



hao3100590的博客文章

作者: hao3100590 <http://hao3100590.iteye.com>

我的博客文章精选

目 录

1. Android开发

1.1 TabActivity中的Tab标签详细设置 5

1.2 实现ListView的条目下自动隐藏显示Button的方法 17

1.3 Android之父深入解析Android 43

1.4 Android下基于XML的 Graphics (转载) 47

1.5 Android事件模型之interceptTouchEvent ,onTouchEvent关系正解 51

1.6 Android图片处理 (Matrix,ColorMatrix) 61

1.7 Android Matrix理论与应用详解 83

1.8 断点续传和下载原理分析 98

1.9 带进度和时间的播放器 130

1.10 手机游戏的优化 147

2. Box2d游戏

2.1 Box2d碰撞筛选 149

3. android游戏开发

3.1 手机游戏的优化 147

4. 算法学习

4.1 1000亿以内素数计数算法 154

4.2 素数的求解逐步改进 174

4.3 关于序列的几个算法 188

4.4 二分查找的递归和非递归 203

- 4.5 求幂的递归和非递归205
- 4.6 主元素算法210
- 4.7 整数的随机置换215
- 4.8 线性查找二维数组220
- 4.9 关于最长递增子序列的实际应用--动态规划225
- 4.10 Josephus问题237
- 4.11 后缀表达式的值240
- 4.12 中缀表达式转换为后缀249
- 4.13 栈的各种实现255
- 4.14 多种队列的实现269
- 4.15 二叉树基本操作大全288
- 4.16 线索二叉树306
- 4.17 构造哈夫曼树314
- 4.18 B-树317
- 4.19 二叉排序树331
- 4.20 平衡二叉树344
- 4.21 B-树实现365
- 4.22 红黑树的插入总结399
- 4.23 set和map的简单实现402
- 4.24 二叉堆的实现426
- 4.25 KMP算法解析433
- 4.26 常见字符串操作大全441

4.27 排序方法总结455

5. 基本概念

5.1 指针与复制构造函数467

5.2 C++动态分配470

5.3 c++中malloc与free477

5.4 结构体482

5.5 动态规划493

5.6 单链表的面向对象实现503

5.7 C++ 迭代器失效的问题506

5.8 单链表的简单实践509

5.9 vector的基本实现（ c++ ）515

1.1 TabActivity中的Tab标签详细设置

发表时间: 2011-03-24 关键字: Android, OS, BBS, thread

参考链接：

<http://www.iteye.com/topic/602737>

这个写的很不错，我是跟着一步步写下来的，不过到最后也遇到了麻烦，就是不能将Tab标签的文字和图片分开，始终是重合的，而且每个具体的代码，还是搞了许久才出来，故而分享之，希望能给大家带来方便，也谢谢下面的高人，呵呵！

<http://www.youmi.net/bbs/thread-102-1-4.html>

这个和上面的代码是一样的，不过代码不全，对于初学者来说，考验的时候来了，完善就是提高的过程，不要怕麻烦！

下面就根据上面的参考自己写的，当然大部分是相同的，不过有详细的注释，完整的代码

如果有什么不明白就留言吧！呵呵

首先结果图：

图1：



图2：

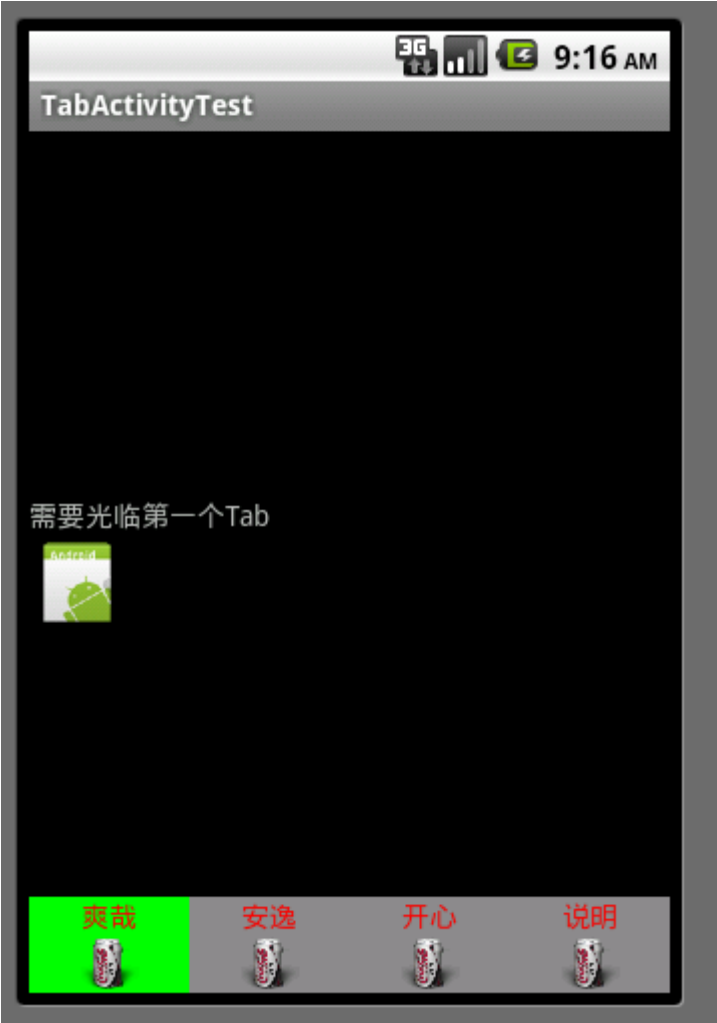


图3：



当然界面没有前面的仁兄好看，我是讲究实用，美化需要自己慢慢做了

呵呵

下面直接代码：

```
package com.woclub.tabactivitytest;

import android.app.TabActivity;
import android.content.res.ColorStateList;
import android.graphics.Color;
import android.os.Bundle;
import android.util.Log;
```

```
import android.view.Gravity;
import android.view.View;
import android.widget.ImageView;
import android.widget.LinearLayout;
import android.widget.TabHost;
import android.widget.TabWidget;
import android.widget.TextView;
import android.widget.TabHost.OnTabChangeListener;

/**
 * 总结：在设置Tab的布局的时候首先需要newTabSpec再在其设置setIndicator(Tab名字，Tab的图标)，
 * 尤其需要注意setContent()，它有三种使用方法setContent(int)它是直接在布局文件中设置其布局，
 * setContent(Intent)可以用Intent指定一个Activity，
 * setContent(TabContentFactory)可以用一个类来指定其布局的方式
 * @author Administrator
 *
 */
public class MainActivity extends TabActivity {

    private static final String Tab1 = "Tab1";
    private static final String Tab2 = "Tab2";
    private static final String Tab3 = "Tab3";
    private static final String Tab4 = "Tab4";

    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        //1得到TabHost对象，正对TabActivity的操作通常都有这个对象完成
        final TabHost tabHost = this.getTabHost();
        final TabWidget tabWidget = tabHost.getTabWidget();

        //2生成一个Intent对象，该对象指向一个Activity，当然现在例子比较简单就没有绑定其他的Activity故而
        //3生成一个TabSpec对象，这个对象代表了一个Tab页
```



```
    TabHost.TabSpec tabSpec = tabHost.newTabSpec(Tab1);
//设置该页的indicator(指示器)设置该Tab页的名字和图标,以及布局
    tabSpec.setIndicator(composeLayout("爽哉", R.drawable.coke))
        .setContent(R.id.view1);
//4将设置好的TabSpec对象添加到TabHost当中
    tabHost.addTab(tabSpec);

//第二个Tab
    tabHost.addTab(tabHost.newTabSpec(Tab2).setIndicator(composeLayout("安逸", R.drawable.c
        .setContent(R.id.view2));

//第三个Tab
    tabHost.addTab(tabHost.newTabSpec(Tab3).setIndicator(composeLayout("开心", R.drawable.c
        .setContent(R.id.view3));

//第四个Tab
    tabHost.addTab(tabHost.newTabSpec(Tab4).setIndicator(composeLayout("说明", R.drawable.c
        .setContent(R.id.view4));

//setContent(TabContentFactory)可以用一个类来指定其布局的方式,前三个都是用的setContent(int)方式
//    CustomLayout custom = new CustomLayout(this);
//    tabHost.addTab(tabHost.newTabSpec(Tab4).setIndicator("Tab4", getResources()
//        .getDrawable(R.drawable.icon))
//        .setContent(custom));
//*****这是对Tab标签本身的设置*****
    int width =45;
    int height =48;
    for(int i = 0; i < tabWidget.getChildCount(); i++)
    {
        //设置高度、宽度,不过宽度由于设置为fill_parent,在此对它没效果
        tabWidget.getChildAt(i).getLayoutParams().height = height;
        tabWidget.getChildAt(i).getLayoutParams().width = width;
        /**
         * 下面是设置Tab的背景,可以是颜色,背景图片等
         */
        View v = tabWidget.getChildAt(i);
        if (tabHost.getCurrentTab() == i) {
            v.setBackgroundColor(Color.GREEN);
        }
    }
}
```

```
        //在这里最好自己设置一个图片作为背景更好
        //v.setBackgroundDrawable(getResources().getDrawable(R.drawable.coke));
    } else {
        v.setBackgroundColor(Color.GRAY);
    }
}

//*****

//设置Tab变换时的监听事件
tabHost.setOnTabChangeListener(new OnTabChangeListener() {

    @Override
    public void onTabChanged(String tabId) {
        // TODO Auto-generated method stub
        //当点击tab选项卡的时候，更改当前的背景
        for(int i = 0; i < tabWidget.getChildCount(); i++)
        {
            View v = tabWidget.getChildAt(i);
            if (tabHost.getCurrentTab() == i) {
                v.setBackgroundColor(Color.GREEN);
            } else {
                //这里最后需要和上面的设置保持一致,也可以用图片作为背景最好
                v.setBackgroundColor(Color.GRAY);
            }
        }
    }

});

}

//*****这是设置TabWidget的布局
/**
 * 这个设置Tab标签本身的布局，需要TextView和ImageView不能重合
 * s:是文本显示的内容
 * i:是ImageView的图片位置
 * 将它设置到setIndicator(composeLayout("开心", R.drawable.coke))中
 */
public View composeLayout(String s, int i){
```

```
        Log.e("Error", "composeLayout");
        LinearLayout layout = new LinearLayout(this);
        layout.setOrientation(LinearLayout.VERTICAL);

        TextView tv = new TextView(this);
        tv.setGravity(Gravity.CENTER);
        tv.setSingleLine(true);
        tv.setText(s);
        tv.setTextColor(Color.RED);
        layout.addView(tv,
                        new LinearLayout.LayoutParams(LinearLayout.LayoutParams.FILL_PARENT, Li

        ImageView iv = new ImageView(this);
        iv.setImageResource(i);
        layout.addView(iv,
                        new LinearLayout.LayoutParams(LinearLayout.LayoutParams.FILL_PARENT, Li

        return layout;
    }

    //#####
}
```

我都有详细的注释，估计大家都能看懂的，有些地方给了提示，扩展的需要就需要自己去完成了

下面是一个两个布局文件

一个是在layout中设置：

```
<?xml version="1.0" encoding="utf-8"?>
<!--
一个典型的标签Activity 是由2 部分构成的 且其id都有规定 即：
* TabWidget 用于展示标签页 id=tabs
* FrameLayout 用于展示隶属于各个标签的具体布局 id=tabcontent
* TabHost 用于整个Tab布局 id=TabHost
还需注意要将Tab显示在最下面就需要这只LinearLayout时用Bottom
```

```
-->
<TabHost xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@android:id/tabhost"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
>
    <LinearLayout
        android:orientation="vertical"
        android:gravity="bottom"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent">
        <FrameLayout
            android:id="@android:id/tabcontent"
            android:layout_width="fill_parent"
            android:layout_height="200dip" >
            <RelativeLayout
                android:id="@+id/view1"
                android:orientation="vertical"
                android:layout_width="fill_parent"
                android:layout_height="fill_parent">
                <TextView
                    android:id="@+id/text1"
                    android:layout_width="wrap_content"
                    android:layout_height="wrap_content"
                    android:text="需要光临第一个Tab"/>
                <ImageView
                    android:id="@+id/image1"
                    android:layout_height="wrap_content"
                    android:layout_below="@id/text1"
                    android:layout_width="wrap_content"
                    android:src="@drawable/icon"
                />
            </RelativeLayout>

            <TextView
                android:id="@+id/view2"
                android:layout_width="fill_parent"
```

```
        android:layout_height="fill_parent"
        android:text="需要光临第二个Tab"/>
    <TextView
        android:id="@+id/view3"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:text="需要光临第三个Tab"/>
    <TextView
        android:id="@+id/view4"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"

    />
</FrameLayout>
<TabWidget
    android:id="@android:id/tabs"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content">
    </TabWidget>
</LinearLayout>
</TabHost>
```

还有一个在类中设置，设置都差不多，在此类中设置只是针对每个Tab页面的设置

```
package com.woclub.tabactivitytest;
import android.app.Activity;
import android.view.Gravity;
import android.view.LayoutInflater;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;
import android.widget.ImageView;
import android.widget.LinearLayout;
import android.widget.RadioButton;
import android.widget.RadioGroup;
```

```
import android.widget.TabHost;
import android.widget.TextView;

/**
 * 此类的功能是设置每个Tab标签的布局方式
 * 使用方法
 * CustomLayout ct = new CustomLayout(this);
 * tHost.addTab(tHost.newTabSpec(Tab4).setIndicator("Tab 4").setContent(ct));
 * @author Administrator
 *
 */
public class CustomLayout implements TabHost.TabContentFactory{

    private Activity myActivity;
    private LayoutInflater layoutHelper;//用于实例化布局
    private LinearLayout layout;
    //构造函数，从外面传递参数Activity
    public CustomLayout(Activity myActivity)
    {
        this.myActivity = myActivity;
        //通过getLayoutInflater从Activity中得到一个实例化的LayoutInflater
        layoutHelper = myActivity.getLayoutInflater();
    }
    /**
     * 根据不同的Tab创建不同的视图
     */
    @Override
    public View createTabContent(String tag) {
        // TODO Auto-generated method stub
        return addCustomView(tag);
    }

    /**
     * 根据Tab的id设置不同Tab的view
     * 这是普通的设置方式，设置每个Tab的布局
     * @param id
     * @return
     */
}
```

```
private View addCustomView(String id)
{
    layout = new LinearLayout(myActivity);
    layout.setOrientation(LinearLayout.HORIZONTAL);

    if(id.equals("Tab1"))
    {
        ImageView iv = new ImageView(myActivity);
        iv.setImageResource(R.drawable.chat);
        //设置layout的布局，将一个ImageView添加到其中，并设置ImageView的布局格式,ac
        layout.addView(iv, new LinearLayout.LayoutParams(LinearLayout.LayoutParams.MATCH_PARENT,
        LinearLayout.LayoutParams.WRAP_CONTENT));
    }
    else if(id.equals("Tab2"))
    {
        //第一个控件，注意每添加一个空间都需要用addView添加到layout中
        EditText edit = new EditText(myActivity);
        layout.addView(edit, new LinearLayout.LayoutParams(LinearLayout.LayoutParams.MATCH_PARENT,
        LinearLayout.LayoutParams.WRAP_CONTENT));
        //第二个控件
        Button button = new Button(myActivity);
        button.setText("确定");
        button.setWidth(100);
        layout.addView(button, new LinearLayout.LayoutParams(LinearLayout.LayoutParams.MATCH_PARENT,
        LinearLayout.LayoutParams.WRAP_CONTENT));
        //第三个控件
        RadioGroup rGroup = new RadioGroup(myActivity);
        rGroup.setOrientation(LinearLayout.HORIZONTAL);
        RadioButton radio1 = new RadioButton(myActivity);
        radio1.setText("Radio A");
        rGroup.addView(radio1);
        RadioButton radio2 = new RadioButton(myActivity);
        radio2.setText("Radio B");
        rGroup.addView(radio2);

        layout.addView(rGroup,
            new LinearLayout.LayoutParams(LinearLayout.LayoutParams.MATCH_PARENT,
            LinearLayout.LayoutParams.WRAP_CONTENT));
    }
    else if(id.equals("Tab3"))
    {
    }
```

```
        TextView text = new TextView(myActivity);
        text.setText("the third TextView");
        text.setGravity(Gravity.CENTER);
        layout.addView(text);
    }
    else if(id.equals("Tab4"))
    {
        LinearLayout.LayoutParams param3 =
            new LinearLayout.LayoutParams(LinearLayout.LayoutParams.FILL_PARENT, LinearLayo
//在这里面又引用了布局文件来设置控件
        layout.addView(layoutHelper.inflate(R.layout.hello, null),param3);
    }
    return layout;
}
}
```

好了，该说的都在代码中说明了

希望大家喜欢，做的粗糙，就由大家去改进吧！

呵呵！

欢迎大家的讨论

附件下载:

- TabActivityTest.rar (1.6 MB)
- dl.iteye.com/topics/download/2f6ee026-c88e-33c9-a012-92a7ddd7f693

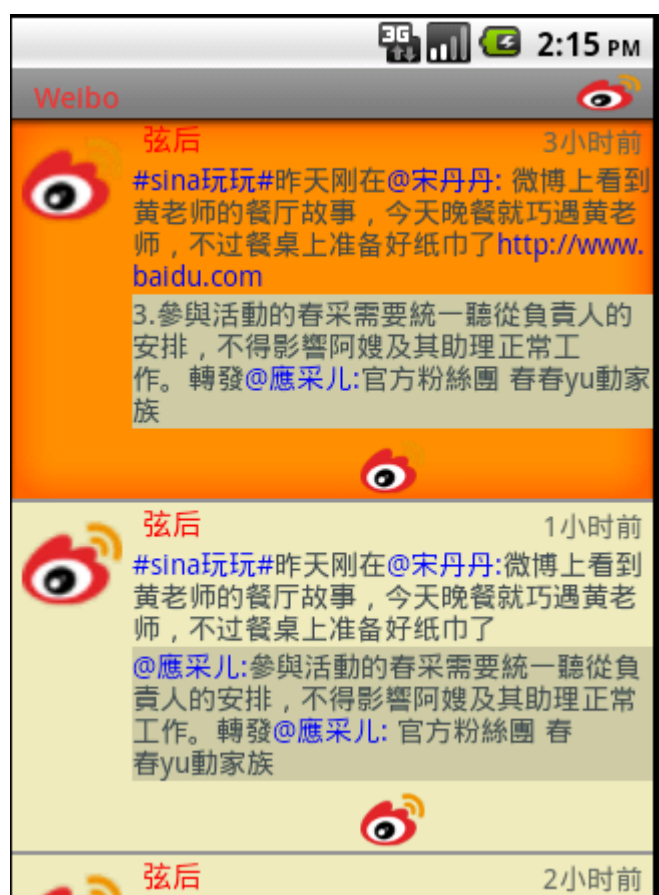
1.2 实现ListView的条目下自动隐藏显示Button的方法

发表时间: 2011-04-17 关键字: Android, XML, MVC, 工作, Blog

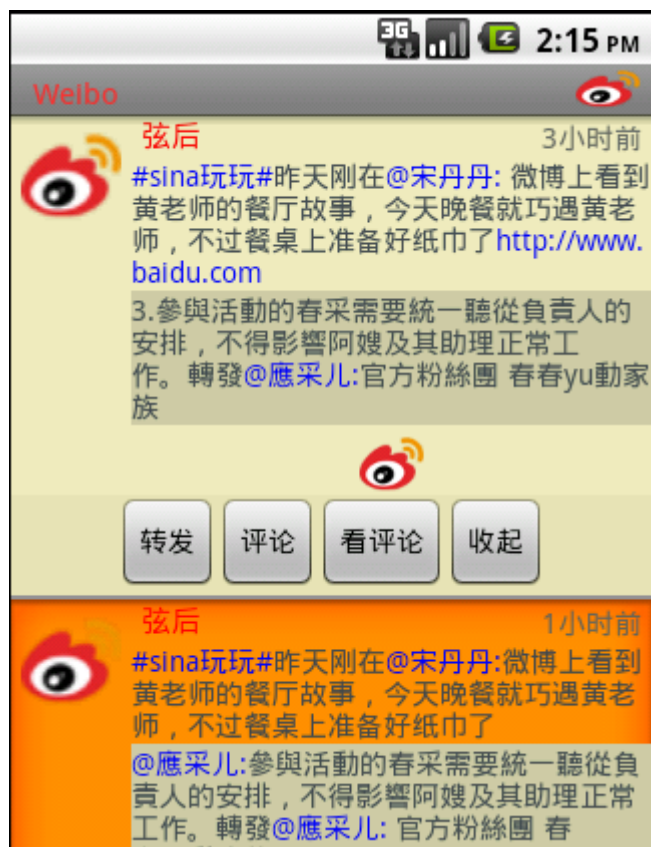
这个想法是我在看了sina微博的塞班客户端的微博显示效果而想移植到Android平台上，因为它的体验很好，而我们做的效果就是要方便，要用户有很好的体验，但是可惜的是在sina官方的Android客户端没有实现这种效果！

废话少说先贴图，看效果：

1. 没有点击ListView之前



2. 点击之后弹出四个Button



3. 当点收起的时候又回到1的状态

当然这个只是刚做出来的效果，界面还没有具体的美化，所以看起来还勉强，呵呵！

追求完美，还需要很多的工作，废话少说，代码+讲解如下：

我之前从来没有做过这种的效果，出发点也不知道，跟大多数的人一样，沉思！

但是之后baidu+google，一遍不行再来，反复的找，但是也没找到解决的办法，目前还没有做这种的效果

莫得办法，自谋出路！ok

第一步：找高人博客（就看关于ListView的介绍）

我们看的不是别人怎么写出来的代码，看的是思想，人家是怎么考虑的，我看过的如下：

<http://blog.csdn.net/flowingflying/archive/2011/03/28/6283942.aspx>

<http://blog.csdn.net/flowingflying/archive/2011/03/29/6286767.aspx>

<http://blog.csdn.net/flowingflying/archive/2011/03/31/6292017.aspx>

这位对ListView的研究很深啊！上面三篇都是，逐步深入，对于初学者绝对是推荐

看了你就爱上他吧！总之一句话！我的神啊！哈哈哈

好了看到这儿，我想你的第一步就是因该马上收藏哥的这篇博客，赶紧看他的吧！他才是哥的灵感，然后再看我的blog

我想这才会理解，我为什么会这么做！ok！just do it！

下面是我看了后的感觉：吸取到的精华就是：

1.**MVC模式**：初学者不知者无罪，但是要知道M V C 三个代表什么，那部分代码因该是那部分的实现。

2.设计自己的**ListView**的**关键**是什么：我告诉你，是ListView和数据的桥梁Adapter，不管是什么BaseAdapter,SimpleAdapter一样，莫得区别了。

3.你需要知道**Adapter**的**每个函数是怎么工作的**，譬如说在调用getView之前你知道它会干什么吗？调用getCount？为什么会调用getCount呢？我说为什么不呢？因为偶去测试了，它就去先去调用了getCount，不会又神来告诉你了！需要你自己去do！不如你去试试吧，把它的返回值设为0，你放心你的getView不灵光了！呵呵

4.在**Adapter**中**最重要的是什么？**想想吧！告诉你是getView，然后它是怎么工作的？调用的机制是什么？怎么调用的？

它是一行行绘制，还是一下子搞完？ok？do it！

好了，还有许多就不一一列举，希望你看了这些也能获得这些知识，一个字悟道吧！就是这么出来的！不知道的可以交流，呵呵

第二步：编码

1.布局文件list.xml

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="horizontal"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
    <LinearLayout
        android:id="@+id/mainlayout"
        android:orientation="horizontal"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content">
        <LinearLayout
            android:id="@+id/layout1"
            android:orientation="horizontal"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content">
            <ImageView
                android:id="@+id/wbIcon"
                android:src="@drawable/sina"
                android:layout_width="50dip"
                android:layout_height="50dip"
                android:layout_marginLeft="5dip"
                android:layout_marginTop="5dip"
                android:layout_marginRight="5dip"
                android:adjustViewBounds="true">
            </ImageView>
        </LinearLayout>
        <LinearLayout
            android:id="@+id/layout2"
            android:orientation="vertical"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_toRightOf="@+id/layout1"
            android:layout_alignParentRight="true">
            <RelativeLayout
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
```

```
        android:layout_alignParentRight="true">
            <TextView
                android:id="@+id/wbUser"
                android:text="username"
                android:textColor="#FF0000"
                android:textSize="15dip"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:paddingLeft="5dip">
            </TextView>
            <TextView
                android:id="@+id/wbTime"
                android:text="updatetime"
                android:textColor="#FF666666"
                android:textSize="14dip"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:paddingRight="5dip"
                android:layout_alignBottom="@+id/wbUser"
                android:layout_alignParentRight="true">
            </TextView>
            <TextView
                android:id="@+id/wbText"
                android:layout_height="wrap_content"
                android:layout_width="fill_parent"
                android:textSize="14dip"
                android:text="the comment fill with it iii\niiiiiiiiiiiiii"
                android:textColor="#424952"
                android:layout_below="@+id/wbUser"
                android:layout_alignParentLeft="true">
            </TextView>
            <LinearLayout
                android:orientation="vertical"
                android:layout_below="@+id/wbText"
                android:layout_width="fill_parent"
                android:layout_height="wrap_content">
                <TextView
```


第一：折中方案，不在每个List的下面显示，而是都显示在最下面，但是可能会覆盖下面的菜单栏，故而舍弃，不过比上面简单的多，贴上xml

list.xml

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
    <!--
    <LinearLayout
        android:id="@+id/buttonlayout"
            android:orientation="horizontal"
            android:paddingLeft="12dip"
            android:background="#EED8AE"
            android:layout_alignParentBottom="true"
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"
            style="@android:style/ButtonBar">
                <Button
                    android:id="@+id/wbTransmit"
                    android:layout_width="wrap_content"
                    android:layout_height="wrap_content"
                    android:layout_weight="1"
                    android:layout_alignParentBottom="true"
                    android:visibility="gone"
                    android:text="转发">
                </Button>
                <Button
                    android:id="@+id/wbComment"
                    android:layout_width="wrap_content"
                    android:layout_height="wrap_content"
                    android:layout_weight="1"
                    android:layout_alignParentBottom="true"
                    android:visibility="gone"
```

```
        android:text="评论">
    </Button>
    <Button
        android:id="@+id/wbRetract"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_weight="1"
        android:layout_alignParentBottom="true"
        android:visibility="gone"
        android:text="收起">
    </Button>
    <Button
        android:id="@+id/wbWatchComment"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_weight="1"
        android:layout_alignParentBottom="true"
        android:visibility="gone"
        android:text="看评论">
    </Button>
</LinearLayout>
-->
<ListView
    android:id="@id/android:list"
    android:divider="#EEB422"
    android:dividerHeight="3dip"
    android:fadeScrollbars="true"
    android:fastScrollEnabled="true"
    android:background="@layout/list_corner"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">
</ListView>
</RelativeLayout>
```


注：在代码中的注释部分即是，他能够做出这种效果，但是在其中有许多的细节需要注意的，最外层需要RelativeLayout，其次list需要在Button之上，不能覆盖list，但是我做出来之后出现了一件诡异的事，list条目乱跳

方案二：直接将button放在list.xml中，作为list的一部分，并且隐藏，当点击list的时候显示list.xml如下：

list.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">
    <LinearLayout
        android:id="@+id/layout01"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content">
        <RelativeLayout
            android:id="@+id/relativelayout"
            android:orientation="vertical"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:paddingLeft="12dip"
            android:paddingBottom="4dip"
            >
            <ImageView
                android:id="@+id/itemImage"
                android:layout_margin="5dip"
                android:src="@drawable/icon"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"/>
            <LinearLayout
                android:id="@+id/layout2"
                android:orientation="vertical"
                android:layout_toRightOf="@+id/itemImage"
                android:layout_width="wrap_content"
```

```
        android:layout_height="wrap_content">
            <TextView
                android:id="@+id/itemTitle"
                android:layout_width="fill_parent"
                android:layout_height="wrap_content"
                android:text="1111111111"
                android:textSize="20dip"/>
            <TextView
                android:id="@+id/itemText"
                android:layout_width="fill_parent"
                android:layout_height="wrap_content"
                android:text="2222222222"
                android:textSize="17dip"/>
        </LinearLayout>
    </RelativeLayout>
</LinearLayout>
<!--
<RelativeLayout
    android:layout_below="@+id/layout01"
    android:orientation="vertical"
    android:paddingLeft="12dip"
    android:background="#EED8AE"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content">
        <Button
            android:id="@+id/wbTransmit"
            android:focusable="false"
            android:focusableInTouchMode="false"
            android:clickable="false"
            android:layout_marginLeft="5dip"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:visibility="gone"
            android:text="转发">
        </Button>
        <Button
            android:id="@+id/wbComment"
```

```
        android:focusable="false"
    android:focusableInTouchMode="false"
    android:clickable="false"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginLeft="28dip"
        android:layout_toRightOf="@+id/wbTransmit"
        android:layout_alignTop="@+id/wbTransmit"
        android:visibility="gone"
        android:text="评论">
</Button>
<Button
        android:id="@+id/wbRetract"
        android:focusable="false"
    android:focusableInTouchMode="false"
    android:clickable="false"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginLeft="28dip"
        android:layout_toRightOf="@+id/wbComment"
        android:layout_alignTop="@+id/wbTransmit"
        android:visibility="gone"
        android:text="收起">
</Button>
<Button
        android:id="@+id/wbWatchComment"
        android:focusable="false"
    android:focusableInTouchMode="false"
    android:clickable="false"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentRight="true"
        android:layout_alignTop="@+id/wbTransmit"
        android:visibility="gone"
        android:text="看评论">
</Button>
```

```
        </RelativeLayout>

-->
</LinearLayout>
```

注：android:focusableInTouchMode="false"

android:focusable="false"

android:clickable="false"这三个是关键，一开始它是不显示的，

当点击list的时候显示，需要在点击事件中去做。但是结果是，点击后效果是出来了，但是每次四个button都出现在list

的下面，最后我分析，是由于getView，我们需要重写这个方法。其关键就是需要理解getView的工作机制。

第三步：重写BaseAdapter函数

这个是最关键的东西，代码如下：

BaseAdapter.java

```
package com.woclub.utils;

import java.util.List;

import android.content.Context;
import android.graphics.Color;
import android.graphics.drawable.Drawable;
import android.text.Spannable;
import android.text.SpannableStringBuilder;
import android.text.style.ForegroundColorSpan;
import android.view.Gravity;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.view.View.OnClickListener;
import android.widget.BaseAdapter;
import android.widget.Button;
import android.widget.ImageView;
```

```
import android.widget.LinearLayout;
import android.widget.TextView;

import com.woclub.beans.WeiboHolder;
import com.woclub.beans.WeiboInfo;
import com.woclub.utils.AsyncImageLoader.ImageCallback;
import com.woclub.weibo.R;

public class MyBaseAdapter extends BaseAdapter{

    private List<WeiboInfo> wbList;
    private LayoutInflater mInflater;
    private AsyncImageLoader asyncImageLoader;
    private Context mContext;
    private WeiboHolder holder = null;
    private int SelectListItem = 0;

    public MyBaseAdapter(Context context, List<WeiboInfo> mData)
    {
        this.mInflater = (LayoutInflater) context.getSystemService(Context.LAYOUT_INFLATER_SERVICE);
        this.wbList = mData;
        mContext = context;
    }

    // private AsyncImageLoader asyncImageLoader;
    @Override
    public int getCount() {
        // TODO Auto-generated method stub
        return wbList.size();
    }

    @Override
    public Object getItem(int position) {
        // TODO Auto-generated method stub
        return wbList.get(position);
    }
}
```

```
}

@Override
public long getItemId(int position) {
    // TODO Auto-generated method stub
    return position;
}

@Override
public View getView(int position, View convertView, ViewGroup parent) {
    // TODO Auto-generated method stub
    if (position < 0 || wbList.size() <= 0)
        return null;
    View row = convertView;
    LinearLayout layout = new LinearLayout(mContext);
    layout.setOrientation(LinearLayout.VERTICAL);

    if(convertView == null)
    {
        //一行行的加载list
        layout.addView(addListView(position, row));
    }
    else
    {
        holder = (WeiboHolder)row.getTag();
        //这是增加四个Button控件
        layout.addView(addListView(position, row));
        int ID = this.getSelectedListItem();
        //如果选择的行是当前正重新刷新(即当前载入的行)时，我们在下面加入四个button
        if(ID==position){
            layout.addView(addButtonView());
        }
    }
    return layout;
}
```

```
/**
 * 设置text中显示的格式
 * @param wbText TextView
 * @param string 开始的字符
 * @param string2 结束字符
 */
private void textHighlight(TextView textView,String start,String end){
    Spannable sp = (Spannable) textView.getText();
    String text = textView.getText().toString();
    int n = 0;
    int s = -1;
    int e = -1;
    while (n < text.length()) {
        s = text.indexOf(start, n);
        if (s != -1) {
            e = text.indexOf(end, s + start.length());
            if (e != -1) {
                e = e + end.length();
            } else {
                e = text.length();
            }
            n = e;
            sp.setSpan(new ForegroundColorSpan(Color.BLUE), s, e,
                Spannable.SPAN_EXCLUSIVE_EXCLUSIVE);
            s = e = -1;
        } else {
            n = text.length();
        }
    }
}

public void setSelectListItem(int position)
{
    this.SelectListItem = position;
}

public int getSelectListItem()
{
}
```

```
        return this.SelectListItem;
    }

    /**
     * 这个函数的作用是加载整个List列表
     * 是一行行的加载整个list
     * @param position行list的位置
     * @return
     */
    private View addListView(int position, View row)
    {
        asyncImageLoader = new AsyncImageLoader();
        row = mInflater.inflate(R.layout.list, null);
        //先取的控件
        holder = new WeiboHolder();
        //下面是获取xml中的实例化控件对象
        holder.wbIcon = (ImageView)row.findViewById(R.id.wbIcon);
        holder.wbUser = (TextView)row.findViewById(R.id.wbUser);
        holder.wbTime = (TextView)row.findViewById(R.id.wbTime);
        holder.wbText = (TextView)row.findViewById(R.id.wbText);
        holder.wbCommentText = (TextView)row.findViewById(R.id.wbCommentText);
        holder.wbImage = (ImageView)row.findViewById(R.id.wbImage);
        //Sets the tag associated with this view.这个tags是包含了view里面的控件，但不一定
        //即相当于这个view重新获得了一个完整的实例化控件集合(实例由上面四步完成)
        WeiboInfo wb = wbList.get(position);
        if(wb != null)
        {
            row.setTag(wb.getId());
            holder.wbUser.setText(wb.getUserName());
            holder.wbTime.setText(wb.getTime());
            // vw.setText("Italic, highlighted, bold.", TextView.BufferType.SPANNABLE);
            // to force it to use Spannable storage so styles can be attached.
            // Or we could specify that in the XML.
            holder.wbText.setText(wb.getText(), TextView.BufferType.SPANNABLE);
            //设置微博内容的显示格式
            textHighlight(holder.wbText, "#", "#");
            textHighlight(holder.wbText, "@", ":");
        }
    }
}
```



