```
List<? extends Fruit> flist = new ArrayList<Apple>();
flist.add(new Apple()); // 编译错误
flist.add(new Fruit()); // 编译错误
flist.add(new Object()); // 编译错误
/*
List<? extends Fruit>也可以合法的指向一个List<Orange>, 显然往里面放Apple、
Fruit、Object都是非法的。编译器不知道List<? extends Fruit>所持有的具体类型是什么,
所以一旦执行这种类型的向上转型, 你就将丢失掉向其中传递任何对象的能力
*/
```

### 3. ? super E 向上限定，E及其父类

可以添加最低级的那个类及其子类（只有这个可以添加）

```
Collection<? super Animal> c1 = new ArrayList<Object>();
Collection<? super Animal> c2 = new ArrayList<Animal>();
c1.add(new Animal());
c2.add(new Animal());
c2.add(new Dog());
// 子类对象可以看作是一个父类对象，而左边的集合引用指向的集合中
// 存储的对象都是Animal或者Animal的父类，所以可以添加
```

### 泛型擦除

泛型信息只存在于源代码中，编译的时候会把这些信息擦除（字节码文件中没有泛型，还是Object类强转为具体类型）

E -> Object

? extends Animal -> Animal

? super Animal -> Object

练习：

```
/*
练习：在List<String>集合中插入一个int类型的数据。
利用反射绕过泛型检查
*/
List<String> list = new ArrayList<>();
list.add("hello");
list.add("java");
list.add("world");
// list.add(1); 直接加不行

// 获取list的Class对象
Class<? extends List> listClass = list.getClass();
// 获取add方法对应的Method对象
Method add = listClass.getMethod("add", Object.class); // 泛型擦除后 E就是
Object
// 传入对象在Method对象上调用
add.invoke(list, 1);
System.out.println(list); // [hello, java, world, 1]

// 获取该对象
Object obj = list.get(3);
int i = (Integer) obj;
System.out.println(i); // 1
```

# for each

(常用)

**格式:**

```
for (元素数据类型 变量 : 数组或者集合) {  
    使用变量  
}
```

**原理**

1. 编译器对数组进行特殊关照：还是用索引
2. 集合实际上就是用Iterator迭代器

```
// 编译前  
String[] strs = {"a", "b", "c", "d"};  
int[] nums = {1, 2, 3, 4, 5};  
for (String s : strs) {  
    System.out.println(s);  
}  
for (int n : nums) {  
    System.out.println(n);  
}  
  
// 反编译后  
String[] strs = new String[]{"a", "b", "c", "d"};  
int[] nums = new int[]{1, 2, 3, 4, 5};  
String[] var3 = strs;  
int var4 = strs.length;  
  
int var5;  
for(var5 = 0; var5 < var4; ++var5) {  
    // 先取出数再操作，所以不能用for each来进行修改或者添加操作  
    String s = var3[var5];  
    System.out.println(s);  
}  
  
int[] var7 = nums;  
var4 = nums.length;  
  
for(var5 = 0; var5 < var4; ++var5) {  
    int n = var7[var5];  
    System.out.println(n);  
}
```

**优点**

1. 简化了数组和集合的遍历
2. 代码更加简洁



## 缺点

1. 只能查看元素，不能够添加和删除元素（先取出数再操作，所以不能用for each来进行修改或者添加操作）
2. 没有了索引信息，不能对索引进行操作
3. 只能遍历所有元素，不能只遍历一部分元素

## foreach使用条件

必须是数组和实现了Iterable接口的对象（可以返回一个迭代器）

---

## 可变长参数

**格式：**数据类型... 变量名 （0或者多个）

**原理：**底层其实就是数组

**优点：**增加了代码的可读性，参数清晰可见

**注意事项：**可变长参数一定要位于参数列表的最后（否则编译器没法确定边界）

## 问题

1. 一个方法可以有多个可变长参数吗：不可以
2. 如果一个方法需要多个可变长参数，怎么办：前面的用数组代替

---

## 集合与数组的相互转化

### 集合 -> 数组

Object[] toArray()

T[] toArray(**T[]** arr) 加粗的地方为参数化类型（即确定泛型具体是哪种类型）

```
List<String> list = new LinkedList<>();
list.add("hello");
list.add("world");
list.add("java");
// 一般推荐这样，以处理并发环境
String[] stringArr = list.toArray(new String[0]);
```

### 数组 -> 集合

### 视图技术

Arrays类中的

static List asList(T... a)

```
List<String> stringList = Arrays.asList("hello", "kitty", "this", "world");  
System.out.println(stringList); // [hello, kitty, this, world]
```

不能通过视图添加或删除元素（但可以用set方法，因为不改变结构）  
:UnsupportedOperationException

心得：

1. 所有对象共用的数据可以定义为static变量，这样一个类存一份，而不是每个对象存一份
2. 如果输入参数不符合要求，throw new 异常（别人定义或者自定义）
3. 善用三目运算符使代码简洁

---

## day 2

---

### 数据结构

概念：相互之间存在一种或多种特定关系的数据元素的集合

Java中：数据：对象（逻辑）结构：对象间的关系

### 四种基本结构

集合：结构中的数据元素除了同属于一个集合的关系之外，别无其他关系

线性：结构中的数据元素间是1对1的关系

树：结构中的数据元素是1对多的关系

图：结构中的数据元素是多对多的关系

### 逻辑结构和物理结构

逻辑结构：数据元素间的逻辑关系

物理结构：数据结构在计算机中的表示

1. 顺序存储(数组)
2. 链式存储（链表）

## 线性表

概念：n个数据元素的有序序列

1. 元素个数有限
2. 元素之间是有位序关系的

局部有序确定了全局有序，元素的唯一前驱或者唯一后继确定了该元素在线性表中的位置（小学生排队）

一个集合对象能拥有多个迭代器

自己实现的ArrayList

```
// 仿照List接口
public interface MyList<E> extends Iterable<E> {

    boolean add(E e);

    void add(int index, E e);

    void clear();

    boolean contains(Object obj);

    E get(int index);

    int indexOf(Object obj);

    int lastIndexOf(Object obj);

    boolean isEmpty();

    E remove(int index);

    boolean remove(Object obj);

    E set(int index, E element);

    int size();

    MyIterator<E> iterator();

    MyIterator<E> iterator(int index);
}
```

```
// 仿照ListIterator接口
public interface MyIterator<E> extends Iterator<E> {

    void add(E e);

    boolean hasNext();

    E next();

    boolean hasPrevious();

    E previous();

    int nextIndex();
}
```

```

    int previousIndex();

    void remove();

    void set(E e);
}

```

```

// 底层以数组形式实现线性表
public class MyArrayList<E> implements MyList<E> {
    // 数组默认初始容量大小
    private static final int DEFAULT_CAPACITY = 10;
    // 数组最大容量大小
    private static final int MAX_CAPACITY = Integer.MAX_VALUE - 8;
    // 对象数组：泛型E擦除后就是Object
    private Object[] elements;
    // 数组中元素个数
    private int size;
    // 集合被修改的次数：用于检查并发修改异常
    private int modCount;

    // 默认大小
    public MyArrayList() {
        elements = new Object[DEFAULT_CAPACITY];
    }

    // 指定大小
    public MyArrayList(int initialCapacity) {
        // 检查输入参数是否合法
        if (initialCapacity <= 0 || initialCapacity > MAX_CAPACITY) {
            throw new IllegalArgumentException("initialCapacity = " +
initialCapacity);
        }
        elements = new Object[initialCapacity];
    }

    // 在序列尾添加元素e
    @Override
    public boolean add(E e) {
        // 复用方法
        add(size, e);
        return true;
    }

    // 在索引为index的位置添加元素e
    @Override
    public void add(int index, E e) {
        // 检查索引是否合法
        checkIndexForAdd(index);
        // 检查是否需要扩容
        if (size == elements.length) {
            int miniCapacity = size + 1;
            // 计算所需的新的数组的长度
            int newLength = calculateCapacity(miniCapacity);
            // 扩容
            grow(newLength);
        }
        // 将索引位置后面（包括索引）的元素都往后移

```

```

        for (int i = size; i > index; i--) {
            elements[i] = elements[i - 1];
        }
        // 插入元素到索引位置
        elements[index] = e;
        // 元素个数+1
        size++;
        // 集合结构发生了改变
        modCount++;
    }

    private void grow(int newLength) {
        // 申请新数组
        Object[] newTable = new Object[newLength];
        // 复制元素到新表
        for (int i = 0; i < size; i++) {
            newTable[i] = elements[i];
        }
        // 使elements指向新数组，完成扩容
        elements = newTable;
    }

    private int calculateCapacity(int miniCapacity) {
        // 如果所需最小长度：原元素个数 + 需新增元素个数过大
        if (miniCapacity > MAX_CAPACITY || miniCapacity < 0) {
            throw new ArrayOverflowException();
        }
        // 原定数组扩为原来的1.5倍
        int newLength = elements.length + (elements.length >> 1);
        // 如果数组的1.5倍过大
        if (newLength > MAX_CAPACITY || newLength < 0) {
            newLength = MAX_CAPACITY;
        }
        // 返回数组1.5倍和所需最小长度的最大值，若为后者，则下次有元素添加又得马上扩容
        return Math.max(newLength, miniCapacity);
    }

    // 插入元素的索引范围[0,size]
    private void checkIndexForAdd(int index) {
        if (index < 0 || index > size) {
            throw new IllegalArgumentException("index = " + index + ", size = "
+ size);
        }
    }

    @Override
    public void clear() {
        // 因为不置为空会出现内存泄漏：该被回收的对象没有及时被回收，导致堆中可用的内存越来越少
        for (int i = 0; i < size; i++) {
            elements[i] = null;
        }
        size = 0;
        // 集合结构发生改变
        modCount++;
    }

    @Override
    public boolean contains(Object obj) {

```

```

        // 复用其他方法
        return indexOf(obj) != -1;
    }

    @Override
    @SuppressWarnings("unchecked")
    public E get(int index) {
        // 按索引去元素：先检查索引后直接返回就行
        checkIndex(index);
        return (E)elements[index];
    }

    // 检查get/set方法的索引
    private void checkIndex(int index) {
        if (index < 0 || index >= size) {
            throw new IllegalArgumentException("index = " + index + ", size = "
+ size);
        }
    }

    // 找到第一个与目标对象相等的元素的索引，若不存在这样的元素则返回-1
    @Override
    public int indexOf(Object obj) {
        // 将输入的对象分为null和非null两种情况讨论
        if (obj == null) {
            for (int i = 0; i < size; i++) {
                if (elements[i] == null) {
                    return i;
                }
            }
        } else {
            for (int i = 0; i < size; i++) {
                if (obj.equals(elements[i])) {
                    return i;
                }
            }
        }
        return -1;
    }

    @Override
    public int lastIndexOf(Object obj) {
        if (obj == null) {
            for (int i = size - 1; i >= 0; i--) {
                if (elements[i] == null) {
                    return i;
                }
            }
        } else {
            for (int i = size - 1; i >= 0; i--) {
                if (obj.equals(elements[i])) {
                    return i;
                }
            }
        }
        return -1;
    }
}

```

```

@Override
public boolean isEmpty() {
    return size == 0;
}

@Override
@SuppressWarnings("unchecked")
public E remove(int index) {
    // 检查索引
    checkIndex(index);
    // 保存被删除的元素
    E retValue = (E)elements[index];
    // 移动元素，覆盖被删除的元素
    for (int i = index; i < size - 1; i++) {
        elements[i] = elements[i + 1];
    }
    // 将最后一个位置的元素置为null，以免将来出现内存泄漏的隐患
    elements[size - 1] = null;
    size--;
    // 集合结构发生改变
    modCount++;
    return retValue;
}

@Override
public boolean remove(Object obj) {
    // 复用方法
    // 获得第一个与指定元素相等的元素索引
    int index = indexOf(obj);
    // 若不存在此元素
    if (index == -1) {
        return false;
    }
    // 删除该索引上的元素
    remove(index);
    return true;
}

@Override
@SuppressWarnings("unchecked")
public E set(int index, E e) {
    // 检查索引
    checkIndex(index);
    // 保存被替换的元素
    E retValue = (E)elements[index];
    // 替换
    elements[index] = e;
    return retValue;
}

@Override
public int size() {
    return size;
}

@Override
public MyIterator<E> iterator() {
    return new Itr();
}

```

```

}

@Override
public MyIterator<E> iterator(int index) {
    checkIndexForAdd(index);
    return new Itr(index);
}

@Override
public String toString() {
    StringBuilder sb = new StringBuilder("");
    for (int i = 0; i < size; i++) {
        sb.append(elements[i]).append(", ");
    }
    if (size != 0) {
        sb.delete(sb.length() - 2, sb.length());
    }
    return sb.append("]").toString();
}

// 迭代器作为列表的实现类的内部类
private class Itr implements MyIterator<E> {
    // 光标
    int cursor;
    // 保存创建时集合的修改次数，以检查并发修改异常
    int expModCount = modCount;
    // 刚创建时没有返回元素
    int lastRet = -1;

    Itr() {}

    // 指定光标后面元素的索引
    Itr(int index) {
        cursor = index;
    }

    @Override
    public void add(E e) {
        // 检查在迭代器对象创建后，集合结构是否发生了变化
        // 如果集合结构发生了变化：添加、删除、清空
        // 则当前迭代器失效
        checkConModException();
        // 调用外部类的当前对象的add方法
        // 在光标后面的位置添加元素
        MyArrayList.this.add(cursor, e);
        // 光标往后移动
        cursor++;
        // 更新修改计数
        expModCount = modCount;
        // 最近返回元素失效
        lastRet = -1;
    }

    @Override
    public boolean hasNext() {
        return cursor != size;
    }
}

```



```

@Override
@SuppressWarnings("unchecked")
public E next() {
    // 检查迭代器是否有效
    checkConModException();
    // 检查迭代器后面是否还存在元素
    if (!hasNext()) {
        throw new NoSuchElementException();
    }
    // 保存返回的元素的下标
    lastRet = cursor;
    // 返回越过的元素，并使光标右移一位
    return (E)elements[cursor++];
}

private void checkConModException() {
    if (expModCount != modCount) {
        throw new ConcurrentModificationException();
    }
}

@Override
public boolean hasPrevious() {
    return cursor != 0;
}

@Override
@SuppressWarnings("unchecked")
public E previous() {
    checkConModException();
    if (!hasPrevious()) {
        throw new NoSuchElementException();
    }
    lastRet = --cursor;
    return (E)elements[lastRet];
}

@Override
public int nextIndex() {
    return cursor;
}

@Override
public int previousIndex() {
    return cursor - 1;
}

@Override
public void remove() {
    // 检查迭代器是否有效
    checkConModException();
    // 如果最近没有返回的元素，或者返回的元素已经失效
    if (lastRet == -1) {
        throw new IllegalStateException();
    }
    // 调用外部类的方法，删除刚刚返回的元素（利用下标）
    MyArrayList.this.remove(lastRet);
    // 更新修改计数

```

```

        expModCount = modCount;
        // 更新光标位置：顺序逆序遍历都一样
        cursor = lastRet;
        // 使最近返回的元素失效
        lastRet = -1;
    }

    @Override
    public void set(E e) {
        // 检查迭代器是否有效
        checkConModException();
        // 如果最近没有返回的元素，或者返回的元素已经失效
        if (lastRet == -1) {
            throw new IllegalStateException();
        }
        elements[lastRet] = e;
    }
}

```

特殊问题：

```

MyArrayList<String> list = new MyArrayList<>();
list.add("hello");
list.add("java");
list.add("world");

for(MyIterator<String> it = list.iterator(); it.hasNext(); ) {
    String s = it.next();
    if ("java".equals(s)) {
        int index = it.previousIndex();
        list.remove(index);
    }
}
System.out.println(list); // [hello, world]
/*
    在用迭代器遍历时
    用集合API删除倒数第二个元素不会报并发修改异常：
    因为调用集合API：list.remove(1)之后list的size-- 变为2
    而此时cursor也为2，
    hasNext() 中的代码是 return cursor != size;
    此时返回false，不满足循环条件，所以不会进入下一次循环
    而检查并发修改异常的代码在next()方法中，所以不会触发并发修改异常
*/

```

## day 3

linkedlist的实现

分析迭代器的remove方法

 image-20200618202158152

```
if (lastRet != curr) {
    // 正向
    cursor--;
} else {
    // 逆向
    curr = curr.next;
}
lastRet = null;
```

## day 4

### 栈

#### 为什么需要栈

1. 应用场景：FILO
2. 操作更安全：无法直接操作到栈中间或者栈底元素

---

#### 栈的API:

1. push(): 在栈顶添加元素  $O(1)$
2. pop(): 从栈顶删除并返回元素  $O(1)$
3. peek(): 访问栈顶元素  $O(1)$
4. isEmpty(): 判断栈是否为空  $O(1)$

---

#### 实现：顺序/非顺序

##### 顺序映像

```
public class MyStack<E> {
    // 常量：栈的默认容量和最大容量
    private static final int DEFAULT_CAPACITY = 10;
    private static final int MAX_CAPACITY = Integer.MAX_VALUE - 8;
    // top为栈顶指针指向栈顶元素，初始值为-1
    private int top = -1;
    // 元素个数
    private int size;
    // 底层数组
    private Object[] elements;

    public MyStack() {
        elements = new Object[DEFAULT_CAPACITY];
    }

    public MyStack(int initialCapacity) {
        // 检查输入的指定容量
        if (initialCapacity <= 0 || initialCapacity > MAX_CAPACITY) {
            throw new IllegalArgumentException("initialCapacity = " +
initialCapacity);
        }
    }
}
```

```

    }
    elements = new Object[initialCapacity];
}

public void push(E e) {
    // 若当前栈容量已满
    if (size == elements.length) {
        // 计算出新数组的长度
        int newLength = calculateCapacity();
        // 扩容
        grow(newLength);
    }
    // 先移动指针，再添加元素
    elements[++top] = e;
    // 元素个数+1
    size++;
}

private void grow(int newLength) {
    // 申请新数组
    Object[] newTable = new Object[newLength];
    // 复制元素
    for (int i = 0; i < size; i++) {
        newTable[i] = elements[i];
    }
    // 指向新数组
    elements = newTable;
}

private int calculateCapacity() {
    // 若此时底层数组容量已达上限
    if (elements.length == MAX_CAPACITY) {
        throw new ArrayOverflowException();
    }
    // 计划扩容到原数组的1.5倍
    int newLength = elements.length + (elements.length >> 1);
    // 若1.5倍超过最大容量限制，则取最大容量
    if (newLength > MAX_CAPACITY || newLength < 0) {
        newLength = MAX_CAPACITY;
    }
    return newLength;
}

@SuppressWarnings("unchecked")
public E pop() {
    // 先判空
    if (isEmpty()) {
        throw new EmptyStackException();
    }
    // 保存需返回的元素
    E retValue = (E)elements[top];
    // 栈顶位置置空后，指针位置左移
    elements[top--] = null;
    size--;
    return retValue;
}

@SuppressWarnings("unchecked")