```
textHighlight(holder.wbText,"http://"," ");

holder.wbCommentText.setText(wb.getCommentText(),TextView.BufferType.SP
textHighlight(holder.wbCommentText,"#", "#");
textHighlight(holder.wbCommentText,"@", ":");
//textHighlight2(holder.wbCommentText,new char[]{'#'},new char[]{'#'});
//textHighlight2(holder.wbCommentText,new char[]{'@'},new char[]{':', '
//载入头像
Drawable cachedIcon = asyncImageLoader.loadDrawable(wb.getUserIcon(), R

@Override
public void imageLoaded(Drawable imageDrawable, ImageView imageView,
                        String imageUrl) {
    // TODO Auto-generated method stub
    imageView.setImageDrawable(imageDrawable);
}

});
if(cachedIcon == null)
{
    holder.wbIcon.setImageResource(R.drawable.sina);
}
else
{
    holder.wbIcon.setImageDrawable(cachedIcon);
}

//载入图片
if(wb.getImage() != null)
{
    Drawable cachedImage = asyncImageLoader.loadDrawable(wb.getImage(), R

@Override
public void imageLoaded(Drawable imageDrawable, ImageView imageView,
                        String imageUrl) {
    // TODO Auto-generated method stub
    imageView.setImageDrawable(imageDrawable);
}
```

```
        });
        if(cachedImage == null)
        {
            holder.wbImage.setImageResource(R.drawable.sina);
        }
        else
        {
            holder.wbImage.setImageDrawable(cachedImage);
        }
    }

    row.setTag(holder);
return row;
}

/*
 * 这个函数的作用是显示List下面的四个Button
 */
private View addButtonView()
{
    //流式布局放四个button
    LinearLayout layout = new LinearLayout(mContext);
    layout.setOrientation(LinearLayout.HORIZONTAL);
    layout.setPadding(12, 1, 12, 1);
    layout.setBackgroundColor(android.R.color.darker_gray);
    layout.setGravity(Gravity.CENTER);

    final Button bt1 = new Button(mContext);
    bt1.setText("转发");
    bt1.setFocusable(false);
    bt1.setFocusableInTouchMode(false);
    bt1.setClickable(true);
    bt1.setOnClickListener(new OnClickListener() {

        @Override
        public void onClick(View v) {
```

```
        // TODO Auto-generated method stub
        System.out.println("wbTransmit");
    }

    });

    layout.addView(bt1,
        new LinearLayout.LayoutParams(LinearLayout.LayoutParams.WRAP_CONTENT, L

final Button bt2 = new Button(mContext);
bt2.setText("评论");
bt2.setFocusable(false);
bt2.setFocusableInTouchMode(false);
bt2.setClickable(true);
bt2.setOnClickListener(new OnClickListener() {

    @Override
    public void onClick(View v) {
        // TODO Auto-generated method stub
        System.out.println("wbComment");
    }

    });

    layout.addView(bt2,
        new LinearLayout.LayoutParams(LinearLayout.LayoutParams.WRAP_CONTENT, L

final Button bt3 = new Button(mContext);
bt3.setText("看评论");
bt3.setFocusable(false);
bt3.setFocusableInTouchMode(false);
bt3.setClickable(true);
bt3.setOnClickListener(new OnClickListener() {

    @Override
    public void onClick(View v) {
        // TODO Auto-generated method stub
        System.out.println("wbWatchComment");
    }

    });

    layout.addView(bt3,
```

```
        new LinearLayout.LayoutParams(LinearLayout.LayoutParams.WRAP_CONTENT, 1

final Button bt4 = new Button(mContext);
bt4.setText("收起");
bt4.setFocusable(false);
bt4.setFocusableInTouchMode(false);
bt4.setClickable(true);
bt4.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View v) {
        // TODO Auto-generated method stub
        System.out.println("wbRetract");
        bt1.setVisibility(View.GONE);
        bt2.setVisibility(View.GONE);
        bt3.setVisibility(View.GONE);
        bt4.setVisibility(View.GONE);
    }
});
layout.addView(bt4,
        new LinearLayout.LayoutParams(LinearLayout.LayoutParams.WRAP_CONTENT, 1

return layout;
}

}
```

其中关键的地方是在getView里面，这里我只说几点：

1. addListView(position, row)函数的作用是绘制整个list（没有button），这个是在第一次显示的时候去调用
2. addButtonView()函数的作用是添加点击list之后显示button重写绘制的list
3. 图片是从网上获取需要异步加载
4. 整个思想是点击list后重写加载list，当然这不是最好的办法，为了效率还需要改进。

MainActivity.java

```
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import android.app.ListActivity;
import android.graphics.Bitmap;
import android.os.Bundle;
import android.util.Log;
import android.view.View;
import android.view.Window;
import android.widget.AdapterView;
import android.widget.Button;
import android.widget.ImageView;
import android.widget.ListView;
import android.widget.TextView;
import android.widget.AdapterView.OnItemClickListener;

import com.woclub.beans.WeiboHolder;
import com.woclub.beans.WeiboInfo;
import com.woclub.utils.HttpDownloader;
import com.woclub.utils.MyBaseAdapter;
import com.woclub.utils.MySimpleAdapter;

public class MainActivity extends ListActivity {

    private MyBaseAdapter adapter = null;
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        //requestWindowFeature 要在setContentView之前
        //getWindow().setFeatureInit最好在setContentView之后
    }
}
```

```
requestWindowFeature(Window.FEATURE_CUSTOM_TITLE);// 注意顺序
setContentView(R.layout.main);
getWindow().setFeatureInt(Window.FEATURE_CUSTOM_TITLE, R.layout.title);

adapter = new MyBaseAdapter(this, setWeiboList());
setListAdapter(adapter);

//setAdapter(setList());
}

public List<WeiboInfo> setWeiboList()
{
    List<WeiboInfo> list = new ArrayList<WeiboInfo>();
    WeiboInfo infos = new WeiboInfo();
    infos.setId("0");
    infos.setUserId("0");
    infos.setUserName("弦后");
    infos.setTime("3小时前");
    infos.setText("#sina玩玩#昨天刚在@宋丹丹：微博上看到黄老师的餐厅故事，今天晚餐就巧遇黄老师，7");
    infos.setCommentText("3.參與活動的春采需要統一聽從負責人的安排，不得影響阿嫂及其助理正常工作。");
    infos.setUserIcon("http://192.168.1.102:8081/image/1.jpg");
    infos.setImage("http://192.168.1.102:8081/image/2.jpg");
    list.add(infos);

    WeiboInfo infos2 = new WeiboInfo();
    infos2.setId("1");
    infos2.setUserId("1");
    infos2.setUserName("弦后");
    infos2.setTime("1小时前");
    infos2.setText("#sina玩玩#昨天刚在@宋丹丹：微博上看到黄老师的餐厅故事，今天晚餐就巧遇黄老师，7");
    infos2.setCommentText("@應采儿：參與活動的春采需要統一聽從負責人的安排，不得影響阿嫂及其助理正");
    infos2.setUserIcon("http://192.168.1.102:8081/image/2.jpg");
    infos2.setImage("http://192.168.1.102:8081/image/2.jpg");
    list.add(infos2);

    WeiboInfo infos3 = new WeiboInfo();
```

```
        infos3.setId("2");
        infos3.setUserId("2");
        infos3.setUserName("弦后");
        infos3.setTime("2小时前");
        infos3.setText("#sina玩玩#昨天刚在@宋丹丹:微博上看到黄老师的餐厅故事，今天晚餐就巧遇黄老师，2");
        infos3.setCommentText("@應采儿:參與活動的春采需要統一聽從負責人的安排，不得影響阿嫂及其助理正");
        infos3.setUserIcon("http://192.168.1.102:8081/image/2.jpg");
        infos3.setImage("http://192.168.1.102:8081/image/2.jpg");
        list.add(infos3);
        return list;
    }

    @Override
    protected void onListItemClick(ListView l, View v, int position, long id) {
        // TODO Auto-generated method stub
        adapter.setSelectedListItem(position);
        adapter.notifyDataSetChanged();
        super.onListItemClick(l, v, position, id);
    }
}
```

主函数没什么难的，很简单，因为都在Adapter实现了。

下面是MVC中所说的：M

WeiboHolder.java

```
package com.woclub.beans;

import android.widget.Button;
import android.widget.ImageView;
import android.widget.TextView;

public class WeiboHolder {
    public ImageView wbImage;//微博中的图片
    public ImageView wbIcon;//发布人头像
    public TextView wbUser;
    public TextView wbTime;
```

```
public TextView wbText;  
public TextView wbCommentText;  
  
}
```

WeiboInfo.java

```
public class WeiboInfo {  
    private String id;//文章id  
    private String userId;//发布人id  
    private String userName;//发布人名字  
    private String userIcon;//发布人头像  
    private String time;//发布时间  
    private String haveImage;//有图片  
    private String text;//文章内容  
    private String commentText;//评论内容  
  
    public static final String ID = "id";  
    public static final String USERID = "userId";  
    public static final String USERNAME = "userName";  
    public static final String USERICON = "userIcon";  
    public static final String TIME = "time";  
    public static final String TEXT = "text";  
    public static final String COMMENTTEXT = "commentText";  
    public static final String HAVEIMAGE = "haveImage";  
  
    public String getId() {  
        return id;  
    }  
  
    public void setId(String id) {  
        this.id = id;  
    }  
  
    public String getUserId() {  
        return userId;  
    }  
  
    public void setUserId(String userId) {  
        this.userId = userId;  
    }  
}
```



```
}  
public String getUserName() {  
    return userName;  
}  
public void setUserName(String userName) {  
    this.userName = userName;  
}  
public String getUserIcon() {  
    return userIcon;  
}  
public void setUserIcon(String userIcon) {  
    this.userIcon = userIcon;  
}  
public String getTime() {  
    return time;  
}  
public void setTime(String time) {  
    this.time = time;  
}  
public String getImage() {  
    return haveImage;  
}  
public void setImage(String haveImage) {  
    this.haveImage = haveImage;  
}  
public String getText() {  
    return text;  
}  
public void setText(String text) {  
    this.text = text;  
}  
public String getCommentText() {  
    return commentText;  
}  
public void setCommentText(String commentText) {  
    this.commentText = commentText;  
}
```

```
}
```

上面基本上完成了，V就是布局文件了，Adapter属于C。好了，这篇就此结束，希望多提意见，哈哈！

1.3 Android之父深入解析Android

发表时间: 2011-04-22 关键字: Android, Google, 应用服务器, Linux, UP

Android之父深入解析Android

By: 海市蜃楼 | In: [Android开发](#) | [Android新闻](#)

+ 142009

上周末去光谷书城，不经意间看到了程序员杂志2008年合订本，无意中看到这篇经典的文章：Andy Rubin 独家专访，感受颇深，将这篇文章稍微整理了下，在这里与大家一起分享。声明：原文章来源于程序员杂志2008年第一期，本人看到这篇文章后收获很多，将原文稍微修改（原文比较长，去掉无关紧要的内容，保留经典部分）。

整理后的文章如下：

2007 年11 月Google 宣布Android 平台，被众多评论者认为是今年移动领域最具影响力的事件之一。恰在12 月，Android 平台创始人Andy Rubin 访问中国，《程序员》杂志有幸成为采访他的第一家中国媒体，本文就是Andy 与杂志总编孟岩的对话。

.....

孟岩：这么说您也很喜欢iPhone？那么为什么还劳神费力开发Android 这样一个新的手机操作系统？

Andy：没错，iPhone 非常棒，但是它是一家公司的产品，它从里到外的设计都是特定的，只有一个按钮，那么大的屏幕，特有的软件系统。尽管它很棒，但是它就是苹果的，别的制造商用不了。Android 不同，Android对所有人开放，而不只是一家厂商。你可以在摩托、三星、LG等几十个厂商的手机上运行Android。对于开发者来说，这一点意义重大，它意味着你写的手机应用可以无需修改地运行在几十个不同厂商出产的手机上。

孟岩：Android 手机会很贵吗？

Andy：不会。手机硬件越来越便宜。相对来说，软件成本才是居高不下。在整个手机的成本中，软件所占比例越来越大。这跟20多年前发生在PC领域中的情况一模一样。Android是开源软件，能够有效降低软件的成本，从而降低手机的整体价格。

孟岩：如何保证那些手机都能够顺利地运行Android 的呢？各种手机的配置千差万别，难道是像微软那样指定一个硬件规范，要求厂商去遵守吗？

Andy：当然不是。这里面可是有门道的，我们在设计时就努力地让Android变得很容易移植到不同手机设备上。Android也许是目前最具可移植性的手机操作系统。给你讲个实际例子吧。今年感恩节前后，我们的一个工程师打算利用假期尝试把Android移植到一款诺基亚手机上。这是一款与我们的参考设备完全不同的手机，屏幕尺寸、按钮、无线设备统统不同。你猜猜他用了多长时间完成这一工作？

孟岩：少说也得一个星期吧？

Andy：只用了三个多小时，总共改动了4行C语言源代码。

孟岩：太令人震惊了！

Andy：可不是吗，连我也感到震惊。我想这就是开放平台的威力。

孟岩：您提到开放，一个开放的手机软件平台对于手机制造商来说有什么意义呢？

Andy：目前手机操作系统大约占手机成本的20%，而在此之上，手机制造商还必须自己集成若干重要应用软件，如浏览器、短信、图片显示软件等等，这给他们带来了不小的负担。Android提供了一个从操作系统到应用程序的完整软件栈，同时又允许人们定制差异化，以形成自己竞争优势，对于手机制造商来说，其意义不言而喻。开放带来的另一个好处是允许厂商自主解决问题，当你发现Android系统中有bug或者不符合你要求的地方，不必等上18个月，而是可以直接解决。

.....

孟岩：如果Android取得成功的话，Google和你本人肯定无意成为手机软件领域的统治者吗？

Andy：当然不会。开放本身就意味着没有人统治一切。况且如今已经有三十多家公司加入Android联盟，实际上Android不是Google一家的，任何人都可以拥有Android。相信我，没有人能成为Android世界的垄断者。

孟岩：如果是这样的话？Google又能得到什么好处？

Andy：你还记得我刚才说过的，今天的手机软件产业与20多年前的PC软件产业如出一辙。因此我们要注意，如果只有一家公司来提供手机基础软件的话，那么毫无疑问就会再次产生一个巨无霸的垄断者。今天世界上大约12亿PC用户，可是手机用户有30亿之多！这意味着，手机一定会成为人们获取信息的主要设备。如果有一家公司垄断了手机软件市场，它就可以决定人们可以看或不可以看什么样的信息，这是很严重的问题，也是Google不愿意看到的。我们开发Android并且将其开放，就是要防止这种情况发生，让每个人都可以平等自由地访问信息。只要用户能够自由获取信息，Google就可以找到自己的业务模式。

孟岩：好吧，那么告诉我，你为什么要把这个系统命名为Android？

Andy：Android其实是我于2005年1月创办的手机操作系统软件公司的名字，半年后这家公司被Google收购了，不过我们的产品名字继续称为Android。至于其来历嘛，我创办的上一家公司叫做Danger（危险），你说如果上一家公司叫“危险”的话，下一家公司该叫什么名字？总不能叫“完蛋”吧？我是说，高科技公司取名

字尽可以有趣一些，既然Linus Torvalds把自己写的操作系统称为Linux，那么我的名字是Andrew，把这家公司叫做Android有何不可呢？再加上我本身是个机器人迷，所以Android这个名字还是不错的。事实上，在Android之前，我们能已经做出来一款手机，叫做T-Mobile Sidekick。这款产品很成功，好莱坞的很多明星都用它。但是我觉得毕竟它能够触及到的人群还是有限，所以决定做一个手机操作系统，能够运行在各种手机设备上。

孟岩：Android 由Linux+Java 构成。手机操作系统用Linux 我能理解，用Java 也是情理之中。可是Android不是把现有的成熟的Java 拿来直接用，而是从头开始重新实现了一个Java，从虚拟机到相当多的Java 类库。这不是重新发明轮子吗？有这个必要吗？

Andy：对我来说，“Java”这个词意味着四种东西：一种编程语言，一种虚拟机，一个类库，和一个应用程序框架。我们的确重新开发了虚拟机，这是因为要克服现有手机JVM的一些固有缺陷，比如启动时间过长，功能受限。你知道，Java ME为了保证“一次开发，到处运行”，就不得不迁就那些配置非常低端的手机设备，结果导致其功能十分有限。而你看看iPhone之所以这么酷，就是因为它完全不考虑要去兼容什么过时的设备，iPhone的配置完全不亚于一台五年前的PC。我们在设计之初就决定，甩掉不必要的历史包袱，对Android的硬件配置作出一定的要求，从而使得我们可以在比较高的水平上重新设计和优化JVM。此外，我们的确完全重新开发了一个应用程序框架，称之为Android应用框架。这是因为我们对于如何组织手机应用程序有全新的、特别的想法，现有的Java ME完全不能满足我们的需要，所以重新开发一个，这并没有什么大不了的。

孟岩：什么全新的、特别孟岩：什么全新的、特别的想法，能透露一下吗？

Andy：我们希望支持手机应用程序的mash-up。“Mash-up”这个词是从Web 2.0 里偷过来的，你应该知道什么是Web 2.0 mash-up 吧？

孟岩：这我当然知道。比如我从一个网页上抓下一块XHTML 数据，再从另一个Web Services 那里获得一些XML 数据，我就可以把这两块数据mash-up 起来，形成新的XML 数据，并且这一数据可以在此被别人mash-up。不过Web 2.0 中的mash-up 是基于XML数据的，难道Android 中的mash-up 也需要借助XML 来完成吗？

Andy：好问题。在传统的手机开发中，应用程序要么调用操作系统服务，要么通过程序库获得较高级的服务。如果操作系统和程序库都没有提供某项功能，应用程序开发者只好自己实现。在Web上，一个Web应用不仅可以使使用本机上由操作系统提供的服务，还可以以你刚才所说的方式使用其他Web 应用所提供的服务。这样一来，Web 应用程序不仅可以依赖操作系统服务，而且可以彼此相互提供服务。这就是我们想在Android中达成的目标，也是为什么我们要重新开发一个应用框架的根本原因。

孟岩：这个想法是怎么来的？

Andy：你知道Google是一家互联网公司，并且拥有很多世界上最聪明的软件工程师，像mash-up 这样的想法就长在Google的DNA里，所以把这种思想带入手机平台很自然。

孟岩：能详细的讲讲Android 如何实现mash-up 吗？

Andy：Android 的mash-up 中有两个关键概念，一个称为Activity，一个称为Intent。Activity 可以完成某些工作，而Intent 可以表达“要做某事”，一个Activity可以满足若干Intents。我举一个例子好了，比如我在写一个email，打算附加一个图片附件，需要选择图片。我会广播一个Intent，说：“我要选择图片，你们谁能选择图片？”。那么设备中的Android 应用和外部的服务，如Flickr 和Picasa 都可以举手响应说：“我行！” 然后用户就可以选择其中最合适的那个来选择图片。

孟岩：听起来很像GUI 框架中的Signal/Slot ？

Andy：我觉得更好的类比是微软的COM体系。Android 可以把设备内和互联网上的服务都以上述的方式 mash-up 起来。这一能力实在非凡。它使得开发者能够在一个非常高的层次上快速开发高质量的应用。

孟岩：这对于开发者确实有很大的吸引力。

Andy：当然。对于开发者而言，Android 是一个非常有特色的先进的平台，能够放大开发者的能力和效率。同时，Android 的开发语言和环境都是开发者所熟悉的。Java 语言是世界上最流行的语言，而在开发环境方面，我们选择了大家熟悉的Eclipse和IntelliJ。所有的软件、工具和模拟器都是免费的，整个系统稍后还会完全开源，开发者无需做任何痛苦的转型，就可以为Android 开发应用。

.....

读后感

看了以上Android之父对每个问题的解析，我们这些Android应用程序开发人员大脑中的很多疑惑，是不是都消失了？每个Android开发人员都能从这篇文章中受到很大的启发，如果想看原文请到程序员杂志的官网站去下载。在后面的篇幅中我们会抽空，继续与大家分享Android方面的一些经典的文章。

转：<http://www.moandroid.com/?p=1065>

1.4 Android下基于XML的 Graphics (转载)

发表时间: 2011-04-23 关键字: Android, XML, UI

Android下基于XML的 Graphics

以前作图，一般有两种方式，首先是UI把图形设计好，我们直接贴，对于那些简单的图形，如矩形、扇形这样的图形，一般的系统的API会提供这样的接口，但是在Android下，有第三种画图方式，介于二者之间，结合二者的长处，如下代码：

Java 代码

```
<item android:id="@android:id/secondaryProgress">
    <clip>

        <shape>
            <corners android:radius="5dip" />
            <gradient
                android:startColor="#0055ff88"
                android:centerColor="#0055ff00"
                android:centerY="0.75"
                android:endColor="#00320077"
                android:angle="270"
            />
        </shape>
    </clip>
</item>
```

这是一个Progress的style里面的代码，描述的是进度条的为达到的图形，原本以为这是一个图片，后来仔细的跟踪代码，发现居然是 xml，像这种shape corners gradient等等这还是第一次碰到。shape 表示是一个图形，corners表示是有半径为5像素的圆角，然后，gradient表示一个渐变。这样作图简单明了，并且可以做出要求很好的图形，并且节省资源

Java 代码

```
<shape xmlns:android="http://schemas.android.com/apk/res/android" android:shape="rectangle">
    <gradient android:startColor="#FFFF0000" android:endColor="#80FF00FF"
        android:angle="270"/>
    <padding android:left="50dp" android:top="20dp"
        android:right="7dp" android:bottom="7dp" />
    <corners android:radius="8dp" />
</shape>
```


gradient 产生颜色渐变 android:angle 从哪个角度开始变 貌似只有90的整数倍可以

android:shape="rectangle" 默认的也是长方形

Java 代码

```
<shape xmlns:android="http://schemas.android.com/apk/res/android" android:shape="oval">
```

```
<solid android:color="#ff4100ff"/>
```

```
<stroke android:width="2dp" android:color="#ee31ff5e"
```

```
    android:dashWidth="3dp" android:dashGap="2dp" />
```

```
<padding android:left="7dp" android:top="7dp"
```

```
    android:right="7dp" android:bottom="7dp" />
```

```
<corners android:radius="6dp" />
```

```
</shape>
```

#ff4100ff蓝色#ff4100ff绿色

<solid android:color="#ff4100ff"/> 实心的 填充里面

<stroke 描边 采用那样的方式将外形轮廓线画出来

android:dashWidth="3dp" android:dashGap="2dp" 默认值为0

android:width="2dp" android:color="#FF00ff00" 笔的粗细 ,

android:dashWidth="5dp" android:dashGap="5dp" 实现- - 这样的效果 , dashWidth指的是一条小横线的宽度

dashGap 指的是 小横线与小横线的间距。 width="2dp" 不能太宽

shape等特殊xml

1.用 shape 作为背景

```
<shape xmlns:android="http://schemas.android.com/apk/res/android">
```

```
<solid android:color="#f0600000"/>
```

```
<stroke android:width="3dp" color="#ffff8080"/>
```

```
<corners android:radius="3dp" />
```

```
<padding android:left="10dp" android:top="10dp"
```

```
    android:right="10dp" android:bottom="10dp" />
```

```
</shape>
```

一定要注意solid android:color="#f0600000" 是背景色 要用8位 最好不要完全透明不然没有效果啊 这句话本来就不是背景色 的意思

2.类似多选的效果 :

(1) listView.setChoiceMode(ListView.CHOICE_MODE_MULTIPLE);


```
listView.setItemsCanFocus(false);
```

(2) define list item

```
CheckedTextView xmlns:android="http://schemas.android.com/apk/res/android"
```

```
    android:orientation="vertical"
```

```
    android:layout_width="fill_parent"
```

```
    android:layout_height="?android:attr/listPreferredItemHeight"
```

```
    android:textAppearance="?android:attr/textAppearanceLarge"
```

```
    android:gravity="center_vertical"
```

```
    android:paddingLeft="6dip"
```

```
    android:paddingRight="6dip"
```

```
    android:checkMark="?android:attr/listChoiceIndicatorMultiple"
```

```
    android:background="@drawable/txt_view_bg"/>
```

```
(3) define drawable txt_view_bg.xml <item android:drawable="@drawable/selected" android:state_checked="true" /> <item  
android:drawable="@drawable/not_selected" />
```

3.

```
<LinearLayout    android:layout_width = "100dp"    android:layout_height="wrap_content" />
```

```
LinearLayout ll = new LinearLayout(this);parentView.addView(ll, new LinearLayout.LayoutParams(100,  
LayoutParams.WRAP_CONTENT));
```

4. 当设置 TextView setEnabled(false)时 背景颜色你如果用#ffff之类的话可能不会显示 你最好使用 android:textColor这个属性而不是使用color。

```
<TextView    android:text="whatever text you want"    android:layout_width="wrap_content"
```

```
    android:layout_height="wrap_content"    android:textColor="@color/example" />
```

res/color/example.xml):

```
<?xml version="1.0" encoding="utf-8"?><selector xmlns:android="http://schemas.android.com/apk/res/android">    <item  
    android:state_enabled="false" android:color="@color/disabled_color" />    <item android:color="@color/  
normal_color"/> </selector>
```

<http://developer.android.com/intl/zh-CN/reference/android/content/res/ColorStateList.html>

5.

<http://android.amberfog.com/?p=9>

```
<layer-list xmlns:android="http://schemas.android.com/apk/res/android">
```

```
    <item>
```

```
        <shape>
```

```
            <solid android:color="#FFF8F8F8" />
```

```
</shape>
</item>
<item android:top="23px">
    <shape>
        <solid android:color="#FFE7E7E8" />
    </shape>
</item>
</layer-list>
```

You can simple combine several drawables into one using <layer-list> tag.

note: Unfortunately you cannot resize drawables in layer-list. You can only move it.

```
<layer-list xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:drawable="@drawable/shape_below"/>
    <item android:top="10px" android:right="10px" android:drawable="@drawable/shape_cover"/>
</layer-list>
```

include

You can put similar layout elements into separate XML and use <include> tag to use it.

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="64dip"
    android:gravity="center_vertical"
    android:ignoreGravity="@+id/icon">

    <include layout="@layout/track_list_item_common" />;

</RelativeLayout>
```

track_list_item_common.xml

```
<merge xmlns:android="http://schemas.android.com/apk/res/android">

    <ImageView android:id="@+id/icon"
        android:layout_alignParentLeft="true"
        android:layout_centerVertical="true"
        android:layout_marginLeft="4dip"
        android:layout_width="60px"
        android:layout_height="60px"/>

    ...

</merge>
```

1.5 Android事件模型之interceptTouchEvent ,onTouchEvent关系正解

发表时间: 2011-11-19 关键字: interceptTouchEvent, onTouchEvent, 事件模型

参考文档：

<http://blog.csdn.net/liutao5757124/article/details/6097125>

首先，看Android的官方文档正解

onInterceptTouchEvent()与onTouchEvent()的机制:

1. down事件首先会传递到onInterceptTouchEvent()方法
2. 如果该ViewGroup的onInterceptTouchEvent()在接收到down事件处理完成之后return false ,
那么后续的move, up等事件将继续会先传递给该ViewGroup , 之后才和down事件一样传递给最终的目标view的onTouchEvent()处理
3. 如果该ViewGroup的onInterceptTouchEvent()在接收到down事件处理完成之后return true ,
那么后续的move, up等事件将不再传递给onInterceptTouchEvent() , 而是和down事件一样传递给该ViewGroup的onTouchEvent()处理 , 注意 , 目标view将接收不到任何事件。
4. 如果最终需要处理事件的view的onTouchEvent()返回了false , 那么该事件将被传递至其上一层的view的onTouchEvent()处理
5. 如果最终需要处理事件的view 的onTouchEvent()返回了true , 那么后续事件将可以继续传递给该view的onTouchEvent()处理

这是官方文档的说法，要是自己没亲自去写个程序观察哈，基本上没法理解，所以上程序先，然后分析：

布局文件main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<com.hao.LayoutView1 xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical" android:layout_width="fill_parent"
    android:layout_height="fill_parent">
    <com.hao.LayoutView2
        android:orientation="vertical" android:layout_width="fill_parent"
        android:layout_height="fill_parent" android:gravity="center">
        <com.hao.MyTextView
            android:layout_width="wrap_content" android:layout_height="wrap_content"
            android:id="@+id/tv" android:text="AB" android:textSize="40sp"
            android:textStyle="bold" android:background="#FFFFFF"
            android:textColor="#0000FF" />
        </com.hao.LayoutView2>
    </com.hao.LayoutView1>
```

第一层自定义布局LayoutView1.java

```
package com.hao;

import android.content.Context;
import android.util.AttributeSet;
import android.util.Log;
import android.view.MotionEvent;
import android.widget.LinearLayout;

public class LayoutView1 extends LinearLayout {
    private final String TAG = "LayoutView1";
    public LayoutView1(Context context, AttributeSet attrs) {
        super(context, attrs);
```

```
        Log.e(TAG, TAG);
    }

    @Override
    public boolean onInterceptTouchEvent(MotionEvent ev) {
        int action = ev.getAction();
        switch(action){
            case MotionEvent.ACTION_DOWN:
                Log.e(TAG, "onInterceptTouchEvent action:ACTION_DOWN");
                //      return true; 在这就拦截了，后面的就不会得到事件
                break;
            case MotionEvent.ACTION_MOVE:
                Log.e(TAG, "onInterceptTouchEvent action:ACTION_MOVE");
                break;
            case MotionEvent.ACTION_UP:
                Log.e(TAG, "onInterceptTouchEvent action:ACTION_UP");
                break;
            case MotionEvent.ACTION_CANCEL:
                Log.e(TAG, "onInterceptTouchEvent action:ACTION_CANCEL");
                break;
        }
        return false;
    }

    @Override
    public boolean onTouchEvent(MotionEvent ev) {
        int action = ev.getAction();
        switch(action){
            case MotionEvent.ACTION_DOWN:
                Log.e(TAG, "onTouchEvent action:ACTION_DOWN");
                break;
            case MotionEvent.ACTION_MOVE:
                Log.e(TAG, "onTouchEvent action:ACTION_MOVE");
                break;
            case MotionEvent.ACTION_UP:
                Log.e(TAG, "onTouchEvent action:ACTION_UP");
```

```
        break;
    case MotionEvent.ACTION_CANCEL:
        Log.e(TAG,"onTouchEvent action:ACTION_CANCEL");
        break;
    }
    return true;
//    return false;
}

@Override
protected void onLayout(boolean changed, int l, int t, int r, int b) {
    // TODO Auto-generated method stub
    super.onLayout(changed, l, t, r, b);
}

@Override
protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {
    // TODO Auto-generated method stub
    super.onMeasure(widthMeasureSpec, heightMeasureSpec);
}
}
```

第二层布局LayoutView2.java

```
package com.hao;

import android.content.Context;
import android.util.AttributeSet;
import android.util.Log;
import android.view.MotionEvent;
import android.widget.LinearLayout;

public class LayoutView2 extends LinearLayout {

    private final String TAG = "LayoutView2";

    public LayoutView2(Context context, AttributeSet attrs) {
        super(context, attrs);
    }
}
```

```
        Log.e(TAG, TAG);
    }

    @Override
    public boolean onInterceptTouchEvent(MotionEvent ev) {
        int action = ev.getAction();
        switch(action){
            case MotionEvent.ACTION_DOWN:
                Log.e(TAG, "onInterceptTouchEvent action:ACTION_DOWN");
                // return true;
                break;
            case MotionEvent.ACTION_MOVE:
                Log.e(TAG, "onInterceptTouchEvent action:ACTION_MOVE");
                break;
            case MotionEvent.ACTION_UP:
                Log.e(TAG, "onInterceptTouchEvent action:ACTION_UP");
                break;
            case MotionEvent.ACTION_CANCEL:
                Log.e(TAG, "onInterceptTouchEvent action:ACTION_CANCEL");
                break;
        }
        return false;
    }

    @Override
    public boolean onTouchEvent(MotionEvent ev) {
        int action = ev.getAction();
        switch(action){
            case MotionEvent.ACTION_DOWN:
                Log.e(TAG, "onTouchEvent action:ACTION_DOWN");
                break;
            case MotionEvent.ACTION_MOVE:
                Log.e(TAG, "onTouchEvent action:ACTION_MOVE");
                break;
            case MotionEvent.ACTION_UP:
                Log.e(TAG, "onTouchEvent action:ACTION_UP");
```

```
        break;
    case MotionEvent.ACTION_CANCEL:
        Log.e(TAG,"onTouchEvent action:ACTION_CANCEL");
        break;
    }
    //    return true;
    return false;
}
}
```

自定义MyTextView.java

```
package com.hao;

import android.content.Context;
import android.util.AttributeSet;
import android.util.Log;
import android.view.MotionEvent;
import android.view.View;
import android.widget.TextView;

public class MyTextView extends TextView {
    private final String TAG = "MyTextView";
    public MyTextView(Context context, AttributeSet attrs) {
        super(context, attrs);
        Log.e(TAG,TAG);
    }

    @Override
    public boolean onTouchEvent(MotionEvent ev) {
        int action = ev.getAction();
        switch(action){
            case MotionEvent.ACTION_DOWN:
                Log.e(TAG,"onTouchEvent action:ACTION_DOWN");
                break;
            case MotionEvent.ACTION_MOVE:
```