```

```

public E peek() {
    if (isEmpty()) {
        throw new EmptyStackException();
    }
    return (E)elements[top];
}

public boolean isEmpty() {
    return size == 0;
}

public void clear() {
    for (int i = 0; i < size; i++) {
        elements[i] = null;
    }
    size = 0;
    top = -1;
}

public int size() {
    return size;
}

@Override
public String toString() {
    StringBuilder sb = new StringBuilder("[");
    for (int i = 0; i < size; i++) {
        sb.append(elements[i]).append(", ");
    }
    if (size != 0) {
        sb.delete(sb.length() - 2, sb.length());
    }
    return sb.append("]").toString();
}
}

```

## 非顺序映像

```

public class MyStackVer2<E> {
    // 栈顶指针
    private Node top;
    private int size;

    public MyStackVer2() {}

    // 头插法将元素入栈
    public void push(E e) {
        top = new Node(e, top);
        size++;
    }

    public E pop() {
        // 先判空
        if (isEmpty()) {
            throw new EmptyStackException();
        }
        E retValue = (E)top.val;
    }
}

```

```

        // 栈顶指针往后（下）移
        top = top.next;
        size--;
        return retValue;
    }

    public E peek() {
        if (isEmpty()) {
            throw new EmptyStackException();
        }
        return (E)top.val;
    }

    public boolean isEmpty() {
        return size == 0;
    }

    public void clear() {
        top = null;
        size = 0;
    }

    // 结点类最后设为私有的静态内部类，这样一个外部类才用存一份定义
    private static class Node {
        Object val;
        Node next;

        public Node(Object val) {
            this.val = val;
        }

        public Node(Object val, Node next) {
            this.val = val;
            this.next = next;
        }
    }
}

```

## 应用场景

1. 函数调用栈：方法的调用和返回对应着一个栈帧的入栈和出栈
2. 括号匹配问题：左括号对应入栈，右括号对应出栈

```

public static boolean isValid(String s) {
    // jdk中推荐使用Deque + LinkedList, 因为更安全
    Deque<Character> stack = new LinkedList<>();
    for (int i = 0; i < s.length(); i++) {
        char c = s.charAt(i);
        // 这样保证s中含有其他字符时也能匹配正确
        if (c == '[') stack.push(']');
        if (c == '{') stack.push('}');
        if (c == '(') stack.push(')');
        if (c == ')' || c == ']' || c == '}') {
            // 栈为空: ]()

```

```


        if (stack.isEmpty() || stack.pop() != c) {
            return false;
        }
    }
}
// 左括号多了，最后栈会不为空，且空字符串也合法
return stack.empty();
}

```

### 3. 编译器利用栈实现表达式求值

问题2：操作符比较后选择压入还是弹出是为了：

保证优先级高的操作符先参与计算，相同优先级的操作符，  
先压入栈的（在左边的），先参与计算（运算是从左到右结合的）

 image-20200618202032727

### 4. 浏览器的前进后退功能

右下角最后一句是若栈B为空则不能前进

 image-20200618202059340

### 5. 利用栈实现DFS（后面讲）

## 队列

操作受限的线性表，只能在队尾插入，队头删除，FIFO

**队列的API：**均为O(1)

1. enqueue()
2. dequeue()
3. isEmpty()
4. peek()
5. size()

**实现：**顺序映像和非顺序映像

**顺序映像：**循环数组（+png）

```

public class MyQueueVer2<E> {

    private static final int DEFAULT_CAPACITY = 10;
    private static final int MAX_CAPACITY = Integer.MAX_VALUE - 8;
    private Object[] elements;
    // 队头和队尾指针
    private int front;
    private int rear;
    private int size;
}

```

```

public MyQueueVer2() {
    elements = new Object[DEFAULT_CAPACITY];
}

public MyQueueVer2(int initialCapacity) {
    // 检查输入的指定容量
    if (initialCapacity <= 0 || initialCapacity > MAX_CAPACITY) {
        throw new IllegalArgumentException("initialCapacity = " +
initialCapacity);
    }
    elements = new Object[initialCapacity];
}

public void enqueue(E e) {
    if (size == elements.length) {
        // 计算出新数组的长度
        int newLength = calculateCapacity();
        // 扩容
        grow(newLength);
    }
    elements[rear] = e;
    rear = (rear + 1) % elements.length;
    size++;
}

private void grow(int newLength) {
    // 申请新数组
    Object[] newTable = new Object[newLength];
    // 复制元素
    for (int i = 0; i < size; i++) {
        int index = (front + i) % elements.length;
        newTable[i] = elements[index];
    }
    // 指向新数组
    elements = newTable;
    // 在新数组中front和rear也指向新的位置
    // 因为相当于重置了元素的位置
    front = 0;
    rear = size;
}

private int calculateCapacity() {
    // 若此时底层数组容量已达上限
    if (elements.length == MAX_CAPACITY) {
        throw new ArrayOverflowException();
    }
    // 计划扩容到原数组的1.5倍
    int newLength = elements.length + (elements.length >> 1);
    // 若1.5倍超过最大容量限制，则取最大容量
    if (newLength > MAX_CAPACITY || newLength < 0) {
        newLength = MAX_CAPACITY;
    }
    return newLength;
}

public E dequeue() {
    if (isEmpty()) {

```



```

        throw new EmptyQueueException();
    }
    E retValue = (E)elements[front];
    // 避免内存泄漏问题
    elements[front] = null;
    front = (front + 1) % elements.length;
    size--;
    return retValue;
}

public E peek() {
    if (isEmpty()) {
        throw new EmptyQueueException();
    }
    return (E)elements[front];
}

public boolean isEmpty() {
    return size == 0;
}

public int size() {
    return size;
}

public void clear() {
    for (int i = 0; i < size; i++) {
        int index = (front + i) % elements.length;
        elements[index] = null;
    }
    front = 0;
    rear = 0;
    size = 0;
}
}

```

## 非顺序映像

```

public class MyQueue<E> {
    // 属性：两个哑结点和队列中元素个数
    private Node head;
    private Node tail;
    private int size;

    public MyQueue() {
        // 连接链表，此时元素个数为0
        head = new Node(null);
        tail = new Node(null);
        head.next = tail;
        tail.prev = head;
    }

    public void enqueue(E e) {
        // 将元素将入链表表尾
        Node nodeToAdd = new Node(e, tail.prev, tail);
        tail.prev.next = nodeToAdd;
        tail.prev = nodeToAdd;
    }
}

```

```

        // 元素个数+1
        size++;
    }

    public E dequeue() {
        // 出队时先判空
        if (isEmpty()) {
            throw new EmptyQueueException();
        }
        // 记录下队头结点的引用
        Node front = head.next;
        // 保存返回值
        E retValue = (E)front.val;
        // 从链表中删去队头结点
        front.prev.next = front.next;
        front.next.prev = front.prev;
        // 元素个数-1
        size--;
        return retValue;
    }

    public E peek() {
        // 先判空
        if (isEmpty()) {
            throw new EmptyQueueException();
        }
        return (E)head.next.val;
    }

    public void clear() {
        head.next = tail;
        tail.prev = head;
        size = 0;
    }

    public int size() {
        return size;
    }

    public boolean isEmpty() {
        return size == 0;
    }

    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder("");
        Node curr = head.next;
        while (curr != tail) {
            sb.append(curr.val).append(", ");
            curr = curr.next;
        }
        if (size != 0) {
            sb.delete(sb.length() - 2, sb.length());
        }
        return sb.append("]").toString();
    }
}

```

// 结点类

```

private static class Node {

    Object val;
    Node prev;
    Node next;

    public Node(Object val) {
        this.val = val;
    }

    public Node(Object val, Node prev, Node next) {
        this.val = val;
        this.prev = prev;
        this.next = next;
    }
}
}

```

### 队列应用场景：

1. 缓存
2. 队列实现BFS

---

工程中一般用阻塞队列

**阻塞队列：**常用于生产者消费者模型

1. 队列满时，入队操作阻塞
2. 队列空时，出队操作阻塞

自己实现的阻塞队列中：所有方法都需设置为同步方法，以保护数据（size）安全

**BlockingQueue**(接口) 下面为它的两个实现子类

|-- ArrayBlockingQueue 容量大小固定（用这个比较好）

|-- LinkedBlockingQueue

构造方法：

LinkedBlockingQueue() 容量大小不固定

LinkedBlockingQueue(int capacity) 容量大小固定

**API：**

**void put(E e)** 阻塞方法

**E take()** 阻塞方法

boolean offer(E e, long timeout, TimeUnit unit)

E poll(long timeout, TimeUnit unit)

---

**线程池**：就是一个典型的生产者-消费者模型

生产者：提交任务的线程

消费者：线程池中的线程

商品：任务

## Java中什么代表线程池？

Executor接口

void execute(Runnable command) 处理用户提交的任务

## 以前我们是如何创建线程池的？

Executors这个工具类的静态方法

**注意事项**：不要使用Executors中的静态方法创建线程池

原因：Executors中的静态方法创建的线程池，底层的阻塞队列是无界的（链表）

在并发高峰期，容易造成OOM异常（不断申请链表结点）

## 怎么创建线程池？

使用ThreadPoolExecutor创建线程池，并传入一个**有界**的阻塞队列

实现阻塞队列和线程池的注意细节：

1. 线程因wait阻塞后,再次抢到锁时**从wait方法后面**开始执行

```
// 所以需要
while (size == elements.length) {..wait()..} // 可以不断轮询，直到条件满足
// 而非
if (size == elements.length) {..wait()..}
```

2. run方法因为父类中没有抛异常 所以子类中也不能抛异常，所以run方法中只能try-catch
3. 线程池中线程一直存活原理：一个数组中的Thread对象会一直轮询（while(true)）  
BlockingQueue,如果队列为空就会阻塞线程，如果队列中有Runnable对象，该Thread对象就会取任务出来执行

## day 5

---

# 树

子树间互不相交：一对多

**定义**（逻辑结构）：

$n$  ( $\geq 0$ ) 个结点的有限集合，没有结点的树称为空树，在任意一颗非空树中：（递归定义：在树的定义中又用到了树的概念）

1. 有且仅有一个特定称为根的结点
2.  $n > 1$  时，其余结点可分为  $m$  个 **互不相交** 的有限集合，且集合自身也是一棵树，并成为根结点的子树（相交了就不是一对多了）

路径：前驱为后继的父亲

---

## 基本术语

孩子：一个结点的子树的根，该结点称为子树的根的父亲

叶子结点：没有孩子的结点

兄弟：父亲相同的结点

路径： $n_1, n_2, \dots, n_k$  的**路径**就是  $n_1, n_2, \dots, n_k$  的这样一个**结点的序列**，路径的长为路径上**边的个数**： $k-1$ ，从根到每个结点恰好存在一条路径，结点自己到自己的路径长度为 0

## 层、深、高

1. 层：从根开始定义，根为第一层，根的孩子为第二层
  2. 深：从上往下看，结点的深度为根到该结点的**路径的长度**：数边个数就行，根的深度为 0
  3. 高：从下往上看，结点到叶子结点最长路径的长度，还是数边个数就行，叶子结点的高为 0
  4. 所以：树的高 == 根的高，树的深度等于最深的叶子结点的深度，且该深度 == 树的高
- 所以：一棵树的高和深 = 树的层次 - 1

**性质**：树有  $n$  个结点的话，就会只有  $n-1$  条边：因为除了根结点以外每个结点都有一个父亲（有一条边连到自己的父亲）

$$N = E + 1$$

---

## 树的实现

孩子兄弟表示法

```
class TreeNode {
    Object element;
    TreeNode firstChild;
    TreeNode nextSibling;
}
```

## 树的应用:操作系统中的目录结构

1. Unix、Linux：树
2. Windows：森林

---

## 二叉树

**概念：**是一棵树，每个结点最多有两棵子树，子树有左右之分，其次顺序不能颠倒

### 特殊二叉树：

1. 完全二叉树：从上到下，从左到右，依次排列
2. 满二叉树：只有度为0和2的结点，没有度为1的结点
3. 完美二叉树：每一层的结点数目都达到最大

### 二叉树的性质：

1. 二叉树在第*i*层最多有 $2^{(i-1)}$ 个结点
2. 层次为*k*的二叉树，最多有 $2^k - 1$ 个结点
3. 对任何一颗二叉树，叶子结点数为 $n_0$ ，度为2的结点数为 $n_2$ ，则 $n_0 = n_2 + 1$ （用树的结点数 = 边数 + 1去推）
4. 完全二叉树中的编号特性：（用数组存储时用到）  
如果编号*i*为1，则该结点是二叉树的根；  
如果编号*i* > 1，则其双亲结点编号为  $\text{parent}(i) = i/2$ ，  
若  $2i > n$  则该结点没有左孩子，否则其左孩子的编号为  $2i$ ，  
若  $2i + 1 > n$  则该结点没有右孩子，否则其右孩子的编号为  $2i + 1$

### 二叉树的存储结构：

1. 顺序映像：（用完全二叉树的编号）  
优点：随机访问，且可以在 $O(1)$ 的时间内知道孩子结点和父亲结点  
缺点：当树比较稀疏的时候，将浪费极大的内存空间，在最坏的情况下，一颗只有*k*个结点的单支树需要 $2^k$ 个存储单元（索引为0的地方不使用）
2. 非顺序映像：链表形式来存储数据元素和数据元素间的关系

```
class TreeNode {
    Object element;
    TreeNode leftChild;
    TreeNode rightChild;
}
```

**二叉树的遍历：**降维过程，非线性->线性，类似于画出三维图形的正视图、侧视图、俯视图

1. 深度优先遍历 (DFS) 时间复杂度为 $O(n)$  因为访问自己的孩子结点的时间复杂度为 $O(1)$

```
// 注意：访问每个结点一次并不能表示时间复杂度是 $O(n)$ ，如：  
for (int i = 0; i < list.size(); i++) {  
    // visit list.get(i)  
}  
// 时间复杂度为 $O(n^2)$ 
```

NLR (先序) LNR (中序) LRN (后序)

2. 广度优先遍历：层级遍历

**二叉树的建树：**根据遍历序列还原树，类似于根据正视图、侧视图、俯视图还原三维图形  
需知道中序遍历序列和先序、后序中的一种：以确定根和左右子树

## 二叉搜索树

**概念：**树中的结点可以按照某种规则进行比较

1. 左子树中**所有结点**的key比根结点的key小，且左子树也是二叉搜索树
2. 右子树中所有结点的key比根结点的key大，且右子树也是二叉搜索树

为什么说二叉排序树：中序遍历序列是一个有序序列

BST能存储null吗 为什么？

不能，因为null无法比较大小

BST能存储key相同的对象吗 为什么？如果不能，可以改进吗

按照BST的严格定义不能够存储key相同的对象

如何改进：

1. 拉链法（把key相同的元素存放在一条链表中）（常用）

```
class TreeNode {  
    Object val;  
    TreeNode left;  
    TreeNode right;  
    TreeNode next;  
}
```

2. 在结点中添加一个count属性，表示key相同元素的个数（不靠谱）
3. 修改BST的定义
  1. 让左子树中所有结点的key值 $\leq$ 根结点的key
  2. 或者让左子树中所有结点的key值 $\geq$ 根结点的key

**二叉树的实现：**

**API：**

增:

boolean add(E e)

删:

boolean remove(E e)

查:

boolean contains(E e)

遍历:

List preorder()

List inorder()

List postorder()

获取集合属性:

int size()

int height()

建树

```
public static <T extends Comparable<? super T> BinarySeachTree buildTree(List preorder,
List inorder)
```

即传入的T可以用T自己的compareTo也可以用从父类中继承过来的compareTo(父类中定义了)或者自己重写实现的父类的compareTo (父类中没定义)

即只要Son的类中有这个方法存在就行 (继承而来的 (父类中已经重写实现了) 或者自己重写实现的)

```
public int compareTo(Son/Father/GrandFather.. o) {
    // ...
}
// 如果定义泛型的时候是<E extends Comparable<E>>, 则要求E中的类定义中必须有
public int compareTo(Son o) {
    // ...
}
```

```
/* <T extends Comparable<? super T>
Actually, it means that T can implement Comparable<? super T>, not just
Comparable<T>.
For example, it means that a Student class can implement Comparable<Person>,
where Student is a subclass of Person:
*/
public class Person {}

public class Student extends Person implements Comparable<Person> {
    @Override public int compareTo(Person that) {
        // ...
    }
}
```