```
        Log.e(TAG, "onTouchEvent action:ACTION_MOVE");
        break;
    case MotionEvent.ACTION_UP:
        Log.e(TAG, "onTouchEvent action:ACTION_UP");
        break;
    case MotionEvent.ACTION_CANCEL:
        Log.e(TAG, "onTouchEvent action:ACTION_CANCEL");
        break;
    }
    return false;
//    return true;
}

public void onClick(View v) {
    Log.e(TAG, "onClick");
}

public boolean onLongClick(View v) {
    Log.e(TAG, "onLongClick");
    return false;
}
}
```

其实代码很简单，就是自定义了View，在View里面都重写了interceptTouchEvent（），和onTouchEvent（），然后测试其返回值，对监听的影响，关键是自己动手，逐个测试，并预测结果，等你能预测结果的时候，也就懂了，需要修改的地方就是interceptTouchEvent 和onTouchEvent的返回值，他们决定了事件监听的流程，下面我画了一张图，如有不足之处欢迎指正，谢谢！

下面是我的正解：

基本的规则是：

- *1.down事件首先会传递到onInterceptTouchEvent()方法

- *

- * 2.如果该ViewGroup的onInterceptTouchEvent()在接收到down事件处理完成之后return false(不拦截)，

* 那么后续的move, up等事件将继续会先传递给该ViewGroup , 之后才和down事件一样传递给最终的目标view的onTouchEvent()处理。

*

* 3.如果该ViewGroup的onInterceptTouchEvent()在接收到down事件处理完成之后return true(拦截, 那么后面的move,up事件不需要在看因为已经拦截了, 我们直接拿去处理onTouchEvent()就可以了), 那么后续的move, up等事件将不再传递给onInterceptTouchEvent(), 而是和down事件一样传递给该ViewGroup的onTouchEvent()处理, 注意, 目标view将接收不到任何事件。下面例子演示:

* 1:LayoutView1(31375): onInterceptTouchEvent action:ACTION_DOWN

* 2:LayoutView2(31375): onInterceptTouchEvent action:ACTION_DOWN

* 3:LayoutView2(31375): onTouchEvent action:ACTION_DOWN

* 4:LayoutView1(31375): onInterceptTouchEvent action:ACTION_MOVE

* 5:LayoutView2(31375): onTouchEvent action:ACTION_MOVE

* 6:LayoutView1(31375): onInterceptTouchEvent action:ACTION_MOVE

* 7:LayoutView2(31375): onTouchEvent action:ACTION_MOVE

* 8:LayoutView1(31375): onInterceptTouchEvent action:ACTION_UP

* 9:LayoutView2(31375): onTouchEvent action:ACTION_UP

* 该设置为:

* onInterceptTouchEvent : LayoutView1为false,LayoutView2为true

* onTouchEvent : LayoutView2为true

* 故而事件在LayoutView2 (onInterceptTouchEvent : 返回true) 时被拦截并处理, 根据上面说法就是LayoutView2后续的MOVE,UP操作都不在经过onInterceptTouchEvent, 直接

* 交给onTouchEvent处理, 结果也的确如此。(见: LayoutView2的3,5,7,9, 第一次是onInterceptTouchEvent处理如1, 以后交给onTouchEvent)

* 而LayoutView1都还是要经过onInterceptTouchEvent(见LayoutView1的4,6,8)

*

* 4.如果最终需要处理事件的view的onTouchEvent()返回了false(没能处理这个事件, 不能丢在传回来让父继续),

* 那么该事件将被传递至其上一层次的view的onTouchEvent()处理。

* 感觉像是一个圈, 然后一直在找一个能处理这个消息的人, 如果找到了就结束, 没找到就循环, 直到回到发出消息的那个人

* 注(对下面): 没有标注的DOWN表示拦截事件onInterceptTouchEvent, 标注了onTouchEvent就是处理事件

* a.如果都没处理(onInterceptTouchEvent返回false):

A(DOWN)-->B(DOWN)-->C(onTouchEvent DOWN)-->B(onTouchEvent DOWN)-->A(onTouchEvent DOWN),没有执行UP事件, 注意有MOVE的话, 在DOWN和UP之间, 下面的都一样。

*b. B处理(B的onInterceptTouchEvent返回true):

A(DOWN)-->B(DOWN)-->B(onTouchEvent)-->A(onTouchEvent UP)-->B(onTouchEvent UP)-->(over)

*

形象说明：如果父亲不拦截消息就传给儿子，如果儿子要这个消息就处理(DOWN)，结束,然后有父亲1--父亲2--儿子以此释放消息(UP)。 然是如果儿子对这个消息置之不理，那这个消息又传回父亲，由父亲来处理即。

下面给出了5中情况（不拦截表示onInterceptTouchEvent返回false）：

```
* 11** 父亲1(LayoutView1不拦截false)---父亲2(LayoutView2不拦截false)--儿子(MyTextView , onTouchEvent
return true)--结束

* 22** 父亲1(LayoutView1不拦截false)---父亲2(LayoutView2不拦截false)--儿子(MyTextView , onTouchEvent
return false)--回传给父亲2(onTouchEvent return true)--结束

* 33** 父亲1(LayoutView1不拦截false)---父亲2(LayoutView2不拦截false)--儿子(MyTextView , onTouchEvent
return false)--回传给父亲2(onTouchEvent return false)--父亲1(onTouchEvent return true)--结束(如果都没处理不在执行UP ACTION)

* 44** 父亲1(LayoutView1拦截true)--父亲1(onTouchEvent return true)--结束      (DOWN--
DOWN(onTouchEvent)--UP(onTouchEvent))

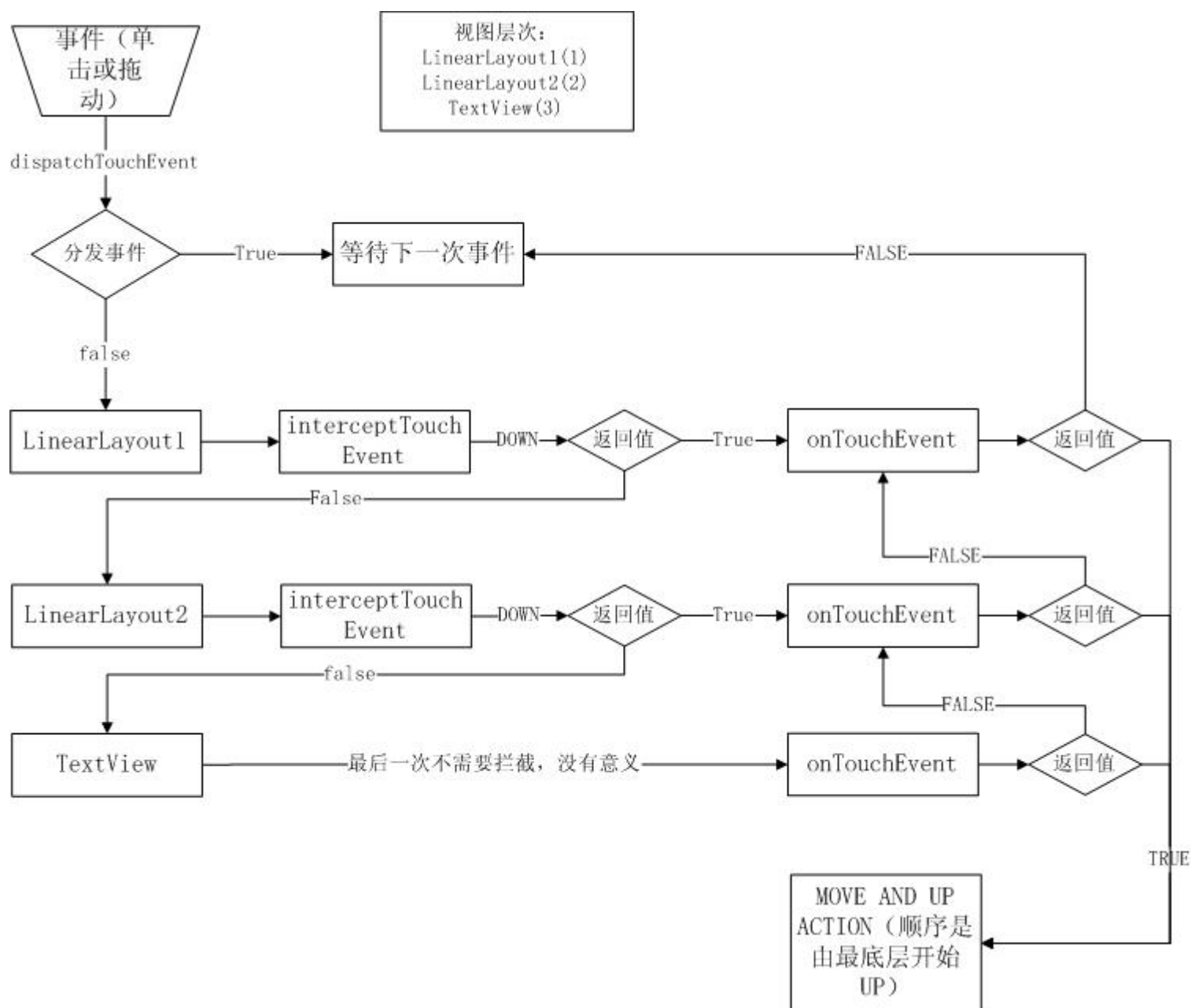
* 55** 父亲1(LayoutView1拦截false)--父亲2(LayoutView2拦截true)--父亲2(onTouchEvent return false)--父亲
1(onTouchEvent return true)--结束    (DOWN1--DOWN2--DOWN(2 onTouchEvent)--DOWN(1 onTouchEvent)--
UP(1 onTouchEvent))(1:父亲2,2：父亲2)

*
* *****

* 5.如果最终需要处理事件的view 的onTouchEvent()返回了true，那么后续事件将可以继续传递给该view的
onTouchEvent()处理。

*/
```

下面给出一张处理的流程图：



下附源代码：

附件下载:

- TouchEventTest.rar (94.3 KB)
- dl.iteye.com/topics/download/aed3307c-42a2-35ba-9a9c-8321cf8e8d20

1.6 Android图片处理 (Matrix,ColorMatrix)

发表时间: 2011-11-20 关键字: Matrix android ColorMatrix

在编程中有时需要对图片做特殊的处理，比如将图片做出黑白的，或者老照片的效果，有时候还要对图片进行变换，以拉伸，扭曲等等。

这些效果在android中有很好的支持，通过颜色矩阵 (ColorMatrix) 和坐标变换矩阵 (Matrix) 可以完美的做出上面的所说的效果。

下面将分别介绍这两个矩阵的用法和相关的函数。

颜色矩阵

android中可以通过颜色矩阵 (ColorMatrix类) 方面的操作颜色，颜色矩阵是一个5x4 的矩阵 (如图1.1)

可以用来方面的修改图片中RGBA各分量的值，颜色矩阵以一维数组的方式存储如下：

[a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t]

他通过RGBA四个通道来直接操作对应颜色，如果会使用Photoshop就会知道有时处理图片通过控制RGBA各颜色通道来做出特殊的效果。

这个矩阵对颜色的作用计算方式如1.3示：

矩阵的运算规则是矩阵A的一行乘以矩阵C的一列作为矩阵R的一行，

C矩阵是图片中包含的ARGB信息，R矩阵是用颜色矩阵应用于C之后的新的颜色分量，运算结果如下：

$$R' = a \cdot R + b \cdot G + c \cdot B + d \cdot A + e;$$

$$G' = f \cdot R + g \cdot G + h \cdot B + i \cdot A + j;$$

$$B' = k \cdot R + l \cdot G + m \cdot B + n \cdot A + o;$$

$$A' = p \cdot R + q \cdot G + r \cdot B + s \cdot A + t;$$

颜色矩阵并不是看上去那么深奥，其实需要使用的参数很少，而且很有规律第一行决定红色第二行决定绿色第三行决定蓝色，第四行决定了透明度，第五列是颜色的偏移量。下面是一个实际中使用的颜色矩阵。

如果把这个矩阵作用于各颜色分量的话， $R=A \cdot C$ ，计算后会发现，各个颜色分量实际上没有任何的改变($R'=R$ $G'=G$ $B'=B$ $A'=A$)。

图1.5所示矩阵计算后会发现红色分量增加100，绿色分量增加100，

这样的效果就是图片偏黄，因为红色和绿色混合后得到黄色，黄色增加了100，图片当然就偏黄了。

改变各颜色分量不仅可以通过修改第5列的颜色偏移量也可如上面矩阵所示将对应的颜色值乘以一个倍数，直接放大。

上图1.6是将绿色分量乘以2变为原来的2倍。相信读者至此已经明白了如何通过颜色矩阵来改变各颜色分量。

下面编写一段代码来，通过调整颜色矩阵来获得不同的颜色效果，JavaCode如下：

MyImage.java

```
package com.hao.view;

import android.content.Context;
import android.graphics.Bitmap;
import android.graphics.BitmapFactory;
import android.graphics.Canvas;
import android.graphics.ColorMatrix;
import android.graphics.ColorMatrixColorFilter;
import android.graphics.Paint;
import android.util.AttributeSet;
import android.util.Log;
import android.view.View;
import com.ho.R;

public class MyImage extends View {
    private Paint mPaint = new Paint(Paint.ANTI_ALIAS_FLAG);
    private Bitmap mBitmap;
    private float[] array = new float[20];

    private float mAngle;

    public MyImage(Context context, AttributeSet attrs) {
        super(context, attrs);

        mBitmap = BitmapFactory.decodeResource(context.getResources(),
                                                R.drawable.sample_4);
        invalidate();
    }

    public void setValues(float[] a) {
        for (int i = 0; i < 20; i++) {
```

```
        array[i] = a[i];
    }

}

@Override
protected void onDraw(Canvas canvas) {
    Paint paint = mPaint;

    paint.setColorFilter(null);
    canvas.drawBitmap(mBitmap, 0, 0, paint);

    ColorMatrix cm = new ColorMatrix();
    // 设置颜色矩阵
    cm.set(array);
    // 颜色滤镜，将颜色矩阵应用于图片
    paint.setColorFilter(new ColorMatrixColorFilter(cm));
    // 绘图
    canvas.drawBitmap(mBitmap, 0, 0, paint);
    Log.i("CMatrix", "----->onDraw");
}
}
```

MatrixTestActivity.java

```
package com.hao;

import com.ho.R;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;

public class MatrixTestActivity extends Activity {
```

```
/** Called when the activity is first created. */
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main_one);
    Button one = (Button) findViewById(R.id.button1);
    one.setOnClickListener(new OnClickListener() {

        @Override
        public void onClick(View v) {
            // TODO Auto-generated method stub
            startActivity(new Intent(MatrixTestActivity.this, ColorMatrixAc

        }

    });

    Button two = (Button) findViewById(R.id.button2);
    two.setOnClickListener(new OnClickListener() {

        @Override
        public void onClick(View v) {
            // TODO Auto-generated method stub
            startActivity(new Intent(MatrixTestActivity.this, MoveMatrixAct

        }

    });
}
```

加上布局文件：

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
```



```
<com.hao.view.MyImage
    android:id="@+id/MyImage"
    android:layout_height="180dip"
    android:layout_width="fill_parent"/>
<TableLayout
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical">
    <TableRow
        android:orientation="horizontal"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content">
        <EditText
            android:id="@+id/indexa0"
            android:text="0"
            android:layout_weight="1"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"/>
        <EditText
            android:id="@+id/indexa1"
            android:text="0"
            android:layout_weight="1"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"/>
        <EditText
            android:id="@+id/indexa2"
            android:text="0"
            android:layout_weight="1"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"/>
        <EditText
            android:id="@+id/indexa3"
            android:text="0"
            android:layout_weight="1"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"/>
        <EditText
```

```
        android:id="@+id/indexa4"
        android:text="0"
        android:layout_weight="1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>
    </TableRow>
    <TableRow
        android:orientation="horizontal"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content">
        <EditText
            android:id="@+id/indexa5"
            android:text="0"
            android:layout_weight="1"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"/>
        <EditText
            android:id="@+id/indexa6"
            android:text="0"
            android:layout_weight="1"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"/>
        <EditText
            android:id="@+id/indexa7"
            android:text="0"
            android:layout_weight="1"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"/>
        <EditText
            android:id="@+id/indexa8"
            android:text="0"
            android:layout_weight="1"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"/>
        <EditText
            android:id="@+id/indexa9"
            android:text="0"
```

```
        android:layout_weight="1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>
    </TableRow>
    <TableRow
        android:orientation="horizontal"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content">
        <EditText
            android:id="@+id/indexa10"
            android:text="0"
            android:layout_weight="1"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"/>
        <EditText
            android:id="@+id/indexa11"
            android:text="0"
            android:layout_weight="1"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"/>
        <EditText
            android:id="@+id/indexa12"
            android:text="0"
            android:layout_weight="1"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"/>
        <EditText
            android:id="@+id/indexa13"
            android:text="0"
            android:layout_weight="1"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"/>
        <EditText
            android:id="@+id/indexa14"
            android:text="0"
            android:layout_weight="1"
            android:layout_width="wrap_content"
```

```
                android:layout_height="wrap_content"/>
</TableRow>
<TableRow
    android:orientation="horizontal"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content">
    <EditText
        android:id="@+id/indexa15"
        android:text="0"
        android:layout_weight="1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>
    <EditText
        android:id="@+id/indexa16"
        android:text="0"
        android:layout_weight="1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>
    <EditText
        android:id="@+id/indexa17"
        android:text="0"
        android:layout_weight="1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>
    <EditText
        android:id="@+id/indexa18"
        android:text="0"
        android:layout_weight="1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>
    <EditText
        android:id="@+id/indexa19"
        android:text="0"
        android:layout_weight="1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>
</TableRow>
```

```
</TableLayout>

<Button

    android:id="@+id/set"

    android:layout_width="fill_parent"

    android:layout_height="wrap_content"

    android:text="colorMartrix变换"

/>

</LinearLayout>
```

CMatrix类主要负责接收颜色矩阵的设置和重绘，没有要说的。MyImage类中进行绘图工作，首先设置颜色矩阵cm.set(..)从一维数组中读取数据20个数据给颜色矩阵赋值，paint.setColorFilter(..)设置颜色滤镜，然后绘图，效果就出来了（这个过程和PS差不多）如下：

看到这里，相信大家对颜色矩阵的作用已经有了一个直观的感受，现在也可以尝试做一个照片特效的软件。但是各种效果并不能让用户手动调节颜色矩阵，这里需要计算公式，由于本人并不是做图形软件的也不能提供，可以参考这个链接：

http://www.adobe.com/devnet/flash/articles/matrix_transformations/ColorMatrixDemo.swf

坐标变换矩阵

坐标变换矩阵是一个3*3的矩阵如图2.1，用来对图形进行坐标变化，将原来的坐标点转移到新的坐标点，因为一个图片是有点阵和每一点上的颜色信息组成的，所以对坐标的变换，就是对每一点进行搬移形成新的图片。

具体的说图形的放大缩小，移动，旋转，透视，扭曲这些效果都可以用此矩阵来完成。

这个矩阵的作用是对坐标x,y进行变换计算结果如下：

$$x' = a * x + b * y + c$$

$$y' = d * x + e * y + f$$

通常情况下g=h=0,这样使1=0*x+0*y+1恒成立。和颜色矩阵一样，坐标变换矩阵真正使用的参数很少也很有

规律。

上图就是一个坐标变换矩阵的简单例子，计算后发现 $x'=x+50, y'=y+50$ 。

可见图片的每一点都在x和y方向上平移到了 (50 , 50) 点处，这种效果就是平移效果，将图片转移到了 (50 , 50) 处。

计算上面得矩阵 $x'=2*x, y'=2*y$ 。经过颜色矩阵和上面转移效果学习，相信读者可以明白这个矩阵的作用了，这个矩阵对图片进行了放大，具体的说是放大了二倍。

下面将介绍几种常用的变换矩阵：

1. 旋转

绕原点逆时针旋转 θ 度角的变换公式是 $x' = x\cos\theta - y\sin\theta$ 与 $y' = x\sin\theta + y\cos\theta$

旋转过程图解：

原点 $A(x_0, y_0) = (r \cos \alpha, r \sin \alpha)$

旋转角后 $B(x, y) = (r \cos(\alpha + \theta), r \sin(\alpha + \theta))$

表

$$X \cdot A = \begin{vmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{vmatrix} \cdot \begin{vmatrix} r \cos \alpha \\ r \sin \alpha \\ 0 \end{vmatrix} = \begin{vmatrix} r \cos(\alpha + \theta) \\ r \sin(\alpha + \theta) \\ 0 \end{vmatrix}$$

X表变换

$$= \begin{vmatrix} x \\ y \\ 0 \end{vmatrix}$$

2. 缩放

变换后长宽分别放大 $x' = \text{scale} * x; y' = \text{scale} * y$.

3. 切变

4. 反射

(,)单位向量

5. 正投影

(,)单位向量

上面的各种效果也可以叠加在一起，既矩阵的组合变换，可以用矩阵乘法实现之，如： $R=B(A*C)=(B*A)C$ ，注意一点就是 $B*A$ 和 $A*B$ 一般是不等的。

下面将编一个小程序，通过控制坐标变换矩阵来达到控制图形的目的，JavaCode如下：

MyMoveView.java

```
package com.hao.view;

import com.ho.R;

import android.graphics.Bitmap;
import android.graphics.BitmapFactory;
import android.graphics.Canvas;
import android.graphics.Matrix;
import android.graphics.Paint;
import android.util.Log;
import android.view.View;
import android.content.Context;
import android.util.AttributeSet;

public class MyMoveView extends View {
    public MyMoveView(Context context, AttributeSet attrs) {
        super(context, attrs);
        // TODO Auto-generated constructor stub
        mBitmap = BitmapFactory.decodeResource(context.getResources(), R.drawable.music);
        invalidate();
    }

    private Paint mPaint = new Paint(Paint.ANTI_ALIAS_FLAG);
    private Bitmap mBitmap;
    private float[] array = new float[9];

    public void setValues(float[] a) {
```



```
        for (int i = 0; i < 9; i++) {
            array[i] = a[i];
        }
    }

    @Override
    protected void onDraw(Canvas canvas) {
        Paint paint = mPaint;
        canvas.drawBitmap(mBitmap, 0, 0, paint);
        // new 一个坐标变换矩阵
        Matrix cm = new Matrix();
        // 为坐标变换矩阵设置响应的值
        cm.setValues(array);
        // 按照坐标变换矩阵的描述绘图
        canvas.drawBitmap(mBitmap, cm, paint);
        Log.i("CMatrix", "----->onDraw");
    }
}
```

MoveMatrixActivity.java

```
package com.hao;

import com.hao.view.MyMoveView;
import com.ho.R;

import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;

public class MoveMatrixActivity extends Activity {
    private Button change;
    private EditText[] et = new EditText[9];
    private float[] carray = new float[9];
    private MyMoveView sv;
```

```
/** Called when the activity is first created. */
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main_two);

    change = (Button) findViewById(R.id.change);
    sv = (MyMoveView) findViewById(R.id.moveView);

    for (int i = 0; i < 9; i++) {
        int id = R.id.ed1 + i;
        et[i] = (EditText) findViewById(id);
        carray[i] = Float.valueOf(et[i].getText().toString());
    }
    change.setOnClickListener(l);
}

private Button.OnClickListener l = new Button.OnClickListener() {
    @Override
    public void onClick(View v) {
        // TODO Auto-generated method stub
        getValues();
        sv.setValues(carray);
        sv.invalidate();
    }
};

public void getValues() {
    for (int i = 0; i < 9; i++) {
        carray[i] = Float.valueOf(et[i].getText().toString());
    }
}
}
```

加上布局：

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">
    <com.hao.view.MyMoveView
        android:id="@+id/moveView"
        android:layout_gravity="center_horizontal"
        android:layout_width="wrap_content"
        android:layout_height="200dip"
    />
    <TableLayout
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:orientation="vertical">
        <TableRow
            android:orientation="horizontal"
            android:layout_width="fill_parent"
            android:layout_height="wrap_content">
            <EditText
                android:id="@+id/ed1"
                android:text="0"
                android:layout_weight="1"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"/>
            <EditText
                android:id="@+id/ed2"
                android:text="0"
                android:layout_weight="1"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"/>
            <EditText
                android:id="@+id/ed3"
                android:text="0"
                android:layout_weight="1"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"/>
        </TableRow>
    </TableLayout>
</LinearLayout>
```

```
</TableRow>
<TableRow
    android:orientation="horizontal"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content">
    <EditText
        android:id="@+id/ed4"
        android:text="0"
        android:layout_weight="1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>
    <EditText
        android:id="@+id/ed5"
        android:text="0"
        android:layout_weight="1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>
    <EditText
        android:id="@+id/ed6"
        android:text="0"
        android:layout_weight="1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>
</TableRow>
<TableRow
    android:orientation="horizontal"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content">
    <EditText
        android:id="@+id/ed7"
        android:text="0"
        android:layout_weight="1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>
    <EditText
        android:id="@+id/ed8"
        android:text="0"
```

```
                android:layout_weight="1"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"/>
            <EditText
                android:id="@+id/ed9"
                android:text="0"
                android:layout_weight="1"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"/>
        </TableRow>
    </TableLayout>
    <Button
        android:id="@+id/change"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="moveMartrix变换"
    />
</LinearLayout>
```

上面的代码中类CooMatrix用于接收用户输入的坐标变换矩阵参数，类MyImage接收参数，通过setValues()设置矩阵参数，然后Canvas调用drawBitmap绘图。效果如下：

上面给出了用坐标变换矩阵做出的各种效果，用坐标变换矩阵可以方面的调节图形的各种效果，但是我们看看Matrix类就可以发现，实际上，matrix类本身已经提供了许多类似的方法，我们只要调用，就可以了。

setScale(float sx, float sy, float px, float py) 放大
setSkew(float kx, float ky, float px, float py) 斜切
setTranslate(float dx, float dy) 平移
setRotate(float degrees, float px, float py) 旋转

上面的函数提供了基本的变换平移，放大，旋转，斜切。为了做出更复杂的变换，同时不必亲手去改动坐标变换矩阵，

Matrix类提供了许多Map方法，将原图形映射到目标点构成新的图形，

下面简述setPolyToPoly(float[] src, int srcIndex, float[] dst, int dstIndex, int pointCount) 的用法，希望起到举一反三的作用。

参数src和dst是分别存储了原图像的点和指定的目标点的一维数组，数组中存储的坐标格式如下：

[x0, y0, x1, y1, x2, y2,...]

这个函数将src中的坐标映射到dst中的坐标，实现图像的变换。

具体的例子可以参考APIDemos里的PolyToPoly，我在这里就不再贴代码了，只讲一下函数是怎么变换图片的。下面是效果：

图中写1的是原图，写有2，3，4的是变换后的图形。现在分析2是怎么变换来的，变换的原坐标点和目的坐标点如下：

```
src=new float[] { 32, 32, 64, 32 }
```

```
dst=new float[] { 32, 32, 64, 48 }
```

从上图标示出的坐标看出原图的（32，32）映射到原图的（32，32），（64，32）映射到原图（64，48）这样的效果是图像放大了而且发生了旋转。这样的过程相当于（32，32）点不动，然后拉住图形（64，32）点并拉到（64，48）点处，这样图形必然会被拉伸放大并且发生旋转。最后用一个平移将图形移动到右边现在的位置。希望能够好好理解这一过程，下面的3，4图是同样的道理。

代码：

```
/*
 * Copyright (C) 2007 The Android Open Source Project
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *     http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
```

```
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.  
* See the License for the specific language governing permissions and  
* limitations under the License.  
*/
```

```
package com.example.android.apis.graphics;
```

```
// Need the following import to get access to the app resources, since this  
// class is in a sub-package.
```

```
//import com.example.android.apis.R;
```

```
import android.app.Activity;
```

```
import android.content.Context;
```

```
import android.graphics.*;
```

```
import android.os.Bundle;
```

```
import android.view.View;
```

```
public class PolyToPoly extends GraphicsActivity {
```

```
    @Override
```

```
    protected void onCreate(Bundle savedInstanceState) {
```

```
        super.onCreate(savedInstanceState);
```

```
        setContentView(new SampleView(this));
```

```
    }
```

```
    private static class SampleView extends View {
```

```
        private Paint mPaint = new Paint(Paint.ANTI_ALIAS_FLAG);
```

```
        private Matrix mMatrix = new Matrix();
```

```
        private Paint.FontMetrics mFontMetrics;
```

```
        private void doDraw(Canvas canvas, float src[], float dst[]) {
```

```
            canvas.save();
```

```
            //这个函数很关键，是利用坐标点的对应关系进行图像变换，从src->dst
```

```
            mMatrix.setPolyToPoly(src, 0, dst, 0, src.length >> 1);
```

```
            canvas.concat(mMatrix);
```

```
            //画一个矩形
```

```
            mPaint.setColor(Color.GRAY);
```

```
mPaint.setStyle(Paint.Style.STROKE);
canvas.drawRect(0, 0, 64, 64, mPaint);
canvas.drawLine(0, 0, 64, 64, mPaint);
canvas.drawLine(0, 64, 64, 0, mPaint);

//在矩形上写字并设置其位置居于矩形中间
mPaint.setColor(Color.RED);
mPaint.setStyle(Paint.Style.FILL);
// how to draw the text center on our square
// centering in X is easy... use alignment (and X at midpoint)
float x = 64/2;
// centering in Y, we need to measure ascent/descent first
float y = 64/2 - (mFontMetrics.ascent + mFontMetrics.descent)/2;
canvas.drawText(src.length/2 + "", x, y, mPaint);
//画一次存一次
canvas.restore();
}

public SampleView(Context context) {
    super(context);

    // for when the style is STROKE
    mPaint.setStrokeWidth(4);
    // for when we draw text
    mPaint.setTextSize(40);
    mPaint.setTextAlign(Paint.Align.CENTER);
    mFontMetrics = mPaint.getFontMetrics();
}

@Override protected void onDraw(Canvas canvas) {
    Paint paint = mPaint;

    canvas.drawColor(Color.WHITE);

    canvas.save();
    //x和y轴都平移10然后画第一个矩形
    canvas.translate(10, 10);
```



```
// translate (1 point),
doDraw(canvas, new float[] { 0, 0 }, new float[] { 5, 5 });
canvas.restore();

canvas.save();
canvas.translate(160, 10);
//原图的 ( 32 , 32 ) 映射到原图的 ( 32 , 32 ) , ( 64 , 32 ) 映射到原图 ( 64 , 48 )
//这样的效果是图像放大了而且发生了旋转。这样的过程相当于 ( 32 , 32 ) 点不动 ,
//然后拉住图形 ( 64 , 32 ) 点并拉到 ( 64 , 48 ) 点处 , 这样图形必然会被拉伸放大并且发生旋转。
//最后用一个平移将图形移动到右边现在的位置。
// rotate/uniform-scale (2 points)
doDraw(canvas, new float[] { 32, 32, 64, 32 },
        new float[] { 32, 32, 64, 48 });
canvas.restore();

canvas.save();
canvas.translate(10, 110);
// rotate/skew (3 points),初始三个点就是矩形左上三个顶点
doDraw(canvas, new float[] { 0, 0, 64, 0, 0, 64 },
        new float[] { 0, 0, 96, 0, 24, 64 });
canvas.restore();

canvas.save();
canvas.translate(160, 110);
// perspective (4 points) , 对四个顶点都拉伸和变换
doDraw(canvas, new float[] { 0, 0, 64, 0, 64, 64, 0, 64 },
        new float[] { 0, 0, 96, 0, 64, 96, 0, 64 });
canvas.restore();
    }
}
}
```

非常感谢

转载自 : <http://www.cnblogs.com/leon19870907/articles/1978065.html>

还有这个<http://blog.csdn.net/kuku20092009/article/details/6740865>

也讲的非常好！再次感谢

1.7 Android Matrix理论与应用详解

发表时间: 2011-11-21 关键字: android Matrix Translate skew

本文转自<http://blog.csdn.net/kuku20092009/article/details/6740865>

非常感谢！呵呵

然后我又做了些补充

Matrix学习——基础知识

以前在线性代数中学习了矩阵，对矩阵的基本运算有一些了解，前段时间在使用GDI+的时候再次学习如何使用矩阵来变化图像，看了之后在这里总结说明。

首先大家看看下面这个3 x 3的矩阵，这个矩阵被分割成4部分。为什么分割成4部分，在后面详细说明。

首先给大家举个简单的例子：现设点P0 (x0 , y0) 进行平移后，移到P (x , y) ，其中x方向的平移量为 Δx ，y方向的平移量为 Δy ，那么，点P (x , y) 的坐标为：

$$x = x0 + \Delta x$$

$$y = y0 + \Delta y$$

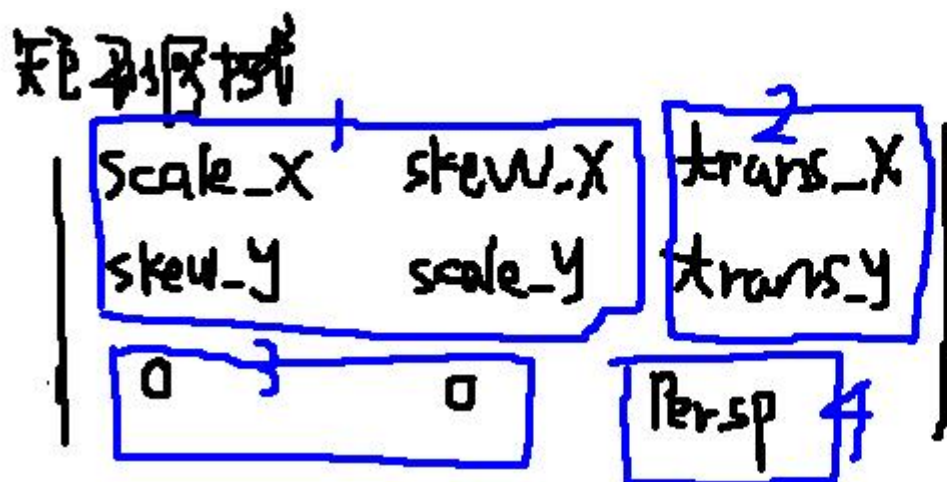
采用矩阵表达上述如下：

clip_image002

上述也类似与图像的平移，通过上述矩阵我们发现，只需要修改矩阵右上角的2个元素就可以了。

我们回头看上述矩阵的划分：

clip_image003



说明1：

上面的矩阵分为了四块：区域1,2,3,4,

其中区域1的功能就如上面所说：能实现缩放，旋转，透视，很重要

区域2：能实现平移

区域3：一般不用，对图也能产生一定的影响，一般为0

区域4：上面没写，能实现缩放的功能，如2表示缩小为原来的1/2,0.5表示扩大两倍，这个需要注意

说明2：

上面四块区域与Matrix函数的对应关系：

注：在函数中的(px,py)都表示一个坐标，表示view变换的一个中心点（可选）。

缩放：

对应与矩阵的关系： $sx \rightarrow scale_x$ ， $sy \rightarrow scale_y$

```
setScale(float sx, float sy, float px, float py)
```

```
setScale(float sx, float sy);
```

透视变换：

对应与矩阵的关系： $kx \rightarrow skew_x$ ， $ky \rightarrow skew_y$

```
setSkew(float kx, float ky, float px, float py)
```

```
setSkew(float kx, float ky);
```

平移：

对应与矩阵的关系： $dx \rightarrow trans_x$ ， $dy \rightarrow trans_y$

```
setTranslate(float dx, float dy)
```

旋转：

对应与矩阵的关系：这个就稍微麻烦点，此时先将原来的点转换为 $(r\cos(a), r\sin(a))$ （ a 表示初始角度）

然后变换角度后点是 $(r\cos(a+\text{degrees}), r\sin(a+\text{degrees}))$ ；...这个具体过程看下面

这里用的话就没这么复杂，直接设置角度，很简单

```
setRotate(float degrees);
```

```
setRotate(float degrees, float px, float py)
```

注意：

原来的矩阵是：mMatrix.setValues(

```
new float[] {
```

```
1, 0, 200,
```

```
0, 1, 200,
```

```
0, 0, 2 });
```

然后我们进行如下操作：

```
System.out.println(mMatrix);
```

```
    mMatrix.setSkew(1, -1);
```

```
    System.out.println(mMatrix);
```

```
    mMatrix.setTranslate(100, 100);
```

```
    System.out.println(mMatrix);
```

输出结果为：

```
11-21 01:56:34.261: INFO/System.out(2046): Matrix{[1.0, 0.0, 200.0][0.0, 1.0, 200.0][0.0, 0.0, 2.0]}-----
```

矩阵A

11-21 01:56:34.261: INFO/System.out(2046): Matrix{[1.0, 1.0, 0.0][-1.0, 1.0, 0.0][0.0, 0.0, 1.0]}-----矩阵B

11-21 01:56:34.271: INFO/System.out(2046): Matrix{[1.0, 0.0, 100.0][0.0, 1.0, 100.0][0.0, 0.0, 1.0]}-----矩阵C

尤其是第二和第三个结果，好像并不是简单的将skew_x和skew_y替换，而是将整个数组替换

set是直接设置Matrix的值，每次set一次，整个Matrix的数组都会变。

相当于setValues只是设置指定的元素，其他默认，上面设置了A相当于吧setValues的初始值覆盖了,setTranslate一样

最后矩阵的值就是setTranslate的值，除了trans_x,trans_y其他都默认

总结：

1.set...方法是重新设置整个数组，等效与setValues只是设置部分值，其他默认。

2.set...,post, pre是矩阵的三种变换方式

set是重新设置数组

post是后乘 $M' = S * M$ (原数组在后)

pre是前乘 $M' = M * S$ (原数组在前)

其中前后乘是不一样的

3.复杂的变换可以用上述三种的组合实现，如一个图片旋

转30度，然后平移到(100,100)的地方

```
1. Matrix m = new Matrix();
2.
3. m.postRotate(30 );
4.
5. m.postTranslate(100 , 100 );
```

4.post和pre操作的时候自己生成了一个矩阵，如下

$$\begin{array}{ccc}
 \begin{array}{c} M \\ \left\{ \begin{array}{ccc} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{array} \right\} \end{array} & \xrightarrow{\text{setTranslate}(100, 100)} & \begin{array}{c} M' \\ \left\{ \begin{array}{ccc} 1 & 0 & 100 \\ 0 & 1 & 100 \\ 0 & 0 & 1 \end{array} \right\} \end{array} \\
 \\
 \begin{array}{c} M \\ \left\{ \begin{array}{ccc} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{array} \right\} \end{array} & \xrightarrow{\text{postTranslate}(100, 100)} & \begin{array}{c} M \times S \\ \left\{ \begin{array}{ccc} 1 & 0 & 100 \\ 0 & 1 & 100 \\ 0 & 0 & 1 \end{array} \right\} \times \left\{ \begin{array}{ccc} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{array} \right\} \end{array} \\
 & & M' =
 \end{array}$$

故而postTranslate(x,y)事实是生成一个矩阵(相当于setTranslate (x,y) 的矩阵)，然后再后乘原矩阵M。

为了验证上面的功能划分，我们举个具体的例子：现设点P0 (x0 , y0) 进行平移后，移到P (x , y)，其中x放大a倍，y放大b倍，

矩阵就是：clip_image004，按照类似前面“平移”的方法就验证。


图像的旋转稍微复杂：现设点P0 (x0 , y0) 旋转θ角后的对应点为P (x , y)。通过使用向量，我们得到如下：

$$x_0 = r \cos \alpha$$

$$y_0 = r \sin \alpha$$

$$x = r \cos(\alpha + \theta) = x_0 \cos \theta - y_0 \sin \theta$$

$$y = r \sin(\alpha + \theta) = x_0 \sin \theta + y_0 \cos \theta$$

于是我们得到矩阵：

如果图像围绕着某个点(a , b)旋转呢？则先要将坐标平移到该点，再进行旋转，然后将旋转后的图像平移回到原来的坐标原点，在后面的篇幅中我们将详细介绍。

Matrix学习——如何使用Matrix

上一篇幅 **Matrix学习——基础知识**，从高等数学方面给大家介绍了Matrix，本篇幅我们就结合Android 中的 android.graphics.Matrix来具体说明，还记得我们前面说的**图像旋转**的矩阵：



从最简单的旋转90度的是：



在android.graphics.Matrix中有对应旋转的函数：

```
Matrix matrix = new Matrix();  
matrix.setRotate(90);  
Test.Log(MAXTRIX_TAG," setRotate(90):%s" , matrix.toString());
```



查看运行后的矩阵的值（通过Log输出）：



与上面的公式基本完全一样（ android.graphics.Matrix采用的是浮点数，而我们采用的整数）。

有了上面的例子，相信大家就可以亲自尝试了。通过上面的例子我们也发现，我们也可以直接来初始化矩阵，比如说要旋转30度：



前面给大家介绍了这么多，下面我们开始介绍**图像的镜像**，分为2种：水平镜像、垂直镜像。先介绍如何实现垂直镜像，什么是垂直镜像就不详细说明。图像的垂直镜像变化也可以用矩阵变化的表示，设点P0 (x0 , y0) 进行镜像后的对应点为P (x , y)，图像的高度为fHeight，宽度为fWidth，原图像中的P0 (x0 , y0) 经过垂直镜像后的坐标变为 (x0 , fHeight- y0) ；

$x = x0$

$y = fHeight - y0$

推导出相应的矩阵是：

clip_image011

```
final float f[] = {1.0F,0.0F,0.0F,0.0F,-1.0F,120.0F,0.0F,0.0F,1.0F};
```

```
Matrix matrix = new Matrix();
```

```
matrix.setValues(f);
```

按照上述方法运行后的结果：

clip_image012

至于水平镜像采用类似的方法，大家可以自己去试试吧。

实际上，使用下面的方式也可以实现垂直镜像：

```
Matrix matrix = new Matrix();
```

```
matrix.setScale (1.0 , -1.0);
```

```
matrix.postTraslate(0, fHeight);
```

这就是我们将在后面的篇幅中详细说明。

Matrix学习——图像的复合变化

Matrix学习——基础知识篇幅中，我们留下一个话题：如果图像围绕着某个点P(a, b)旋转，则先要将坐标系平移到该点，再进行旋转，然后将旋转后的图像平移回到原来的坐标原点。

我们需要3步：

1. **平移**——将坐标系平移到点P(a, b)；

2. **旋转**——以原点为中心旋转图像；
3. **平移**——将旋转后的图像平移回到原来的坐标原点；

相比较前面说的图像的几何变化（基本的图像几何变化），这里需要**平移——旋转——平移**，这种需要多种图像的几何变化就叫做图像的复合变化。

设对给定的图像依次进行了基本变化F1、F2、F3.....、Fn，它们的变化矩阵分别为T1、T2、T3.....、Tn，图像复合变化的矩阵T可以表示为： $T = T_n T_{n-1} \dots T_1$ 。

按照上面的原则，围绕着某个点(a, b)旋转 θ 的变化矩阵序列是：

clip_image013

按照上面的公式，我们列举一个简单的例子：围绕（100，100）旋转30度($\sin 30 = 0.5$ ， $\cos 30 = 0.866$)

```
float f[] = { 0.866F, -0.5F, 63.4F, 0.5F, 0.866F, -36.6F, 0.0F, 0.0F, 1.0F };
```

```
matrix = new Matrix();
```

```
matrix.setValues(f);
```

旋转后的图像如下：

clip_image014

Android为我们提供了更加简单的方法，如下：

```
Matrix matrix = new Matrix();
```

```
matrix.setRotate(30, 100, 100);
```

矩阵运行后的实际结果：

clip_image015

与我们前面通过公式获取得到的矩阵完全一样。

在这里我们提供另外一种方法，也可以达到同样的效果：

```
float a = 100.0F, b = 100.0F;
```

```
matrix = new Matrix();
```

```
matrix.setTranslate(a, b);
```

```
matrix.preRotate(30);
```

```
matrix.preTranslate(-a,-b);
```

将在后面的篇幅中为大家详细解析

通过类似的方法，我们还可以得到：**相对点P(a , b)的比例[sx,sy]变化矩阵**

clip_image016

Matrix学习——Preconcats or Postconcats?

从最基本的高等数学开始，Matrix的基本操作包括： $+$ 、 $*$ 。Matrix的乘法不满足交换律，也就是说 $A*B \neq B*A$ 。

还有2种常见的矩阵：

clip_image017

有了上面的基础，下面我们开始进入主题。由于矩阵不满足交换律，所以用矩阵B乘以矩阵A，需要考虑是左乘（ $B*A$ ），还是右乘（ $A*B$ ）。在Android的android.graphics.Matrix中为我们提供了类似的方法，也就是我们本篇幅要说明的Preconcats matrix 与 Postconcats matrix。下面我们还是通过具体的例子还说明：

clip_image018

通过输出的信息，我们分析其运行过程如下：

clip_image019

看了上面的输出信息。我们得出结论：**Preconcats matrix相当于右乘矩阵，Postconcats matrix相当于左乘矩阵。**

上一篇幅中，我们说到：

clip_image020

其晕死过程的详细分析就不在这里多说了。

Matrix学习——错切变换

什么是**图像的错切变换**（Shear transformation）？我们还是直接看图片错切变换后是的效果：

clip_image021

clip_image022

对图像的错切变换做个总结：

clip_image023

$x = x0 + b*y0;$

$y = d*x0 + y0;$

clip_image024

这里再次给大家介绍一个需要注意的地方：

clip_image025

通过以上，我们发现Matrix的setXXXX()函数，在调用时调用了一次reset()，这个在复合变换时需要注意。

Matrix学习——对称变换（反射）

什么是对称变换？具体的理论就不详细说明了，图像的镜像就是对称变换中的一种。

clip_image026

利用上面的总结做个具体的例子，产生与直线 $y = -x$ 对称的反射图形，代码片段如下：

clip_image027

当前矩阵输出是：

clip_image028

图像变换的效果如下：

clip_image029

附：三角函数公式

两角和公式

$$\sin(a+b) = \sin a \cos b + \cos a \sin b$$

$$\sin(a-b) = \sin a \cos b - \sin b \cos a ?$$

$$\cos(a+b) = \cos a \cos b - \sin a \sin b$$

$$\cos(a-b) = \cos a \cos b + \sin a \sin b$$

$$\tan(a+b) = (\tan a + \tan b) / (1 - \tan a \tan b)$$

$$\tan(a-b) = (\tan a - \tan b) / (1 + \tan a \tan b)$$

$$\cot(a+b) = (\cot a \cot b - 1) / (\cot b + \cot a) ?$$

$$\cot(a-b) = (\cot a \cot b + 1) / (\cot b - \cot a)$$

倍角公式

$$\tan 2a = 2 \tan a / [1 - (\tan a)^2]$$

$$\cos 2a = (\cos a)^2 - (\sin a)^2 = 2(\cos a)^2 - 1 = 1 - 2(\sin a)^2$$

$$\sin 2a = 2\sin a \cos a$$

半角公式

$$\sin(a/2) = \sqrt{(1 - \cos a)/2} \quad \sin(a/2) = -\sqrt{(1 - \cos a)/2}$$

$$\cos(a/2) = \sqrt{(1 + \cos a)/2} \quad \cos(a/2) = -\sqrt{(1 + \cos a)/2}$$

$$\tan(a/2) = \sqrt{(1 - \cos a)/(1 + \cos a)} \quad \tan(a/2) = -\sqrt{(1 - \cos a)/(1 + \cos a)}$$

$$\cot(a/2) = \sqrt{(1 + \cos a)/(1 - \cos a)} \quad \cot(a/2) = -\sqrt{(1 + \cos a)/(1 - \cos a)} ?$$

$$\tan(a/2) = (1 - \cos a)/\sin a = \sin a/(1 + \cos a)$$

和差化积

$$2\sin a \cos b = \sin(a+b) + \sin(a-b)$$

$$2\cos a \sin b = \sin(a+b) - \sin(a-b)$$

$$2\cos a \cos b = \cos(a+b) + \cos(a-b)$$

$$-2\sin a \sin b = \cos(a+b) - \cos(a-b)$$

$$\sin a + \sin b = 2\sin((a+b)/2)\cos((a-b)/2)$$

$$\cos a + \cos b = 2\cos((a+b)/2)\cos((a-b)/2)$$

$$\tan a + \tan b = \sin(a+b)/\cos a \cos b$$

积化和差公式

$$\sin(a)\sin(b)=-1/2*[\cos(a+b)-\cos(a-b)]$$

$$\cos(a)\cos(b)=1/2*[\cos(a+b)+\cos(a-b)]$$

$$\sin(a)\cos(b)=1/2*[\sin(a+b)+\sin(a-b)]$$

诱导公式

$$\sin(-a)=-\sin(a)$$

$$\cos(-a)=\cos(a)$$

$$\sin(\pi/2-a)=\cos(a)$$

$$\cos(\pi/2-a)=\sin(a)$$

$$\sin(\pi/2+a)=\cos(a)$$

$$\cos(\pi/2+a)=-\sin(a)$$

$$\sin(\pi-a)=\sin(a)$$

$$\cos(\pi-a)=-\cos(a)$$

$$\sin(\pi+a)=-\sin(a)$$

$$\cos(\pi+a)=-\cos(a)$$

$$\tan a=\tan a=\sin a/\cos a$$

万能公式

$$\sin(a)=(2\tan(a/2))/(1+\tan^2(a/2))$$

$$\cos(a) = (1 - \tan^2(a/2)) / (1 + \tan^2(a/2))$$

$$\tan(a) = (2 \tan(a/2)) / (1 - \tan^2(a/2))$$

其它公式

$$a \sin(a) + b \cos(a) = \sqrt{a^2 + b^2} \sin(a+c) \text{ [其中, } \tan(c) = b/a]$$

$$a \sin(a) - b \cos(a) = \sqrt{a^2 + b^2} \cos(a-c) \text{ [其中, } \tan(c) = a/b]$$

$$1 + \sin(a) = (\sin(a/2) + \cos(a/2))^2$$

$$1 - \sin(a) = (\sin(a/2) - \cos(a/2))^2$$

其他非重点三角函数

$$\csc(a) = 1/\sin(a)$$

$$\sec(a) = 1/\cos(a)$$

双曲函数

$$\sinh(a) = (e^a - e^{-a})/2$$

$$\cosh(a) = (e^a + e^{-a})/2$$

$$\tgh(a) = \sinh(a)/\cosh(a)$$

1.8 断点续传和下载原理分析

发表时间: 2011-12-07 关键字: 多线程, 下载, 上传, 续传, Android

最近做一个文件上传和下载的应用对文件上传和下载进行了一个完整的流程分析

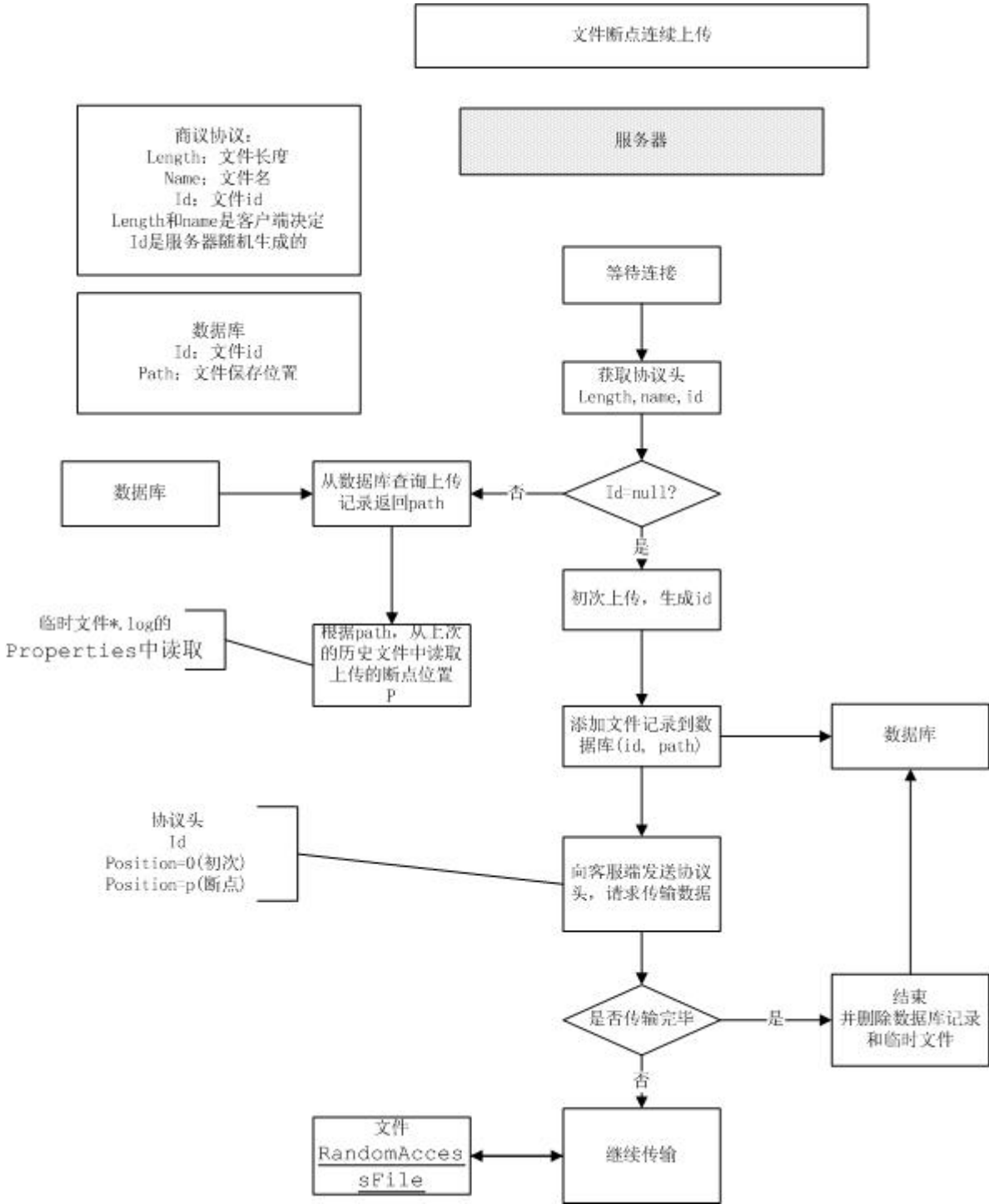
以来是方便自己对文件上传和下载的理解，而来便于团队内部的分享

故而做了几张图，将整体的流程都画下来，便于大家的理解和分析，如果有不完善的地方希望

大家多提意见，

由于参考了网上许多的资料，特此感谢

首先是文件上传，这个要用到服务器



关键代码：

FileServer.java

```
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
```

```
import java.io.InputStream;
import java.io.OutputStream;
import java.io.PushbackInputStream;
import java.io.RandomAccessFile;
import java.net.ServerSocket;
import java.net.Socket;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.HashMap;
import java.util.Map;
import java.util.Properties;
import java.util.Set;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

import util.FileLogInfo;
import util.StreamTool;

public class FileServer {
    private ExecutorService executorService;//线程池
    private int port;//监听端口
    private boolean quit = false;//退出
    private ServerSocket server;
    private Map<Long, FileLogInfo> datas = new HashMap<Long, FileLogInfo>();//存放断点数据,
    public FileServer(int port)
    {
        this.port = port;
        //创建线程池,池中具有(cpu个数*50)条线程
        executorService = Executors.newFixedThreadPool(Runtime.getRuntime().availableProcessors());
    }

    /**
     * 退出
     */
    public void quit()
```

```
{
    this.quit = true;
    try
    {
        server.close();
    }catch (IOException e)
    {
        e.printStackTrace();
    }
}

/**
 * 启动服务
 * @throws Exception
 */
public void start() throws Exception
{
    server = new ServerSocket(port); //实现端口监听
    while(!quit)
    {
        try
        {
            Socket socket = server.accept();
            executorService.execute(new SocketTask(socket)); //为支持多用户并发访问，采用线程
        }catch (Exception e)
        {
            e.printStackTrace();
        }
    }
}

private final class SocketTask implements Runnable
{
    private Socket socket = null;
    public SocketTask(Socket socket)
    {
        this.socket = socket;
    }
}
```

```
}  
@Override  
public void run()  
{  
    try  
    {  
        System.out.println("FileServer accepted connection "+ socket.ge  
        //得到客户端发来的第一行协议数据：Content-Length=143253434;file  
        //如果用户初次上传文件，sourceid的值为空。  
        InputStream inStream = socket.getInputStream();  
        String head = StreamTool.readLine(inStream);  
        System.out.println("FileServer head:"+head);  
        if(head!=null)  
        {  
            //下面从协议数据中提取各项参数值  
            String[] items = head.split(";");  
            String filelength = items[0].substring(items[0].indexOf(''  
            String filename = items[1].substring(items[1].indexOf(''  
            String sourceid = items[2].substring(items[2].indexOf(''  
            //生成资源id，如果需要唯一性，可以采用UUID  
            long id = System.currentTimeMillis();  
            FileLogInfo log = null;  
            if(sourceid!=null && !"".equals(sourceid))  
            {  
                id = Long.valueOf(sourceid);  
                //查找上传的文件是否存在上传记录  
                log = find(id);  
            }  
            File file = null;  
            int position = 0;  
            //如果上传的文件不存在上传记录,为文件添加跟踪记录  
            if(log==null)  
            {  
                //设置存放的位置与当前应用的位置有关  
                File dir = new File("c:/temp/");  
                if(!dir.exists()) dir.mkdirs();  
                file = new File(dir, filename);
```

```
//如果上传的文件发生重名，然后进行改名
if(file.exists())
{
    filename = filename.substring(0, filename.length() - 1) + "1";
    file = new File(dir, filename);
}
save(id, file);
}
// 如果上传的文件存在上传记录,读取上次的断点位置
else
{
    System.out.println("FileServer have exists log record");
    //从上传记录中得到文件的路径
    file = new File(log.getPath());
    if(file.exists())
    {
        File logFile = new File(file.getParentFile() + "/log.txt");
        if(logFile.exists())
        {
            Properties properties = new Properties();
            properties.load(new FileInputStream(logFile));
            //读取断点位置
            position = Integer.valueOf(properties.getProperty("position"));
        }
    }
}
//*****上面是对协议头的处理，下面正
//向客户端请求传输数据
OutputStream outputStream = socket.getOutputStream();
String response = "sourceid="+ id+ ";position="+ position;
//服务器收到客户端的请求信息后，给客户端返回响应信息：sourceid
//sourceid由服务生成，唯一标识上传的文件，position指示客户端从哪个位置开始上传
outputStream.write(response.getBytes());
RandomAccessFile fileOutputStream = new RandomAccessFile(file, "rw");
//设置文件长度
if(position==0) fileOutputStream.setLength(Integer.valueOf(response.split(";")[1]));
//移动文件指定的位置开始写入数据
```

```
        fileOutputStream.seek(position);
        byte[] buffer = new byte[1024];
        int len = -1;
        int length = position;
        //从输入流中读取数据写入到文件中，并将已经传入的文件长度写入
        while( (len=inStream.read(buffer)) != -1)
        {
            fileOutputStream.write(buffer, 0, len);
            length += len;
            Properties properties = new Properties();
            properties.put("length", String.valueOf(length));
            FileOutputStream logFile = new FileOutputStream(logFile);
            //实时记录文件的最后保存位置
            properties.store(logFile, null);
            logFile.close();
        }
        //如果长传长度等于实际长度则表示长传成功
        if(length==fileOutputStream.length()){
            delete(id);
        }
        fileOutputStream.close();
        inStream.close();
        outputStream.close();
        file = null;
    }
}
catch (Exception e)
{
    e.printStackTrace();
}
finally{
try
{
    if(socket!=null && !socket.isClosed()) socket.close();
}
catch (IOException e)
{

```



```
        e.printStackTrace();
    }
}
}

/**
 * 查找在记录中是否有sourceid的文件
 * @param sourceid
 * @return
 */
public FileLogInfo find(Long sourceid)
{
    return datas.get(sourceid);
}

/**
 * 保存上传记录，日后可以改成通过数据库存放
 * @param id
 * @param saveFile
 */
public void save(Long id, File saveFile)
{
    System.out.println("save logfile "+id);
    datas.put(id, new FileLogInfo(id, saveFile.getAbsolutePath()));
}

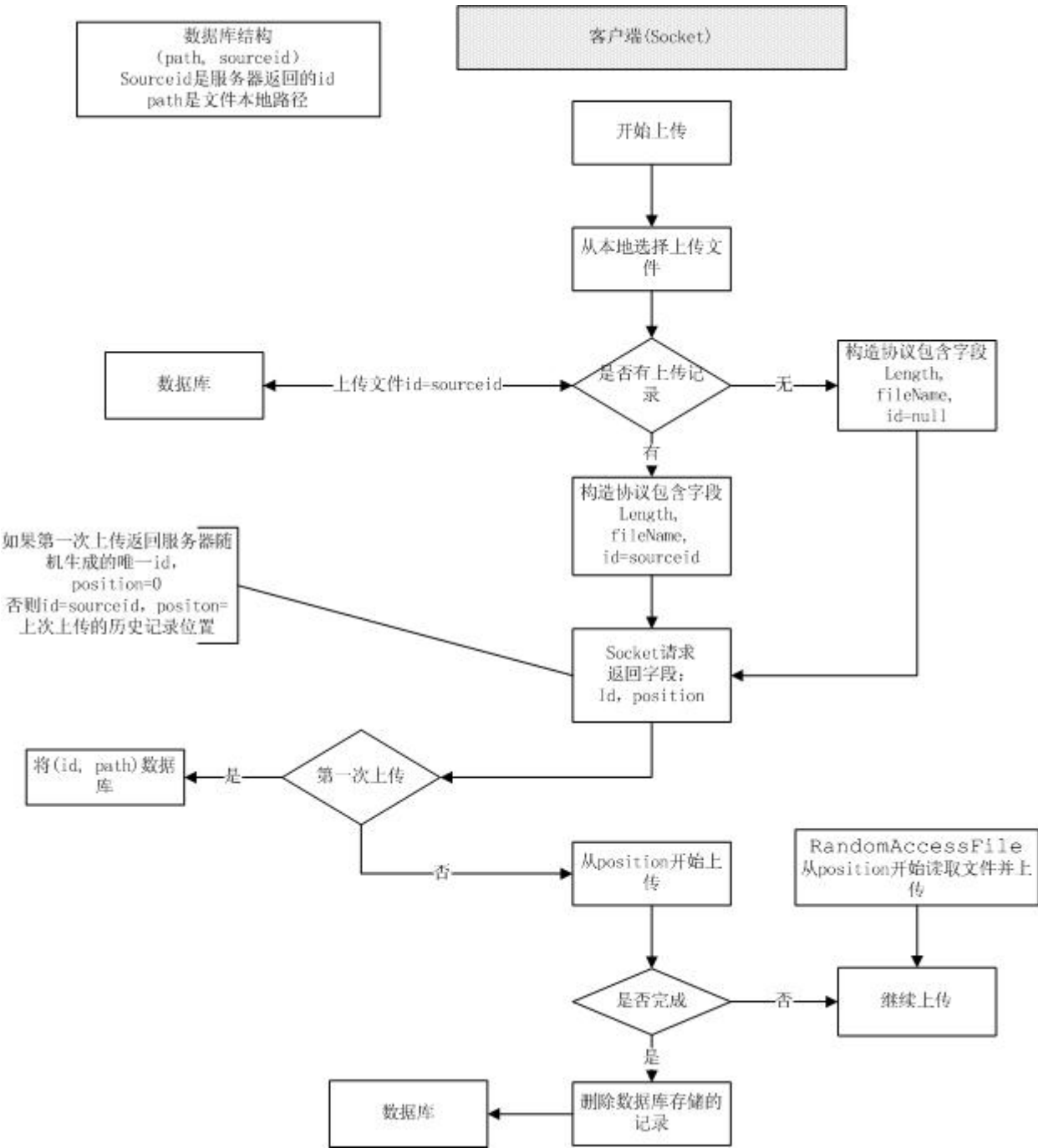
/**
 * 当文件上传完毕，删除记录
 * @param sourceid
 */
public void delete(long sourceid)
{
    System.out.println("delete logfile "+sourceid);
    if(datas.containsKey(sourceid)) datas.remove(sourceid);
}
```

```
}

```

由于在上面的流程图中已经进行了详细的分析，我在这儿就不讲了，只是在存储数据的时候服务器没有用数据库去存储，这儿只是为了方便，所以要想测试断点上传，服务器是不能停的，否则数据就没有了，在以后改进的时候应该用数据库去存储数据。

文件上传客户端：



关键代码：

UploadActivity.java

```
package com.hao;

import java.io.File;
import java.util.List;

import com.hao.upload.UploadThread;
import com.hao.upload.UploadThread.UploadProgressListener;
import com.hao.util.ConstantValues;
import com.hao.util.FileBrowserActivity;

import android.app.Activity;
import android.app.Dialog;
import android.app.ProgressDialog;
import android.content.DialogInterface;
import android.content.Intent;
import android.content.res.Resources;
import android.net.Uri;
import android.os.Bundle;
import android.os.Environment;
import android.os.Handler;
import android.os.Message;
import android.util.Log;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;
import android.widget.TextView;
import android.widget.Toast;

/**
 *
 * @author Administrator
 *
 */
public class UploadActivity extends Activity implements OnClickListener{
    private static final String TAG = "SiteFileFetchActivity";
    private Button download, upload, select_file;
```

```
private TextView info;
private static final int PROGRESS_DIALOG = 0;
private ProgressDialog progressDialog;
private UploadThread uploadThread;
private String uploadFilePath = null;
private String fileName;
/** Called when the activity is first created. */
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.upload);
    initView();
}

private void initView(){
    download = (Button) findViewById(R.id.download);
    download.setOnClickListener(this);
    upload = (Button) findViewById(R.id.upload);
    upload.setOnClickListener(this);
    info = (TextView) findViewById(R.id.info);
    select_file = (Button) findViewById(R.id.select_file);
    select_file.setOnClickListener(this);
}

@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    // TODO Auto-generated method stub
    super.onActivityResult(requestCode, resultCode, data);
    if (resultCode == RESULT_OK) {
        if (requestCode == 1) {
            Uri uri = data.getData();    // 接收用户所选文件的路径
            info.setText("select: " + uri); // 在界面上显示路径
            uploadFilePath = uri.getPath();
            int last = uploadFilePath.lastIndexOf("/");
            uploadFilePath = uri.getPath().substring(0, last+1);
            fileName = uri.getLastPathSegment();
        }
    }
}
```

```
    }
}

protected Dialog onCreateDialog(int id) {
    switch(id) {
    case PROGRESS_DIALOG:
        progressDialog = new ProgressDialog(UploadActivity.this);
        progressDialog.setProgressStyle(ProgressDialog.STYLE_HORIZONTAL);
        progressDialog.setButton("暂停", new DialogInterface.OnClickListener() {
            @Override
            public void onClick(DialogInterface dialog, int which) {
                // TODO Auto-generated method stub
                uploadThread.closeLink();
                dialog.dismiss();
            }
        });
        progressDialog.setMessage("正在上传...");
        progressDialog.setMax(100);
        return progressDialog;
    default:
        return null;
    }
}

/**
 * 使用Handler给创建他的线程发送消息，
 * 匿名内部类
 */
private Handler handler = new Handler()
{
    @Override
    public void handleMessage(Message msg)
    {
        //获得上传长度的进度
        int length = msg.getData().getInt("size");
        progressDialog.setProgress(length);
        if(progressDialog.getProgress()==progressDialog.getMax())//上传成功
```

```
{
    progressDialog.dismiss();
    Toast.makeText(UploadActivity.this, getResources().getString(R.string.upload_ov
}
}
};

@Override
public void onClick(View v) {
    // TODO Auto-generated method stub
    Resources r = getResources();
    switch(v.getId()){
        case R.id.select_file:
            Intent intent = new Intent();
            //设置起始目录和查找的类型
            intent.setDataAndType(Uri.fromFile(new File("/sdcard")), "*/*");//"*/*"
            intent.setClass(UploadActivity.this, FileBrowserActivity.class);
            startActivityForResult(intent, 1);
            break;
        case R.id.download:
            startActivity(new Intent(UploadActivity.this, SmartDownloadActi
            break;
        case R.id.upload:
            if(Environment.getExternalStorageState().equals(Environment.MEDIA_MOUNTED))//判
            {
                if(uploadFilePath == null){
                    Toast.makeText(UploadActivity.this, "还没设置上传文件", 1).show(
                }
                System.out.println("uploadFilePath:"+uploadFilePath+" "+fileName);
                //取得SDCard的目录
                File uploadFile = new File(new File(uploadFilePath), fileName);
                Log.i(TAG, "filePath:"+uploadFile.toString());
                if(uploadFile.exists())
                {
                    showDialog(PROGRESS_DIALOG);
                    info.setText(uploadFile+" "+ConstantValues.HOST+": "+ConstantValues.PORT
                    progressDialog.setMax((int) uploadFile.length());//设置长传文件的
```

```
        uploadThread = new UploadThread(UploadActivity.this, uploadFile, Constants.UPLOAD_SIZE);
        uploadThread.setListener(new UploadProgressListener() {

            @Override
            public void onUploadSize(int size) {
                // TODO Auto-generated method stub
                Message msg = new Message();
                msg.getData().putInt("size", size);
                handler.sendMessage(msg);
            }

        });
        uploadThread.start();
    }
    else
    {
        Toast.makeText(UploadActivity.this, "文件不存在", 1).show();
    }
}
else
{
    Toast.makeText(UploadActivity.this, "SDCard不存在!", 1).show();
}

        break;
    }

}

}
```

UploadThread.java

```
package com.hao.upload;

import java.io.File;
import java.io.IOException;
import java.io.InputStream;
```

```
import java.io.OutputStream;
import java.io.RandomAccessFile;
import java.net.Socket;

import android.content.Context;
import android.util.Log;

import com.hao.db.UploadLogService;
import com.hao.util.StreamTool;

public class UploadThread extends Thread {

    private static final String TAG = "UploadThread";
    /*需要上传文件的路径*/
    private File uploadFile;
    /*上传文件服务器的IP地址*/
    private String dstName;
    /*上传服务器端口号*/
    private int dstPort;
    /*上传socket链接*/
    private Socket socket;
    /*存储上传的数据库*/
    private UploadLogService logService;
    private UploadProgressListener listener;
    public UploadThread(Context context, File uploadFile, final String dstName,final int dstPort) {
        this.uploadFile = uploadFile;
        this.dstName = dstName;
        this.dstPort = dstPort;
        logService = new UploadLogService(context);
    }

    public void setListener(UploadProgressListener listener) {
        this.listener = listener;
    }

    /**
     * 模拟断开连接
     */
}
```



```
    */
    public void closeLink(){
        try{
            if(socket != null) socket.close();
        }catch(IOException e){
            e.printStackTrace();
            Log.e(TAG, "close socket fail");
        }
    }