如何添加

```
public boolean add(E e) {
    // 不允许传入null
    if (e == null) {
```



```

        throw new IllegalArgumentException("null element is not
allowed");
    }
    //      return addIteratively(e);
    // 记录尝试添加元素之前的元素个数
    int oldSize = size;
    // 在以root为根的树中尝试添加e元素
    // 并将所形成的新的树的根结点返回给root
    root = add(root, e);
    // 若树中的元素个数发生了变化，则说明添加成功
    return size > oldSize;
}

private TreeNode add(TreeNode root, E e) {
    // 若当前树为空树，则直接添加，并改变元素个数，并将root指向新元素结点
    if (root == null) {
        size++;
        return new TreeNode(e);
    }
    // 若当前树不为空，则比较待添加元素与根结点元素的大小
    int cmp = e.compareTo(root.val);
    if (cmp < 0) {
        // 若小于根结点则递归添加到左子树
        root.left = add(root.left, e);
    } else if (cmp > 0) {
        // 若大于根结点则递归添加到右子树
        root.right = add(root.right, e);
    }
    // 相等则也不做，不添加，元素个数不发生变化

    // 当if-else if语句结束时，root的左子树和右子树已经构建完成
    // 返回新创建好的树的根结点
    return root;
}

```

## 如何删除

```

public boolean remove(E e) {
    if (e == null) {
        throw new NoSuchElementException();
    }
    // 记录尝试删除前的元素个数
    int oldSize = size;
    // 在以root为根的树中删除e元素，并返回所形成的树的根结点
    root = remove(root, e);
    // 若元素个数发生了变化
    return size < oldSize;
}

private TreeNode remove(TreeNode root, E e) {
    // 若当前树为空树，则返回null
    if (root == null) {
        return null;
    }
    // e与当期树的根结点比较大小
    int cmp = e.compareTo(root.val);
    if (cmp < 0) {

```

```

        // 小于则在左子树中递归删除e，并将所形成的根结点返回给root.left
        root.left = remove(root.left, e);
    } else if (cmp > 0) {
        // 大于则在左子树中递归删除e，并将所形成的根结点返回给root.right
        root.right = remove(root.right, e);
    } else {
        // 找到了需要删除的元素
        // 若度为2
        if (root.left != null && root.right != null) {
            // 先找到其在中序遍历序列的后继，即右子树中的最小结点，不可能有左子树
            // (如果有那它就不是最小结点)
            // 所以下面的remove(root.right, successor.val)一定会落到else分支
            // 中：即度为1或者度为0的情况
            TreeNode successor = root.right;
            while (successor.left != null) {
                successor = successor.left;
            }
            // 与后继结点交换数据域
            root.val = successor.val;
            // 最终还是要靠else中的代码删除
            // 在当前root的右子树中递归删除successor，并将所形成的树的根结点返回
            // 给root.right
            // 将度为2的情况退化为度为1或者0的情况
            root.right = remove(root.right, successor.val);
        } else {
            // 度为1
            // 真正造成size变化的地方在这里
            size--;
            // 将待删除的结点的唯一孩子或者null赋值给上层调用的
            // root.left/root.right
            return root.left != null ? root.left : root.right;
        }
    }
}

// 两个递归方法返回时，root的左子树和右子树已经构建完成，返回根结点
return root;
}

```

层级遍历(广度优先搜索 借助队列)

```

public List<List<E>> levelOrder() {
    List<List<E>> list = new ArrayList<>();
    if (isEmpty()) {
        return list;
    }
    Queue<TreeNode> queue = new LinkedList<>();
    queue.offer(root);
    while (!queue.isEmpty()) {
        // 先记录队列中的元素个数，即当前层的元素个数
        int rowElementSize = queue.size();
        // 暂存当前层元素
        List<E> row = new ArrayList<>();
        for (int i = 0; i < rowElementSize; i++) {
            TreeNode node = queue.poll();
            row.add(node.val);
            if (node.left != null) {
                queue.offer(node.left);
            }
        }
    }
}

```

```

    }
    if (node.right != null) {
        queue.offer(node.right);
    }
}
// 将当前层加入到所有层中
list.add(row);
}
return list;
}

```

### 用栈实现DFS:

用递归的栈需要在**虚拟机栈**中分配空间（方法的调用和返回对应一个栈帧在线程私有的栈中的入栈和出栈）

用数据结构的栈是在**堆**分配空间（`Deque stack = new LinkedList<>();`）

而虚拟机栈(KB,MB级别)的可用空间远小于堆(GB级别)中的可用空间，所以用数据结构的栈可以避免栈溢出

### 先序遍历

```

public List<E> preOrderWithStack() {
    List<E> list = new ArrayList<>();
    if (isEmpty()) {
        return list;
    }
    Deque<TreeNode> stack = new LinkedList<>();
    stack.push(root);
    while (!stack.isEmpty()) {
        TreeNode node = stack.pop();
        list.add(node.val);
        // 注意：因为先序遍历时NLR所以需要先右孩子入栈，再将左孩子入栈
        // 才是先处理左子树再处理右子树
        if (node.right != null) {
            stack.push(node.right);
        }
        if (node.left != null) {
            stack.push(node.left);
        }
    }
    return list;
}

```

### 后序遍历

```

public List<E> postOrderWithStack() {
    List<E> list = new LinkedList<>();
    if (isEmpty()) {
        return list;
    }
    Deque<TreeNode> stack = new LinkedList<>();
    stack.push(root);
    while (!stack.isEmpty()) {
        TreeNode node = stack.pop();
    }
}

```

头部

```
// 后序遍历LRN，用类似头插法的方式实现后序，即每次插入元素的时候都插入list

list.add(0, node.val);
// 因为要先处理右子树再处理左子树，所以先将左孩子压入站
if (node.left != null) {
    stack.push(node.left);
}
if (node.right != null) {
    stack.push(node.right);
}
}
return list;
}
```

中序遍历

```
public List<E> inOrderWithStack() {
    List<E> list = new ArrayList<>();
    if (isEmpty()) {
        return list;
    }
    // 刚开始curr指向根结点
    TreeNode curr = root;
    Deque<TreeNode> stack = new LinkedList<>();
    // 循环直到curr指向空引用或者栈为空
    while (curr != null || !stack.isEmpty()) {
        // 若当前结点不为空，将当前结点入栈，并一直向左走
        while (curr != null) {
            stack.push(curr);
            curr = curr.left;
        }
        // 此时栈顶元素为当前处理的最左元素，没有左子树所以它是LNR中的N
        curr = stack.pop();
        // 将N加入队列
        list.add(curr.val);
        // 在往右子树中找即LNR中的R
        curr = curr.right;
    }
    return list;
}
```

建树

```
public static <T extends Comparable<? super T>> BinarySearchTree<T>
buildTree2(List<T> preOrder, List<T> inOrder) {
    BinarySearchTree<T> tree = new BinarySearchTree<>();
    tree.size = preOrder.size();
    tree.root = tree.build2(preOrder, inOrder);
    return tree;
}

private <T extends Comparable<? super T>> TreeNode build2(List<E>
preOrder, List<E> inOrder) {
    if (preOrder == null || preOrder.isEmpty()) {
        return null;
    }
}
```

```

    E rootValue = preOrder.get(0);
    TreeNode root = new TreeNode(rootValue);
    int index = inOrder.indexOf(rootValue);

    List<E> leftPreOrder = preOrder.subList(1, 1 + index);
    List<E> leftInOrder = inOrder.subList(0, index);
    root.left = build2(leftPreOrder, leftInOrder);

    List<E> rightPreOrder = preOrder.subList(1 + index,
preOrder.size());
    List<E> rightInOrder = inOrder.subList(1 + index, inOrder.size());
    root.right = build2(rightPreOrder, rightInOrder);

    return root;
}

```

### 时间复杂度

在我们自己实现的BST中，添加删除查找元素的时间复杂度是多少呢（与树的高度直接相关）

查找：O(h)

添加：O(h)

删除：O(h)

问题：每次删除度为2的结点，都是从右子树中删除。在动态的添加和删除元素的过程中，会导致树往左倾斜。这样的树是不平衡的，最坏情况下会退化成链表O(n)。

自平衡二叉树

a. AVL树 (对于任意一个结点，它的左子树和右子树的高度之差不超过1)  $h = O(\log n)$

b. 红黑树: 利用红黑规则，保证树的高度  $h = O(\log n)$

AVL树 VS 红黑树

查找：AVL > 红黑树(因为AVL对树的高度规定更为严格，所以添加删除时为了维护特性所需的额外开销也更多)

添加：红黑树 > AVL

删除：红黑树 > AVL

## Week 8

### day 2

### Set

#### 概念

**不包含重复元素**的Collection（重复即为：`e1.equals(e2) == true`），且最多包含一个null元素（如果允许存储null的话）

注意事项：

1. Set集合可以无序 (HashSet)，也可以是有顺序的(TreeSet)
2. Set中的元素都是以key的形式存储在Map中（其对应的value均为一个PRESENT (Object对象)占位对象）

Set中实际存储元素的都是一个Map

---

## HashSet

### 概述

1. 底层数据结构是哈希表 (HashMap: 数组 + 链表)
2. 不保证迭代顺序, 特别是不保证该顺序恒久不变 (哈希表扩容再散列时元素的相对位置会发生变化, 重新计算索引, 并头插入新链表中)
3. 允许存储null元素
4. 不同步

---

**哈希表**(JDK中的实现: HashMap) 数组 + 链表 (拉链法)

Map中的Key是唯一的, 不能重复 (通过添加操作put保证)

### API

**增:** V put(K key, V value)

1. 计算key的hash  
`int hash = hash(key);`
2. 计算索引 (根据索引就能找到链表table[index])  
`int index = hash % table.length;`
3. 遍历链表, 判断key是否存在  
存在: **更新**key关联值, 并把原来关联的值返回  
不存在, 在该链表中链表中添加键值对 (Entry对象), 返回null

```
// 所以set中的add方法是  
return map.put(e, PRESNET) == null;
```

**删:** V remove(K key)

```
int hash = hash(key);  
  
int index = hash % table.length;  
  
遍历链表判断key是否存在  
存在: 删除键值对, 并返回其值  
不存在: 返回null
```

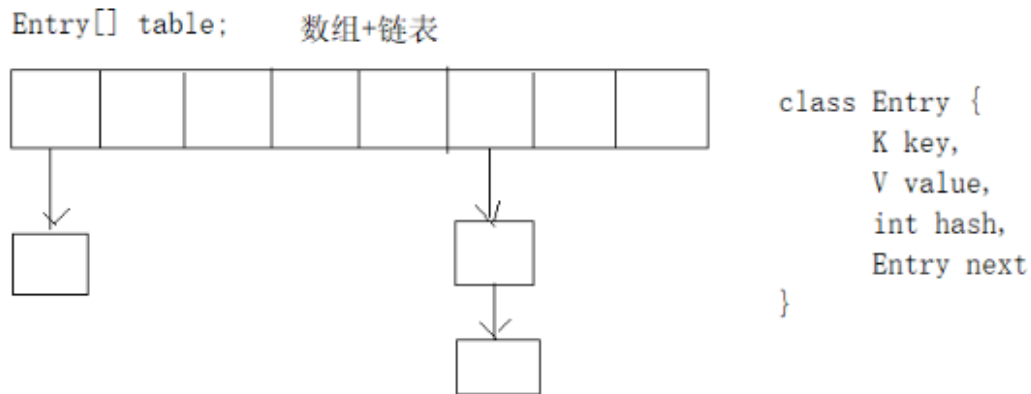
**查:** V get(K key)

```
int hash = hash(key);  
  
int index = hash % table.length;
```

遍历链表判断key是否存在

存在：返回key对应的value

不存在：返回null



在哈希函数已经把键值对均匀散射的前提下，增删查的时间复杂度是多少

$O(L)$   $L$ 表示链表的平均长度（因为其他操作都是 $O(1)$ ，所以操作的时间复杂度取决于遍历链表的时间复杂度）

$O(L) \rightarrow O(1)$ ：控制链表的平均长度不超过某个常数 $M$ ， $M$ 称为加载因子loadFactor

所以在链表平均长度超过 $M$ 时需要进行扩容，然后进行再散列，以控制链表的平均长度，(扩容)再散列时会导致元素间的相对位置发生改变

---

## HashSet源码分析：HashSet如何保证元素唯一性

Set中存储的元素是作为Map的key，而Map的key是唯一的

```
/*HashSet是如何保证元素的唯一性的呢？
HashMap的 key 值是唯一的。
*/

class HashSet<E> implements Set {
    private transient HashMap<E, Object> map; // 里面的value都是PRESENT

    /*占位符 dumb value*/
    private static final Object PRESENT = new Object();

    public boolean add(E e) {
        // put返回null表示Map中添加键值对成功了
        return map.put(e, PRESENT) == null;
    }
}

// JDK1.7
class HashMap<K, V> implements Map<K, V> {

    public V put(K key, V value) {
        //看哈希表是否为空，如果空，就开辟空间(延缓了申请数组空间的时间：从创建对象->加入元素)
        if (table == EMPTY_TABLE) {
            inflateTable(threshold);
        }
    }
}
```

```

//判断对象是否为null
if (key == null)
    return putForNullKey(value);

int hash = hash(key); // 和key对象的hashCode()方法相关

//在哈希表中查找hash值
int i = indexFor(hash, table.length);

// 遍历链表
for (Entry<K,V> e = table[i]; e != null; e = e.next) {
    Object k;
    // 同种类型的对象，hash值相同对象不一定相等，但hash不同两个对象一定不相等
    // e.hash = hash, 判断e.hash = hash(key) 判断e的，hash一样，它们的值不一定
    一样。

    // 第二个判断是判断两个是否是同一个对象
    // 最终才调用equals方法：为了提高效率
    if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {
        // key 与 e.key 是相等的
        V oldValue = e.value;
        e.value = value;
        e.recordAccess(this);
        return oldValue;
        //走这里其实是没有添加元素
    }
}

// for loop中没有返回，走到这里说明没有相等的key，所以添加元素
// 添加键值对
modCount++;
addEntry(hash, key, value, i); //把元素添加
return null;
}

transient int hashSeed = 0;

final int hash(Object k) { //k="hello", hash(Object k) 只有k的hashCode()方法有
    关。

    int h = hashSeed;
    if (0 != h && k instanceof String) {
        return sun.misc.Hashing.stringHash32((String) k);
    }

    h ^= k.hashCode(); //这里调用的是对象的hashCode()方法

    // 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);
}
}

```

注意事项：

1. HashSet的存储依赖于对象的两个方法：int hashCode()(影响对象的hash值，继而影响到索引值)  
boolean equals(Object obj) (影响对象是否能找到相等的对象即Entry对象中的key)



(实际上是通过底层Map的put()来保证其元素唯一性的)

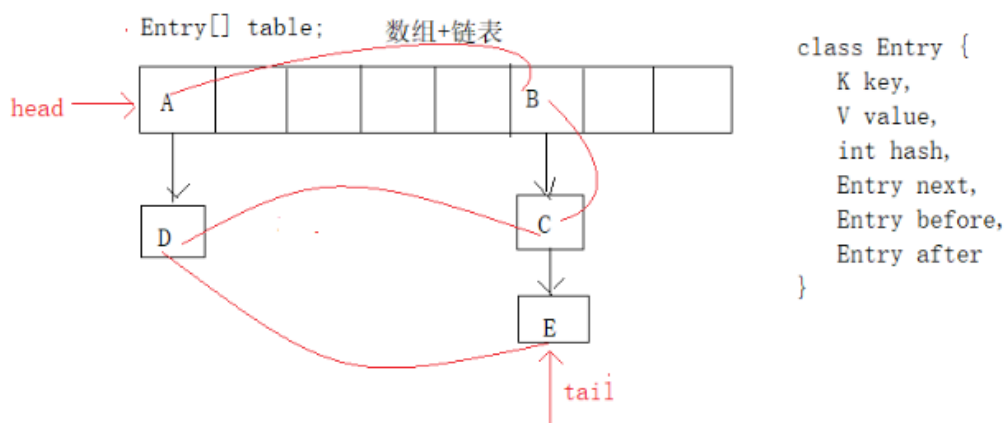
2. 千万不要修改HashSet元素的属性值 (不再唯一, 但是仍可以存在在Set中, 破坏了Set的性质)
3. 所以自己定义的类放入HashSet中, 要保证**唯一存储**必须重写hashCode和equals方法

---

## LinkedHashSet

### 概述

1. 底层数据结构是哈希表 + 双向链表 (在Entry类的定义中多加了两个指针域before & after)
2. 哈希表保证元素的唯一性
3. 链表定义了迭代顺序 (按元素的插入位置决定)
4. 不同步
5. 是HashSet的子类
6. 可以存储null



LinkedHashSet具有哈希表和链表的好处, 添加删除查找的时间复杂度为O(1), 并且具有可预知的迭代顺序。

---

## TreeSet

### 概述

1. 底层是TreeMap, TreeMap的底层是红黑树(自平衡的二叉搜索树)
2. 如果创建对象时, 没有传入Comparator对象, 则根据自然顺序进行排序
3. 如果创建对象时, 传入了Comparator对象, 则根据Comparator进行排序
4. 不能存储null元素, 除非在Comparator中定义了null的比较规则
5. 不同步

**自然排序:** 按照Comparable中定义的比较规则进行排序, int compareTo(Object obj)

---

### 构造方法

TreeSet()

创建一个空的TreeSet，根据自然顺序进行排序

TreeSet(Comparator<? super E> comparator)

创建一个空的TreeSet，根据comparator进行排序

Comparator接口

int compare(T o1, T o2)