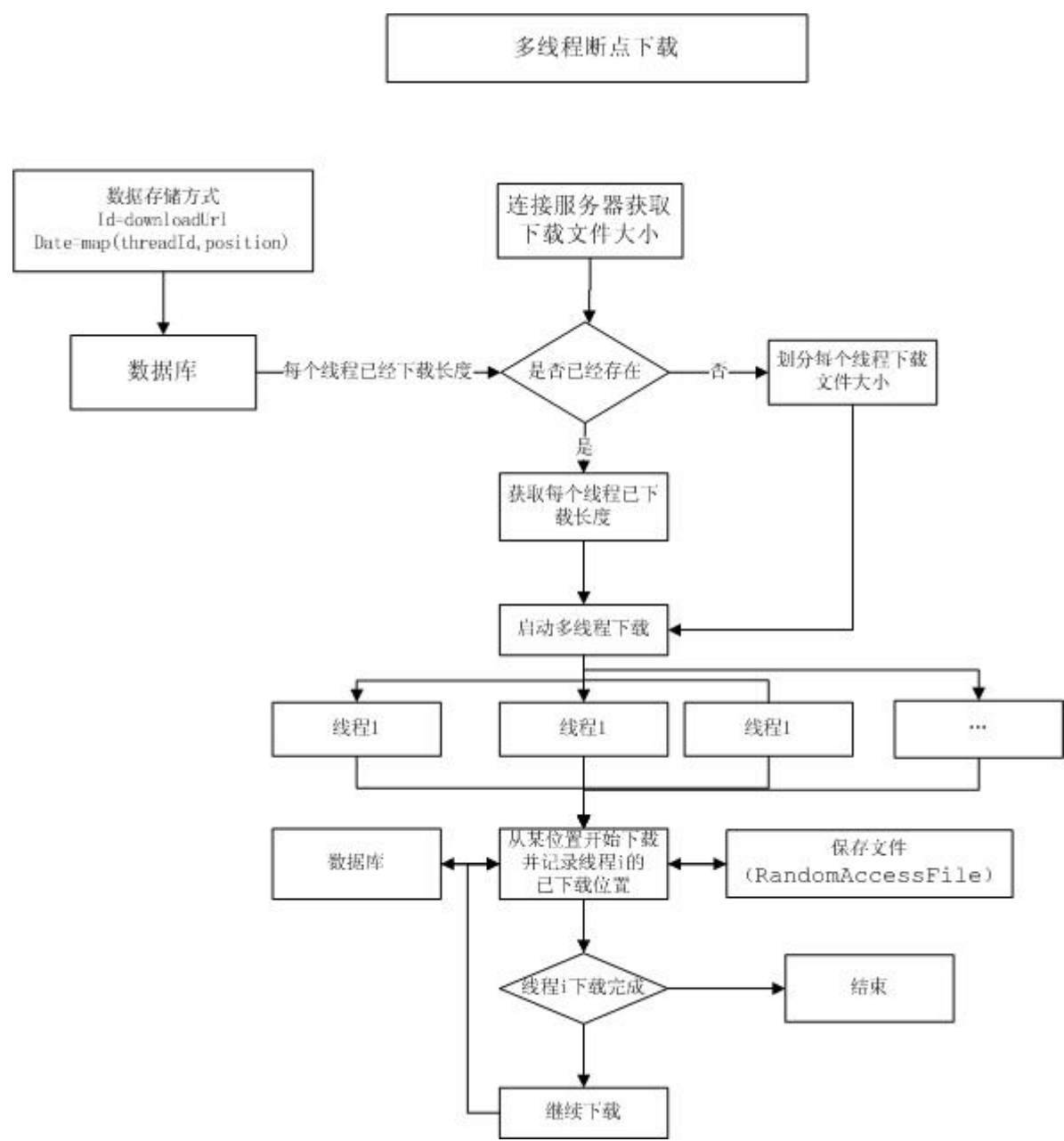
    @Override
    public void run() {
        // TODO Auto-generated method stub
        try {
            // 判断文件是否已有上传记录
            String souceid = logService.getBindId(uploadFile);
            // 构造拼接协议
            String head = "Content-Length=" + uploadFile.length()
                + ";filename=" + uploadFile.getName() + ";sourceid="
                + (souceid == null ? "" : souceid) + "%";
            // 通过Socket取得输出流
            socket = new Socket(dstName, dstPort);
            OutputStream outStream = socket.getOutputStream();
            outStream.write(head.getBytes());
            Log.i(TAG, "write to outStream");

            InputStream inStream = socket.getInputStream();
            // 获取到字符流的id与位置
            String response = StreamTool.readLine(inStream);
            Log.i(TAG, "response:" + response);
            String[] items = response.split(";");
            String responseid = items[0].substring(items[0].indexOf("=") + 1);
            String position = items[1].substring(items[1].indexOf("=") + 1);
            // 代表原来没有上传过此文件，往数据库添加一条绑定记录
            if (souceid == null) {
                logService.save(responseid, uploadFile);
            }
        }
    }
}
```

```
        RandomAccessFile fileOutputStream = new RandomAccessFile(uploadFile, "r");
        // 查找上次传送的最终位置, 并从这开始传送
        fileOutputStream.seek(Integer.valueOf(position));
        byte[] buffer = new byte[1024];
        int len = -1;
        // 初始化上传的数据长度
        int length = Integer.valueOf(position);
        while ((len = fileOutputStream.read(buffer)) != -1) {
            outputStream.write(buffer, 0, len);
            // 设置长传数据长度
            length += len;
            listener.onUploadSize(length);
        }
        fileOutputStream.close();
        outputStream.close();
        inputStream.close();
        socket.close();
        // 判断上传完则删除数据
        if (length == uploadFile.length())
            logService.delete(uploadFile);
    } catch (Exception e) {
        e.printStackTrace();
    }
}

public interface UploadProgressListener{
    void onUploadSize(int size);
}
}
```

下面是多线程下载



SmartDownloadActivity.java

```
package com.hao;

import java.io.File;

import com.hao.R;
import com.hao.R.id;
import com.hao.R.layout;
import com.hao.download.SmartFileDownloader;
import com.hao.download.SmartFileDownloader.SmartDownloadProgressListener;
import com.hao.util.ConstantValues;
```

```
import android.app.Activity;
import android.os.Bundle;
import android.os.Environment;
import android.os.Handler;
import android.os.Message;
import android.view.View;
import android.widget.Button;
import android.widget.ProgressBar;
import android.widget.TextView;
import android.widget.Toast;

/**
 *
 * @author Administrator
 *
 */
public class SmartDownloadActivity extends Activity {
    private ProgressBar downloadbar;
    private TextView resultView;
    private String path = ConstantValues.DOWNLOAD_URL;
    SmartFileDownloader loader;
    private Handler handler = new Handler() {
        @Override
        // 信息
        public void handleMessage(Message msg) {
            switch (msg.what) {
                case 1:
                    int size = msg.getData().getInt("size");
                    downloadbar.setProgress(size);
                    float result = (float) downloadbar.getProgress() / (float) downloadbar.getMax();
                    int p = (int) (result * 100);
                    resultView.setText(p + "%");
                    if (downloadbar.getProgress() == downloadbar.getMax())
                        Toast.makeText(SmartDownloadActivity.this, "下载成功", Toast.LENGTH_SHORT).show();
                    break;
                case -1:
            }
        }
    }
```

```
        Toast.makeText(SmartDownloadActivity.this, msg.getData().getStr  
        break;  
    }  
  
}  
  
};  
  
public void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
    setContentView(R.layout.download);  
  
    Button button = (Button) this.findViewById(R.id.button);  
    Button closeConn = (Button) findViewById(R.id.closeConn);  
    closeConn.setOnClickListener(new View.OnClickListener() {  
  
        @Override  
        public void onClick(View v) {  
            // TODO Auto-generated method stub  
            if(loader != null){  
                finish();  
            }else{  
                Toast.makeText(SmartDownloadActivity.this, "还没有开始下  
            }  
        }  
    });  
  
    downloadbar = (ProgressBar) this.findViewById(R.id.downloadbar);  
    resultView = (TextView) this.findViewById(R.id.result);  
    resultView.setText(path);  
    button.setOnClickListener(new View.OnClickListener() {  
        @Override  
        public void onClick(View v) {  
            if (Environment.getExternalStorageState().equals(Environment.ME  
                download(path, ConstantValues.FILE_PATH);  
            } else {  
                Toast.makeText(SmartDownloadActivity.this, "没有SDCard"  
            }  
        }  
    }  
}
```

```
        });  
    }  
  
    // 对于UI控件的更新只能由主线程(UI线程)负责, 如果在非UI线程更新UI控件, 更新的结果不会反映在屏幕上  
    private void download(final String path, final File dir) {  
        new Thread(new Runnable() {  
            @Override  
            public void run() {  
                try {  
                    loader = new SmartFileDownloader(SmartDownloadActivity.  
int length = loader.getFileSize();// 获取文件的长度  
downloadbar.setMax(length);  
loader.download(new SmartDownloadProgressListener() {  
            @Override  
            public void onDownloadSize(int size) {// 可以实  
                Message msg = new Message();  
                msg.what = 1;  
                msg.getData().putInt("size", size);  
                handler.sendMessage(msg);  
            }  
        });  
    } catch (Exception e) {  
        Message msg = new Message();// 信息提示  
        msg.what = -1;  
        msg.getData().putString("error", "下载失败");// 如果下载  
        handler.sendMessage(msg);  
    }  
    }  
    }).start();// 开始  
}  
}
```

这个单个的下载线程

SmartDownloadThread.java

```
package com.hao.download;

import java.io.File;
import java.io.InputStream;
import java.io.RandomAccessFile;
import java.net.HttpURLConnection;
import java.net.URL;

import android.util.Log;

/**
 * 线程下载
 * @author Administrator
 *
 */
public class SmartDownloadThread extends Thread {
    private static final String TAG = "SmartDownloadThread";
    private File saveFile;
    private URL downUrl;
    private int block;
    /* *下载开始位置 */
    private int threadId = -1;
    private int downLength;
    private boolean finish = false;
    private SmartFileDownloader downloader;

    public SmartDownloadThread(SmartFileDownloader downloader, URL downUrl,
                               File saveFile, int block, int downLength, int threadId) {
        this.downUrl = downUrl;
        this.saveFile = saveFile;
        this.block = block;
        this.downloader = downloader;
        this.threadId = threadId;
        this.downLength = downLength;
    }

    @Override
```

```
public void run() {
    if (downLength < block) { // 未下载完成
        try {
            HttpURLConnection http = (HttpURLConnection) downUrl
                .openConnection();
            http.setConnectTimeout(5 * 1000);
            http.setRequestMethod("GET");
            http.setRequestProperty("Accept", "image/gif, image/jpeg, image/");
            http.setRequestProperty("Accept-Language", "zh-CN");
            http.setRequestProperty("Referer", downUrl.toString());
            http.setRequestProperty("Charset", "UTF-8");
            int startPos = block * (threadId - 1) + downLength; // 开始位置
            int endPos = block * threadId - 1; // 结束位置
            http.setRequestProperty("Range", "bytes=" + startPos + "-" + endPos);
            http.setRequestProperty("User-Agent", "Mozilla/4.0 (compatible;");
            http.setRequestProperty("Connection", "Keep-Alive");

            InputStream inStream = http.getInputStream();
            byte[] buffer = new byte[1024];
            int offset = 0;
            print("Thread " + this.threadId + " start download from position " + startPos);
            RandomAccessFile threadfile = new RandomAccessFile(this.saveFile, "rw");
            threadfile.seek(startPos);
            while ((offset = inStream.read(buffer, 0, 1024)) != -1) {
                threadfile.write(buffer, 0, offset);
                downLength += offset;
                downloader.update(this.threadId, downLength);
                downloader.saveLogFile();
                downloader.append(offset);
            }
            threadfile.close();
            inStream.close();
            print("Thread " + this.threadId + " download finish");
            this.finish = true;
        } catch (Exception e) {
            this.downLength = -1;
            print("Thread " + this.threadId + ":" + e);
        }
    }
}
```



```
        }  
    }  
}  
  
private static void print(String msg) {  
    Log.i(TAG, msg);  
}  
  
/**  
 * 下载是否完成  
 * @return  
 */  
public boolean isFinish() {  
    return finish;  
}  
  
/**  
 * 已经下载的内容大小  
 * @return 如果返回值为-1,代表下载失败  
 */  
public long getDownLength() {  
    return downLength;  
}  
}
```

总得下载线程

SmartFileDownloader.java

```
package com.hao.download;  
  
import java.io.File;  
import java.io.RandomAccessFile;  
import java.net.HttpURLConnection;  
import java.net.URL;  
import java.util.LinkedHashMap;  
import java.util.Map;
```

```
import java.util.UUID;
import java.util.concurrent.ConcurrentHashMap;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

import com.hao.db.DownloadFileService;

import android.content.Context;
import android.util.Log;

/**
 * 文件下载主程序
 * @author Administrator
 *
 */
public class SmartFileDownloader {
    private static final String TAG = "SmartFileDownloader";
    private Context context;
    private DownloadFileService fileService;
    /* 已下载文件长度 */
    private int downloadSize = 0;
    /* 原始文件长度 */
    private int fileSize = 0;
    /*原始文件名*/
    private String fileName;
    /* 线程数 */
    private SmartDownloadThread[] threads;
    /* 本地保存文件 */
    private File saveFile;
    /* 缓存各线程下载的长度 */
    private Map<Integer, Integer> data = new ConcurrentHashMap<Integer, Integer>();
    /* 每条线程下载的长度 */
    private int block;
    /* 下载路径 */
    private String downloadUrl;

    /**
```

```
    * 获取文件名
    */
    public String getFileName(){
        return this.fileName;
    }

    /**
    * 获取线程数
    */
    public int getThreadSize() {
        return threads.length;
    }

    /**
    * 获取文件大小
    * @return
    */
    public int getFileSize() {
        return fileSize;
    }

    /**
    * 累计已下载大小
    * @param size
    */
    protected synchronized void append(int size) {
        downloadSize += size;
    }

    /**
    * 更新指定线程最后下载的位置
    * @param threadId 线程id
    * @param pos 最后下载的位置
    */
    protected void update(int threadId, int pos) {
        this.data.put(threadId, pos);
    }
}
```

```
/**
 * 保存记录文件
 */
protected synchronized void saveLogFile() {
    this.fileService.update(this.downloadUrl, this.data);
}

/**
 * 构建文件下载器
 * @param downloadUrl 下载路径
 * @param fileSaveDir 文件保存目录
 * @param threadNum 下载线程数
 */
public SmartFileDownloader(Context context, String downloadUrl,
    File fileSaveDir, int threadNum) {
    try {
        this.context = context;
        this.downloadUrl = downloadUrl;
        fileService = new DownloadFileService(this.context);
        URL url = new URL(this.downloadUrl);
        if (!fileSaveDir.exists()) fileSaveDir.mkdirs();
        this.threads = new SmartDownloadThread[threadNum];
        HttpURLConnection conn = (HttpURLConnection) url.openConnection();
        conn.setConnectTimeout(5 * 1000);
        conn.setRequestMethod("GET");
        conn.setRequestProperty("Accept", "image/gif, image/jpeg, image/pjpeg, image/pjpeg");
        conn.setRequestProperty("Accept-Language", "zh-CN");
        conn.setRequestProperty("Referer", downloadUrl);
        conn.setRequestProperty("Charset", "UTF-8");
        conn.setRequestProperty("User-Agent", "Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 5.1; Trident/4.0; .NET CLR 3.0.30729.54; .NET CLR 3.5.30729.3; .NET CLR 3.0.04506.686; .NET CLR 2.0.50724.30334)");
        conn.setRequestProperty("Connection", "Keep-Alive");
        conn.connect();
        printResponseHeader(conn);
        if (conn.getResponseCode() == 200) {
            this.fileSize = conn.getContentLength(); // 根据响应获取文件大小
            if (this.fileSize <= 0)
                throw new RuntimeException("Unkown file size ");
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

```
        fileName = getFileName(conn);
        this.saveFile = new File(fileSaveDir, fileName);/* 保存文件 */
        Map<Integer, Integer> logdata = fileService.getData(downloadUrl);
        if (logdata.size() > 0) {
            for (Map.Entry<Integer, Integer> entry : logdata.entrySet()) {
                data.put(entry.getKey(), entry.getValue());
            }
        }
        //划分每个线程下载文件长度
        this.block = (this.fileSize % this.threads.length) == 0 ? this.fileSize / this.threads.length : this.fileSize / this.threads.length + 1;
        if (this.data.size() == this.threads.length) {
            for (int i = 0; i < this.threads.length; i++) {
                this.downloadSize += this.data.get(i + 1);
            }
            print("已经下载的长度" + this.downloadSize);
        }
    } else {
        throw new RuntimeException("server no response ");
    }
} catch (Exception e) {
    print(e.toString());
    throw new RuntimeException("don't connection this url");
}
}

/**
 * 获取文件名
 */
private String getFileName(HttpURLConnection conn) {
    String filename = this.downloadUrl.substring(this.downloadUrl.lastIndexOf('/') + 1);
    if (filename == null || "".equals(filename.trim())) {
        // 如果获取不到文件名称
        for (int i = 0; i < conn.getHeaderFields().size(); i++) {
            String mine = conn.getHeaderField(i);
            print("ConnHeader:" + mine + " ");
            if (mine == null) {
                break;
            }
        }
    }
}
```

```
        if ("content-disposition".equals(conn.getHeaderFieldKey(i).toLowerCase())) {
            Matcher m = Pattern.compile(".*filename=(.*)").matcher(conn.getHeaderField(i));
            if (m.find())
                return m.group(1);
        }
    }

    filename = UUID.randomUUID() + ".tmp";// 默认取一个文件名
}

return filename;
}

/**
 * 开始下载文件
 *
 * @param listener
 *      监听下载数量的变化,如果不需要了解实时下载的数量,可以设置为null
 * @return 已下载文件大小
 * @throws Exception
 */
public int download(SmartDownloadProgressListener listener)
    throws Exception {
    try {
        RandomAccessFile randOut = new RandomAccessFile(this.saveFile, "rw");
        if (this.fileSize > 0)
            randOut.setLength(this.fileSize);
        randOut.close();
        URL url = new URL(this.downloadUrl);
        if (this.data.size() != this.threads.length) {
            this.data.clear();// 清除数据
            for (int i = 0; i < this.threads.length; i++) {
                this.data.put(i + 1, 0);
            }
        }
        for (int i = 0; i < this.threads.length; i++) {
            int downLength = this.data.get(i + 1);
            if (downLength < this.block && this.downloadSize < this.fileSize)
                this.threads[i] = new SmartDownloadThread(this, url,
```

```
        this.saveFile, this.block, this.data.ge
        this.threads[i].setPriority(7);
        this.threads[i].start();
    } else {
        this.threads[i] = null;
    }
}
this.fileService.save(this.downloadUrl, this.data);
boolean notFinish = true; // 下载未完成
while (notFinish) { // 循环判断是否下载完毕
    Thread.sleep(900);
    notFinish = false; // 假定下载完成
    for (int i = 0; i < this.threads.length; i++) {
        if (this.threads[i] != null && !this.threads[i].isFinis
            notFinish = true; // 下载没有完成
            if (this.threads[i].getDownLength() == -1) { //
                this.threads[i] = new SmartDownloadThre
                    url, this.saveFile, thi
                this.threads[i].setPriority(7);
                this.threads[i].start();
            }
        }
    }
    if (listener != null)
        listener.onDownloadSize(this.downloadSize);
}
fileService.delete(this.downloadUrl);
} catch (Exception e) {
    print(e.toString());
    throw new Exception("file download fail");
}
return this.downloadSize;
}

/**
 * 获取Http响应头字段
 *

```

```
* @param http
* @return
*/
public static Map<String, String> getHttpResponseHeader(
    HttpURLConnection http) {
    Map<String, String> header = new LinkedHashMap<String, String>();
    for (int i = 0;; i++) {
        String mine = http.getHeaderField(i);
        if (mine == null)
            break;
        header.put(http.getHeaderFieldKey(i), mine);
    }
    return header;
}

/**
 * 打印Http头字段
 *
 * @param http
 */
public static void printResponseHeader(HttpURLConnection http) {
    Map<String, String> header = getHttpResponseHeader(http);
    for (Map.Entry<String, String> entry : header.entrySet()) {
        String key = entry.getKey() != null ? entry.getKey() + ":" : "";
        print(key + entry.getValue());
    }
}

// 打印日志
private static void print(String msg) {
    Log.i(TAG, msg);
}

public interface SmartDownloadProgressListener {
    public void onDownloadSize(int size);
}
}
```


好了这里只是将主要的代码分享出来，主要是为了了解他的基本流程，然后自己可以扩展，和优化

1.9 带进度和时间的播放器

发表时间: 2011-12-19 关键字: android, ProgressBar, 播放器, 进度

最近由于需要，做了一个音乐播放控制view，在上面需要能

- *控制播放

- *显示剩余时间

- *显示进度（整个view就是一个进度条）

- *实现播放暂停，以及ProgressBar的第一二进度功能

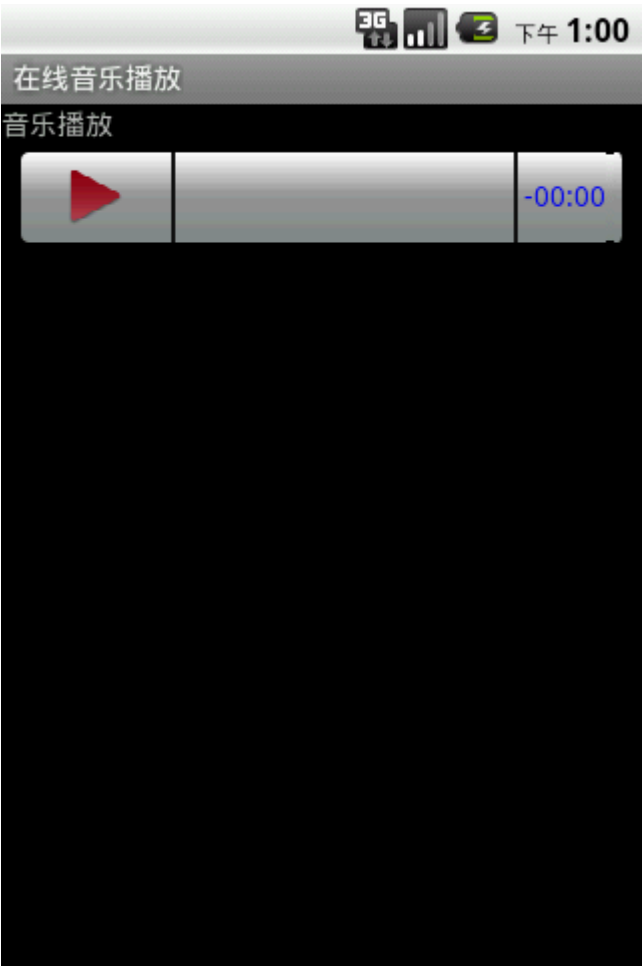
开始想到用组合的方式实现，然后重写ProgressBar的方式实现，但是发现很困难而且文字显示也不行

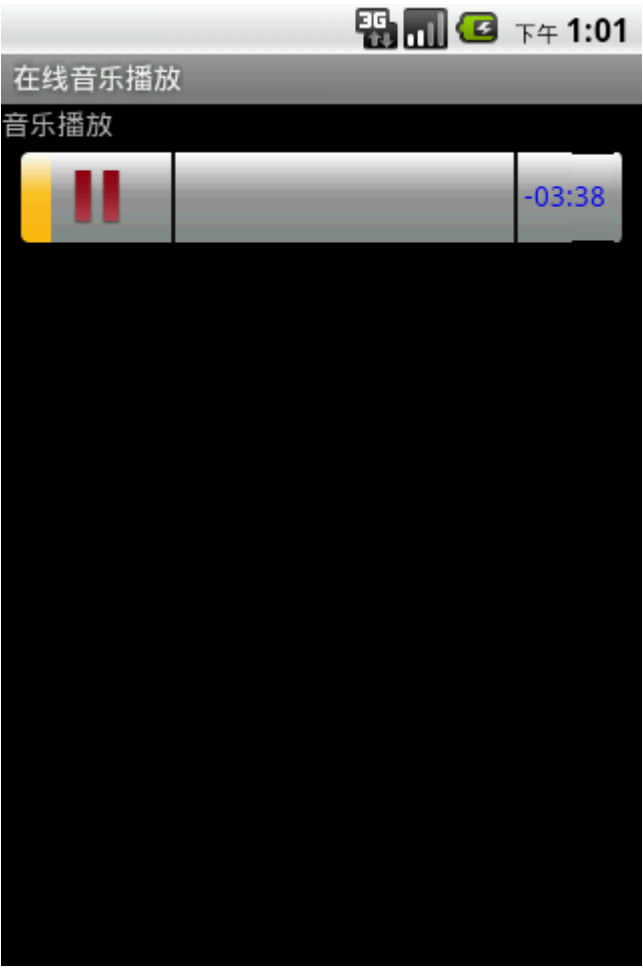
最后只有自己动手写一个新的控制条

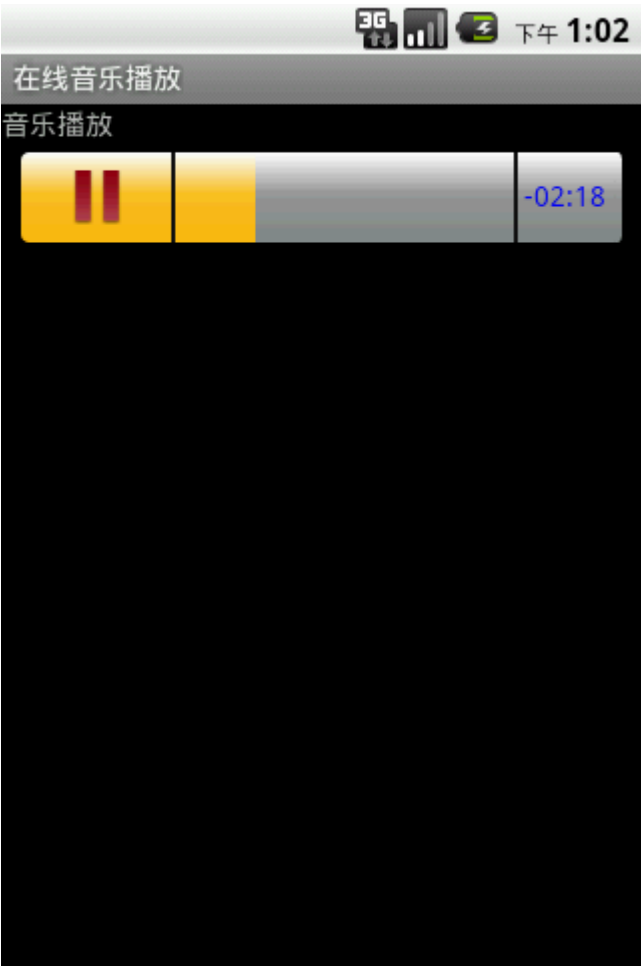
主要的原理就是绘制视图的时候控制onDraw，然后在上面画图片和文字

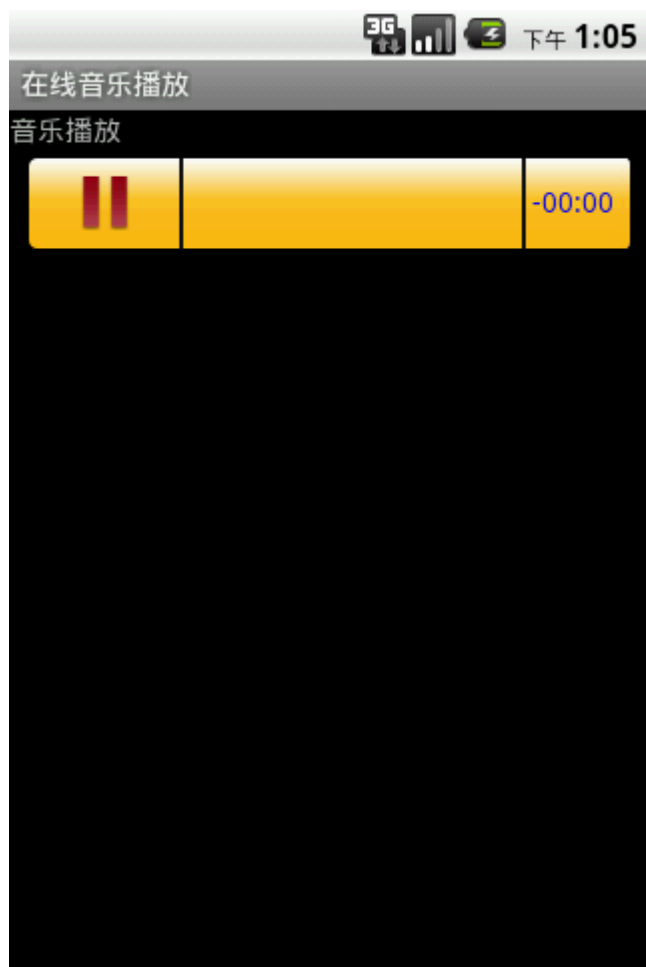
然后在进度变化的时候不断重绘View就可以了

先上图：









其主要的View部分：

```
package com.hao;

import android.content.Context;
import android.graphics.Bitmap;
import android.graphics.BitmapFactory;
import android.graphics.Canvas;
import android.graphics.Color;
import android.graphics.Paint;
import android.graphics.PixelFormat;
import android.graphics.Rect;
import android.graphics.drawable.BitmapDrawable;
import android.graphics.drawable.Drawable;
```

```
import android.util.AttributeSet;
import android.util.Log;
import android.view.MotionEvent;
import android.view.View;
import android.view.View.MeasureSpec;
import android.view.View.OnClickListener;
import android.widget.ImageButton;
import android.widget.ImageView;
import android.widget.TextView;

public class ProgressButton extends View{
    private Bitmap begin , bm_gray, bm_yellow, bm_second, end_gray, end_yellow, line, begin_
    private Bitmap pausePressedImg;
    private Bitmap playPressedImg;
    private int bitmapWidth = 0 , bitmapHeight = 0, btWidth = 0, btHeight = 0;
    private int progress = 0, secondProgress = 0;
    private double perLen = 0, max = 0, maxSize = 0;
    private OnProgressChanged mOnProgressChanged;
    private boolean isPlaying = false;
    private Paint mTextPaint;
    private String time = "00:00";
    private int color = Color.BLUE;

    public ProgressButton(Context context) {
        super(context);
        // TODO Auto-generated constructor stub
        init();
    }

    public ProgressButton(Context context, AttributeSet attrs, int defStyle){
        super(context, attrs, defStyle);
        init();
    }

    public ProgressButton(Context context, AttributeSet attrs){
        super(context, attrs);
        init();
    }
}
```

```
}
```

```
private void init(){
```

```
    begin = drawableToBitmap(getResources().getDrawable(R.drawable.rectangle_left_)
```

```
    begin_gray = drawableToBitmap(getResources().getDrawable(R.drawable.rectangle_l
```

```
    bm_gray = drawableToBitmap(getResources().getDrawable(R.drawable.rectangle_gra
```

```
    bm_yellow = drawableToBitmap(getResources().getDrawable(R.drawable.rectangle_)
```

```
    bm_second = drawableToBitmap(getResources().getDrawable(R.drawable.rectangle_s
```

```
    end_gray = drawableToBitmap(getResources().getDrawable( R.drawable.rectangle_r
```

```
    end_yellow = drawableToBitmap(getResources().getDrawable(R.drawable.rectangle_
```

```
    line = drawableToBitmap(getResources().getDrawable(R.drawable.rectangle_line));
```

```
    pausePressedImg = BitmapFactory.decodeResource(getResources(), R.drawable.pause
```

```
    playPressedImg = BitmapFactory.decodeResource(getResources(), R.drawable.play_t
```

```
    bitmapHeight = begin.getHeight();
```

```
    bitmapWidth = begin.getWidth();
```

```
    btWidth = pausePressedImg.getWidth();
```

```
    btHeight = pausePressedImg.getHeight();
```

```
    mTextPaint = new Paint();
```

```
    mTextPaint.setAntiAlias(true);
```

```
    mTextPaint.setTextSize(14);
```

```
    mTextPaint.setColor(color);
```

```
    setPadding(3, 3, 3, 3);
```

```
}
```

```
public static Bitmap drawableToBitmap(Drawable drawable) {
```

```
    int width = drawable.getIntrinsicWidth();
```

```
    int height = drawable.getIntrinsicHeight();
```

```
    Bitmap bitmap = Bitmap.createBitmap(width, height, drawable
```

```
        .getOpacity() != PixelFormat.OPAQUE ? Bitmap.Config.ARGB_8888
```

```
        : Bitmap.Config.RGB_565);
```

```
    Canvas canvas = new Canvas(bitmap);
```

```
    drawable.setBounds(0, 0, width, height);
```

```
    drawable.draw(canvas);
```

```
    return bitmap;
```

```
}
```



```
@Override
protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {
    // TODO Auto-generated method stub
    Log.e("*****", "onMeasure");
    setMeasuredDimension(measureWidth(widthMeasureSpec),
        measureHeight(heightMeasureSpec));
    perLen = maxSize/max;
}

@Override
protected void onDraw(Canvas canvas) {
    // TODO Auto-generated method stub
    Log.e("*****", "onDraw");
    int middle1 = (int) (progress*perLen), middle2 = (int) (secondProgress*perLen)
    if(progress == 0 && secondProgress == 0){
        //draw background
        canvas.drawBitmap(begin_gray, new Rect(0,0,bitmapWidth,bitmapHeight),
            new Rect(0, 0, bitmapWidth, bitmapHeight), null);
        canvas.drawBitmap(bm_gray, new Rect(0,0,end-middle1,bitmapHeight),
            new Rect(bitmapWidth, 0, end, bitmapHeight), null);
        canvas.drawBitmap(end_gray, new Rect(0,0,4,bitmapHeight),
            new Rect(end, 0, end+4, bitmapHeight), null);
        //draw button and line
        canvas.drawBitmap(playPressedImg, new Rect(0, 0, btWidth, btHeight),
            new Rect(0, 0, btWidth, bitmapHeight), null);
        canvas.drawBitmap(line, new Rect(0, 0, 2, bitmapHeight),
            new Rect(btWidth, 0, btWidth+2, bitmapHeight), null);
        //draw time and line
        if(time.length() == 5){
            canvas.drawBitmap(line, new Rect(0, 0, 2, bitmapHeight),
                new Rect(end - 50, 0, end-48, bitmapHeight), null);
            canvas.drawText("-"+time, end-45, bitmapHeight/2+5, mTextPaint);
        }else{
            canvas.drawBitmap(line, new Rect(0, 0, 2, bitmapHeight),
                new Rect(end - 60, 0, end-58, bitmapHeight), null);
            canvas.drawText("-"+time, end-55, bitmapHeight/2+5, mTextPaint);
        }
    }
}
```

```
    }
}
}else{
    //begin
    canvas.drawBitmap(begin, new Rect(0,0,bitmapWidth,bitmapHeight),
        new Rect(0, 0, bitmapWidth, bitmapHeight), null);
    canvas.drawBitmap(bm_yellow, new Rect(0,0,middle1-bitmapWidth,bitmapHeight),
        new Rect(bitmapWidth, 0, middle1, bitmapHeight), null);
    //middle
    if(secondProgress != 0 && secondProgress > progress){
        canvas.drawBitmap(bm_second, new Rect(0,0,bitmapWidth,bitmapHeight),
            new Rect(middle1, 0, middle2, bitmapHeight), null);
        canvas.drawBitmap(bm_gray, new Rect(0,0,bitmapWidth,bitmapHeight),
            new Rect(middle2, 0, end, bitmapHeight), null);
    }else{
        canvas.drawBitmap(bm_gray, new Rect(0,0,end-middle1,bitmapHeight),
            new Rect(middle1, 0, end, bitmapHeight), null);
    }
    //end
    canvas.drawBitmap(end_gray, new Rect(0,0,4,bitmapHeight),
        new Rect(end, 0, end+4, bitmapHeight), null);
    if(middle2 >= end || middle1 >= end){
        canvas.drawBitmap(end_yellow, new Rect(0,0,4,bitmapHeight),
            new Rect(end, 0, end+4, bitmapHeight), null);
    }
    //draw button
    if(!isPlaying) {
        canvas.drawBitmap(playPressedImg, new Rect(0, 0, btWidth, btHeight),
            new Rect(0, 0, btWidth, bitmapHeight), null);
    }else{
        canvas.drawBitmap(pausePressedImg, new Rect(0, 0, btWidth, btHeight),
            new Rect(0, 0, btWidth, bitmapHeight), null);
    }
    //draw line and time
    canvas.drawBitmap(line, new Rect(0, 0, 2, bitmapHeight),
        new Rect(btWidth, 0, btWidth+2, bitmapHeight), null);
    if(time.length() == 5){
        canvas.drawBitmap(line, new Rect(0, 0, 2, bitmapHeight),
```

```
        new Rect(end - 50, 0, end-48, bitmapHeight), null,
        canvas.drawText("-"+time, end-45, bitmapHeight/2+5, mTextPaint);
    }else{
        canvas.drawBitmap(line, new Rect(0, 0, 2, bitmapHeight),
            new Rect(end - 60, 0, end-58, bitmapHeight), null,
            canvas.drawText("-"+time, end-55, bitmapHeight/2+5, mTextPaint);
    }
}

super.onDraw(canvas);
}

@Override
public boolean onTouchEvent(MotionEvent event) {
    // TODO Auto-generated method stub
    //在这里因为要换按钮，故而需要更新整个视图
    if(event.getAction() == MotionEvent.ACTION_DOWN){
        onClickListener.onClick(this);
        invalidate();
    }
    return true;
}

/**
 * 这个方法必须设置，当播放的时候
 * @param isPlaying
 */
public void setStateChanged(boolean isPlaying){
    this.isPlaying = isPlaying;
}

public void setTextColor(int color){
    this.color = color;
    invalidate();
}
```

```
/**
 * Determines the width of this view
 * @param measureSpec A measureSpec packed into an int
 * @return The width of the view, honoring constraints from measureSpec
 */
private int measureWidth(int measureSpec) {
    int result = 0;
    int specMode = MeasureSpec.getMode(measureSpec);
    int specSize = MeasureSpec.getSize(measureSpec);
    if (specMode == MeasureSpec.EXACTLY) {
        // We were told how big to be
        result = specSize;
    } else {
        result = (int) ((int)max*perLen + getPaddingLeft() + getPaddingRight());
        if (specMode == MeasureSpec.AT_MOST) {
            // Respect AT_MOST value if that was what is called for by measureSpec
            result = Math.min(result, specSize);
        }
    }
    System.out.println("width:"+result);
    maxSize = result;
    return result;
}

/**
 * Determines the height of this view
 * @param measureSpec A measureSpec packed into an int
 * @return The height of the view, honoring constraints from measureSpec
 */
private int measureHeight(int measureSpec) {
    int result = 0;
    int specMode = MeasureSpec.getMode(measureSpec);
    int specSize = MeasureSpec.getSize(measureSpec);

    if (specMode == MeasureSpec.EXACTLY) {
        // We were told how big to be
        result = specSize;
    }
}
```

```
    } else {
        // Measure the text (beware: ascent is a negative number)
        result = (int) getPaddingTop() + getPaddingBottom() + bitmapHeight;
        if (specMode == MeasureSpec.AT_MOST) {
            // Respect AT_MOST value if that was what is called for by measureSpec
            result = Math.min(result, specSize);
        }
    }
    System.out.println("Height:"+result);
    return result;
}

/**
 * set the time
 * @param currentTime 当前播放时间
 * @param totalTime 总播放时间
 */
public void setTime(int currentTime, int totalTime){
    int time = totalTime - currentTime;
    if(time <= 1000){
        this.time="00:00";
        return;
    }
    time/=1000;
    int minute = time/60;
    int hour = minute/60;
    int second = time%60;
    minute %= 60;
    if(hour == 0){
        this.time = String.format("%02d:%02d", minute,second);
    }else{
        this.time = String.format("%02d:%02d:%02d", hour, minute,second);
    }
}

/**
 *
```

```
* @param viewWidth 组件的宽度
*/
public void setMax(int max){
    this.max = max;
}

public int getMax(){
    return (int)max;
}

/**
 * 设置第一进度
 * @param progress
 */
public void setProgress(int progress){
    if(progress>max){
        progress = (int) max;
    }
    else if(progress<0){
        progress = 0;
    }
    if(mOnProgressChanged!=null){
        mOnProgressChanged.onProgressUpdated();
    }
    this.progress = progress;
    invalidate();
}

/**
 * 设置第二进度
 * @param secondProgress
 */
public void setSecondProgress(int secondProgress){
    if(secondProgress>max){
        secondProgress = (int) max;
    }
    else if(secondProgress<0){
```

```
        secondProgress = 0;
    }
    if(mOnProgressChanged!=null){
        mOnProgressChanged.onSecondProgressUpdated();
    }
    this.secondProgress = secondProgress;
    invalidate();
}

/**
 * 设置进度监听器
 * @param mOnProgressChanged
 */
public void setmOnProgressChanged(OnProgressChanged mOnProgressChanged) {
    this.mOnProgressChanged = mOnProgressChanged;
}

public interface OnProgressChanged{
    void onProgressUpdated();
    void onSecondProgressUpdated();
}

@Override
public void setOnClickListener(OnClickListener l) {
    // TODO Auto-generated method stub
    if(l != null) onClickListener = l;
    super.setOnClickListener(l);
}

private View.OnClickListener onClickListener;
}
```