### 注意事项

1. TreeSet不依赖于equals和hashCode方法
2. TreeSet依赖于元素的compareTo方法或者是传入的Comparator对象的compare方法
3. 不要修改TreeSet中元素的属性值

### TreeSet如何保证元素的唯一性

依赖于元素的compareTo方法或者是传入的Comparator对象的compare方法

若是比较方法返回0，则表明两个对象相等，**则不添加该对象**（所以在写compare方法时一定要小心，把所有的成员变量都比一遍）

```
public class TreeSet<E> implements Set<E> {

    private transient NavigableMap<E, Object> m;

    // Dummy value to associate with an Object in the backing Map
    private static final Object PRESENT = new Object();

    TreeSet(NavigableMap<E, Object> m) {
        this.m = m;
    }

    public TreeSet() {
        this(new TreeMap<E, Object>());
    }

    public TreeSet(Comparator<? super E> comparator) {
        this(new TreeMap<>(comparator));
    }

    public boolean add(E e) { // "Hello"
        return m.put(e, PRESENT) == null;
    }
}

public class TreeMap<K, V> implements Map<K, V> {
    private final Comparator<? super K> comparator;

    private transient Entry<K, V> root;

    public TreeMap() {
```

```

        comparator = null;
    }

    public TreeMap(Comparator<? super K> comparator) {
        this.comparator = comparator;
    }

    public V put(K key, V value) { // key="Hello", value=PRESENT
        Entry<K,V> t = root;
        // 空树
        if (t == null) {
            compare(key, key); // type (and possibly null) check

            root = new Entry<>(key, value, null);
            size = 1;
            modCount++;
            return null;
        }
        int cmp;
        Entry<K,V> parent;
        // split comparator and comparable paths
        Comparator<? super K> cpr = comparator;
        // 传入了比较器
        if (cpr != null) {
            do {
                parent = t;
                // 通过比较器中定义的规则进行标胶
                cmp = cpr.compare(key, t.key);
                if (cmp < 0)
                    t = t.left;
                else if (cmp > 0)
                    t = t.right;
                else
                    // 更新value值
                    return t.setValue(value);
            } while (t != null);
        }
        // 没有传入比较器
        else {
            // 如果key为null, 抛出空指针异常。
            if (key == null)
                throw new NullPointerException();
            // 强转成Comparable类型, 如果类没有实现Comparable接口就会抛出
            ClassCastException.
                @SuppressWarnings("unchecked")
                Comparable<? super K> k = (Comparable<? super K>) key;
            do {
                parent = t;
                cmp = k.compareTo(t.key);
                if (cmp < 0)
                    t = t.left;
                else if (cmp > 0)
                    t = t.right;
                else
                    // 更新value值
                    return t.setValue(value);
            } while (t != null);
        }
    }

```

```

        // 添加结点
        Entry<K,V> e = new Entry<>(key, value, parent);
        if (cmp < 0)
            parent.left = e;
        else
            parent.right = e;
        // 重新调整平衡
        fixAfterInsertion(e);
        size++;
        modCount++;
        return null;
    }
}

```

---

## API

获取大于或者小于某个对象的结点

E ceiling(E e) >=

E floor(E e) <=

E higher(E e) >

E lower(E e) <

对最小最大结点的操作

E first()

E last()

E pollFirst()

E pollLast()

视图技术

NavigableSet subSet(E fromElement, boolean fromInclusive, E toElement, boolean toInclusive)

---

try with resource

```

try (PrintWriter writer = new PrintWriter("student.txt")) {
    // 放在括号内，输出流会自动关闭
} catch (...) {

}

```

## Collection小结

Collection接口下面有三个接口

List

--| ArrayList

--| LinkedList

--| Vector

--| stack

Set

--| HashSet

--| LinkedHashSet

--| TreeSet

Queue

--| Deque

--| LinkedList

---

## day 3

---

### Map

#### 映射含义

字典：单词 -> 单词的信息（看作一个整体（对象））

索引：关键字 -> 关键字所在页码

定义域中的值在值域中有唯一对应的值

---

#### 概述

Map<K, V>

1. 将键映射到值的对象（可以通过键快速地查找到值）
2. Map中键是不重复的（而值可以是重复的）
3. 每个键最多只能映射到一个值

---

#### API

##### 增（改）

V put(K key, V value)

void putAll(Map<? extends K,? extends V> m)

##### 删

V remove(Object key)  
void clear()

## 查

V get(Object key)  
boolean containsKey(Object key)  
boolean containsValue(Object value)

提供containsKey原因: 不能用 if (map.get(key) != null)来判断key是否存在, 因为value可能就是null

## 获取集合属性

boolean isEmpty()  
int size()

**遍历:** Map接口没有extends Iterable 所以没有 iterator()这个抽象方法 所以不能用迭代器遍历

```
public interface Map<K,V> {  
    interface Entry<K, V> {  
        K getKey()  
        V getValue()  
        V setValue(V value)  
    }  
}
```

Map.Entry Map接口中定义的接口 (类似于外部类和内部类)

1. Set<Map.Entry<K,V>> entrySet() 用于遍历键值对
2. Set keySet() 遍历键
3. Collection values() 遍历值, 因为值可以是重复的 (对比定义域和值域), 所以不能用Set装, 而需要用集合装

---

## HashMap

哈希表 (抽象结构) HashMap (具体实现)

### 概述

1. 基于哈希表的Map接口实现
2. 允许null键和null值
3. 不保证映射的顺序, 特别是它不保证该顺序恒久不变
4. 不同步

### 构造方法

HashMap()  
默认容量为16,默认加载因子为0.75f

HashMap(int initialCapacity)

容量大小为大于等于initialCapacity的最小2的n次幂, 默认加载因子为0.75f

HashMap(int initialCapacity, float loadFactor) 赋值小数形式的loadfactor需加f, 因为小数默认是double类型

容量大小为大于等于initialCapacity的最小2的n次幂, 加载因子为loadFactor

HashMap(Map<? extends K,? extends V> m)

创建HashMap对象, 并把m中键值对添加进去

## HashMap vs Hashtable (也是Map接口的子类)

同: 都是采用数组 + 链表的方法实现了哈希表

异:

1. HashMap不同步, Hashtable同步
2. HashMap允许存储null键和null值, Hashtable不允许

注意事项:

不要忘记重写key的hashCode和equals方法 (map的put方法只保证key不重复)

---

## LinkedHashMap

### 概述

1. HashMap的子类
2. Map接口的哈希表和 (双向) 链表实现, 具有可预知的迭代顺序
3. 哈希表保证了键的唯一性
4. 链表定义了迭代顺序 (为键值对的插入顺序)
5. 可以存储null键和null值
6. 不同步

---

## TreeMap

### 概述

1. 底层数据结构是红黑树
2. 如果创建对象时没有传入Comparator对象, 键将按自然顺序排序 (此时要求键实现Comparable接口)
3. 如果创建对象时传入了Comparator对象, 键将按Comparator进行排序 (此时不要求键实现Comparable接口)
4. 不同步

### 构造方法:

TreeMap()

TreeMap(Comparator<? super K> comparator)

## API:

K ceilingKey(K key) >=  
K floorKey(K key) <=  
K higherKey(K key) >  
K lowerKey(K key) <  
  
K firstKey()  
K lastKey()  
Map.Entry<K,V> pollFirstEntry()  
Map.Entry<K,V> pollLastEntry()  
  
(Outer.Inner的形式)

## 视图方法

NavigableMap<K,V> subMap(K fromKey, boolean fromInclusive, K toKey, boolean toInclusive)

---

## Propertities(属性集)

### 概述:

1. Hashtable<Object, Object>的子类
2. Propertities类表示了一个可持久的属性集（不要把它跟properties文件搞混了，Propertities对象的内容是存储在堆上的）（持久：内存中的数据就是不持久的，外存，网络中存放的文件是持久的）
3. Propertities可以把数据保存到流中（可持久），或者从流中加载数据
4. Propertities中每个键和值都应该是一个字符串

## API:

### 增（改）：

Object setProperty(String key, String value)

### 查:

String getProperty(String key)  
String getProperty(String key, String defaultValue)

删：用父类的remove(Object obj) 删除键值对

### 遍历:

Set stringPropertyNames()



## 和流相关的API

`void store(OutputStream out, String comments)`

**void store(Writer writer, String comments)** (writer中指定编码表) (将Properties对象中的内容通过流存储到外存中, 并添加注释)

**void storeToXML(OutputStream os, String comment)**

`void storeToXML(OutputStream os, String comment, String encoding)`

`void load(InputStream inStream)`

**void load(Reader reader)** (reader中也可以指定字符集)

**void loadFromXML(InputStream in)**

### 注意事项:

a. store和load方法中, 字节流默认使用ISO 8859-1 (所以尽量不用) (欧洲地区用的, 1字节表示字符)

b. storeToXML和loadFromXML方法中, 字节流默认使用UTF-8编码。

1. 不要使用Hashtable里面定义的方法(put)添加键值对, 因为可以插入不是String类型的数据
2. 如果一个Properties中含有非String的键值对, 那么和这个Properties是“不安全的”, 调用store或者save方法将失败(无法输出到流中)

## 小结

Map

|-- HashMap

|-- LinkedHashMap

|-- Hashtable

|-- properties

|-- TreeMap

## day 5

---

## 符号表

### 概述

符号表: 描述一张抽象的表格, 会将信息(值, 多种信息可以封装成一个对象)存储在其中, 然后按照指定的键来搜索并获取这些信息(对象)

也称为字典: 键: 单词 值: 单词对应的定义、发音和词源

也称为索引: 键: 术语 值: 书中该术语出现的页码

### 符号表应用

应用	目的	键	值
字典	查找单词的释义	单词	释义
图书索引	找出相关页码	术语	一串页码
文件共享	找到歌曲下载地址	歌曲名	计算机IP(URL)
账号管理	处理交易	账号号码	交易详情
网络搜索	找出相关网页	关键字	网页名称
编译器	找出符号的类型和值	变量名	类型和值

无序符号表 (HashMap) (哈希表)

有序符号表(TreeMap) (红黑树)

## 哈希表

### 概述

如果所有的键都是小整数(有确定范围的整数)，可以用一个数组来实现的符号表，将数组的索引作为键，而数组中对应的位置存储键关联的值。

哈希表能够处理更加复杂类型的键，但我们需要用哈希函数将键转换成数组的索引。

哈希表的核心算法可以分为两步。

1.用哈希函数将键转换为数组中的一个索引。理想情况下不同的键都能转换成不同的索引值。当然这只是理想情况下，所以我们需要处理两个或者多个键都散列到相同索引值的情况 (哈希碰撞)。

2.处理碰撞冲突。

a. 开放地址法

线性探测法, 平方探测法, 再散列法...

b. 拉链法

### 哈希函数

一个优秀的 hash 算法，满足以下特点：

1. 正向快速：给定明文和 hash 算法，在有限时间和有限资源内能计算出 hash 值。
2. 逆向困难：给定（若干） hash 值，在有限时间内很难（基本不可能）逆推出明文。
3. 输入敏感：原始输入信息修改一点信息，产生的 hash 值看起来应该都有很大不同。（自己写 hash()的原因）
4. 冲突避免：很难找到两段内容不同的明文，使得它们的 hash 值一致（发生冲突）。即对于任意两个不同的数据块，其hash值相同的可能性极小；对于一个给定的数据块，找到和它hash值相同的数据块极为困难。

(我们可以简单地把哈希函数理解为在模拟随机映射，然后从随机映射的角度去理解哈希函数)

定义域	值域
x1	y1
x2	y2
...	...

哈希函数只能是模拟随机映射，不能真正做到随机映射。因为算法其实就是一个有规律的。

以hashCode为例，定义域范围： $\infty$ ，值域范围： $2^{32}$

随机一个数据，hash刚好和h相等的概率：1/43亿

## 经典的哈希函数：

MD4: 128位，MD4已被证明不够安全。

MD5: MD5 已被证明不具备"强抗碰撞性"。（就是可以被破解）（小企业）

SHA1: SHA-1 已被证明不具"强抗碰撞性"。（大企业）

SHA2:

SHA3: 相关算法已经被提出。

## 哈希算法的应用：

- 指纹 (身份的标识)

a. 证书, 文件

b. 加密

...

- 散列

a. 集群, 分布式(负载均衡)

b. 数据结构

...

加密：

注册：密码  $\rightarrow$  hash(密码)  $\rightarrow$  hash值

登录：密码  $\rightarrow$  hash(密码)  $\rightarrow$  hash值

因为数据库被盗，密码遭到破解又是怎么回事呢？

撞库攻击

黑客会维护一个很大的Map

hash值	明文
-------	----

...	abc
-----	-----

...	123
-----	-----

如何防备撞库攻击呢？

a. 设计密码尽量复杂

b. 加盐

注册：密码  $\rightarrow$  密码 + 盐  $\rightarrow$  hash(加盐后的密码)  $\rightarrow$  hash值

注册：密码  $\rightarrow$  密码 + 盐  $\rightarrow$  hash(加盐后的密码)  $\rightarrow$  hash值

破解：hash值  $\rightarrow$  明文

hash算法可以用于加密领域，那么hash算法是一种加密算法吗？

不是。

注意事项：hash算法可以用于加密领域，但hash算法不是一种加密算法，加密是相对于解密而言的，没有解密何谈加密呢，hash的设计以无法解密为目的的。并且如果我们不附加一个随机的salt值，hash口令是很容易被字典攻击入侵的。

## 哈希函数在数据结构中的应用：

只要求哈希函数能满足：

1. 计算速度快
2. Hash值平均分布

**容量问题：**为什么哈希表中数组的长度要设置成 $2^n$ ：方便取余运算

取余时模为2，可以直接用位运算  $a \% b \Leftrightarrow a \& (b - 1)$

问题：取余操作是一个相对复杂的操作。

```
int hash = hash(key)
```

```
int index = hash % table.length;
```

取余操作是一个相对复杂的操作： $a \% b = a - (a / b) * b$ ;

取余操作包含一次除法, 一次乘法, 一次减法。

问题：如何简化取余操作？

当模是2的幂时，我们可以把取余操作简化成位操作。

$a \% b = a \& (b-1)$  ( $b=2^n$ )

```
1101 1010
% 0010 0000
0001 1010
```

```
1101 1010
& 0001 1111
0001 1010
```

求  $\geq \text{initialCapacity}$  的最小的  $2^n$

如何计算大于等于cap的最小2的幂？

```
int n = cap - 1;
```

```
n |= n >> 1;
```

```
n |= n >> 2;
```

```
n |= n >> 4;
```

```
n |= n >> 8;
```

```
n |= n >> 16;
```

```
return n + 1;
```

假设字长为8(也就是用8位表示一个整数)  $\text{cap} = 11$

$n = 11 - 1 = 10$

```
0000 1010
| 0000 0101
0000 1111
```

```
0000 1111
| 0000 0011
0000 1111
```

```
0000 1111
0000 0000
0000 1111
```

## 并发场景

HashMap类不同步，在并发场景中可能会出现许多问题，如查找的时候可能会出现死循环

线程1扩容的时候切换到了线程2，线程2扩容后改变了A、B的相对位置，而线程1继续往下执行就会将两个结点之间连成一个环（假设A,B哈希值相同）

在使用HashMap的过程中可能会出现死循环。

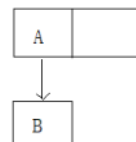
```
private void grow(int newCapacity) {
    Entry[] newTable = new Entry[newCapacity];
    for (Entry e : table) {
        while (e != null) {
            Entry next = e.next; ← 线程1, e --> A, next --> B
            int index = indexFor(e.hash, newCapacity);
            e.next = newTable[index];
            newTable[index] = e;
            e = next;
        }
    }
    table = newTable;
    threshold = (int) (newCapacity * loadFactor);
}
```

线程1：执行grow方法

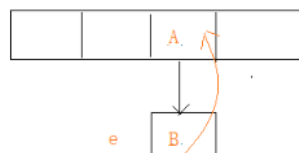
线程2：执行完grow方法

线程1：接着执行

扩容前：



扩容后：



### 1. HashMap和Hashtable的区别

HashMap允许null键和null值，不同步

Hashtable不允许null键和null值，同步

### 2. 应该选择哪个并发安全的Map

a. Hashtable 锁对象为this对象

b. Collections.synchronizedMap(Map map) 锁对象为this对象

c. **ConcurrentHashMap**锁对象是链表的头结点，提高了并发度（在一个链表上增删元素与另一个链表无关）

### 3. JDK8对HashMap的改进

某条链表长度超过8，就把这条链表转化成红黑树（防止有人恶意存储很多哈希值相同的元素，使得某个链表的长度过长，导致哈希表退化为链表）

某条链表长度低于6，就把红黑树变成链表

## Week 9

### day 1

### 2-3-4树

(就是4阶B树)

概念：

- 在二叉搜索树上进行了扩展，允许有多个键（1-3）
- 树保持完美平衡

完美平衡：根到每个叶子结点的路径都是一样长的

### 结点:

- 2-node: 1个键, 两个孩子
- 3-node: 2个键, 三个孩子
- 4-node: 3个键, 四个孩子

特点: 动态添加和删除元素时, 都能保持树的完美平衡

### 查找:

- 和当前结点的所有的键进行比较
- 如果当前结点中没有, 就找到对应的区间
- 依据链接找到下一个结点 (递归)

### 插入:

- 查找找到键应该的插入位置 (树底)
- 2node --> 3node
- 3node --> 4node
- 4node -> 分裂

### 分裂4-node的两种解决方案

- bottom-up (从上到下后还需要再从下到上)
- top-down (实际使用)

### top-down: (一次遍历完成)

- 沿着查找路径分裂4-node(局部变换, 不会影响到其他无关的结点)
- 然后在底部插入元素

注: 分解后, 不会出现连续的两个4-node

父结点在当前结点的前面, 所以肯定被分裂过了, 所以不可能是4-node 最多只可能是3-node

#### Case 3: 父节点是3-node



### top-down结果:

- 4-node的父亲一定不是4-node
- 到达的叶子结点是2-node或者3-node

## 2-3-4树性能分析：

树的高度（决定了树的性能 $O(h)$ ）：因为完美平衡的性质

- 最坏情况:  $\lg N$  [全部是2-node]
- 最好情况:  $\log_4 N = \frac{1}{2} \lg N$  [全部是4-node]
- 100万个结点高度在[10, 20]
- 10亿个结点高度在[15, 30]

但是2-3-4树实现困难

---

## 红黑树

### 定义：

一颗红黑树是满足下面红黑性质的二叉搜索树（就是一颗BST）

- 每个结点是红色或者黑色的
- 根节点是黑色的
- 叶子结点（null）是黑色的
- 如果一个结点是红色的，那么它的两个子结点都是黑色的（4-node只有一种编码方式）
- 对每个结点，从该结点到其所有后代结点的叶结点的简单路径上，均包含相同数目的黑色结点（黑高平衡，也就是说2-3-4是一颗完美平衡的树）