控制非常简单，在这里设置了第一和二进度：

```
package com.hao;

import java.util.Timer;
import java.util.TimerTask;

import com.hao.ProgressView.OnProgressChanged;

import android.app.Activity;
import android.graphics.drawable.Drawable;
import android.os.Bundle;
import android.os.Handler;
import android.os.Message;
import android.view.Gravity;
import android.view.View;
import android.view.View.OnClickListener;
import android.view.ViewGroup.LayoutParams;
import android.widget.ImageButton;
import android.widget.LinearLayout;
import android.widget.SeekBar;
import android.widget.SeekBar.OnSeekBarChangeListener;

public class SeekBarTestActivity extends Activity {
    ProgressBar bp;
    int time = 60000 , currentTime = 0;
    boolean flag = true;
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.my_progress_bar);
        bp = (ProgressBar) findViewById(R.id.pbt);
        bp.setOnClickListener(new OnClickListener() {

            @Override
```



```
        public void onClick(View v) {
            // TODO Auto-generated method stub
            System.out.println("main onck");
            bp.setStateChanged(flag);
            flag = !flag;
        }
    });
    bp.setMax(60000);
    Timer timer = new Timer();
    timer.schedule(new TimerTask() {

        @Override
        public void run() {
            // TODO Auto-generated method stub
            handler.sendMessage(0);
        }
    }, 0, 2000);

}

Handler handler = new Handler(){

    @Override
    public void handleMessage(Message msg) {
        // TODO Auto-generated method stub
        currentTime +=1000;
        bp.setProgress(currentTime);
        bp.setSecondProgress(currentTime+1000);
        bp.setTime(currentTime, time);
    }

};
}
```

布局文件：

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_height="fill_parent"
    android:layout_width="fill_parent"
    android:orientation="vertical">
    <TextView android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="音乐播放"/>
    <LinearLayout
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:paddingTop="5dip"
        android:gravity="center_horizontal">
        <com.hao.ProgressButton
            android:id="@+id/pb1"
            android:layout_width="300dp"
            android:layout_height="45dp"
            android:background="@drawable/button_left_gray_background"/>
        </LinearLayout>
    </LinearLayout>
</LinearLayout>
```

我用到的进度图片都是通过xml定义的在附件中

附件下载:

- [drawable.rar \(13.8 KB\)](#)
- dl.iteye.com/topics/download/e038a327-fa42-379e-a93a-a89dc0936d61

1.10 手机游戏的优化

发表时间: 2012-05-14 关键字: 游戏, 内存重用, 图片优化, android

手机游戏设计中由于设备性能限制，可能出现资源不足的情况，这就需要优化实现技术，游戏的优化有很多的技巧，在我做的这个游戏中使用了以下方法：

1. 关于异常的处理

Java中提供了try/catch来方便用户捕捉异常，进行异常处理。但是如果使用不当，也会给J2ME程序的性能带来影响，所以在程序的编写过程中，应注意以下两点：如果可以使用if，while等逻辑语句来处理，就尽量不使用异常处理；对于必须要进行异常的处理时，尽可能的重用已经存在的异常对象。

2. 图片优化

J2ME的内存杀手无疑图片莫属，一张3kb的png图片可以占用20多kb的内存。对于图片的优化在我做的这个游戏中使用了几种方法：将所有图片存储为8位色，而不是16位或24位；也可以将同一规格的图片合并，做成一张长条的图片，并在导入时使用createImage（）函数裁剪。如原来为130*50的图片，取其左面的一部分
写道

```
playerSprite=new Sprite(Image.createImage("/res/MyPlaneFrames.png"), 65, 50);
```

3. 代码优化

Java的垃圾回收器并非是实时的，从某种角度来讲在J2ME上所有垃圾必须由手工释放，因为简单类型外所有对象都必须显示置空。例如在游戏程序中对于生命面包类对象bread的声明：

```
waterLayer=new TiledLayer(50,200,Image.createImage("/res/backg.png"),32,32);
```

该段代码是先创建了对象然后再进行赋值操作，也就是说在这期间有两个对象同时存在，这就很可能会产生溢出，同时这样做也会妨碍垃圾回收器的工作，较好的编程如下：

写道

```
waterLayer=null; waterLayer=newTiledLayer(50,200,Image.createImage("/res/backg.png"),32,32);
```

把所有对象的初始化放在构造函数中是理所当然的，大多数人通常的做法是把当前逻辑所要用到的资源全部初始化完毕。但是很大一部分的内存溢出都是发生在构造函数中，内存使用的高峰期都是在构造函数中，避开这个高峰期就能有效地防止溢出，所以程序汇总对于变量的初始化尽量在第一次调用时进行。

Java代码访问成员变量比局部变量所需的时间更长。其原因和两种变量的内存如何访问有关。实际上，这意味着在性能关键的时候，你应该使用局部变量而不是成员变量。例如有一个循环重复访问的成员变量，那么，应该考虑在开始循环之前把这个成员变量存储到一个局部变量中。下面是游戏中在一个循环中访问成员变量：

写道

```
for(int i=0;i<MAX_BP;i++){  
if(bigPlane[i].getY()>downLimit-bigPlane[i].getHeight())  
bigPlane[i].setVisible(false);};
```

可以看到，bigPlane对象的成员变量在循环中被访问了MAX_BP次。这段代码的一种快速优化包是把bigPlane[i].getY()设置给一个局部变量，然后在循环中访问这个局部变量，可以在for循环之前定义局部变量String a[]，然后在循环中用a[]代替bigPlane[i].getY。

4. 减少内存的使用

在很多时候，手机内存的限制比设备的有限处理能力的限制要显著的多，因此尽可能地减少手机游戏的内存使用显得及其重要，可以利用一些实际的开发方法来减少一个游戏MIDlet所需的内存。一般情况下，对于减少内存的使用所用的方法是使用对象时重用对象。

对象重用就是重新使用已有的对象，而不是创建一个新的对象。这种方法只有在需要重复使用相同类的对象时才奏效。对象重用避免了不必要的内存分配。例如如果创建一个对象然后终止使用它，java垃圾回收器最终会释放分配给它的内存。如果需要另一个相同的类型的对象并且创建了新的，对象所需要的内存会自动重新分配，那么，可以把最初的对象重新初始化而不再创建一个新的，这要也就重用了对象。

From : <http://gaochaojs.blog.51cto.com/812546/187893>

2.1 Box2d碰撞筛选

发表时间: 2012-04-26 关键字: box2d 碰撞 筛选 过滤 groupindex categorybits maskbits

应用博客：http://blog.sina.com.cn/s/blog_6a2061a20100n0or.html

碰撞筛选就是一个防止某些形状发生碰撞的系统。按照具体需求设置哪些物体跟那些物体发生碰撞，跟哪些物体不发生碰撞。

Box2D通过种群跟组索引支持碰撞筛选。

组索引比较简单，设置其shapeDef的groupIndex值即可，例如boxDef.filter.groupIndex = 1。

通过groupIndex值的正负来确定同一个组的所有形状总是发生碰撞（正）或永远不发生碰撞（负），需要特别注意的是两个不同的付索引是依然会发生碰撞的，例如一个圆的GroupIndex值为-1，一个矩形GroupIndex值为-2。因为不同组索引之间是按照种群跟掩码来筛选的，也就是讲，组索引是有着更高的优先权的。

Box2D支持16个种群，因此我们可以指定任何一个形状属于哪个种群，同时也可以指定这一形状和哪些其它的种群发生碰撞。这一过程就是通过设置其shapeDef的categoryBits值与maskBits值完成的。

categoryBits用于定义自己所属的碰撞种类，maskBits则是指定碰撞种类。

举个例子讲，如果body1的boxDef.filter.categoryBits = 0x0002，body2的boxDef.filter.categoryBits = 0x0004，则如果另外一个body想与他们两个都发生碰撞，则其boxDef.filter.maskBits = 0x0006；简单讲，一个body要与其它种群的body发生碰撞，则其maskBits值应该为其它种群的body的categoryBits之和。

但是同时也不是那么简单，如果三个body的categoryBits分别为0x0001,0x0002,0x0003,那另外一个body的 maskBits值如果是0x0003的话，那它是跟categoryBits值为0x0001和0x0002的两个body碰撞呢，还是单独只跟 categoryBits值0x0003的body发生碰撞呢，亦或是跟三个body都发生碰撞呢？

一个游戏中的种群一般有多少种呢？

现在我们就做一个测试，具体要求：四个刚体。

矩形：只跟自身，圆形，三角形发生碰撞。

圆形：只跟自身，矩形，五边形发生碰撞。

三角形：只跟自身，矩形发生碰撞。

五边形：只跟自身，圆形碰撞。

categoryBits值：矩形[0x0001],圆形[0x0002],三角形[0x0003],五边形[0x0004]。

maskBits值：矩形[1+2+3=6=0x0006]

圆形[1+2+4=7=0x0007]

三角形[1+3=4=0x0004]

五边形[2+4=6=0x0006]

这个碰撞具体会怎样呢？矩形跟五边形竟然一样！6跟7少说也有两种组合方法，还是直接看Demo好了。

结果：

矩形：只跟圆形发生碰撞。6跟2有什么关系？

圆形：只跟自身，矩形，五边形发生碰撞。符合要求。赞一个！

三角形：只跟五边形发生碰撞。4跟4，是不是单一比组合优先？

五边形：只跟自身，圆形，三角形发生碰撞。6 = 2+3+4??我勒个去！

还是先看看正确的把！

一切一切的错误就是categoryBits值有些取值是违规的，categoryBits值是必须为2的倍数的。

即有如下的16个种群：

0x0000 = 0

0x0001 = 1

0x0002 = 2

0x0004 = 4

0x0008 = 8

0x0010 = 16

0x0020 = 32

0x0040 = 64

0x0080 = 128

0x0100 = 256

0x0200 = 512

0x0400 = 1024

0x0800 = 2048

0x1000 = 4096

0x2000 = 8192

0x4000 = 16384

0x8000 = 32768

这样子，6就只能跟2+4配对，8就只能跟8自己配对。

3.1 手机游戏的优化

发表时间: 2012-05-14 关键字: 游戏, 内存重用, 图片优化, android

手机游戏设计中由于设备性能限制，可能出现资源不足的情况，这就需要优化实现技术，游戏的优化有很多的技巧，在我做的这个游戏中使用了以下方法：

1. 关于异常的处理

Java中提供了try/catch来方便用户捕捉异常，进行异常处理。但是如果使用不当，也会给J2ME程序的性能带来影响，所以在程序的编写过程中，应注意以下两点：如果可以使用if，while等逻辑语句来处理，就尽量不使用异常处理；对于必须要进行异常的处理时，尽可能的重用已经存在的异常对象。

2. 图片优化

J2ME的内存杀手无疑图片莫属，一张3kb的png图片可以占用20多kb的内存。对于图片的优化在我做的这个游戏中使用了几种方法：将所有图片存储为8位色，而不是16位或24位；也可以将同一规格的图片合并，做成一张长条的图片，并在导入时使用createImage（）函数裁剪。如原来为130*50的图片，取其左面的一部分
写道

```
playerSprite=new Sprite(Image.createImage("/res/MyPlaneFrames.png"), 65, 50);
```

3. 代码优化

Java的垃圾回收器并非是实时的，从某种角度来讲在J2ME上所有垃圾必须由手工释放，因为简单类型外所有对象都必须显示置空。例如在游戏程序中对于生命面包类对象bread的声明：

```
waterLayer=new TiledLayer(50,200,Image.createImage("/res/backg.png"),32,32);
```

该段代码是先创建了对象然后再进行赋值操作，也就是说在这期间有两个对象同时存在，这就很可能会产生溢出，同时这样做也会妨碍垃圾回收器的工作，较好的编程如下：

写道

```
waterLayer=null; waterLayer=newTiledLayer(50,200,Image.createImage("/res/backg.png"),32,32);
```

把所有对象的初始化放在构造函数中是理所当然的，大多数人通常的做法是把当前逻辑所要用到的资源全部初始化完毕。但是很大一部分的内存溢出都是发生在构造函数中，内存使用的高峰期都是在构造函数中，避开这个高峰期就能有效地防止溢出，所以程序汇总对于变量的初始化尽量在第一次调用时进行。

Java代码访问成员变量比局部变量所需的时间更长。其原因和两种变量的内存如何访问有关。实际上，这意味着在性能关键的时候，你应该使用局部变量而不是成员变量。例如有一个循环重复访问的成员变量，那么，应该考虑在开始循环之前把这个成员变量存储到一个局部变量中。下面是游戏中在一个循环中访问成员变量：

写道


```
for(int i=0;i<MAX_BP;i++){  
if(bigPlane[i].getY()>downLimit-bigPlane[i].getHeight())  
bigPlane[i].setVisible(false);};
```

可以看到，bigPlane对象的成员变量在循环中被访问了MAX_BP次。这段代码的一种快速优化包是把bigPlane[i].getY设置给一个局部变量，然后在循环中访问这个局部变量，可以在for循环之前定义局部变量String a[]，然后在循环中用a[]代替bigPlane[i].getY。

4. 减少内存的使用

在很多时候，手机内存的限制比设备的有限处理能力的限制要显著的多，因此尽可能地减少手机游戏的内存使用显得及其重要，可以利用一些实际的开发方法来减少一个游戏MIDlet所需的内存。一般情况下，对于减少内存的使用所用的方法是使用对象时重用对象。

对象重用就是重新使用已有的对象，而不是创建一个新的对象。这种方法只有在需要重复使用相同类的对象时才奏效。对象重用避免了不必要的内存分配。例如如果创建一个对象然后终止使用它，java垃圾回收器最终会释放分配给它的内存。如果需要另一个相同的类型的对象并且创建了新的，对象所需要的内存会自动重新分配，那么，可以把最初的对象重新初始化而不再创建一个新的，这要也就重用了对象。

From : <http://gaochaojs.blog.51cto.com/812546/187893>

4.1 1000亿以内素数计数算法

发表时间: 2012-06-02 关键字: 算法, 素数, 1000亿

转载自：

是一篇很好的文章，效率相当高，可惜注释少了些，看起来有些恼火

1000亿以内素数计数算法

```

/*****
copyright (C) 2007 Huang Yuanbing
version 1.1, 2007 PrimeNumber
mailto: bailuzhou@163.com1 (remove the last digit 1 for "laji" mail)
free use for non-commercial purposes
*****/

main ideal come from a paper by M.DELEGLISE ADN J.LAGARIAS
"COMPUTING  $\pi(x)$ : THE MEISSEL, LEHMER, LAGARIAS, MILLER, ODLYZKO METHOD"

a =  $\pi(y)$ ; ( $y \gg x^{1/3}$  and  $y \leq x^{1/2}$ )

 $\pi(x)$           =  $\phi(x, a) + a - 1 - P2Xa(x, a)$ ;
 $\phi(x, a)$       =  $S_0 + S$ 
=  $S_0 + S_1 + S_3 + S_2$ 
=  $S_0 + S_1 + S_3 + U + V$ 
=  $S_0 + S_1 + S_3 + U + V_1 + V_2$ 
=  $S_0 + S_1 + S_3 + U + V_1 + W_1 + W_2 + W_3 + W_4 + W_5$ 
it need  $O(n^{2/3})$  space and  $MAXN > 10$  and  $MAXN \leq 1e11$ 
*****/

#include <time.h>
#include <stdio.h>
```

```
#include <math.h>
#include <memory.h>
#include <assert.h>

#define COMP 4
#define MASKN(n) (1 << ((n >> 1) & 7))
#define min(a, b) (a) < (b) ? (a) : (b)
#define MULTI_THREAD
#define THRED_NUMS 4
// #define PRINT_DEBUG

typedef unsigned int uint;
#ifdef _WIN32
typedef __int64 int64;
#else
typedef long long int64;
#endif

int *Prime;
int *PI;

int64 MAXN = (int64)1e11;
int pt[130000]; // MAXN < 1e11, size = np

int *phiTable;
int *minfactor;

int x23, x12, x13, x14, y;
int64 x;

int ST = 7, Q = 1, phiQ = 1;
// Q = 2*3*5*7*11*13 = 30030
// phiQ = 1*2*4*6*10*12 = 5760

int phi(int x, int a)
{
    if (a == ST)
```

```
        return (x / Q) * phiQ + phiTable[x % Q];
    else if (a == 0)
        return x;
    else if (x < Prime[a])
        return 1;
    else
        return phi(x, a - 1) - phi(x / Prime[a], a - 1);
}

#ifdef MULTI_THREAD
struct ThreadInfo{
    int pindex;
    int theadnums;
    int64 pnums;
};Threadparam[2 * THRED_NUMS + 2];

#ifdef _WIN32
#include <windows.h>
DWORD WINAPI WIN32ThreadFun(LPVOID pinfo)
#else
#include <pthread.h>
void* POSIXThreadFun(void *pinfo)
#endif
{
    ThreadInfo *pThreadInfo = (ThreadInfo *) (pinfo);
    int addstep = pThreadInfo->theadnums * 2;

    for (int i = pThreadInfo->pindex; pt[i] != 0; i += addstep){
        if (pt[i] > 0)
            pThreadInfo->pnums -= phi(pt[i], pt[i + 1]);
        else
            pThreadInfo->pnums += phi(-pt[i], pt[i + 1]);
    }
    // printf("ThreadID = %4d, phi(%d) = %d\n", GetCurrentThreadId(), pThreadInfo->pindex, pThreadInfo->pnums);
    // printf("ThreadID = %4d, phi(%d) = %d\n", pthread_self(), pThreadInfo->pindex, pThreadInfo->pnums);
    return 0;
}
```

```
int64 multiThread(int theadnums)
{
    int i;
    int64 pnums = 0;

    assert(theadnums < 2 * THRED_NUMS);

    for (i = 0; i < theadnums; i++){
        Threadparam[i].pindex = 2 * i + 1;
        Threadparam[i].theadnums = theadnums;
        Threadparam[i].pnums = 0;
    }

#ifdef _WIN32
    HANDLE tHand[THRED_NUMS * 2];
    DWORD threadID[THRED_NUMS * 2];
    for (i = 0; i < theadnums; i++){
        tHand[i] = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)WIN32ThreadFun, (LPVOID>(&Threadparam[i]), 0));
        if (tHand[i] == NULL)
            printf("create Win32 thread error\n");
    }
    WaitForMultipleObjects(theadnums, tHand, true, INFINITE);
    for (i = 0; i < theadnums; i++){
        pnums += Threadparam[i].pnums;
        CloseHandle(tHand[i]);
    }
#else
    pthread_t tid[THRED_NUMS];
    for (i = 0; i < theadnums; i++){
        int error = pthread_create(&tid[i], NULL, POSIXThreadFun, &Threadparam[i]);
        if (error != 0)
            printf("Create pthread error: %d\n", error);
    }
    for (i = 0; i < theadnums; i++){
        pthread_join(tid[i], NULL);
        pnums += Threadparam[i].pnums;
    }
#endif
}
```

```
    }
#endif
    return pnums;
}
#endif

int freememory(int alloc)
{
    int memsize = (x / y) + 100;
    if (alloc == 0){
        int psize = (int) (memsize / log(memsize));
        //printf("psize = %d memeszie = %d\n", psize, (int)(psize * 1.2) );
        PI = new int [memsize + 100];
        Prime = new int[(int)(psize * 1.2) + 100];
        assert(PI && Prime);
    }else{
        delete phiTable;
        delete minfactor;
        delete Prime;
        delete PI;
    }
    return alloc;
}

void init_phiTable( )
{
    clock_t start = clock();

    int p, i, j;

    if (x < 1e10)
        ST = 6;
    if (ST > PI[y])
        ST = PI[y];

    for (i = 1; i <= ST; ++i){
        Q *= Prime[i];
    }
}
```

```
        phiQ *= Prime[i] - 1;
    }

    phiTable = new int[Q + 10];
    for (i = 0; i < Q; ++i)
        phiTable[i] = i;

    for (i = 1; i <= ST; ++i){
        p = Prime[i];
        for (j = Q - 1; j >= 0; --j)
            phiTable[j] -= phiTable[j / p];
    }

    printf("    Q = %d, PhiQ = %d\n", Q, phiQ);
    printf("    init_phiTable time = %ld ms\n", clock() - start);
}

void init_minFactor( )
{
    clock_t start = clock();

    int i, j, maxy = y + 10;
    int sqrt = (int)sqrt(maxy) + 1;

    minfactor = new int[maxy + 10];
    for (i = 0; i < maxy; i++)
        minfactor[i] = i;

    for (i = 1; Prime[i] <= maxy; i++){
        for (j = Prime[i]; j <= maxy; j += Prime[i]){
            if (minfactor[j] == -j || minfactor[j] == j)
                minfactor[j] = -Prime[i];
            else
                minfactor[j] = -minfactor[j];
        }
    }
}
```

```
for (i = 1; Prime[i] <= sqrt; i++){
    int powp = Prime[i] * Prime[i];
    for (j = powp; j <= maxy; j += powp)
        minfactor[j] = 0;
}
printf("    init_minFactor time = %ld ms\n", clock() - start);
}

int sieve( )
{
    clock_t start = clock();

    int primes = 1;
    int maxp = (x / y) + 10;
    Prime[1] = 2;

    unsigned char *bitMask = (unsigned char *) PI;
    memset(bitMask, 0, (maxp + 64) >> COMP);

    for (int p = 3; p < maxp; p += 2){
        if ( !(bitMask[p >> COMP] & MASKN(p)) ){
            Prime[++primes] = p;
            if (p > maxp / p)
                continue;
            for (int j = p * p; j < maxp; j += p << 1)
                bitMask[j >> COMP] |= MASKN(j);
        }
    }

    Prime[0] = primes;
    Prime[primes] = 0;
    printf("pi(%d) = %d\n", maxp, primes);

    printf("    sieve time = %ld ms\n", clock() - start);
    return primes;
}
```



```
void init_x( )
{
    x = (int64)MAXN;
    x23 = (int)(pow(x, 2.0 / 3) + 0.01);
    x12 = (int)(pow(x, 1.0 / 2) + 0.01);
    x13 = (int)(pow(x, 1.0 / 3) + 0.01);
    x14 = (int)(pow(x, 1.0 / 4) + 0.01);

    assert((int64)x12 * x12 <= x);
    assert((int64)x13 * x13 * x13 <= x);
    assert((int64)x14 * x14 * x14 * x14 <= x);

    y = x13;

    printf("    x14 = %d, x13 = %d, x12 = %d x23 = %d, y = %d\n", x14, x13, x12, x23, y);

    freememory(0);
}

int64 cal_S0( )
{
    int64 S0 = x;
    for (int j = 2; j <= y; j++){
        if (minfactor[j] > 0)
            S0 += x / j;
        else if (minfactor[j] < 0)
            S0 -= x / j;
    }

    printf("S0 = %I64d\n", S0);

    return S0;
}

// so bad performance in this function
int64 cal_S3( )
{
```

```
clock_t start = clock();

int i, p, a;
int np = 1;
int64 S3 = 0;

for (i = 1; i <= PI[x14]; i++){
    p = Prime[i];
    a = PI[p] - 1;
#ifdef PRINT_DEBUG
    assert(p <= x14 && a >= 0);
#endif
    for (int m = y / p + 1; m <= y; m++){
        int xx = x / (int64)(m * p);
#ifdef MULTI_THREAD
        if (minfactor[m] > p)
            S3 -= phi(xx, a);
        else if (minfactor[m] < -p)
            S3 += phi(xx, a);
#else
        if (minfactor[m] > p){
            pt[np++] = xx;
            pt[np++] = a;
        }
        else if (minfactor[m] < -p){
            pt[np++] = -xx;
            pt[np++] = a;
        }
#endif
    }
}

#ifdef MULTI_THREAD
    printf("np = %d\n", np);
    if (np < 100)
        S3 = multiThread(1);
    else
```

```
S3 = multiThread(THRED_NUMS);
#endif

printf("S3 = %I64d\n", S3);
printf("    cal S3 time = %ld ms\n", clock() - start);
return S3;
}

void init_PI( )
{
    clock_t start = clock();

    int Px = 1;
    PI[0] = PI[1] = 0;
    for (int i = 1; Prime[i]; i++, Px++){
        for (int j = Prime[i]; j <= Prime[i + 1]; j++)
            PI[j] = Px;
    }

    //printf("    Px = %d, primes = %d\n", Px, Prime[0]);
    printf("    PI[%d] = %d\n", Px, PI[Px - 1]);
    printf("    init_PI time = %ld ms\n", clock() - start);
}

int64 cal_P2xa( )
{
    int64 phi2 = (int64)PI[y] * (PI[y] - 1) / 2 - (int64)PI[x12] * (PI[x12] - 1) / 2;

    for (int i = PI[y] + 1; i <= PI[x12] + 0; i++){
        int p = Prime[i];
#ifdef PRINT_DEBUG
        assert(p > y && p <= x12);
#endif
        phi2 += PI[x / p];
    }
    printf("P2xa(%I64d, %d) = %I64d\n", x, PI[y], phi2);
}
```

```
    return phi2;
}

int64 cal_S1( )
{
    int64 temp = PI[y] - PI[x13];
    int64 S1 = temp * (temp - 1) / 2;
    printf("S1 = %I64d\n", S1);

    return S1;
}

int64 cal_U( )
{
    int64 p, U = 0;
    int sqrt_xdivy = (int)sqrt(x / y);
    for (int i = PI[sqrt_xdivy] + 1; i <= PI[x13]; i++){
        p = Prime[i];
        U += PI[y] - PI[x / (p * p)];
    }

    printf("U = %I64d\n", U);

    return U;
}

int64 cal_V1( )
{
    int64 V1 = 0;
    int MINq, p;
    for (int i = PI[x14] + 1; i <= PI[x13]; i++){
        p = Prime[i];
        MINq = min((uint)y, x / ((int64)p * p));
#ifdef PRINT_DEBUG
        assert(p > x14 && p <= x13 && MINq >= p);
#endif
        V1 += (int64)(2 - PI[p]) * (PI[MINq] - PI[p]); //!!!!!!
    }
```

```
}

printf("V1 = %I64d\n", V1);

return V1;
}

int64 cal_V2( )
{
    int64 V2 = 0;
    int i, sqrt_xdivy = (int)sqrt(x / y);
    //    int sqrt_xdivy2 = (int)sqrt(x / (y * y));
    //    int sqrt_xdivp = (int)sqrt(x / 1);
    int xdivy2 = x / (y * y);

#if 0
    uint p, q;
    uint W[6] = {0};
    //W1
    for (p = x14 + 1; p <= xdivy2; p++)
        for (q = p + 1; q <= y; q++)
            W[1] += PI[x / (p * q)];

    //W2
    for (p = xdivy2 + 1; p <= sqrt_xdivy; p++){
        sqrt_xdivp = (uint)sqrt(x / p);
        for (q = p + 1; q <= sqrt_xdivp; q++)
            W[2] += PI[x / (p * q)];
    }

    //W3
    for (p = xdivy2 + 1; p <= sqrt_xdivy; p++)
        for (q = sqrt(x / p) + 1; q <= y; q++)
            W[3] += PI[x / (p * q)];

    //W4
    for (p = sqrt_xdivy + 1; p <= x13; p++){
```

```
    sqrt_xdivp = (uint)sqrt(x / p);
    for (q = p; q <= sqrt_xdivp; q++)
        W[4] += PI[x / (p * q)];
}

//W5
for (p = sqrt_xdivy + 1; p <= x13; p++)
    for (q = sqrt(x / p) + 1; q <= xdivy2; q++)
        W[5] += PI[x / (p * q)];
V2 = W[1] + W[2] + W[3] + W[4] + W[5];

#else

    for (i = PI[x14] + 1; i <= PI[sqrt_xdivy]; i++){
        int64 p = Prime[i];
#ifdef PRINT_DEBUG
        assert(p > x14 && p <= sqrt_xdivy);
#endif
        for (int j = PI[p] + 1; j <= PI[y]; j++){
            int q = Prime[j];
#ifdef PRINT_DEBUG
            assert(q > p && q <= y);
#endif
            V2 += PI[x / (p * q)];
        }
    }

    for (i = PI[sqrt_xdivy] + 1; i <= PI[x13]; i++){
        int64 p = Prime[i];
#ifdef PRINT_DEBUG
        assert(p > sqrt_xdivy && p <= x13);
#endif
        xdivy2 = x / (p * p);
        for (int j = PI[p] + 1; j <= PI[xdivy2]; j++){
            int q = Prime[j];
            V2 += PI[x / (p * q)];
        }
    }
}
```

```
    }

#endif

    printf("V2 = %I64d\n", V2);

    return V2;
}

int main(int argc, char* argv[])
{
    clock_t start = clock();

    if (argc > 1)
        MAXN = atoll(argv[1]); //mingw

    init_x( );

    sieve( );

    init_PI( );

    init_minFactor( );
    init_phiTable( );

    int64 pix = PI[y] - 1;
    pix += cal_S3( ); //mul thread

    pix -= cal_P2xa( );
    pix += cal_S0( );

    if (y != x13){
        pix += cal_S1( );
        pix += cal_U( );
    }
}
```

```
    pix += cal_V2( );
    pix += cal_V1( );

#if 0
    printf("phi(%u, 209) = %d, time = %ld ms\n", x, phi(x, 209), clock() - start);
    delete phiTable;
    return 0;
#endif

    puts("\nPi(x) = phi(x, a) + a - 1 - phi2(x,a):");

    printf("pi(%I64d) = %I64d, time use %ld ms\n", x, pix, clock() - start);

    freememory(1);
    return 0;
}

//benchmark, Windows Xp SP2, 1G, Mingw G++ 3.4.5
// pi( 2147483647) = 105097565, time use 108 ms (machine AMD 3600+ 2.0G)
// pi(100000000000) = 4118054813, time use 2171 ms (machine intel PD 940 3.2G)
```

最后顺便附上论文。。。。。。

下面在CSDN上面还有一篇比较细致的文章分析素数：

<http://blog.csdn.net/hexiios/article/details/4400068>

也贴在下面，大家观摩

问题描述：寻找素数

求小于自然数N的所有素数。

解决方案

程序 1-1 经典算法

经典的素数判定算法是这样：给定一个正整数 n ，用2到 \sqrt{n} 之间的所有整数去除 n ，如果可以整除，则 n 不是素数，如果不可以整除，则 n 就是素数。所以求小于 N 的所有素数程序如下：

```
#include <stdio.h>
#include <stdlib.h>

#define N 1000000

int main(int argc, char *argv[])
{
    int m, n;
    for (n = 2; n < N; n++)
    {
        for (m = 2; m * m <= n; m++)
            if (n % m == 0) break;
        if (m * m > n) printf("%6d", n);
    }
    system("PAUSE");
    return 0;
}
```

算法的时间复杂度是 $O(N \cdot \sqrt{N})$ ，求解规模较小的输入，尚且没有问题。但对于规模较大的 N ，算法就力不从心了。有一种算法叫厄拉多塞筛（sieve of Eratosthenes），它在求解时具有相当高的效率，但是要牺牲较多的空间。

程序 1-2 厄拉多塞筛算法

这个程序的目标是，若 i 为素数，则设置 $a[i] = 1$ ；如果不是素数，则设置为0。首先，将所有的数组元素设为1，表示没有已知的非素数。然后将已知为非素数（即为已知素数的倍数）的索引对应的数组元素设置为0。如果将所有较小素数的倍数都设置为0之后， $a[i]$ 仍然保持为1，则可判断它是所找的素数。

```
#include <stdio.h>
#include <stdlib.h>

#define N 1000000

int a[N];

int main(int argc, char *argv[])
{
}
```

```
int i, j;
for (i = 2; i < N; i++) a[i] = 1;
for (i = 2; i < N; i++)
{
    if (a[i])
        for (j = i + i; j < N; j += i)
            a[j] = 0;
}
for (i = 2; i < N; i++)
    if (a[i]) printf("%6d ", i);
system("PAUSE");
return 0;
}
```

例如，计算小于32的素数，先将所有数组项初始化为1（如下第二列），表示还未发现非素数的数。接下来，将索引为2、3、5倍数的数组项设置成0，因为2、3、5倍数的数是非素数。保持为1的数组项对应索引为素数（如下最右列）。

i	2	3	5	a[i]
2	1			1
3	1			1
4	1	0		
5	1			1
6	1	0		
7	1			1
8	1	0		
9	1		0	
10	1	0		
11	1			1
12	1	0		
13	1			1
14	1	0		
15	1		0	
16	1	0		
17	1			1
18	1	0		
19	1			1
20	1	0		
21	1		0	
22	1	0		
23	1			1
24	1	0		
25	1		0	
26	1	0		
27	1		0	
28	1	0		
29	1			1
30	1	0		
31	1			1

如何理解厄拉多塞筛算法呢？

我们一定记得小学时候就学过的素数和合数的性质：任何一个合数一定可以表示成若干个素数之积。如： $4 = 2 * 2$ ， $6 = 2 * 3$ ， $12 = 2 * 2 * 3$ 。也就是说，合数N一定是小于N的某个（或若干）素数的整数倍，反之，如果N不是任何比它小的素数的倍数，则N必然是素数。

程序 1-3 经典算法的改进版本

经典素数判定算法中，并不需要用2到 \sqrt{n} 之间的所有整数去除n，只需要用其间的素数就够了，原因也是合数一定可以表示成若干个素数之积。算法的改进版本如下：

```
#include <stdio.h>
#include <stdlib.h>

#define N 1000000

int prime[N];

int main(int argc, char *argv[])
{
    int i, n, flag;
    prime[0] = 0; //保存素数的总个数
    for (n = 2; n < N; n++)
    {
        flag = 1;
        for (i = 1; i <= prime[0] && prime[i] * prime[i] <= n; i++)
            if (n % prime[i] == 0)
            {
                flag = 0;
                break;
            }
        if (flag)
        {
            prime[0]++;
            prime[prime[0]] = n;
            printf("%6d ", n);
        }
    }
    system("PAUSE");
    return 0;
}
```

算法虽然在效率下，比先前版本有所提高，但需要牺牲空间记住已确定的素数。

程序 1-4 厄拉多塞筛算法的改进版

程序1-2使用一个数组来包含最简单的元素类型，0和1两个值，如果我们使用位的数组，而不是使用整数的数组，则可获得更高的空间有效性。

```
#include <stdio.h>
#include <stdlib.h>

#define N 1000000
#define BITSPERWORD 32
#define SHIFT 5
#define MASK 0x1F
#define COUNT 1+N/BITSPERWORD //Bit i 对映数i

void set(int a[],int i);
void clr(int a[],int i);
int test(int a[],int i);

int a[COUNT];

int main(int argc, char *argv[])
{
    int i, j;
    for (i = 0; i < COUNT; i++) a[i] = -1;
    for (i = 2; i < N; i++) {
        if (test(a,i))
            for (j = i + i; j < N; j += i)
                clr(a,j);
    }
    for (i = 2; i < N; i++)
        if (test(a,i)) printf("%4d ",i);
    system("PAUSE");
    return 0;
}

void set(int a[],int i) {
```

```
a[i >> SHIFT] |= (1<<(i & MASK));  
}  
  
void clr(int a[],int i) {  
    a[i >> SHIFT] &= ~(1<<(i & MASK));  
}  
  
int test(int a[],int i) {  
    return a[i >> SHIFT] & (1<<(i & MASK));  
}
```

附件下载:

- [COMPUTING_PI_x__THE_MEISSEL__LEHMER__LAGARIAS__MILLER__ODLYZKO_METHOD.pdf](#)
(271.8 KB)
- dl.iteye.com/topics/download/47bf35fa-4c97-3282-82f9-579dfafb7b28

4.2 素数的求解逐步改进

发表时间: 2012-06-02 关键字: 素数, 改进, 厄拉多塞筛算法

我的注释都写在代码里面了，就不在赘述了！如果有任何疑问欢迎留言

参考博客：

1.位操作总结：<http://blog.csdn.net/morewindows/article/details/7354571>

2.找素数算法总结：<http://blog.csdn.net/hexiios/article/details/4400068>

非常感谢上面两篇博客的仁兄，致谢！

尤其是读了位操作的那篇文章，写的非常好，希望各位都可以一次尝试哈，没准给你的程序增加一个亮点

2的总结也很好，我只是在它的基础上在详细的给了些注释，如果有不懂的地方就好理解了！

1.基础补习

什么是合数？什么是素数？百度一下你就知道

有一个重要的性质:任何一个合数一定可以表示成若干个素数之积。

如： $4 = 2 * 2$ ， $6 = 2 * 3$ ， $12 = 2 * 2 * 3$ 。也就是说，合数N一定是小于N的某个（或若干）素数的整数倍，反之，如果N不是任何比它小的素数的倍数，则N必然是素数。

这个性质就决定了下面的算法基本思想

2.检查是否是素数（这个是从基本概念出发的检测算法）

主要的改进就是检测的范围缩小从0---sqrt(n)

```
bool isPrime(int n){
    if(n ==0 || n==1) return 0;
    for(int i=2; i<=sqrt(n); i++){
        if(n%i == 0){
            cout<<i<<endl;
            return 0;
        }
    }
    return 1;
}
```

3.经典素数求解算法

它对每个数的检测就是利用2的方法

```
/**
 *经典素数求解算法
 *prime[0]中保存素数总个数
 */
int* getPrime(){
    int prime[LEN] = {0}; //判断是否为素数,初始都为素数
    prime[0] = 0; //保存素数的总个数
    for(int n=2; n<LEN; n++){
        bool flag = 1;
        //对每个数A都按照：2到sqrt(A)依次除，看是否整除，如果能则不是素数
        //这样是很低效的m*m<=n也可写成m<=sqrt(n)
        //注意：必须有m*m<=n 中的 "="
        for(int m=2; m*m<=n; m++){
            //如果能整除，不是素数
            if(n%m == 0){
                flag = 0;
                break;
            }
        }
    }
}
```

```
        }
        if(flag){
            prime[0]++;
            prime[prime[0]] = n;
        }
    }
    cout<<"\ncount:"<<prime[0]<<endl;
    return prime;
}
```

4.经典算法改进版

其主要的改进的地方就是，检测的思想，前面的检测方法就是2到sqrt(A)依次除，看是否整除

改进的就不同了，它利用了合数的性质，利用已经找到的素数来除，这样大大的减少了需要整除的次数，这个改进不错！

```
/**
 *它是上面经典算法改进版
 *这里prime数组中存的是素数,prime[0]中保存素数总个数
 *获取0-n的所有素数
 */
int* getPrime1(){
    int prime[LEN] = {0}; //判断是否为素数,初始都为素数
    int i, n, flag;
    prime[0] = 0; //保存素数的总个数
    for (n = 2 ; n < LEN; n++)
    {
        flag = 1;
        //判断当前n是否被"已经找出的素数"整除（如果能，说明不是素数,根据合数性质）
        //使用条件prime[i] * prime[i] <= n，就是判断一个数A是否是素数的时候，不需要
        //从2-A的所有数都去看是否能整除，只需要看2-sqrt(A)即可，即最终整除位置为sqrt(A),见上面isPrime
        //这里是用prime[i]去整除，故而只需要prime[i]*prime[i]<=n也可写成prime[i]<=sqrt(n)
    }
}
```



```
for (i = 1; i <= prime[0] && prime[i] * prime[i] <= n; i++)
    //如果不是素数, break, flag=0
    if (n % prime[i] == 0)
    {
        flag = 0;
        break;
    }
//flag=1, 则说明是素数, 存储
if (flag)
{
    prime[0]++; //素数个数
    prime[prime[0]] = n; //将确定的素数放入数组
}
}
cout<<"\ncount:"<<prime[0]<<endl;
return prime;
}
```

5. 厄拉多塞筛算法

所谓厄拉多塞筛算法就是：利用一个空间换时间的思想，利用一个辅助数组来存储素数的位置，然后利用的是筛选法，筛选出素数的倍数

那你就会发出一个疑问？只是筛选出素数的倍数，那是不是还有遗漏啊？这个就看看1的性质吧，即所有合数，都可以分解成若干个素数。

*首先，将所有的数组元素设为0，表示没有已知的非素数。

*然后，将已知为非素数（即为已知素数的倍数）的索引对应的数组元素设置为1。

*如果将所有较小素数的倍数都设置为1之后，a[i]仍然保持为0，则可判断它是所找的素数。

```
/**
 *厄拉多塞筛算法:prime[0]中保存素数总个数
 */
int* getPrime2(){
    int count = 0;
    int primes[LEN / 3] = {0}, pi=1;
    int flag[LEN] = {0}; //判断是否为素数,初始都为素数
    //0,1不是素数
    flag[0]=1;
    flag[1]=1;
    for(int i=2; i<LEN; i++){
        //倍数必然不是素数,把不是素数的都筛选出来
        if(!flag[i]){
            primes[pi++] = i;
            // 注意:这里只用素数来筛,因为合数筛时肯定被素数筛过了,所以没必要.
            //为什么?因为合数的性质:任何一个合数一定可以表示成若干个素数之积.如:4 = 2
            //即所有合数,都可以分解成若干个素数.
            for(int j=i; j<LEN; j+=i){
                //为什么9624会出现在素数中?
                //if(j == 9624) cout<<"-----"<<i<<" "<<j<<endl;
                flag[j]=1;
                count++;
            }
        }
    }
    //从count可以看出,对于一个位置重复访问了许多次,可以优化
    cout<<"访问数组总数:"<<count<<endl;
    primes[0] = pi - 1;
    cout<<"count:"<<primes[0]<<endl;
    return primes;
}
```

6, 厄拉多塞筛算法改进版

由于对于5来说,最大的缺憾就是利用了辅助数组,那么我们可以减少空间,反正都只是存0,1,为什么不用位来存储呢?这样就大大的减少了空间的浪费了!

首先看看一个为操作的事例:

```
/**
 *用于位测试
 *
 */
#include <iostream>
#include <cstdio>
using namespace std;

int main(){
    //在数组中在指定的位置上写1
    int b[5] = {0}; //占用连续的20个字节空间(每个int占用4字节),每个字节8位(1Byte=8bit)
    cout<<sizeof(b)<<" "<<sizeof(int)<<endl;
    int i;
    //在第i个位置上写1(一共有20*8=160位,现在之使用了40位)
    for (i = 0; i < 40; i += 3)
        //每int占用4字节即32位,当b[0]的32位用完,就b[1]
        b[i / 32] |= (1 << (i % 32));
    //输出整个bitset
    for (i = 0; i < 40; i++)
    {
        //将数组依次右移一位,然后和1求与,判断该位是0还是1
        if ((b[i / 32] >> (i % 32)) & 1)
            putchar('1');
        else
            putchar('0');
    }
    putchar('\n');
    return 0;
}
```

```
    return 0;
}
```

在下面的算法中就是利用了上面的操作，只要去看看我第一个链接，里面更详细！

```
/**
 *厄拉多塞筛算法改进版:prime[0]中保存素数总个数
 */
int* getPrime3(){
    int primes[LEN / 3] = {0}, pi=1; //实际素数的个数最多就是LEN / 3,故而没用LEN
    //用位替代int prime[LEN] = {0}; //判断是否为素数,初始都为素数
    int flag[LEN/32] = {0}; //每个int是4字节=32bit (32位) 可以存储32个0,1, 故而只需要LEN/32个长
    for(int i=2; i<LEN; i++){
        //倍数必然不是素数, 把不是素数的都筛选出来
        if(!((flag[i/32]>>(i%32))&1)){
            primes[pi++] = i;
            for(int j=i; j<LEN; j+=i){
                //将第j位设置为1, 表示不是素数
                flag[j/32] |= (1<<(j%32));
            }
        }
    }
    primes[0] = pi - 1;
    cout<<"count:"<<primes[0]<<endl;
    return primes;
}
```

7.时间复杂度分析

10万个数：这个数来测试有些小，可以大些，这样效果比较明显

时间花费依次：62ms, 15ms, 0ms, 0ms

当然最好两个是0ms这个大些才能看出来，非常明显的效果提升

8.所有代码

```
/**
 *说明：
 *有一个重要的性质:任何一个合数一定可以表示成若干个素数之积。
 *如：4 = 2 * 2 , 6 = 2 * 3 , 12 = 2 * 2 * 3。也就是说，
 *合数N一定是小于N的某个（或若干）素数的整数倍,反之，如果N不是任何比它小的素数的倍数，
 *则N必然是素数。
 */

* Author: Blakequ@gmail.com
* Data   : 2012-06-02 22:20
*
*
*/

#include <iostream>
#include <cmath>
#include <ctime>

const int LEN = 100000;
using namespace std;

//检查n是不是素数
bool isPrime(int n){
    if(n == 0 || n == 1) return 0;
    for(int i=2; i<=sqrt(n); i++){
        if(n%i == 0){
            cout<<i<<endl;
            return 0;
        }
    }
    return 1;
}
```

```
/**
 *经典素数求解算法
 *prime[0]中保存素数总个数
 */
int* getPrime(){
    int prime[LEN] = {0}; //判断是否为素数,初始都为素数
    prime[0] = 0; //保存素数的总个数
    for(int n=2; n<LEN; n++){
        bool flag = 1;
        //对每个数A都按照: 2到sqrt(A)依次除, 看是否整除, 如果能则不是素数
        //这样是很低效的 $m*m \leq n$ 也可写成 $m \leq \sqrt{n}$ 
        //注意: 必须有 $m*m \leq n$  中的 "="
        for(int m=2; m*m<=n; m++){
            //如果能整除, 不是素数
            if(n%m == 0){
                flag = 0;
                break;
            }
        }
        if(flag){
            prime[0]++;
            prime[prime[0]] = n;
        }
    }
    cout<<"\ncount:"<<prime[0]<<endl;
    return prime;
}

/**
 *它是上面经典算法改进版
 *这里prime数组中存的是素数, prime[0]中保存素数总个数
 *获取0-n的所有素数
 */
int* getPrime1(){
    int prime[LEN] = {0}; //判断是否为素数,初始都为素数
    int i, n, flag;
```

```
prime[0] = 0; //保存素数的总个数
for (n = 2 ; n < LEN; n++)
{
    flag = 1;
    //判断当前n是否被"已经找出的素数"整除（如果能，说明不是素数,根据合数性质）
    //使用条件prime[i] * prime[i] <= n，就是判断一个数A是否是素数的时候，不需要
    //从2-A的所有数都去看是否能整除，只需要看2-sqrt(A)即可，即最终整除位置为sqrt(A),见上面isPrime
    //这里是用prime[i]去整除，故而只需要prime[i]*prime[i]<=n也可写成prime[i]<=sqrt(n)
    for (i = 1; i <= prime[0] && prime[i] * prime[i] <= n; i++)
        //如果不是素数，break，flag=0
        if (n % prime[i] == 0)
        {
            flag = 0;
            break;
        }
    //flag=1，则说明是素数，存储
    if (flag)
    {
        prime[0]++; //素数个数
        prime[prime[0]] = n; //将确定了的素数放入数组
    }
}
cout<<"\ncount:"<<prime[0]<<endl;
return prime;
}
```

//-----下面两个算法是典型的以空间换时间，不过空间改进即可-----

/**

- *厄拉多塞筛算法:prime[0]中保存素数总个数
- *首先，将所有的数组元素设为0，表示没有已知的非素数。
- *然后将已知为非素数（即为已知素数的倍数）的索引对应的数组元素设置为1。
- *如果将所有较小素数的倍数都设置为1之后，a[i]仍然保持为0，
- *则可判断它是所找的素数。

*/

```
int* getPrime2(){
    int count = 0;
```

```
int primes[LEN / 3] = {0}, pi=1;
int flag[LEN] = {0}; //判断是否为素数,初始都为素数
//0,1不是素数
flag[0]=1;
flag[1]=1;
for(int i=2; i<LEN; i++){
    //倍数必然不是素数,把不是素数的都筛选出来
    if(!flag[i]){
        primes[pi++] = i;
        // 注意:这里只用素数来筛,因为合数筛时肯定被素数筛过了,所以没必要.
        //为什么?因为合数的性质:任何一个合数一定可以表示成若干个素数之积。如:4 = 2
        //即所有合数,都可以分解成若干个素数。
        for(int j=i; j<LEN; j+=i){
            //为什么9624会出现在素数中?
            //if(j == 9624) cout<<"-----"<<i<<" "<<j<<endl;
            flag[j]=1;
            count++;
        }
    }
}
//从count可以看出,对于一个位置重复访问了许多次,可以优化
cout<<"访问数组总数:"<<count<<endl;
primes[0] = pi - 1;
cout<<"count:"<<primes[0]<<endl;
return primes;
}

/**
 *厄拉多塞筛算法改进版:prime[0]中保存素数总个数
 *使用一个数组来包含最简单的元素类型,0和1两个值,
 *如果我们使用位的数组,而不是使用整数的数组,则可获得更高的空间有效性
 */

int* getPrime3(){
    int primes[LEN / 3] = {0}, pi=1; //实际素数的个数最多就是LEN / 3,故而没用LEN
    //用位替代int prime[LEN] = {0}; //判断是否为素数,初始都为素数
    int flag[LEN/32] = {0}; //每个int是4字节=32bit (32位) 可以存储32个0,1,故而只需要LEN/32个长
```



```
for(int i=2; i<LEN; i++){
    //倍数必然不是素数，把不是素数的都筛选出来
    if(!((flag[i/32]>>(i%32))&1)){
        primes[pi++] = i;
        for(int j=i; j<LEN; j+=i){
            //将第j位设置为1，表示不是素数
            flag[j/32] |= (1<<(j%32));
        }
    }
}
primes[0] = pi - 1;
cout<<"count:"<<primes[0]<<endl;
return primes;
}
```

```
int main(){
    clock_t start, finish;
    start = clock();
    int n = 0;
    /*
    cout<<"输入判断的数："<<endl;
    cin>>n;
    if(isPrime(n)){
        cout<<"是素数"<<endl;
    }else{
        cout<<"不是素数"<<endl;
    }
    */

    //-----方法1(经典算法62ms)-----
    //getPrime();
    /*
    int *a = getPrime();
    for(int i=1; i<LEN; i++){
        if(a[i]==0) break;
        cout<<a[i]<<" ";
    }
    */
}
```

```
    }  
    */  
  
    //-----方法1(经典算法改进15ms)-----  
    //getPrime1();  
    /*  
    int *a = getPrime1();  
    for(int i=1; i<LEN; i++){  
        if(a[i]==0) break;  
        cout<<a[i]<<" ";  
    }  
    */  
  
    //-----方法3(厄拉多塞筛算法<1ms)-----  
    //getPrime2();  
    /*  
    int *a = getPrime2();  
    for(int i=1; i<LEN; i++){  
        if(a[i]==0) break;  
        cout<<a[i]<<" ";  
    }  
    */  
  
    //-----方法3(厄拉多塞筛算法改进<1ms)-----  
    //getPrime3();  
  
    int *a = getPrime3();  
    for(int i=1; i<LEN; i++){  
        if(a[i]==0) break;  
        cout<<a[i]<<" ";  
    }  
  
    finish = clock();  
    cout<< "\nCost Time: " << finish - start << " ms";  
    return 0;  
}
```


4.3 关于序列的几个算法

发表时间: 2012-06-03 关键字: 子序列, 算法

1.求最小子序列的和

就是对于连续的序列，找出连续序列中和最小的

例如:int a[LEN] = {4,-1,5,-2,-1,2,6,-2,1,-3};

最小的子序列就是：-2,1,-3

对于下面的最大子序列就是：4,-1,5,-2,-1,2,6。

```
/**
 *最小子序列和
 *n
 */
int subMinSum(int a[], int length){
    int thismin = 0, min = INT_MAX;
    for(int i=0; i<length; i++){
        thismin += a[i];
        if(thismin < min){
            min = thismin;
        }else if(thismin > 0){
            thismin = 0;
        }
    }
    return min;
}
```

2.最大子序列的和，其思想和求最小是一样的

```
/**
 *最大子序列和
 *
 */
int subMaxSum(int *a, int length){
    int thismax = a[0], max=INT_MIN;//注：此处修正了thismax = 0,因为全负数的时候返回0，这里初值
    for(int i=0; i<length; i++){
        thismax += a[i];
        if(thismax > max){
            max = thismax;
        }else if(thismax < 0){
            thismax = 0;
        }
    }
    return max;
}
```

针对上面的两种算法，效率是比较高的，但是正确性不容易看出来，需要自己仔细的揣摩和证明，但是优点也很明显，就是对数据只需要一次的扫描，顺序读入，这对于大量数据来说是有利的，只要全部读入数据就可以得出结果，这个在算法分析一书中叫做“联机算法(on-line algorithm)”，意思就是上面描述的，线性时间完成的联机算法基本上算法完美算法。

3.求最大子序列,并记录子序列位置

这里我们肯定很容易记录下最终的位置，就是不容易确定序列的起始位置，主要是由于该算法的特性，最后想了好久才想到用：减最大的值的方法，当等于0的时候就找到起始位置了，从而确定序列范围

```
#include <iostream>
using namespace std;

int maxsub(const int* a, int length, int* loc){
    //maxsum记录最大的值，thissum记录当前的值
    int maxsum=0, thissum=0, i=0;
```

```
if(length <= 0) return 0;
for(i=0; i<length; i++){
    thissum += a[i];
    if(maxsum < thissum){
        maxsum = thissum;
        loc[1] = i;
        //这里有一个思想就是，为负数的子序列不可能成为最优子序列的前缀，
    }else if(thissum < 0){
        thissum = 0;
    }
}
thissum = maxsum;
for(i=loc[1]; i>=0; i--){
    thissum -= a[i];
    if(thissum == 0){
        loc[0] = i;
        break;
    }
}
return maxsum;
}

int main(){
    int a[] = {2, -3, 7, -4, -2, -2, 6, -2};
    int loc[2]={0};
    cout<<maxsub(a, 8, loc)<<endl;
    cout<<"from:"<<loc[0]<<"---to:"<<loc[1]<<endl;
    return 0;
}
```

4.最小正子序列和

这个是在一博客中看到的：<http://blog.csdn.net/qq675927952/article/details/6790726>

然后我有一个疑问就是，究竟什么是最小正子序列和？这个在网上找了也没有个好的答案

(<http://tengtime.com/a/gaoxingnenWEBkaifa/20120406/3217.html>)，如果哪位仁兄知道，不胜感激！！

对于下面求出的sum我也没搞的太清楚是为什么？然后又循环什么的？

```
struct Node
{
    int sum;
    int xiabiao;
};

int cmp(const Node& t1,const Node& t2)
{
    return t1.sum < t2.sum;
}

/**
    *最小正子序列和
    *n*logn
    */
int positiveSubMinSum(int *data, int len){
    Node* temp = new Node[len];
    temp[0].sum = data[0];
    temp[0].xiabiao = 0;
    for(int i=1;i<len;i++)
    {
        temp[i].sum = temp[i-1].sum+data[i];
        temp[i].xiabiao = i;
    }
    //对temp.sum[]进行从小到大排序，sum[]中只有相邻的两个数才有可能 得到 最小正子序列和
    sort(temp,temp+len,cmp);
    int sum = INT_MAX;
    for(int i=0;i<len-1;i++)
    {
        if(temp[i].xiabiao < temp[i+1].xiabiao)
```

```
{
    if(temp[i+1].sum - temp[i].sum > 0 && temp[i+1].sum - temp[i].sum < sum)
        sum = temp[i+1].sum - temp[i].sum;
}
}
delete temp;
temp=0;
return sum;
}
```

5.最大子序列的乘积

对网上的一些算法进行了下比较，发现这个算法还可以，比较简洁：<http://blog.csdn.net/ztj111/article/details/1909339>

对该算法的说明：

这个问题其实可以简化成这样：数组中找一个子序列，使得它的乘积最大；同时找一个子序列，使得它的乘积最小（负数的情况）。虽然我们只要一个最大积，但由于负数的存在，我们同时找这两个乘积做起来反而方便。

我们让maxCurrent表示当前最大乘积的candidate，minCurrent反之，表示当前最小乘积的candidate。这里我用candidate这个词是因为只是可能成为新一轮的最大/最小乘积，而maxProduct则记录到目前为止所有最大乘积candidates的最大值。

由于空集的乘积定义为1，在搜索数组前，maxCurrent,maxProduct,minCurrent都赋为1。

假设在任何时刻你已经有了maxCurrent和minCurrent这两个最大/最小乘积的candidates，

新读入数组的元素x(i)后，新的最大乘积candidate只可能是maxCurrent或者minCurrent

与x(i)的乘积中的较大者，如果x(i)<0导致maxCurrent<minCurrent，需要交换这两个

candidates的值。

当任何时候 $\text{maxCurrent} < 1$ ，由于1（空集）是比 maxCurrent 更好的candidate，所以更新

maxCurrent 为1，类似的可以更新 minCurrent 。任何时候 maxCurrent 如果比最好的

maxProduct 大，更新 maxProduct 。

```
void swap(int& a, int& b){
    int temp = a;
    a = b;
    b = temp;
}

/**
 *最大子序列乘积(同时也求出了最小的子序列乘积)
 *在找最大的值得时候，必须记录最小值，因为有负数存在，最小的数可能变成最大的数
 */
int mutiSubMax(int *a, int length){
    int i;
    int maxProduct = 1;
    int minProduct = 1;
    int maxCurrent = 1;
    int minCurrent = 1;

    for( i=0; i<length ;i++)
    {
        maxCurrent *= a[i];
        minCurrent *= a[i];
        if(maxCurrent > maxProduct)
            maxProduct = maxCurrent;
        if(minCurrent > maxProduct)
            maxProduct = minCurrent;
```

```
    if(maxCurrent < minProduct)
        minProduct = maxCurrent;
    if(minCurrent < minProduct)
        minProduct = minCurrent;
    //注意交换
    if(minCurrent > maxCurrent)
        swap(maxCurrent,minCurrent);
    //这个必须在最后（防止为0的时候）
    if(maxCurrent<1)
        maxCurrent = 1;
    //if(minCurrent>1)//这里不需要，因为通过交换即可，只需要一个
    // minCurrent =1;
}
return maxProduct;
}
```

6.最长递增子序列

参考：

<http://blog.csdn.net/hhygcy/article/details/3950158>

<http://www.programfan.com/blog/article.asp?id=13086>

a , 问题描述

设 $L=\langle a_1,a_2,\dots,a_n \rangle$ 是 n 个不同的实数的序列， L 的递增子序列是这样一个子序列 $L_{in}=\langle a_{k1},a_{k2},\dots,a_{km} \rangle$ ，其中 $k_1 < k_2 < \dots < k_m$ 且 $a_{k1} < a_{k2} < \dots < a_{km}$ 。求最大的 m 值

b , 问题分析

最长递增子序列可以看成是一个**动态规划**的问题，关于动态规划可以参见:

<http://hi.baidu.com/hacklzt/blog/item/81e6b8fc795d251f09244de7.html>

<http://www.cnblogs.com/brokencode/archive/2011/06/26/2090702.html>

其实质就是：将**问题细化**，**逐步求解**

当然实际的过程相对复杂些，就拿该题来说

如：子序列{1, 9, 3, 8, 11, 4, 5, 6, 4, 19, 7, 1, 7}这样一个字符串的最长递增子序列就是{1,3,4,5,6,7}或者{1,3,4,5,6,19}。

首先，该问题可以抽象为：子序列为 $L(1..n)$ ， $A(1..i)$ 是一个从1到i的优化的子结构,也就是最长递增子序列,求 $A(n)$ 。

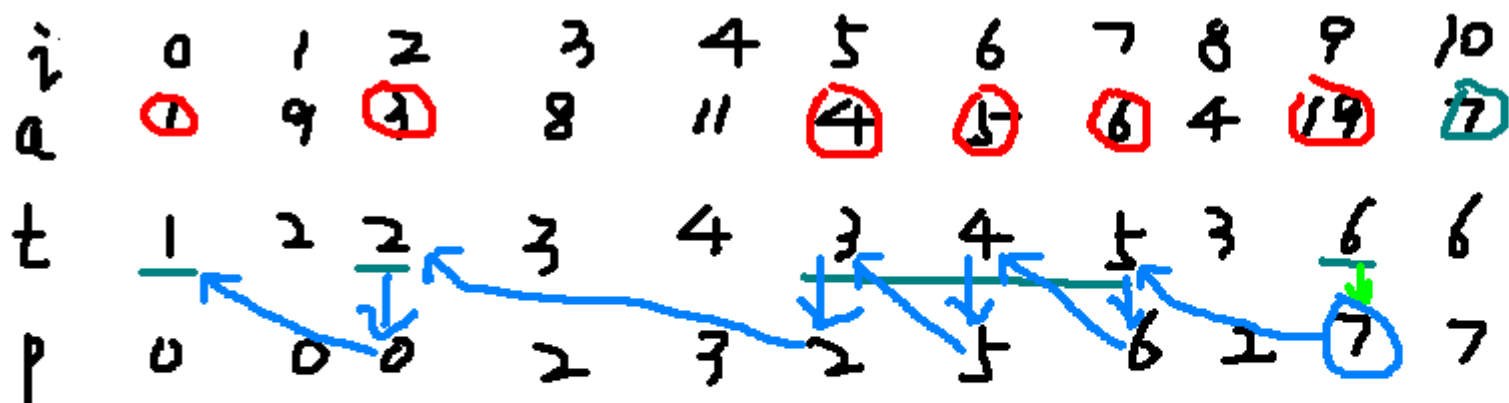
其次，根据上面的抽象，求的就是 $A(n)$ ，那么问题就转化为求 $A(j)$ 与 $A(1..i)(j > i)$ 的关系（状态转移方程）。

最后，根据动态规划的概念，找的就是上面的这样一个关系，如何将 $A(j)$ 与 $A(1..i)$ 联系起来？从而将问题细化，
 $L(j) = \max \{L(i), i < j \ \&\& \ A(i) < A(j) \} + 1;$

也就是说 $L(j)$ 等于之前所有的 $L(i)$ 中最大的 $L(i)$ 加一。这样的 $L(i)$ 需要满足的条件就是 $A(i) < A(j)$ 。这个推断还是比较容易理解的。就是选择j之前所有的满足小于当前数组的最大值。

然后就是将上面的关系式转换为代码，这个就很简单了。

过程如下：



说明:

1. i是数组下标; a是原始数组; t是最长递增子序列的长度; p是最长递增子序列最优子序列的前一个值下标。

2. t的作用: 记录最优子序列的长度, 从中可以找出最优子序列 (它是从1递增到max, 本题中从1-6)。

p的作用: 记录了每次的最优子序列前一个的下标, 如t[9]是最终的长度6, 而长度为5的递增子序列的下标在哪里呢? 其中, p[9]=7, 则只需要在t[7]中找就是5, 而对应的值是a[7]=6, 从而将最终子序列找出来

下面就来讨论哈, 如何求递增子序列的过程?

$t[i] = \max(t[j], a[i] > a[j] \ \&\& \ t[i] < t[j] + 1) + 1$ 这就是求解的状态转移方程。

1. 上面方程说明了, 求t[i]必须就是将0到i-1的最大递增子序列找出来, 然后加1。

例如, 在求t[4], 扫描整个t[0...3], 发现t[3]=3是最大的递增子序列, 而且满足限制条件, 故而t[4]=t[3]+1, 而当前子序列就求出了t, 而在求t的过程中记录下下标到p

2. 可以尝试自己先按照这个思想, 将数组a, t, p自己模仿写出来, 会理解更深, 对动态规划这个过程也能理解。

3. 对于最长递增子序列的最大长度6很容易求出来, 那么怎样找那个子序列的序列1, 3, 4, 5, 6, 19 (7) 呢?

这个就是利用了p数组, 它里面就记录了所有的信息, 就是按照蓝色箭头的方式, 一步步找到这些元素的

设6即最大递增子序列长度为k, 通过循环k=p[k]就可以找到所有。这个见具体的代码

首先, 查找最大递增子序列长度max=6, 和所在下标index=9

其次, 依据所得下标, 找到最后一个元素a[index]=19

然后, t[index]=7表示子序列上一个值的下标位置, index=7, a[index]=6, max=t[index]=5

最后, 一直循环这样一个过程, 直到找到所有。

```
/**
 *最长递增子序列
 *
 */
#include <iostream>
#include <cstring>
using namespace std;