因为红黑树里面黑色结点对应2-3-4树里面一个结点

而红色结点对应2-3-4树中的一个结点(3-node或者4-node)的一部分

所以红黑树的黑高平衡就是说它对应的2-3-4树的完美平衡

**左倾红黑树：**规定了3node的红色边只能是左倾的，与相应的234树一一对应

1. 用 BST 来表现 2-3-4 树
2. 用“内部的”红色边来表示 3-node 和 4-node
3. 3-node 的红色边是左倾的

注：子结点的颜色表示边的颜色，因为每个结点最多只有一个父结点，即最多只有一条指向父结点的边，刚好可以一一对应

### 查找

查找与普通的BST一样

**旋转：**局部变换

旋转后：

- 仍然满足查找树的性质

- 仍然是黑高平衡的

**插入：**如果底部插入的结点是4-node，则要分裂

- top-down：在查找过程中，遇到4-node就直接分裂：保证当前结点不是4-node
- bottom-up：自底向上在修复结点中分裂4-node，每次插入修复后：树中没有4-node, 变成2-3树，

### 插入过程

- 自顶向下，查找插入位置
- 在底端添加新结点
- 自底向上修复结点

### 自底向上修复步骤：

修复原因：因为插入新结点可能会造成右倾的3-node，或者是连续的红边（不规范的4-node）

1. 遇到右倾的红边：左旋
2. 遇到连续的两条左倾红边：右旋
3. 遇到4-node：分裂

注：分裂后，会向上插入一个新结点，与在底部插入新结点一样，也可能导致上面一层出现不规范的结点，可由上面一层的1 2步骤消除(递归调用返回的过程，在树中就是沿着原来的查找路径向上)

### 分裂4-node：颜色反转

- 局部变换
- 保持了黑高平衡
- 将红边连接到父结点，相当于在父结点中添加了一个结点（插入后也会跟在底部插入一样，产生不规范的结点）

---

### 删除最大值

- 沿着最右边的分支向下查找
- 遇到左倾红边，右旋（保证最大值没有左孩子，如果有，删除了最大值就会丢失左孩子）
- 如果自己只有右孩子，则要保证右孩子不是2-node，是2-node则从兄弟结点中借结点（顺时针方向）
  - 以保证黑高平衡：如果是2-node的话，结点颜色是黑色的，且刚好是我们要删除的结点  
删除了之后这条路径的黑高-1，而其他路径的黑高不变，所以就破坏了黑高平衡的性质
- 在最底端删除最大值
- 自底向上修复结点

### 删除最小值

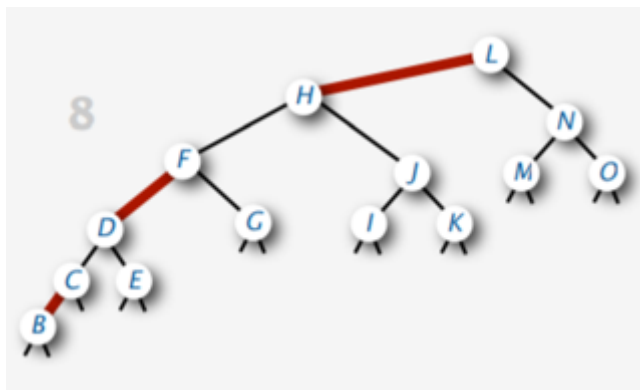


- 沿着最左边的分支向下查找
- 如果有左孩子，则要保证左孩子不是2-node，是2-node则从兄弟结点中借结点（逆时针方向）
  - 以保证黑高平衡
  - 又因为我们用的是2-3树，所以不存在右倾的红边（不规范的3-node或者4-node），所以比删除最大值少一个步骤
- 在最底端删除最小值
- 自底向上修复结点

## 删除

- 比较键的大小，得出查找的方向
- 如果往左边走，则要保证左孩子不是2-node，然后递归删除
- 如果往右边走，则要保证右孩子不是2-node，然后递归删除
- 如果当前结点就是要删除的结点
  - 如果是底端结点：直接删除
  - 如果还有右子树，则先找到右子树的最小值（中序遍历序列后继），用其后继的数据域覆盖掉当前结点的数据域，然后在右子树中递归删除最小结点（跟我们在BST中删除度为2的结点的思路一样）
- 自底向上修复结点

**最坏情况：**红边与黑边间隔出现，此时仍满足黑高平衡，但查找效率最低（即树高最高），最长的路径是最短的路径的2倍



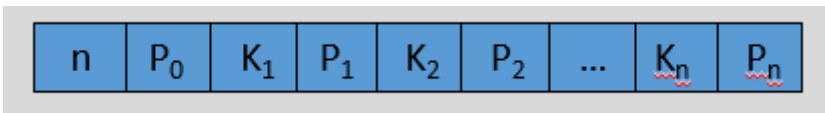
## B树：

### 性质：

B树，又称多路平衡查找树，B树中所有结点的孩子结点数的最大值又称为B树的阶，

通常用m表示。一棵m阶B树或为空树，或为满足如下特性的m叉树：

- 1) 树中每个结点至多有m棵子树（即至多含有m-1个关键字）
- 2) 若根结点不是叶子结点(null)，则至少有两棵子树 (null)
- 3) 除根结点外的所有非叶结点至少有  $\lceil m/2 \rceil$  棵子树（即至少含有  $\lceil m/2 \rceil - 1$  个关键字）（方便合并和分裂）
- 4) 所有非叶子结点的结构如下：



其中,  $K_i$  ( $i=1, 2, \dots, n$ ) 为结点的关键字, 且满足  $K_1 < K_2 < \dots < K_n$ ,  $P_i$  ( $i=1, 2, \dots, n$ ) 为指向子树根结点的指针, 且指针  $P_{i-1}$  所指子树中所有结点的关键字均小于  $K_i$ ,  $P_i$  所指子树中所有结点的关键字均大于  $K_i$ ,  $n(\lceil m/2 \rceil - 1 \leq n \leq m-1)$  为结点中关键字的个数。

5) 所有叶子结点都出现在同一层次上, 并且不带信息。(完美平衡)

## day 3

### 数据库基础

#### 定义

Database: A database is an organized collection of data, stored and accessed electronically

数据库: 数据库是按照数据结构来组织、存储和管理数据的仓库

#### 易混淆术语

- **数据库系统DBS**: 指在计算机系统中引入数据库后的系统, 一般由数据库、数据库管理系统、应用系统、数据库管理员 (DBA) 构成
- **数据库管理系统DBMS**: 是一种操作和管理数据库的大型软件, 用于建立、使用和维护数据库 (MySQL)
- **数据库DB**: 数据库是按照数据结构来组织、存储和管理数据的仓库

#### 分类

- 关系型数据库:
 

不仅存储数据本身, 还存储数据之间的关系, 比如说用户信息和订单信息, 关系型数据库模型把复杂的数据结构归结为简单的二维表 (都支持SQL) (数据和信息不一样 10:0是数据 巴西10:0中国是信息)

Oracle MySQL Microsoft SQL Server
- 非关系型数据库:
 

也被称为NoSQL数据库, 其产生并不是为了否定关系型数据库, 而是作为关系型数据库的一个有效补充 (不能用SQL查询数据)

#### 关系型数据库发展历史

1970 E.F.Codd 提出

1974 SEQUEL 论文发表

1979 第一个商用关系型数据库 Oracle 2 诞生

1995 开源数据库 MySQL 诞生

2008 MySQL被SUN公司收购

2010 SUN 被 Oracle 收购

## 非关系型数据库类型

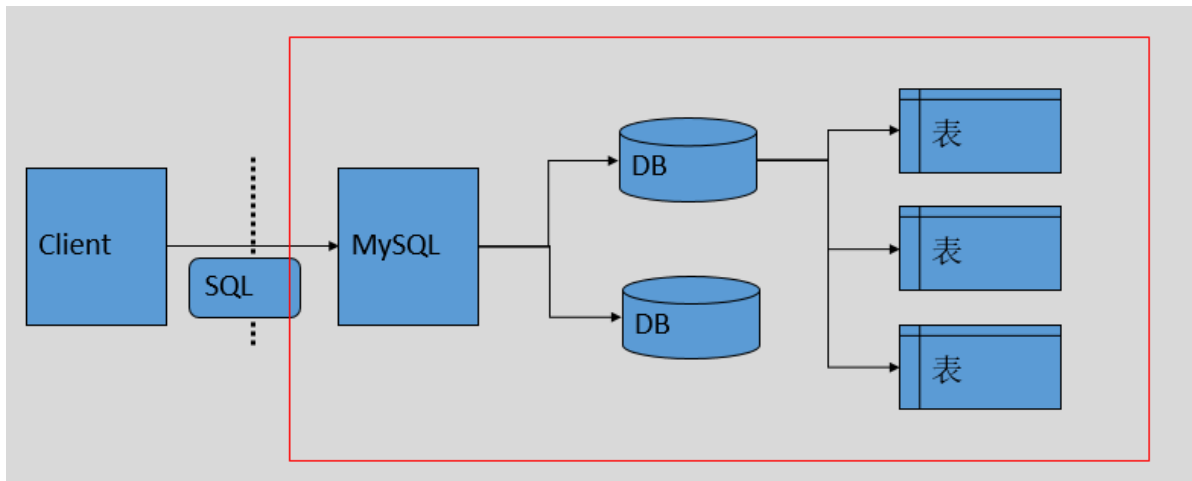
- 文档 MongoDB
- 键值 Redis
- 搜索引擎 Elastic Search
- 列存储（分布式中压缩存储）Cassandra
- 图形数据库 Neo4j

为什么NoSQL最后证实了关系型数据库的重要性呢？：SQL有统一的标准

## 关系型数据库的架构

C/S架构

通信：TCP/IP（实现进程与进程间的通信）

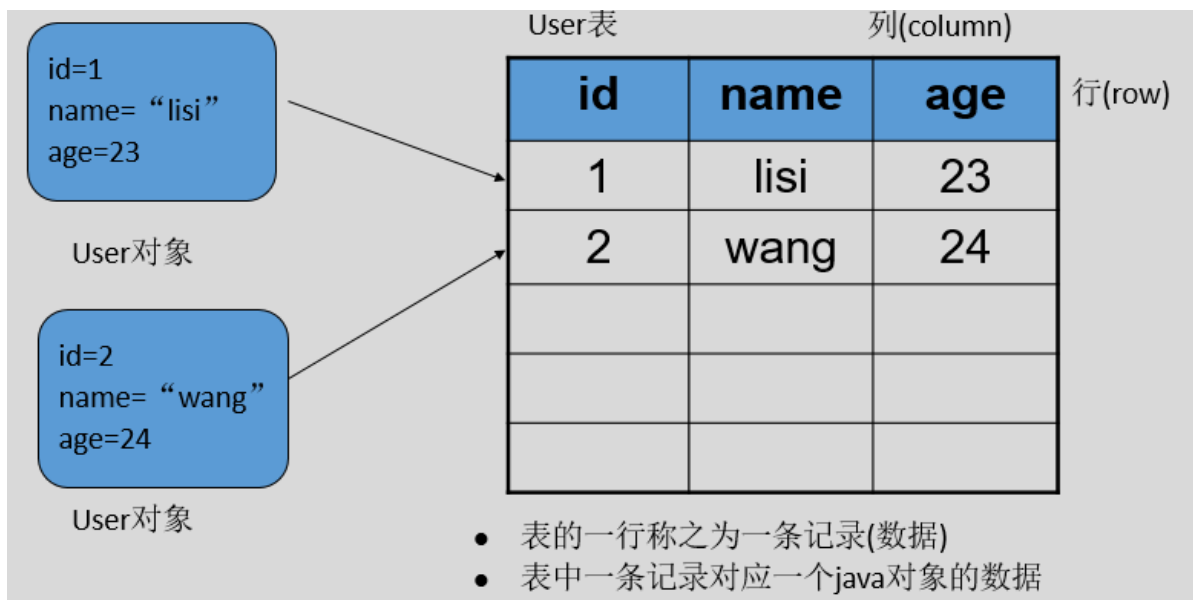


## 数据在表中的形式

表中的一行称为一条记录

一条记录对于一个Java对象

- 对象与行对应
- 属性与列（字段）对应



ORM框架 object relation mapping: 希望通过操作对象的方式 操作数据库

## SQL简介

### 概念

**SQL**: 结构化查询语言structured query language的缩写，专门用来与**关系型数据库**沟通的语言

### 优点

- 是一种通用语言，几乎所有的关系型数据库都支持SQL
- 简单易学
- 可以进行非常复杂的数据库操作
- 半衰期很长：SQL92 SQL99（学了之后很久知识不用更新）

### 说明：

SQL有统一的标准，但DBMS厂商会对SQL进行扩展：SQL方言

### 注意

SQL不区分大小写是关键字不区分大小写

但是表名、列名和值可能区分（依赖于具体的DBMS和系统环境），所以表名、列名最好小写

## SQL组成

**DDL**：数据定义语言

**DML**：数据操作语言（增删改）

**DQL**：数据查询语言（查）

**DCL**：数据控制语言：

used to control access to data stored in a database (Authorization)

**TPL**(transaction processing language): 事务处理语言

## DDL

DDL: data definition language(数据定义语言)

作用: 创建和管理数据库和表的结构

常用关键字: *CREATE ALTER DROP*

### 字符集和校对规则

校对规则: 字符集的比较规则

ci:不区分大小写  $A == a$

bin: 区分大小写  $A < a$

```
show character set; # 查看所有的字符集
show variables like '%char%'; # 查看系统默认字符集
select @@character_set_database; # 查看具体的变量

show collation; # 查看所有的校对集
```

## DDL

### 数据库层面

- 查看数据库

```
show databases; # 查看所有的数据库
show create database mydb1; # 查看数据库的创建语句
show create database mydb2;
show create database mydb3;
```

- 创建数据库

```
# 语法: create database [if not exists] db_name [specifications]
# 推荐使用if not exists (程序健壮性)
# 练习:
# 创建一个名称为mydb1的数据库。
create database mydb1;
create database mydb1;
create database if not exists mydb1;
# 创建一个使用gbk字符集的mydb2数据库。
create database if not exists mydb2 character set gbk;
# 创建一个使用gbk字符集, 并带校对规则(gbk_bin)的mydb3数据库。
create database if not exists mydb3 character set gbk collate gbk_bin;
```

- 删除数据库

```
# 语法: drop database [if exist] db_name;
# 练习: 删除前面创建的mydb3数据库
drop database mydb3;
drop database mydb3;
drop database if exists mydb3;
```

- 修改数据库

```
# 修改字符集或者校对规则
# 语法: alter database db_name [specifications]
# 练习: 把mydb2的字符集修改为utf8
alter database mydb2 character set utf8;
```

## 表层面

- 创建表

```
# 语法:
-- create table tb_name (
--   field_name1 dtype1,
--   field_name2 dtype2,
--   field_name3 dtype3,
-- )[specifications];
# 注意事项: 创建表之前要使用数据库
use mydb1;
# 练习: 创建User表, 包含(id, name, password, birthday)字段
create table t_user(
    id int,
    name varchar(255),
    password varchar(255),
    birthday date
);
```

- 查看表

```
show tables; # 查看当前数据所有表
show tables in world; # 查看数据库中所有表
describe t_user; # 查看表的结构
desc t_user; # describe t_user的缩写
show create table t_user; # 查看表的创建语法
```

- 修改表

```
# a. 添加列
# 语法1: alter table tb_name add [column] 列的定义 [, add column 列的定义]
# 练习: 添加gender列, 类型为varchar(255).
alter table t_user add column gender varchar(255);
# 语法2: alter table tb_name add column 列的定义 after field_name;
# 练习: 在name后面添加balance列, 类型为int
alter table t_user add column balance int after name;
# 语法3: alter table tb_name add column 列的定义 first;
# 练习: 在前面添加a列, 类型为int
alter table t_user add column a int first;
```

```

# 练习：一次性添加b和c列，类型都为int类型。
alter table t_user add column b int, add column c int;

# b. 修改列
# 修改列的名称
# 语法1: alter table tb_name change column col_name new_col_name dtpye;
# 练习：把balance修改成salary
alter table t_user change column balance salary int;
# 修改列的定义
# 语法2: alter table tb_name modify column col_name dtpye;
# 练习：把gender的类型修改为 bit(1)
alter table t_user modify column gender bit(1);

# c. 删除列
# 语法: alter table drop column col_name;
# 练习：删除a列
alter table t_user drop column a;

# 练习：删除b,c列，同时将salary的名字改成balance
alter table t_user drop column b, drop column c, change column salary
balance int;

# d. 修改表的名称
# 语法: rename table tb_name to new_tb_name;
# 练习：将t_user修改成t_student;
rename table t_user to t_student;
# 迁移表
# 练习：将t_student迁移到mydb2;
rename table t_student to mydb2.t_student;
show tables;
show tables in mydb2;
# 练习：将mydb2中的t_student表迁移到mydb1，并将表的名字修改为t_user;
rename table mydb2.t_student to t_user;

# e. 修改表的字符集和校对规则
# 语法: alter table tb_name character set charset_naem collate
collation_name;
# 练习：把t_user的字符集修改成utf8
alter table t_user character set utf8;
show create table t_user;

```

- 删除表

```

# 语法: drop table tb_name;
# 练习：删除t_user表
drop table t_user;
show tables;

```

## day 4

字段的数据类型：数值类型 + 字符串类型 + 日期类型

### 数值类型

## 整数类型

数据类型	占用字节	表示范围
TINYINT	1	256
SMALLINT	2	65536
MEDIUMINT	3	...
INT	4	约43亿
BIGINT	8	...

没有赋值的就是null(如果没有说明not null的话)

## 浮点数和定点数类型

类型名称	占用字节
FLOAT(M, D)	4
DOUBLE(M, D)	8
DECIMAL(M, D)最好用这个	M + 2(2用于记录M和D)

M:精度, 整数位 + 小数位

D:标度, 小数位

## 日期类型

类型名称	日期格式	占用字节
YEAR	YYYY (2018)	1
TIME	HH:MM:SS (10:20:00)	3
DATE	YYYY-MM-DD (2018-7-23)	3
DATETIME	YYYY-MM-DD HH:MM:SS	8
TIMESTAMP	YYYY-MM-DD HH:MM:SS	4

timestamp(时间戳): 存储自基准时间以来的秒数

### **datetime vs timestamp**

- 系统默认值: datetime:null; timestamp:now()
- 存储的时间与时区是否有关: datetime无关, timestamp有关
- datetime占用8个字节, timestamp占用4个字节

## 字符串类型



类型名称	占用字节	说明
CHAR(M)	M, 1 <= M <= 255	固定长度字符串
VARCHAR(M)	L+1, L <=M, 1 <=M <=255	变长字符串
TINYTEXT	L+1, L < 2^8 (多出来的一个字节表示长度)	非常小的文本字符串
TEXT	L+2, L < 2^16	小的文本字符串
MEDIUMTEXT	L+3, L < 2^24	中等大小的文本字符串
LONGTEXT	L+4, L < 2^32	大的文本字符串
ENUM常用表示性别	1 或者 2个字节，取决于枚举的数目，最大65535个	枚举类型
SET	1,2,3,4或8个字节	集合类型

enum类型的默认值：null / 列表的第一个元素（声明为not null时）

## 二进制类型

类型名称	占用字节	说明
BIT(M)	[(m+7)/8]	位字节类型
BINARY(M)	M	固定长度的二进制数据
VARBINARY(M)	L+1	可变长度的二进制数据
TINYBLOB(M)	L+1, L < 2^8	非常小的 BLOB
BLOB(M)	L+2, L < 2^16	小的 BLOB
MEDIUMBLOB(M)	L+3, L < 2^24	中等大小的BLOB
LOB(M)	L+4, L < 2^32	非常大的BLOB

存储二进制数据（字节）：视频 音频

实际中很少使用，一般是用地址

## DML

### 概念

DML: data manipulation language

作用：用于向数据库表中插入、删除、修改数据

常用关键字：INSERT DELETE UPDATE

## insert

insert into tb\_name(..) values(..);

值和列一一对应 没值就插入null或者默认值

- 插入的数据应与字段的数据类型相同
- 数据大小应在字段的规定范围内
- **字符串和日期类型的数据应包含在单引号中**
- 插入空值 null

## update

update tb\_name set col\_name = value where ..

- 如果没有where子句，则更新所有的行

## delete

delete from tb\_name where ..

- 如果不使用where，则删除的是表中的所有行
- delete语句不能删除某一列的值（可用update 置为null），删除的单位是行
- delete语句仅删除记录，不删除表本身（drop table tb\_name删除表本身）

## 数据的导入和导出（都是在cmd内）

数据导入：

- mysql -u \$user -p db\_name < file.sql
- mysql -u root -p  
mysql>use 数据库  
然后使用source命令，后面参数为脚本文件(如这里用到的.sql)  
mysql>source d:/dbname.sql

数据导出：

- mysqldump -u \$user -p db\_name > file.sql

## 常用运算符

- 算术运算符 + - \* / %
- 比较运算符：

运算符	作用	运算符	作用
=	等于	<=>	安全的等于
<> (!=)	不等于	<=	小于等于
>=	大于等于	>	大于
IS NULL	是否为NULL	IS NOT NULL	是否不为NULL
BETWEEN AND	是否在闭区间内	IN	是否在列表内
NOT IN	是否不在列表内	LIKE	通配符匹配

注：

null不能用等号判断，但是可以用安全的等于判断（不推荐使用）

LIKE 通配符规则：

1. '%' 匹配任何数目的字符，甚至包括零字符

2. '\_' 只能匹配一个字符

- 逻辑运算符

NOT(!) AND(&&) OR(||)

- 位操作运算符

& | ~ ^ << >>

## DQL

### [简单查询](#)

DQL: data query language

作用：查询表中的数据

关键字：SELECT

#### 1. 计算表达式和函数的值

```
SELECT 3*2;  
SELECT NOW();  
SELECT CONCAT('ab','cd');
```

#### 2. 查询表中的字段

查询单个字段的值

```
SELECT name FROM heros;
```

查询多个字段的值

```
SELECT name, hp_max, mp_max, role_main FROM heros;
```

还可以用 `*` 指代所有的字段

```
SELECT * FROM heros
```

注意：在生产环境中，尽量不要使用 `*` 通配符。因为查询不必要的数据会降低查询和应用程序的效率！

### 3. 使用WHERE子句过滤数据

where子句后面接逻辑表达式，如果逻辑表达式的结果为真，这条记录就会添加到结果集中，否则就不会添加到结果集

### 4. 用AS给字段起别名

注意：

- ① AS 关键字可以省略，但是不推荐这样做。
- ② AS 关键字不仅仅可以给字段起别名，还可以给表起别名。

### 5. 用DISTINCT去除重复行

注意：

- ① DISTINCT 是对所有查询字段的组合进行去重，也就是说每个字段都相同，才认为两条记录是相同的。
- ② DISTINCT 关键字必须放在所有查询字段的前面。

### 6. 用ORDER BY 排序

- ORDER BY 可以对结果集进行排序。ASC 表示升序，DESC 表示降序，默认情况为升序
- 还可以对多个字段进行排序。即先按照第一个字段排序，当第一个字段相同时，再按照第二个字段排序，依此类推。
- ORDER BY 可以对非选择字段进行排序，也就是说排序的字段不一定要在结果集中。
- 甚至，我们还可以对计算字段进行排序。

### 7. 限制结果集的数量

- LIMIT 可以限制结果集的数量。它有两种使用方式：  
LIMIT offset, nums 和 LIMIT nums OFFSET offset(推荐使用后者)
- 当 OFFSET 为 0 的时候，我们可以将其省略。
- 使用 LIMIT 可以很方便地实现分页查询 `LIMIT rows OFFSET (page - 1) * rows`

### 8. 计算字段

计算字段并不实际存在于数据库表中，它是由表中的其它字段计算而来的。一般我们会给计算字段起一个别名

## 9. 聚合函数

聚合函数是对某个字段的值进行统计的（竖着的），而不是对某条记录进行统计。如果想计算某个学生各科成绩的总分，那么你应该使用计算字段（横着的）。

聚合函数往往是搭配分组使用的。如果没有分组，那么聚合函数统计的是整个结果集的数据；如果分组了，那么聚合函数统计的是结果集中每个组的数据。

SQL 中一共有 5 个聚合函数。分别为 `COUNT()`，`SUM()`，`AVG()`，`MAX()`，`MIN()`。

### a. COUNT()

`COUNT(*)` 可以统计记录数（包括null）

`COUNT()` 作用于某个具体的字段，可以统计这个字段的非 NULL 值的个数。

### b. SUM()

`SUM()` 用于统计某个字段非 NULL 值的和。

### c. AVG()

`AVG()` 用于统计某个字段非 NULL 值的平均值。

### d. MAX()

`MAX()` 用于统计某个字段非 NULL 值的最大值。

### e. MIN()

`MIN()` 用于统计某个字段非 NULL 值的最小值。

### f. DISTINCT

我们还可以对字段中不同的值进行统计。先用 `DISTINCT` 去重，再用聚合函数统计。

## 10. 分组

`GROUP BY` 可以对记录进行分组。

### a. 搭配聚合函数使用

按照主要角色定位进行分组，并统计每一组的英雄数目。

按照次要角色定位进行分组，并统计每一组的英雄数目。

你会发现 NULL 值也会被列为一个分组。在 `heros` 表中有 40 个英雄没有次要角色定位。

### b. GROUP\_CONCAT

如果我们想知道每种角色定位的英雄都有哪些，可以使用 `GROUP_CONCAT()` 函数。

### c. 多字段分组

我们可以对多个字段进行分组。也就是说，每个字段的值都相同的记录为一组。

### d. HAVING 过滤分组

`HAVING` 可以过滤分组。比如：我们想要按照英雄的主要角色定位，次要角色定位进行分组，并且筛选分组中英雄数目大于 5 的组，最后根据每组的英雄数目从高到低进行排序。

`WHERE` 和 `HAVING` 的区别：

`WHERE` 和 `HAVING` 都可以用来过滤数据，但是两者有着很明显的区别。

`WHERE` 是分组前用来过滤记录的，`HAVING` 是分组后用来过滤分组的。

注意：

虽然 DBMS 实现的时候，往往会对分组进行排序。但是如果没有明确的 `ORDER BY` 子句，我们就不应该假定结果集是有序的。

## 11. SELECT的顺序

SELECT 是 RDBMS 中执行最多的操作。我们不仅仅要理解 SELECT 的语法，还要理解它底层执行的原理。

有两个关于 SELECT 的顺序，我们需要记住。

a. 语法中关键字的顺序

```
SELECT ... FROM ... WHERE ... GROUP BY ... HAVING ... ORDER BY ... LIMIT ...
```

b. 语句的执行顺序

不同的 RDBMS，它们 SELECT 语句的执行顺序基本是相同的。

```
FROM --> WHERE --> GROUP BY --> HAVING --> SELECT (选列) --> DISTINCT --> ORDER BY --> LIMIT
```

举个例子：

```
SELECT DISTINCT team_id, team_name, count(*) AS num # 顺序5
FROM player JOIN team ON player.team_id = team.team_id # 顺序1
WHERE height > 1.80 # 顺序2
GROUP BY player.team_id # 顺序3
HAVING num > 2 # 顺序4
ORDER BY num DESC # 顺序6
LIMIT 2; # 顺序7
```

详细解释一下 SQL 的执行顺序：

1. 首先是从 FROM 开始执行的。在这个阶段，如果是多表连接查询，还会经历以下几个步骤：
  - 1.1 通过 CROSS JOIN 求得笛卡尔乘积，得到虚拟表 vt1-1;
  - 1.2 通过 ON 进行连接，在 vt1-1 的基础上进行筛选，得到虚拟表 vt1-2;
  - 1.3 如果是外连接，还会在 vt1-2 的基础上添加外部行，得到虚拟表 vt1-3;
  - 1.4 如果连接的表不止两张，还会重复上面步骤，直到所有表都处理完成。这个过程完成之后，我们就得到了虚拟表 vt-1，也就是我们的原始数据。
2. WHERE 会在 vt-1 的基础上进行筛选，得到虚拟表 vt-2。
3. GROUP BY 会在 vt-2 的基础上进行分组，得到虚拟表 vt-3。
4. HAVING 会在 vt-3 的基础上对分组进行筛选，得到虚拟表 vt-4。
5. SELECT 会在 vt-4 的基础上提取想要的字段，得到虚拟表 vt-5。
6. DISTINCT 会在 vt-5 的基础上，去掉重复行，得到虚拟表 vt-6。
7. ORDER BY 会按照指定的字段对 vt-6 进行排序，得到虚拟表 vt-7。
8. LIMIT 会在 vt-7 的基础上提取指定的记录，得到虚拟表 vt-8。

当然，我们在写 SQL 语句的时候不一定存在所有的关键字，那么相应的阶段就会省略。

### sql语句中on和where的区别

语义上就是不同的

- on后面接的是表与表之间的连接条件（形成一个虚拟表的条件）
- where表示过滤掉中间表的记录的过滤条件

group by 与 distinct 一样 也可以用于去重

如

```
select salary
from salaries
where to_date = '9999-01-01'
group by salary
order by salary desc;
# 等价于
select distinct salary
from salaries
where to_date = '9999-01-01'
order by salary desc;
```

## day 5

---

### 约束

[约束](#)

mysql: key == 索引

### 数据库范式

[数据库范式](#)

#### 数据库有哪些范式

范式越高 数据库中表就越多 冗余度越低

反范式：查询比较多

### 复杂查询

[复杂查询](#)

标量查询（一个值）

矢量：（一列值）

## day 6

---

### 事务

概念

**事务** (transaction) :

构成单一逻辑工作单元的操作集合，即使出现故障，数据库系统也必须保证事务的正确执行，要么执行整个事务，要么属于该事务的操作一个也不执行(转账操作)

事务是一个不可分割的工作单位，事务中的操作要么都发生，要么都不发生

## 事务的基本操作

start transaction/ begin:开启一个事务，标记事务的起点

commit:提交事务，表示事务成功被执行

rollback/ rollback to sp:回滚事务，回滚到事务开始前的状态或者回滚点

savepoint sp:回滚点

release savepoint sp:删除回滚点

set transaction isolation level:设置隔离级别

注意事项:

- MySQL中可以用set autocommit = 0;表示事务的开始（之后一直commit就行）  
MySQL默认开启了自动提交,每条SQL语句都会被当成一个事务执行
- 结束事务的情况只有两种
  - commit:事务成功执行，结束事务
  - 发生故障，结束事务，回到事务开启前的状态
- rollback:不表示发生故障，也是一个数据库操作，属于事务的一部分，要想rollback（永久）生效，必须commit

---

## 事务的性质—ACID

原子性是基础，隔离性是手段，一致性是约束条件，持久性是目的

### 原子性 (atomicity)

事务是一个不可分割的工作单位，事务中的操作要么都发生，要么都不发生（就算操作系统崩溃，计算机停止运行也要保证）

### 一致性: (consistency)

事务的操作必须满足一定的约束（如字段、外键的约束）

没执行成功（不满足一致性）不表示发生故障（断网 断电）

### 隔离性(isolation)

尽管多个事务可能并发执行，但系统保证：对任何一对事务A和B，在A看来，B要么在A开始之前已经完成，要么在A完成之后才开始执行（串行）。每个事务都感觉不到系统中有其他事务在并发地执行

### 持久性 (durability)

一个事务成功完成后，它对数据库的改变是永久的，即便出现系统故障

---



## 并发执行引发的问题

**脏写：**两个事务并发地对同一项数据进行写操作

当多个事务并发写同一数据时，先执行的事务所写的数据会被后写的数据覆盖，会导致**更新丢失**，就好像先提交的事务没有执行一样（非常严重，必须避免）

例子：

假设开始 A 开始为 1000。T1 和 T2 两个事务并发运行。T1 表示给 A 存 100 元，T2 也表示给 A 存 100 元。调度如下：

T1	T2
read(A); A := A + 100;	
	read(A); A := A + 100; write(A); commit;
write(A); commit;	

最终，A 账户有多少钱呢？

## 脏读

如果一个事务 A 向数据库写数据，但该事务**还没提交或终止**，另一个事务 B 就看到了事务 A 写入数据库的数据

（两个事务并发执行，一个事务写数据，另一个事务读数据，且可以读取未提交的数据）

例子：

假设开始 A 和 B 开始都为 1000。T1 和 T2 两个事务并发运行。T1 表示 A 给 B 转账 100 元，T2 表示计算 A 和 B 的资金总额。调度如下：

T1	T2
read(A); A := A - 100; write(A);	
	read(A); read(B); sum := A + B; commit;
read(B); B := B + 100; write(B); commit;	

那么 T2 计算的 A, B 总额是多少呢？

## 不可重复读

一个事务有对同一个数据项的多次读取，但是在某前后两次读取之间，另一个事务更新该数据项，并且提交了。在后一次读取时，感知到了提交的更新（隔离性：感知不到其他事务的执行）

例子：

假设开始 A 为 1000。T1 和 T2 两个事务并发运行。T1 表示 A 中存入 100 元，T2 表示为 A 生成两张不同的报表。调度如下：

T1	T2
	read(A); 生成报表1;
read(A); A := A + 100; write(A); commit;	
	read(A); 生成报表2; commit;

那么你会发现，同一个事务中生成的两张报表是不一致的。

## 幻读

一个事务需要进行前后两次统计，在这两次统计期间，另一个事务插入了新的符合统计条件的记录，并且提交了，导致前后两次统计的数据不一致（平白无故多了一些记录，像产生幻觉一样）

例子：

表 t 中现有三个数据 A B C。T1 表示向表 t 中插入新数据 X，T2 表示读取表 t 的所有数据，并生成两张报表。调度如下：

T1	T2
	read(A); read(B); read(C); 生成报表1;
write(X); commit;	
	read(A); read(B); read(C); read(X); 生成报表2; commit;

那么你会发现，同一个事务中生成的两张报表是不一致的。

**隔离级别：**应对并发执行事务带来的问题

SQL标准中规定的四种隔离级别：

- read uncommitted  
允许读未提交的数据，不存在脏写
- read committed  
只允许读已经提交的数据，不存在脏读
- repeatable read  
只允许读已经提交的数据，且在一个事务两次读取一个数据项期间，其他事务不得更新该数据，不存在不可重复读
- serializable  
**看起来**事务就好像串行执行的一样，一个事务结束后，另一个事务才开始执行，不存在幻读

	脏写	脏读	不可重复读	幻读
read uncommitted	×	√	√	√
read committed	×	×	√	√
repeatable read	×	×	×	√
serializable	×	×	×	×

隔离性依次增高：

read uncommitted -> read committed -> repeatable read -> serializable

```
# 演示隔离级别
select @@tx_isolation; # 查看会话的隔离级别
select @@session.tx_isolation; # 查看会话的隔离级别
select @@global.tx_isolation; # 查看系统的级别
# MySQL默认的隔离级别 Repeatable Read

set transaction isolation level read uncommitted; # 设置下一个事务的隔离级别
set session transaction isolation level read uncommitted; # 设置会话的隔离级别
set global transaction isolation level serializable; # 设置系统的隔离级别
```

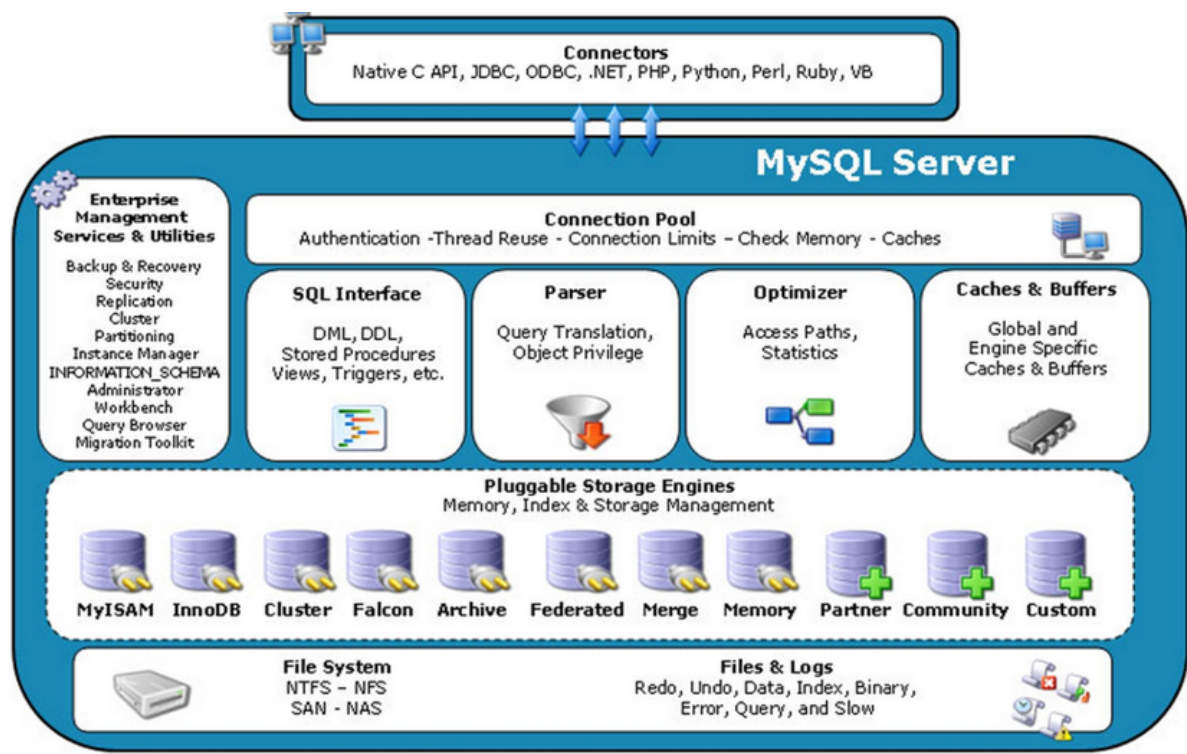
**注意事项：**

- MySQL实现了4种隔离级别, 默认隔离级别是RR
- MySQL RR隔离级别实现比较特殊,能够在大多数情况下避免幻读现象出现.
- Oracle只实现read committed, serializable两种隔离级别,默认隔离级别是read committed.

## Week 10

### day 1

# MySQL逻辑架构



## MySQL分层:

- Server层: 连接器、查询缓存、解析器、优化器、执行器
- 存储引擎层: 负责数据的存储和提取, 插件式架构, 支持多种存储引擎

## MySQL如何工作

1. 连接器: 和客户端建立连接、获取权限 (实际执行时使用)、维持和管理连接 (默认8小时后无操作就关闭连接)
2. 查询缓存: 先查询缓存, 如果缓存命中就返回, 否则就扔给解析器 (默认关闭缓存, 因为只要表更新了, 表的所有缓存都会失效)  
专门缓存查询结果  
缓存类似于Map: 键: sql语句 值: 查询结果
3. 解析器: 词法分析: 分析词的含义, 语法分析: 是否满足SQL语法
4. 优化器: 优化SQL语句, 形成最终的执行方案  
如: 选择索引、连表时: 选择哪个表在前在后, 一般是小表驱动大表, 尽量减少随机查询的次数 (表的数据基本上都是连续的)
5. 执行器: 判断权限 (对表), 没有就报错, 有就调用相应的存储引擎API, 执行语句, 并将结果集返回给客户端

# 存储引擎

任务：负责数据的存储和提取，与文件系统打交道

MySQL的存储引擎是插件式的，不同的引擎支持不同的特性（一般使用InnoDB）

```
# 查看MySQL支持哪些存储引擎
SHOW ENGINES;

# 查看默认存储引擎
SHOW VARIABLES LIKE '%storage_engine%';

# 查看某张表的存储引擎
SELECT ENGINE FROM information_schema.TABLES
WHERE TABLE_SCHEMA='$db'
AND TABLE_NAME='$table';
```

## 各种存储引擎

- **MyISAM**:MySQL 5.5 之前默认的存储引擎
  - 特点：查询速度快、支持表锁、支持全文索引、不支持事务
  - 用其创建的表会生成的文件：frm, mvd, mvi，分别存放表的结构、数据、索引
    - 索引和数据分开存放：非聚集索引
- **InnoDB**:MySQL 5.5 以及以后版本默认的存储引擎。没有特殊应用，Oracle官方推荐使用InnoDB引擎。
  - 特点：支持事务、支持行锁、支持MVCC、支持崩溃回复、支持外键一致性约束（MVCC：多版本控制 实现隔离级别）
  - 用其创建的表会生成的文件：frm, ibd，分别存储表结构和索引与数据
    - 索引和数据存放在一起：聚集索引
- **Memory**:
  - 特点：所有数据都存放在内存中（所以查询速度是最快的），所以数据库重启后数据会丢失、支持表锁、支持Hash（等值查询）和BTree索引、不支持Blob和Text字段
  - 一般用来存放临时表（临时表：一般还是用innodb）
    - 临时表：单个连接中可见，当连接断开的时候临时表消失

---

## 机械磁盘IO原理

磁盘中数据的坐标：柱面号、盘号、块号（扇区）

读取数据的单位：以一个盘块为代价：4KB（也是文件系统中的最小管理单位）

读/写数据的步骤

1. 移动磁头到指定的柱面号（机械移动，最为耗时）
2. 确定读写哪个盘
3. 盘组旋转，指定的块移动到磁头下
4. 传输数据

**结论：**从磁盘上随机读写的速度是很慢很慢的，我们应该尽量少地随机读写磁盘

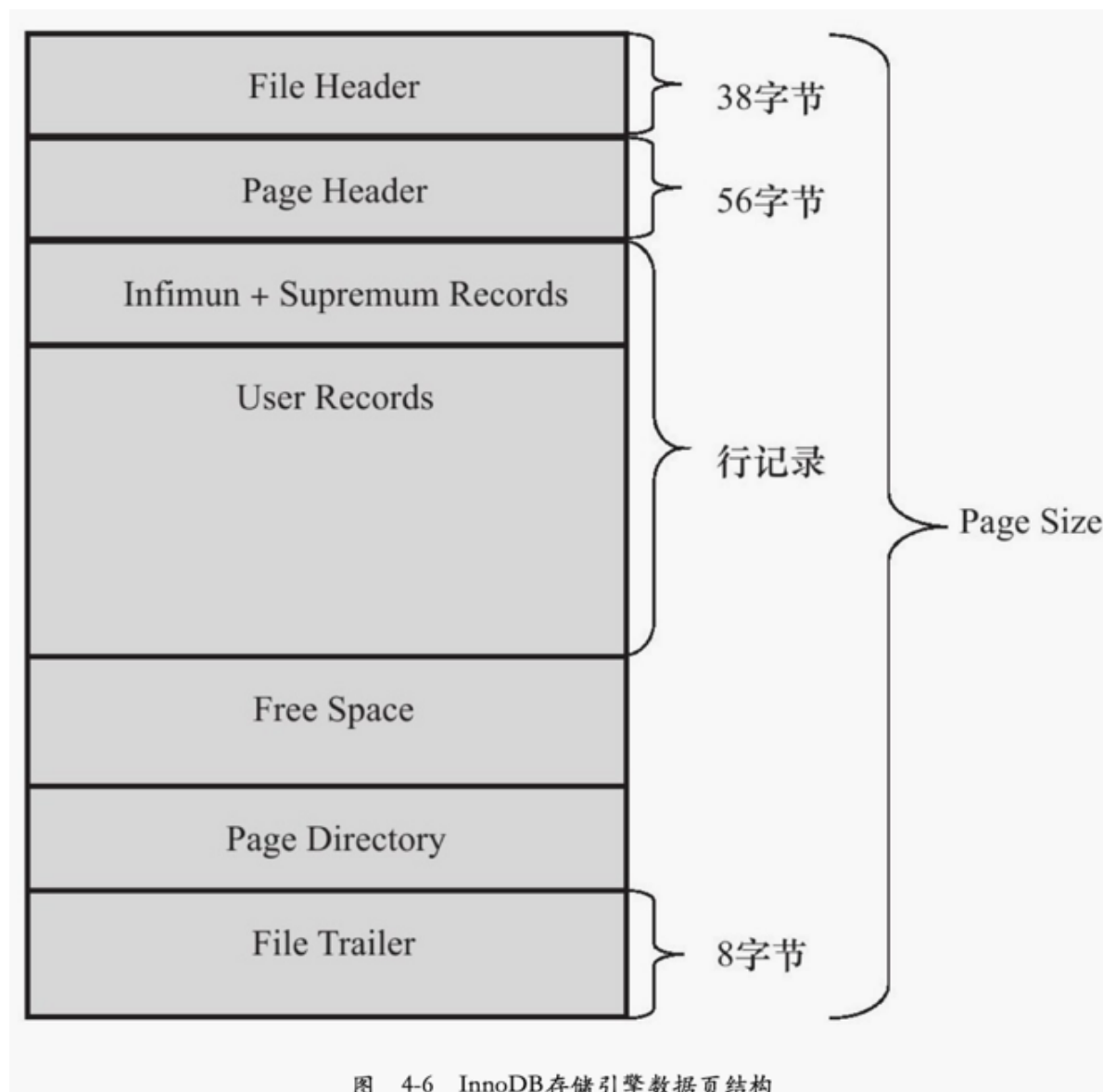
## InnoDB数据页格式

**页：**InnoDB 磁盘管理的最小单位，每次至少会读写一个页的数据，默认大小为16KB。

(可通过参数 `innodb_page_size` 将页的大小设置为 4K、8K 和 16K)

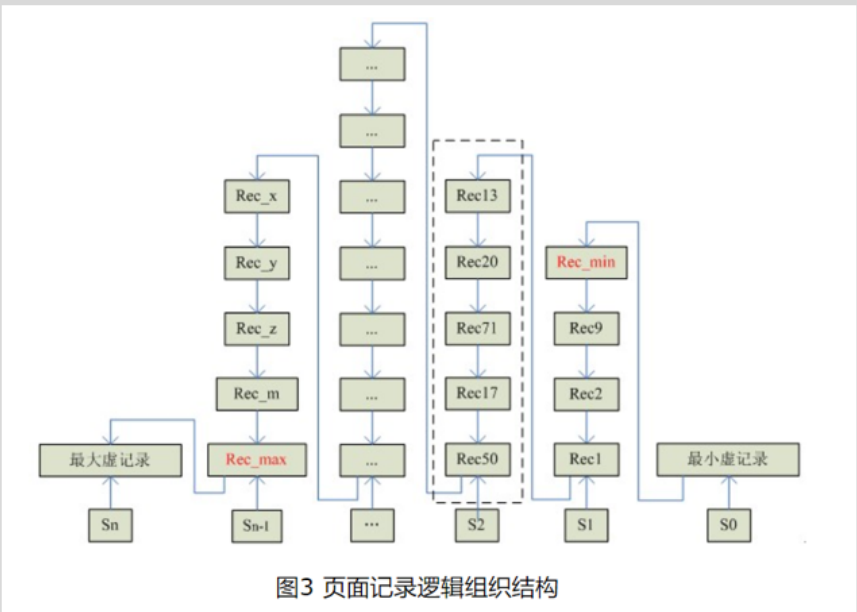
InnoDB存储引擎中的众多页类型中最重要的是数据页：B-tree Node，里面存储了索引和数据的信息（聚集索引的叶子结点）

### 数据页结构



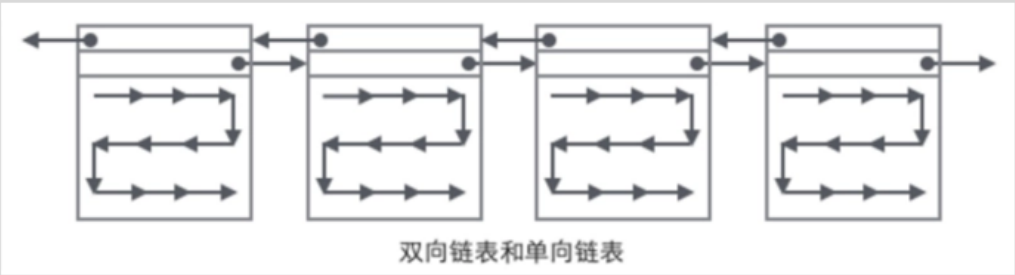
数据页组成部分	作用
file header	存储表空间相关信息
page header	存储描述数据页的信息
Infimum + Supremum Records	两条虚拟记录 就像哨兵、哑结点
user records	用户数据，实际存储的行记录
free space	空闲空间
page directory	页目录，存放记录的相对位置
file trailer	检测页是否完整地写入磁盘

**页面记录逻辑组织结构：**一个页内部在逻辑上是怎么组织的



行记录是用链表形式组织的，最小最大记录相当于两个哨兵。**Page Directory**是一个数组，里面包含很多指向记录的指针(又叫 Slot)，S0指向最小记录的链表, Sn指向最大记录的链表。S1 ~ Sn-1 的每条链的长度范围为 [4, 8]。

**页与页之间是怎么组织的**（所以需要索引提高查询记录的效率，要不然就是单纯的遍历链表）



**File Header** 里面有两个字段：**FIL\_PAGE\_PREV** 和 **FIL\_PAGE\_NEXT** 用来表示上一个页和下一个页，因此，页与页之间是用双链表链接的。页内的记录是由单链表从大到小依次链接的。

# 索引

**索引：**存储引擎用于快速找到记录的一种数据结构，在MySQL中又叫键（key）

作用：提高数据的查询效率，类似于书的目录

## 可以作为索引的数据结构

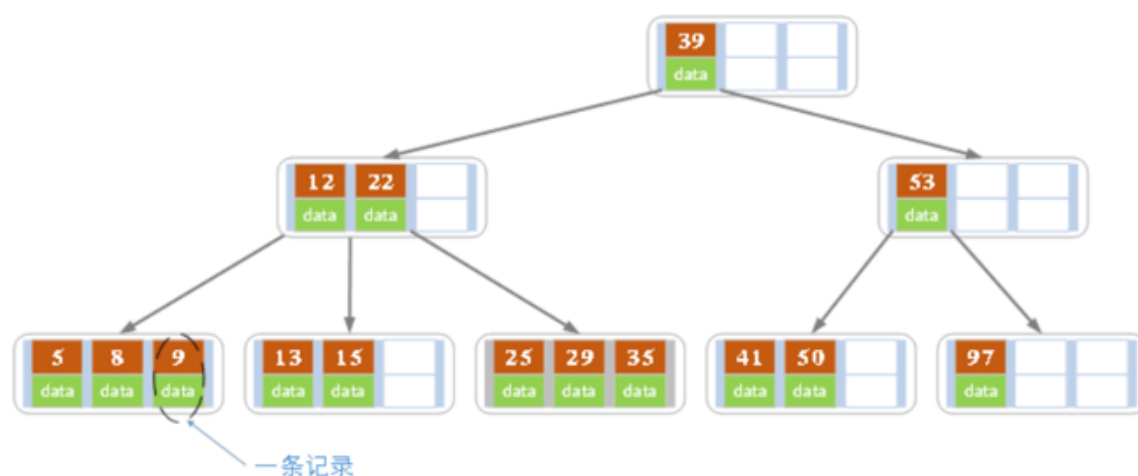
- 有序数组：等值和范围查找快，但增删慢
- 哈希表（hash索引）：增删和等值查找快，但是不方便排序、分组和范围查找
- 平衡二叉树：一个结点存储一个数据页，当页数很多时，二叉树的高度会比较高（几十层，IO几十下）
- B树
- B+树

## B树索引

B 树,它是一颗多路平衡查找树。我们描述一颗 B 树时需要指定它的阶数，阶数表示了一个结点最多有多少个孩子，一般用字母m表示。一颗m阶的B树定义如下：

- 1) 除根结点外，每个结点的度 $[\text{ceil}(m/2), m]$ 。（方便结点的分裂和合并）
- 2) 根结点的度 $[2, m]$
- 3) 每个结点中的关键字都按照从小到大的顺序排列，每个关键字的左子树中的所有关键字都小于它，而右子树中的所有关键字都大于它。（是一颗查找树）
- 4) 所有叶子结点都位于同一层，或者说根结点到每个叶子结点的路径长度都相同。（是一颗完美平衡的树）

索引和数据一起存放



## B树的问题：

- 当一行记录的数据很大时：如1K，此时B树的高度会急速增高，性能下降

这时一个结点最多只能放16条记录（一个结点就是一个数据页，能存储的空间有限）



16 \* 16

16 \* 16 \* 16

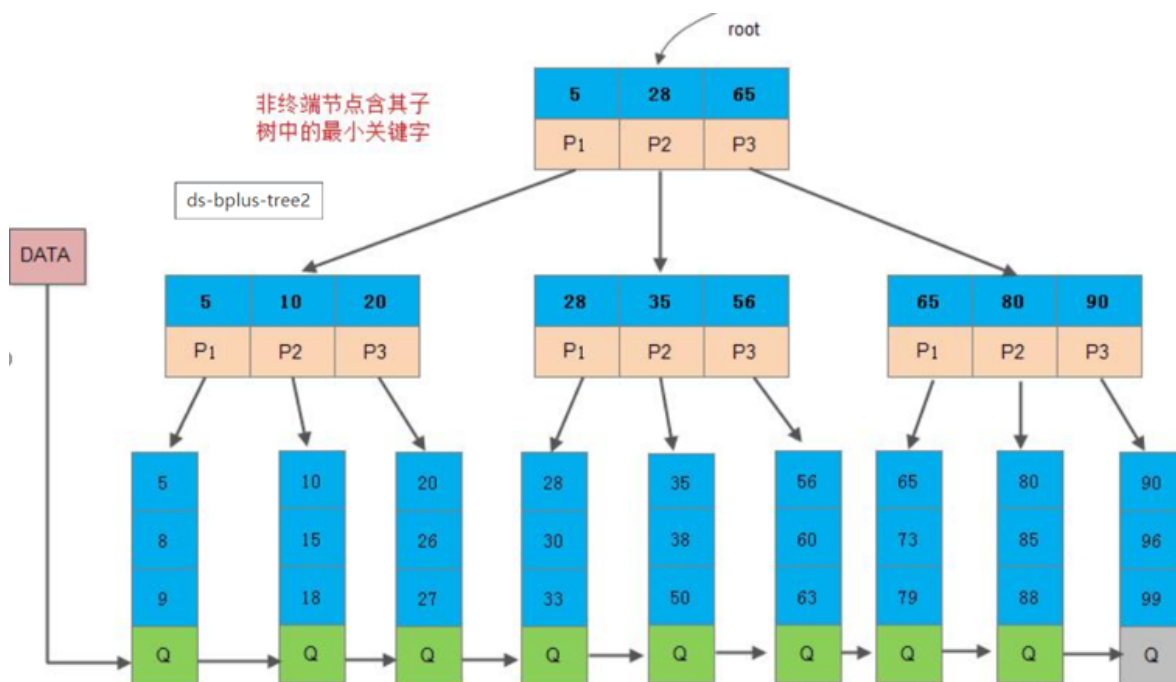
...

- 范围查找不方便：如查找25-50的数据，需要多访问两个结点：（12,22）和（53）

## B+树索引

B+ 树是在B树上做了些改进。一棵 m 阶的 B+ 树定义如下：

- 1) B+树包含2种类型的结点：内部结点(也称索引结点)和叶子结点。根结点即可以是内部结点，也可以是叶子结点。根结点的关键字个数可以只有1个。
- 2) B+树与B树最大的不同是内部结点不保存数据，只保存关键字，所有数据(记录)都保存在叶子结点中。
- 3) m阶B+树表示了内部结点最多有m个关键字。至少有  $\text{ceiling}(m/2)$ 个关键字。
- 4) 内部结点中的key都按照从小到大的顺序排列，叶子结点中的记录也按照key的大小排列。
- 5) 每个叶子结点都存有相邻叶子结点的指针，叶子结点依关键字大小依次链接（遍历全表走的就是这里，横向的指针）



好处：

- 索引结点（内部结点）只存储索引，一行记录的大小不会影响到一个索引结点可以存储的关键字，保证了树的高度不会太高（3、4层）
- 由于叶子节点是由链表按大小依次连接的（在InnoDB 中是双链表）。范围查找（先在B+树中找，然后往右走就行了）的时候，也可以避免过多的IO次数

## 索引的好处和坏处

好处：

提高数据检索的效率，降低数据库的IO成本（没有索引需要像遍历链表一样找）

- 查找
- 排序
- 分组
- 表的连接

坏处：

- 占用额外的空间，有时候索引占用的空间甚至比数据占用的还多
- 索引在提高查询速度的同时也降低了更新表的速度，因为更新数据的同时还要更新索引信息

## InnoDB索引介绍

InnoDB支持以下三种索引：

- **B+树索引**：分为聚集索引和辅助索引
  - 聚集索引：叶子结点存储一行记录的信息（就是数据页）
  - 辅助索引：叶子结点存储关键字和主键
- 全文索引：搜索关键字，类似于搜索引擎
- 哈希索引：是自适应的，引擎会自动给热点数据建立哈希索引：记录->数据页

## 聚集索引

按照每张表的**主键**构建一颗**B+树**，叶子结点存储一行记录的信息，InnoDB中记录只能存放在聚集索引中，所以每张表有且只有一个聚集索引（查询优化器倾向于采用聚集索引）

问题：如果一张表中没有主键，该怎么办呢？

a.找第一个定义的唯一键去构建聚集索引

b.如果没有定义唯一键，又该怎么办呢？InnoDB会提供隐藏的主键，根据隐藏的主键去构建聚集索引。

所以，建议创建表的时候一定要创建主键，且主键最好是占字节较少的数值类型的，因为辅助索引的叶子结点中也要存储主键，主键太大会占用很多空间

## 辅助索引

叶子结点包含自己的键和主键

身份证号不能当主键原因：就是让辅助索引里面主键占的空间小一点

## 创建索引的语法

```
# 创建：
0) create table t (
    id int primary key auto_increment,
    a int,
    b int,
    c int,
    index idx_a_b (a,b)
); # 推荐使用,建表时即指定索引
```

- 1) 创建表的时候指定。会自动给 **primary key** 和 **unique** 创建索引。
- 2) **CREATE [UNIQUE] INDEX** 索引名 **ON** 表名(字段列表);
- 3) **ALTER** 表名 **ADD [UNIQUE] INDEX** 索引名 (字段列表);

#删除:

**DROP INDEX** 索引名 **ON** 表名;

# 查看:

**SHOW INDEX FROM** 表名;

## 回表

当通过辅助索引来查找数据的时候, InnoDB会先通过辅助索引找到主键, 然后通过聚集索引(主键的B+树) 来找到一个完整的行记录

(如果一个辅助索引的高度为3, 聚集索引的高度为3。那么我们需要6次IO操作, 才可以访问最终的数据, 所以应该尽量避免回表)

**查询优化器的作用:** 计算分析系统收集的统计信息, 提供它认为最优的执行计划

**Explain:** 查看执行计划(查询优化器指定的计划) 的信息

语法: explain + select 语句

信息	含义
id	每个 select 子句的标识
select_type	select语句的类型: simple,primary, union, subquery, derived
table	显示的这一行数据是哪个表的
type	ALL, index, range, ref, eq_ref, const, system, null连接类型; (从左到右, 性能越来越好)
possible_keys	可以选择的索引
key	实际选择的索引
key_len	使用索引的长度 (以字节为单位)
ref	与索引比较的字段
rows:	大概要检索的行数
extra	额外信息。

### Extra:

- using filesort: MySQL中无法利用索引完成的排序操作称为“文件排序”, 常见于排序和分组查询。(文件排序不一定在外存)
- using temporary: 表示MySQL需要使用临时表来存储结果集, 常见于排序和分组查询。

using filesort 和 using temporary, 这两项比较耗时, 需要特别小心。

## select\_type

```
# 1. select_type
# simple
use nba;
explain select * from player;
# primary, union
explain
select * from t_girls union select * from t_boys;
# primary, subquery(非关联子查询)
explain
select * from player where height = (
    select max(height) from player
);
# primary, dependent subquery(关联子查询)
explain
select * from player as t1 where height > (
    select avg(height) from player as t2 where t2.team_id = t1.team_id
);
# derived
explain
select player_name, height from player join (
    select team_id, avg(height) as avg_height from player group by team_id
) as temp using(team_id)
where height > avg_height;
```

## type

```
# 2. type: null > system > const > eq_ref > ref > range > index > ALL
# null: 不需要查询表
explain select @@autocommit;
# system: 表中数据最多只有一行
explain select * from (select 'xixi' as name, 16 as age) as temp;
# const: 满足查询条件的结果最多只有一条记录，并且和关键字比较的值是常数。(唯一性索引或者主键索引)
explain select * from player where player_id = 10025;

use index_db;
create table t1(
    id int primary key,
    a int
);
create table t2(
    id int primary key,
    b int
);
insert into t1 values (1,1),(2,2),(3,3),(4,4);
insert into t2 values (1,1),(2,2),(3,3),(4,4);
# eq_ref: 满足查询条件的结果最多只有一条记录，和关键字比较的值不是常数。
explain
select * from t1 where a = (
    select b from t2 where t2.id = t1.id
);
# ref: 可能有多条满足条件的查询结果，关键字进行等值比较（这里是：先在辅助索引的B+树中找，然后再拿着几个主键到聚集索引中找）
```

```

create index idx_a on t1(a);
explain select * from t1 where a = 1;
# range: 范围查找（先在B+树中找，然后再在叶子结点的链表中找）
explain select * from t1 where a between 1 and 2;
# index: 遍历辅助索引的叶子结点
explain select a from t1;
# ALL: 遍历聚集索引的叶子结点(全表扫描)
alter table t1 add column b int;
show index from t1;
explain select * from t1;

```

## day 2

### 联合索引

**联合索引**：对表的多个字段创建索引

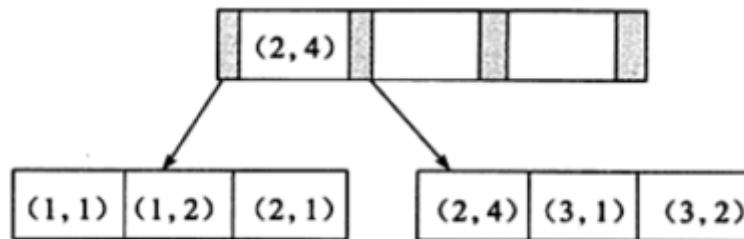


图 5-22 多个键值的 B+ 树

**特别注意**：联合索引(a, b)的键的存放次序：(1, 1), (1, 2), (2, 1), (2, 4), (3, 1), (3, 2)，先对第一个字段a进行排序，如果a相同再对第二个字段b进行排序

#### 好处

- 最左前缀法则
- 键已经排好序了，不需要额外的排序操作

#### 最左前缀法则

对于B+树，可以利用索引的最左前缀来定位记录（用于联合索引或者字符串的模糊查询）

```

# 联合索引 (name, age)
# 右边为实际选择的索引 key = , 即先从哪个B+树中查找
explain select * from t_civil where name = 'xixi' and age = 18; # idx_name_age
explain select * from t_civil where name = 'xixi'; # idx_name_age
explain select * from t_civil where age = 18; # null ALL 遍历全表
explain select * from t_civil where name like 'ab%'; # idx_name_age
explain select * from t_civil where name like '%ab'; # null ALL 遍历全表

```

# 练习：下面哪些 SQL 语句可以使用联合索引？

# 联合索引(a, b)

```
explain select * from t where a = 1 and b = 2; # Y
```

```
explain select * from t where a = 1; # Y
```

```
explain select * from t where b = 2; # N 因为b在联合索引中不是连续存放的
```

```
create table buy_record (  
    id int primary key auto_increment,  
    user_id int not null,  
    buy_date date,  
    money decimal(10,2)  
);
```

```
insert into buy_record(user_id, buy_date) values (1, '2019-01-01'), (2, '2019-01-01'), (3, '2019-01-01'), (1, '2019-02-01'),  
(3, '2019-02-01'), (1, '2019-03-01'), (1, '2019-04-01');
```

```
alter table buy_record add index idx_uid(user_id);
```

```
alter table buy_record add index idx_uid_date(user_id, buy_date);
```

```
show index from buy_record;
```

# 因为这里需要查找所有的字段，而完整的行记录是存储在聚集索引的叶子结点中的

# 所以肯定需要回表

```
explain select * from buy_record where user_id = 2; #idx_uid
```

```
explain select * from buy_record where user_id = 2 order by buy_date desc limit 3; #idx_uid_date (因为找到了user_id, 就能找到一连串有序的buy_date, 所以不用额外进行文件排序)
```

```
explain select * from buy_record force index(idx_uid) where user_id = 2 order by buy_date desc limit 3; # 强行使用user_id作为索引, 会导致文件排序
```

# 对于联合索引 (a, b, c) 下面哪些 SQL 可以利用联合索引直接得到结果，而不需要额外的排序操作？

```
create index idx_a_b_c on t(a,b,c);
```

```
explain select * from t where a = 2 order by b; Y
```

```
explain select * from t where a = 2 and b = 2 order by c; Y
```

```
explain select * from t where a = 2 order by c; N
```

## 覆盖索引

**覆盖索引**（并不是一种索引类型）：从辅助索引中就能得到要查询的信息（行记录的字段），不需要回表（辅助索引中包含自己的键还有主键）

**覆盖索引的好处：**

- 避免回表
- 统计查询会优先选择比较小的覆盖索引

覆盖索引不包含整行记录的信息，所以一般情况下其大小远小于聚集索引，因此可以减少大量的IO操作（指的是辅助索引中所需的数据页比聚集索引的少，也就是链表比较短）

```
# 思考：若主键为 (id1, id2)，辅助索引为 (a, b)。
# 判断下面哪些 SQL 语句需要回表：都不需要，因为辅助索引的叶子结点中存储了a b id1 id2的信息
explain select b from t1 where a = 3; # idx_a_b, ref B+树（最左前缀）
explain select a from t1 where b = 3; # idx_a_b, index 链表
explain select id2, b from t1 where a = 3; # idx_a_b, ref B+树
explain select id1, a from t1 where b = 3; # idx_a_b, index 链表
# all/index和ref就是链表和B+树的查询效率对比
```

```
# 思考：下面 SQL 语句会用到哪个索引，是否会使用文件排序？
# 主键为 (id1, id2)，辅助索引为 (a, b)。count(*)都是要遍历链表即：all或者是index
explain select count(*) from t1 group by a; # idx_a_b, N
explain select count(*) from t1 group by b; # idx_a_b, Y
explain select count(*) from t1 group by id1; # primary, N
explain select count(*) from t1 group by id2; # idx_a_b, Y 特别注意这里，因为辅助索引中也有所有记录的数量，而且辅助索引中的数据页比较少，所以优化器会选择遍历它的链表来统计所有记录的数量
explain select count(*) from t1 group by c; # 主键的链表, Y
```

## 什么时候应该创建索引

- 字段的值是唯一的（MySQL会自动建）（区分度很大）
- 频繁作为where条件的字段
- 频繁作为group by / order by的字段（避免文件排序）
- distinct后接的字段（有序序列中去重时间复杂度比较低）
- 多表连接的时候，作为连接条件的字段

## 什么时候不应该创建索引

- 字段不起定位作用，即不在where, group by, order by, distinct, on中出现的字段
- 表中的记录太少
- 字段的值区分度不高，比如国籍，性别（起不到筛选数据的作用）  
特例：比如女儿国中女人的数目100w, 男人的数目为4个, 需要经常去查询男人。  
这种情形就应该给性别这个字段创建索引。
- 频繁更新的字段，不一定要创建索引

## 什么时候索引会失效（即不会在B+树中找，而是直接遍历链表）

- 索引字段参与了计算
- 对索引字段使用了函数
- or中存在一个条件没有使用索引
- 模糊查询，以%开头
- 用不等于和 is not null作为条件
- 区间查找的范围太大

## 实践策略

- 尽量使用覆盖索引，尽量不要使用select \*（避免回表）
- 尽量使用最左前缀法则（充分利用查找树或者是联合索引的性质）
- 不要在索引列上做运算
- 范围查找尽量不要太大
- 尽量不要使用不等于

---

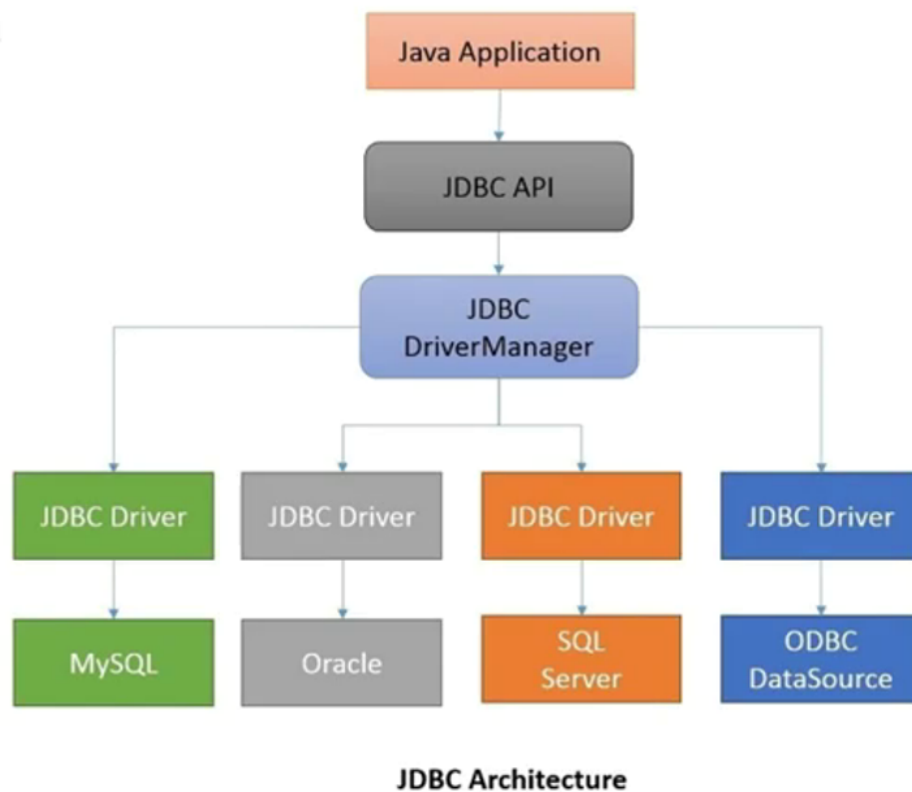
## 单元测试

- 在module下创建一个lib文件夹，把单元测试的jar包放进去
  - 把文件夹add as library，设置为module level
  - 用注解@Test 产生测试方法，并在方法中用Assert类的静态方法判断结果
- 测试方法的要求：public void test() 参数列表为空

---

## JDBC

[JDBC](#)





```
Driver
    |-- Connection connect(String url, Properties info)
DriverManger
    |-- static void registerDriver(Driver driver)
    |-- static Connection getConnection(String url, String user, String
password)
Statement
    |-- boolean execute(String sql)
    |-- int executeUpdate(Strin sql)
    |-- ResultSet executeQuery(String sql)
注意事项：警惕SQL注入问题
```

## day 4

---

### 数据库连接池

[数据库连接池](#)

包装设计模式（装饰者设计模式）IO流

### DbUtils

[DbUtils](#)