//利用动态规划法O(n^2)
```

```
void longestIncreaseSub(int* a, int length, int* sub){
    int n = length;
    //利用一个辅助数列，记录子问题的值
    int *t = new int[n]; //需要将t所有都初始化1, t记录子序列（子问题）的最长递增子序列长度
    int *p = new int[n];
    memset(p, 0, sizeof(int));
    //初始的最大子序列长度默认为1，即自身组成一个最长递增子序列
    t[0]=1;
    for(int i=1; i<n; i++){
        t[i]=1;
        for(int j=0; j<i; j++){
            //这里面就是动态优化的状态转移方程
            if(a[j]<a[i] && t[j]>t[i]-1){
                //里面存的是(0到i)的子序列中最长递增子序列的长度
                t[i]=t[j]+1;
                //符合要求的子序列位置(就是上一个子序列中的最后一个值的位置)
                p[i]=j;
            }
        }
    }

    int m=0, k=0;
    for (int i=1; i<=n; i++)
    {
        if (m<t[i])
        { //m存t中最大值(就是要找的最长递增子序列)，i是其所在的索引
            m = t[i];
            k = i;
        }
    }

    while(m>=0){
        sub[m-1] = a[k];
        m--;
        //p[k]中存着子序列中下一个值的下标位置
        k = p[k];
    }

    for(int d=0; d<n; d++) cout<<t[d]<<" ";
}
```

```
        cout<<endl;
        for(int d=0; d<n; d++) cout<<p[d]<<" ";
        delete[] t;
        delete[] p;
        p = NULL;
        t = NULL;
    }

int main(){
    int a[] = { 1, 9, 3, 8, 11, 4, 5, 6, 4, 19, 7, 1, 7 };
    int length = sizeof(a)/sizeof(int);
    int* sub = new int[length];
    memset(sub,0,sizeof(int));//初始化sub

    longestIncreaseSub(a,length, sub);

    cout<<endl;
    for(int k=0; k<length; k++) cout<<sub[k]<<" ";
    delete[] sub;
    sub = NULL;
}
```

7.上面的1,2,4,5的代码全部

```
/**
 *2.17
 *求最小子序列和，最小正子序列和，最大子序列乘积
 *
 */
```

```
#include <iostream>
#include <algorithm>
using namespace std;
#define INT_MIN -32768
#define INT_MAX 32767
//在里面有使用#define和const定义常量的比较，推荐使用const
//#define LEN 10
const int LEN = 10;

/**
 *最大子序列和
 *
 */
int subMaxSum(int *a, int length){
    int thismax = 0, max=INT_MIN;
    for(int i=0; i<length; i++){
        thismax += a[i];
        if(thismax > max){
            max = thismax;
        }else if(thismax < 0){
            thismax = 0;
        }
    }
    return max;
}

/**
 *最小子序列和
 *n
 */
int subMinSum(int a[], int length){
    int thismin = 0, min = INT_MAX;
    for(int i=0; i<length; i++){
        thismin += a[i];
        if(thismin < min){
            min = thismin;
        }else if(thismin > 0){
```

```
                thismin = 0;

            }

        }

        return min;
    }

struct Node
{
    int sum;
    int xiabiao;
};

int cmp(const Node& t1,const Node& t2)
{
    return t1.sum < t2.sum;
}

/**
 *最小正子序列和
 *n*logn
 */
int positiveSubMinSum(int *data, int len){
    Node* temp = new Node[len];
    temp[0].sum = data[0];
    temp[0].xiabiao = 0;
    for(int i=1;i<len;i++){
        {
            temp[i].sum = temp[i-1].sum+data[i];
            temp[i].xiabiao = i;
        }
    }
    //对temp.sum[]进行从小到大排序，sum[]中只有相邻的两个数才有可能 得到 最小正子序列和
    sort(temp,temp+len,cmp);
    int sum = INT_MAX;
    for(int i=0;i<len-1;i++){
        {
            if(temp[i].xiabiao < temp[i+1].xiabiao)
            {
```



```
        if(temp[i+1].sum - temp[i].sum > 0 && temp[i+1].sum - temp[i].sum < sum)
            sum = temp[i+1].sum - temp[i].sum;
    }
}
delete temp;
temp=0;
return sum;
}
```

```
void swap(int& a, int& b){
    int temp = a;
    a = b;
    b = temp;
}
```

```
/**
```

```
    *最大子序列乘积(同时也求出了最小的子序列乘积)
```

```
    *在找最大的值得时候，必须记录最小值，因为有负数存在，最小的数可能变成最大的数
```

```
    */
```

```
int mutiSubMax(int *a, int length){
    int i;
    int maxProduct = 1;
    int minProduct = 1;
    int maxCurrent = 1;
    int minCurrent = 1;

    for( i=0; i<length ;i++)
    {
        maxCurrent *= a[i];
        minCurrent *= a[i];
        if(maxCurrent > maxProduct)
            maxProduct = maxCurrent;
        if(minCurrent > maxProduct)
            maxProduct = minCurrent;
        if(maxCurrent < minProduct)
            minProduct = maxCurrent;
        if(minCurrent < minProduct)
```

```
        minProduct = minCurrent;
//注意交换
if(minCurrent > maxCurrent)
    swap(maxCurrent,minCurrent);
//这个必须在最后（防止为0的时候）
if(maxCurrent<1)
    maxCurrent = 1;
//if(minCurrent>1)//这里不需要，因为通过交换即可，只需要一个
// minCurrent =1;
}
return maxProduct;
}

int main(){
    int a[LEN] = {4,-1,5,-2,-1,2,6,-2,1,-3};
    cout<<"列表："<<endl;
    for(int i=0; i<10; i++){
        cout<<a[i]<<" ";
    }
    cout<<endl;
    cout<<"最大子序列和："<<subMaxSum(a, LEN)<<endl;
    cout<<"最小子序列和："<<subMinSum(a, LEN)<<endl;
    cout<<"最小正子序列和："<<positiveSubMinSum(a, LEN)<<endl;
    cout<<"最大子序列乘积："<<mutiSubMax(a, LEN)<<endl;
    return 0;
}
```

4.4 二分查找的递归和非递归

发表时间: 2012-06-03 关键字: 二分查找, 算法

二分查找，这个适用于已经排序好了的数组，没有排序那就先排序，不过要根据实际的情况，要是排序代价很小，这样很好了

```
/**
 *2.15,在有序数组中查找，利用二分查找的方法
 *
 */
#include <iostream>
using namespace std;

/**
 *二分查找(也可以通过递归实现)
 *
 */
int sort(int *a, int length, int value){
    int left = 0, right = length - 1;
    while(left <= right){
        int center = (left + right)/2;
        if(value < a[center]){
            right = center - 1;
        }else if(value > a[center]){
            left = center + 1;
        }else{
            return center;
        }
    }
    return -1;
}

//递归形式
int sort(int *a, int left, int right, int value){
    int center = (left + right)/2;
    //异常时的检测
    if(left == right && a[center] != value) return -1;
```

```
//递归查找
if(value > a[center]) sort(a, center+1, right, value);
else if(value < a[center]) sort(a, left, center-1, value);
else return center;
}

int main(){
    int a[10] = {2,4,5,6,8,12,32,45,55,65};
    cout<<"列表："<<endl;
    for(int i=0; i<10; i++){
        cout<<a[i]<<" ";
    }
    cout<<endl;
    int v;
    cin>>v;
    cout<<"在位置："<<sort(a, 10, v)<<endl;
    cout<<"递归在位置："<<sort(a, 0,9, v)<<endl;
}
```

4.5 求幂的递归和非递归

发表时间: 2012-06-03 关键字: 幂运算, 递归, 非递归

本文的非递归部分转载自：<http://www.cnblogs.com/wallace/archive/2009/12/27/1633683.html>

先上算法

1.递归算法

//幂运算的递归算法

```
long pow(long x, int n){
    if(n == 0) return 1;
    if(n == 1) return x;
    if(n % 2 == 0){
        return pow(x*x, n/2);
    }else{
        return pow(x*x, n/2)*x;
    }
}
```

注意：在算法分析中还说明了：

用：

```
return pow(pow(x, 2), n/2);
return pow(pow(x, n/2), 2);
return pow(x, n/2)*pow(x, n/2)
```

都是不行的，具体原因参考算法分析，主要就是明白，我们的目的是**降次和分解**

2.非递归算法

```
//幂运算的非递归运算,由低次幂逐渐升级即: x, x^2, x^4, x^8, ...
long pow2(long x, int n){
    long pw = 1;
    while (n > 0) {
        //奇数二进制最后一位必定是1。如果是奇数,则乘以x(下面x在不断增大,到最后必定n=1,即奇数)
        if (n & 1)          // n & 1 等价于 (n % 2) == 1
            pw *= x;
        x *= x;
        n >>= 1;           // n >>= 1 等价于 n /= 2
    }
    return pw;
}
```

里面都是用了位运算,这个能提高效率,需要掌握!具体的过程分析见下面

快速求正整数次幂,当然不能直接死乘。举个例子:

$$3^{999} = 3 * 3 * 3 * \dots * 3$$

直接乘要做998次乘法。但事实上可以这样做,先求出 2^k 次幂:

$$3^2 = 3 * 3$$

$$3^4 = (3^2) * (3^2)$$

$$3^8 = (3^4) * (3^4)$$

$$3^{16} = (3^8) * (3^8)$$

$$3^{32} = (3^{16}) * (3^{16})$$

$$3^{64} = (3^{32}) * (3^{32})$$

$$3^{128} = (3^{64}) * (3^{64})$$

$$3^{256} = (3^{128}) * (3^{128})$$

$$3^{512} = (3^{256}) * (3^{256})$$

再相乘:

$$\begin{aligned} & 3^{999} \\ &= 3^{(512 + 256 + 128 + 64 + 32 + 4 + 2 + 1)} \\ &= (3^{512}) * (3^{256}) * (3^{128}) * (3^{64}) * (3^{32}) * (3^4) * (3^2) * 3 \end{aligned}$$

这样只要做16次乘法。即使加上一些辅助的存储和运算，也比直接乘高效得多（尤其如果这里底数是成百上千位的大数字的话）。

我们发现，把999转为2进制数：1111100111，其各位就是要乘的数。这提示我们利用求二进制位的算法（其中mod是模运算）：

```
REVERSE_BINARY(n)
1 while (n > 0)
2   do output (n mod 2)
3   n ← n / 2
```

这个算法给出正整数n的反向二进制进位，如6就给出011（6的二进制表示为110）。事实上这个算法对任意的p进制数是通用的，只要把其中的2换成p就可以了。

如何把它改编为求幂运算？我们发现这个算法是从低位向高位做的，而恰好我们求幂也想从低次幂向高次幂计算（参看前面的例子）。而且我们知道前面求出的每个 2^k 次幂只参与一次乘法运算，这就提示我们并不把所有的中间结果保存下来，而是在计算出它们后就立即运算。于是，我们要做的就是将输出语句改为要做的乘法运算，并在n减少的同时不断地累积求 2^k 次幂。

还是看算法吧：

```
POWER_INTEGER(x, n)
1 pow ← 1
2 while (n > 0)
3   do if (n mod 2 = 1)
4     then pow ← pow * x
5   x ← x * x
6   n ← n / 2
7 return pow
```

不难看出这个算法与前面算法的关系。在第1步给出结果的初值1，在while循环内进行运算。3、4中的if语句就来自REVERSE_BINARY的输出语句，不过改成了如果是1则向pow中乘。5句则是不断地计算x的 2^k 次幂，如对前面的例子就是计算 2^2 、 2^4 、 2^8 、...、 2^{512} 。

应该指出，POWER_INTEGER比前面分析的要再多做两次乘法，一次是向pow中第一次乘x，如 2^1 也要进行这个乘法；另一次则是在算法的最后，n除以2后该跳出循环，而前面一次x的自乘就浪费掉了（也可以考虑改

变循环模式优化掉它)。另外，每趟while循环都要进行一次除法和一次模运算，这多数情况下除法和模运算都比乘法慢许多，不过好在我们往往可以用位运算来代替它。

相应的C++代码如下

```
NumberType pow_n(NumberType x, unsigned int n)
{
    NumberType pw = 1;

    while (n > 0) {
        if ((pw % 2) == 1)
            pw *= x;
        x *= x;
        n /= 2;
    }

    return pw;
}
```

进行简单的优化后则有：

```
NumberType optimized_pow_n(NumberType x, unsigned int n)
{
    NumberType pw = 1;

    while (n > 0) {
        if (n & 1)    // n & 1 等价于 (n % 2) == 1
            pw *= x;
        x *= x;
        n >>= 1;    // n >>= 1 等价于 n /= 2
    }

    return pw;
}
```

注1：快速求幂算法POWER_INTEGER常被写成递归的形式，算法实质完全相同，但却是无必要的。

注2：这个算法并不是做乘法数最少的，但多数情况下是足够快并且足够简单的。如果单纯追求做乘法数最少，则未必应该用 2^k 次幂进行计算。如果还允许做除法，则问题会进一步复杂化。

如：

$$x^2 = x * x$$

$$x^4 = (x^2) * (x^2)$$

$$x^8 = (x^4) * (x^4)$$

$$x^{16} = (x^8) * (x^8)$$

$$x^{31} = (x^{16}) * (x^8) * (x^4) * (x^2) * x$$

要8次乘法。

$$x^2 = x * x$$

$$x^4 = (x^2) * (x^2)$$

$$x^8 = (x^4) * (x^4)$$

$$x^{10} = (x^8) * (x^2)$$

$$x^{20} = (x^{10}) * (x^{10})$$

$$x^{30} = (x^{20}) * (x^{10})$$

$$x^{31} = (x^{30}) * x$$

只要7次乘法。

$$x^2 = x * x$$

$$x^4 = (x^2) * (x^2)$$

$$x^8 = (x^4) * (x^4)$$

$$x^{16} = (x^8) * (x^8)$$

$$x^{32} = (x^{16}) * (x^{16})$$

$$x^{31} = (x^{32}) / x$$

只要6次乘或除法。

不过具体得出上述乘（除）法数更少的算法会变得相当复杂，在许多情况下时间收益还会得不偿失。因此往往并不实用。ACM Japan 2006中有一道题即要求计算最少乘法数，可参看：

<http://acm.pku.edu.cn/JudgeOnline/problem?id=3134>

zz from: <http://www.cnblogs.com/wallace/archive/2009/12/27/1633683.html>

4.6 主元素算法

发表时间: 2012-06-04 关键字: 主元素, 快速排序, 中位法

1.算法描述 (算法分析2.26)

大小为N的数组A,其主元素是一个出现超过N/2次的元素 (从而这样的元素最多只有一个)。例如 , 数组

3,3,4,2,4,4,2,4,4只有一个主元素4 ; 3,3,4,2,4,4,2,4没有主元素

求出主元素 , 没有请指出

2.书中列出了一种算法 , 暂且叫递归法 , 这可以自己看书 , 其复杂度也只有O(n)

下面介绍两种其他的方法。

在网上还有其他一些方法 : <http://wintys.blog.51cto.com/425414/100688/>

3.方法一

思想就是利用主元素的个数是 $> N/2$ 的这一性质,复杂度O(n)

*首先 , 随机选取一个作为主元素

*其次 , 依次遍历数组 , 遍历中遇到相同的count++ , 否则count--

*如果 ≤ 0 时在换主元素

*最后 , 剩下的就作为主元素的人选

*当然选出来的不一定就是主元素 , 还要检测 , 如果检测成功 , 那必定就是主元素 , 否则就没有

```
int getMain(int* a, int length){
    int count =0;
    int seed = a[0];
    for(int i=0; i<length; i++){
        if(a[i]==seed) count++;
        else if(count>0) count--;
        else seed = a[i];
    }
    return seed;
}
```

这个方法很简单，简洁明了

4.方法二

方法二：利用快速排序，最后选取N/2处的元素作为主元素,复杂度为快速排序的复杂度

*当然选出来的不一定就是主元素，还要检测，如果检测成功，那必定就是主元素，否则就没有

```
int getMain1(int* a, int length){
    quickSort(a, 0, length-1);
    return a[length/2];
}
```

就是利用了快速排序

5.全部代码

```
/**
 *2.26找超过n/2次出现的主元素
 **/
```

```
#include <iostream>

using namespace std;

//检查是否是主元素
bool check(int* a, int length, int n){
    int count = 0;
    for(int i=0; i<length; i++){
        if(a[i]==n) count++;
    }
    if(count>length/2) return true;
    return false;
}

void swap(int &a, int &b){
    int temp = a;
    a = b;
    b = temp;
}

//一次划分
int partition(int *a, int first, int end){
    int i=first; //初始化
    int j=end;
    while(i < j){
        while(i<j && a[i] <= a[j]) j--;
        if(i<j){
            swap(a[i], a[j]);
            i++;
        }
        while(i<j && a[i] <= a[j]) i++;
        if(i<j){
            swap(a[i], a[j]);
            j--;
        }
    }
    return i;
}
```

```
}

//快速排序
void quickSort(int* a, int first, int end){
    if(first < end){
        int pivot=partition(a, first, end); //一次划分
        quickSort(a, first, pivot-1); //递归地对左侧子序列进行快速排序
        quickSort(a, pivot+1, end); //递归地对右侧子序列进行快速排序
    }
}

/**
 *方法一：思想就是利用主元素的个数是>N/2的这一性质,复杂度O(n)
 *首先，随机选取一个作为主元素
 *其次，依次遍历数组，遍历中遇到相同的count++,否则count--
 *如果<=0时在换主元素
 *最后，剩下的就作为主元素的人选
 *当然选出来的不一定是主元素，还要检测，如果检测成功，那必定就是主元素，否则就没有
 */
int getMain(int* a, int length){
    int count =0;
    int seed = a[0];
    for(int i=0; i<length; i++){
        if(a[i]==seed) count++;
        else if(count>0) count--;
        else seed = a[i];
    }
    return seed;
}

/**
 *方法二：利用快速排序，最后选取N/2处的元素作为主元素,复杂度为快速排序的复杂度
 *当然选出来的不一定是主元素，还要检测，如果检测成功，那必定就是主元素，否则就没有
 */
int getMain1(int* a, int length){
```

```
    quickSort(a, 0, length-1);
    return a[length/2];
}

int main(){
    int a[] = {6,1,6,3,6,6,4,2,6};
    //动态获取数组的长度
    int length = sizeof(a)/sizeof(int);
    //
    int v = getMain(a, length);
    int v = getMain1(a, length);
    cout<<"length:"<<length<<",seed:"<<v<<endl;
    if(check(a, length, v)){
        cout<<"主元素是:"<<v<<endl;
    }else{
        cout<<"没找到主元素!"<<endl;
    }
    return 0;
}
```

4.7 整数的随机置换

发表时间: 2012-06-05 关键字: 随机, 置换, 算法改进

算法描述:生成前N个整数的随机置换, 如{4,3,1,5,2}, {4,5,3,2,1}是合法的, 而{5,4,1,1,2}不合法, 因为3没出现。

1.基本算法

该算法效率比较低, $O(n*n*\log n)$, 主要就是随机的生成一个数, 然后再一直数组中去检测是否存在, 如果不存在才插入。从而效率低下

```
//使用的是思想1( $O(n*n*\log n)$ )
int* getRandom(int* a, int length){
    for(int i=0; i<length; i++){
        int j = randInt(0, length);
        //循环, 直到找到一个在数组中不存在的值, 这里的复杂度是 $n*\log n$ 
        while(true){
            if(findExist(a, i, j)){
                j = randInt(0, length);
            }else{
                break;
            }
        }
        a[i] = j;
    }
    return a;
}
```

2.对算法1进行了改进, 主要就是增加了一个标志数组, 对于已经使用过的数进行了标记, 从而不用再遍历原数组, 复杂度 $O(n*\log n)$ 降低了不少, 但以空间换时间

```
//使用思想2( $O(n*\log n)$ )
int* getRandom2(int* a, const int length){
    int used[LEN] = {0};
```

```
for(int i=0; i<length; i++){
    int j = randInt(0, length);
    //循环，直到找到一个在数组中不存在的值
    while(used[j]){
        j = randInt(0, length);
    }
    used[j] = 1;
    a[i] = j;
}
return a;
}
```

3.使用了逆向思维，为什么我们不一次性插入所有数据，然后再随机交换位置，从而达到此效果呢？该算法就是这样，复杂度 $O(n)$ ，线性的

```
//使用思想3( $O(n)$ )
int* getRandom3(int* a, const int length){
    int i;
    //初始化0-9
    for(i=0; i<length; i++){
        a[i] = i;
    }
    //任意交换位置，打乱次序
    for(i=0; i<length; i++){
        int temp = a[i];
        int j = randInt(0, i);
        a[i] = a[j];
        a[j] = temp;
    }
    return a;
}
```


从1-3就是一个算法改进的过程体现，不断的减少时间复杂度，空间消耗，从而实现一个优秀的算法所必须的过程。

4.全部代码

```
#include <iostream>
#include <cstdlib>
#include <time.h> //用于做种子
const int LEN = 1000;
using namespace std;

int randInt(int start, int end){
    return start+(end-start)*rand()/(RAND_MAX + 1.0);
}

int findExist(const int* a, int length, int value){
    for(int i=0; i<length; i++){
        if(a[i] == value) return true;
    }
    return false;
}

//使用的是思想1( $O(n*n*\log n)$ )
int* getRandom(int* a, int length){
    for(int i=0; i<length; i++){
        int j = randInt(0, length);
        //循环，直到找到一个在数组中不存在的值,这里的复杂度是 $n*\log n$ 
        while(true){
            if(findExist(a, i, j)){
                j = randInt(0, length);
            }else{
                break;
            }
        }
    }
}
```

```
        a[i] = j;
    }
    return a;
}

//使用思想2( $O(n \cdot \log n)$ )
int* getRandom2(int* a, const int length){
    int used[LEN] = {0};
    for(int i=0; i<length; i++){
        int j = randInt(0, length);
        //循环,直到找到一个在数组中不存在的值
        while(used[j]){
            j = randInt(0, length);
        }
        used[j] = 1;
        a[i] = j;
    }
    return a;
}

//使用思想3( $O(n)$ )
int* getRandom3(int* a, const int length){
    int i;
    //初始化0-9
    for(i=0; i<length; i++){
        a[i] = i;
    }
    //任意交换位置,打乱次序
    for(i=0; i<length; i++){
        int temp = a[i];
        int j = randInt(0, i);
        a[i] = a[j];
        a[j] = temp;
    }
    return a;
}
```

```
int main(){
//      srand(unsigned(time(0)));//,随机数种子,使用这个就可以每次不同
    int a[LEN] = {0};
    srand(time(NULL));
//      int *b = getRandom(a, LEN);
//      int *b = getRandom2(a, LEN);
    int *b = getRandom3(a, LEN);
    for(int i=0; i<LEN; i++){
        cout<<b[i]<<" ";
    }
    return 0;
}
```

4.8 线性查找二维数组

发表时间: 2012-06-05 关键字: 二维数组

1.算法描述

有序（行有序，列有序，且每行从左至右递增，列从上至下递增）二维数组查找，要求复杂度 $O(n)$

2.使用到的相关知识：

结构体定义和使用，二维数组传递（<http://blog.csdn.net/yzhbmhm/article/details/2045816>）

3.使用数组名传递

这个的不便之处很明显，一旦确定就不能设置列值

//使用数组名实现(不便之处很明显：列值确定了，不能灵活)

```
loc* findValue(int a[][5], int row, int value){
    int col = sizeof(a)/sizeof(int)*row;
    cout<<"size:"<<col<<endl;
    //先按列比较(第0列)，找到所在的行
    int currRow = 0;
    for(int i=0; i<row; i++){
        if(a[i][0] > value){
            if(i != 0) currRow = i - 1;
            break;
        }
    }

    int index = search(a[currRow], 5, value);
    //利用结构体指针
    loc *l;
    l->row = currRow;
    l->col = index;
    return l;
}
```

4.使用指针数组传递

```
//使用指针数组实现
loc findValue(int* a[], int row, int col, int value){
    //先按列比较(第0列),找到所在的行
    int currRow = 0;
    for(int i=0; i<row; i++){
        if(a[i][0] > value){
            if(i != 0) currRow = i - 1;
            break;
        }
    }

    int index = search(a[currRow], col, value);
    loc l;
    l.row = currRow;
    //在给定的行中搜索
    l.col = index;
    return l;
}
```

5.所有代码

```
/**
 *有序(行有序,列有序,且每行从左至右递增,列从上至下递增)二维数组查找
 *要求复杂度O(n)
 */
#include <iostream>
using namespace std;
struct loc{
    int row;
    int col;
};
```

```
//如果找到放回下标，否则-1
int search(int *a, int length, int value){
    int i=0,j=length-1;
    while(i<=j){
        int center = (i+j)/2;
        if(a[center]<value) i=center+1;
        else if(a[center]>value) j=center-1;
        else return center;
    }
    return -1;
}
```

//使用数组名实现(不便之处很明显：列值确定了，不能灵活)

```
loc* findValue(int a[][5], int row, int value){
    int col = sizeof(a)/sizeof(int)*row;
    cout<<"size:"<<col<<endl;
    //先按列比较(第0列)，找到所在的行
    int currRow = 0;
    for(int i=0; i<row; i++){
        if(a[i][0] > value){
            if(i != 0) currRow = i - 1;
            break;
        }
    }

    int index = search(a[currRow], 5, value);
    //利用结构体指针
    loc *l;
    l->row = currRow;
    l->col = index;
    return l;
}
```

//使用指针数组实现

```
loc findValue(int* a[], int row, int col, int value){
```

```
//先按列比较(第0列), 找到所在的行
int currRow = 0;
for(int i=0; i<row; i++){
    if(a[i][0] > value){
        if(i != 0) currRow = i - 1;
        break;
    }
}

int index = search(a[currRow], col, value);
loc l;
l.row = currRow;
//在给定的行中搜索
l.col = index;
return l;
}

int main(){
    int a[5][5],k=0;
    for(int i=0; i<5;i++){
        for(int j=0; j<5; j++){
            a[i][j] = k++;
        }
    }

    int value;
    cout<<"输入要查找的值:";
    cin>>value;
    //-----1-----
    loc* ll = findValue(a, 5,value);
    if(ll->col != -1){
        cout<<"位置在:("<<ll->row<<","<<ll->col<<")"<<endl;
    }else{
        cout<<"没有找到!"<<endl;
    }

    //-----2-----
```

```
//使用指针数组，必须先将二维数组转换为下面的形式
int *p[] = {*a, *(a+1),*(a+2),*(a+3),*(a+4)};
loc l = findValue(p, 5,5,value);
if(l.col != -1){
    cout<<"位置在:("<<l.row<<","<<l.col<<")"<<endl;
}else{
    cout<<"没有找到！"<<endl;
}
return 0;
}
```


4.9 关于最长递增子序列的实际应用--动态规划

发表时间: 2012-06-07 关键字: 搭桥问题, 叠箱子, 最大递增子序列, 动态规划

参考链接：

a.<http://www.programfan.com/blog/article.asp?id=13086>

b.<http://blog.csdn.net/hhygcy/article/details/3950158>

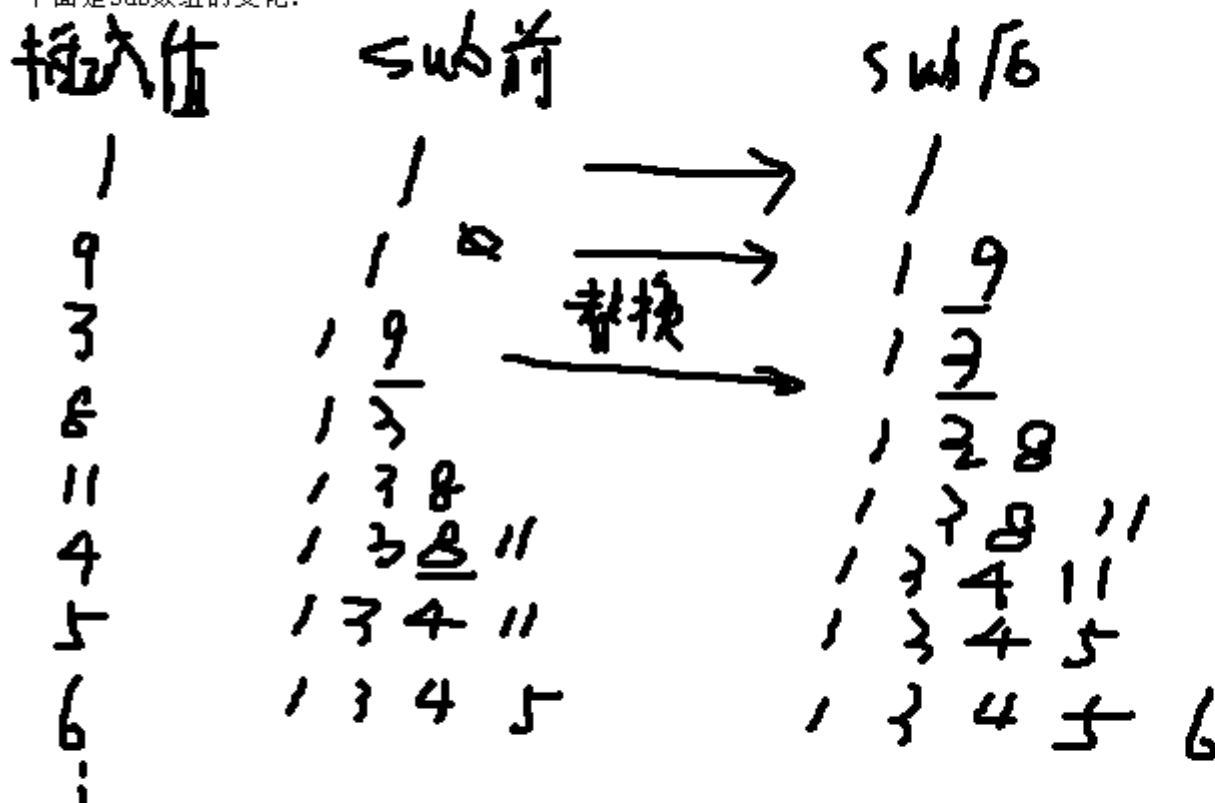
1.对 (<http://hao3100590.iteye.com/blog/1548135>) 中问题6：最长递增子序列的改进，减少时间复杂度

算法的思想：

它不是使用的动态规划法，而是普通的算法思想，就是在数组中直接存储最长递增子序列，在循环的过程中不断的查找插入位置，直到最终找到。下面列出实现的过程：

1 9 → 8 11 4 5 6 4 19 7 17

下面是Sub数组的变化：



从上面的过程可以看出该算法的实现过程，在查找sub合适插入位置的时候，使用了二分查找，提高了查找速度，这个算法本身也比前面利用动态规划法简单，但是该算法复杂之处不在算法

而在算法正确性的证明！，算法证明见：<http://www.programfan.com/blog/article.asp?id=13086>

代码：

```
#include <iostream>
#include <cstring>
using namespace std;
const int MIN = -32768;
```

```
/**
    *a                :原始数组
    *sub              :最终获取的最大递增子序列（注意，sub[0]是哨兵，结果从1开始）
    *length           :数组长度
    */

int longestSub(int* a, int* sub, int length){
    if(length<=0) return 0;
    sub[0]=MIN; //为了初始化的比
    sub[1]=a[0]; //初始序列就是a[0]
    int len = 1; //最大递增子序列长度
    int mid, left, right; //二分查找时的index
    for(int i=0; i<length; i++){
        //在sub数组中二分查找合适的插入位置
        left = 0;
        right = len;
        //获取的最终插入位置就如a[i]=6,{1,3,5,7}位置就是5后面index=3，会替代7
        while(left<=right){
            mid = (left+right)/2;
            if(sub[mid]<a[i]) left=mid+1;
            else right=mid-1;
        }
        //策略是使用替代，不断的寻找合适的值，插入sub，最后sub中剩余的值就是要寻找的最长递增子
        sub[left]=a[i];
        if(left>len) len++;
    }
    return len;
}

int main(){
    int a[] = { 1, 9, 3, 8, 11, 4, 5, 6, 4, 19, 7, 1, 7 };
    int* sub = new int[13];
    memset(sub,0,sizeof(int));
    int length = longestSub(a,sub,13);
    cout<<"length:"<<length<<endl;
    for(int i=1; i<=length; i++) cout<<sub[i]<<" ";
}
```

```
delete[] sub;  
sub=NULL;  
return 0;  
}
```

这个算法的时间复杂度只有 $O(n\log n)$ ，效率明显提高了而且也更简单了

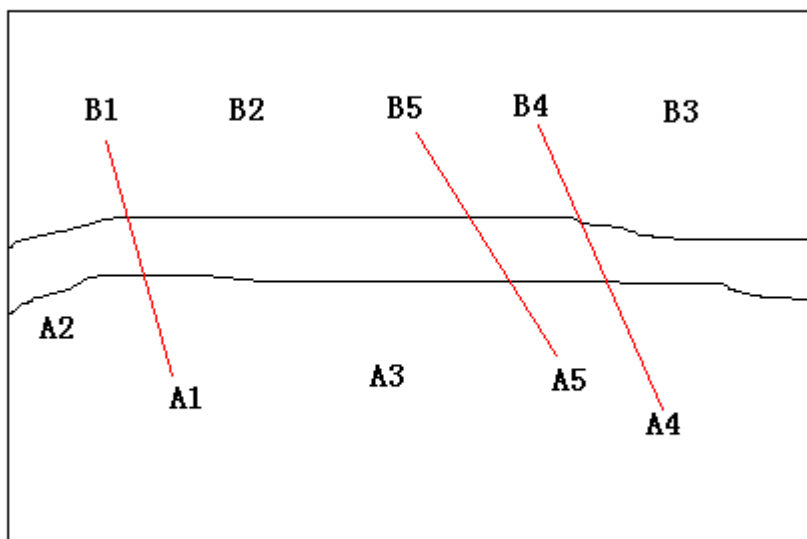
不过其结果和改进之前的有些不同，前面是{1,3,4,5,6,19}，改进之后是{1,3,4,5,6,7}结果都是正确的！

2.实际应用的两个例子

a.造桥问题

问题描述：

要在一条河的南北两边的各个城市之间造若干座桥。桥两边的城市分别是 $a(1) \dots a(n)$ 和 $b(1) \dots b(n)$ 。这里的要求 $a(i)$ 只可以和 $b(i)$ 之间造桥，同时两座桥之间不能交叉。希望可以得到一个尽量多座桥的方案。



问题分析：

首先，这是一个动态规划的问题，即解答有许多中，寻找一个最优的解决方案。

其次，问题抽象，就是怎样将这个问题抽象成一个数学模型进行解决

第一步：就是搞清楚问题是什么？-----就是对号入座A对应于B的序号，这样的解决方案有许多如1,5,4和2,5,4以及1,3

第二步：抽象-----抽象成两个数组S1(1,2,5,4,3), S2(2,1,3,5,4)，然后找S1和S2中相同的数字，且序号必须在数组下标递增，找出最多的对数

(即S1中的1对应了S2中的1，S1中的2对应了S2中的2,虽然S1中的下标是1,2 但是在S2中的下标是2,1没有递增，故而不行的---更简单的说就是上面图不能交叉)

第三步：找规律-----这是最难的，如何才能从中找到突破口，我们在寻找的过程中发现，联系S1和S2的桥梁是什么？就是组成对的两个数在各自数组中的下标

那我们把这些下标都写出来，只是写一方就可以了从S1到S2的，就设为S3(2,1,4,5,3)---就表示S3[0]=2表示S1数组第一个值对应与S2中的第二个值（这个2就是在S2中的下标），

从而依次就把S3写出来了，这样找对数最多，而S3表示的是连线时对方的下标，那么是不是只要下标递增就不会交叉了？事实就是这样！！

就是找S3中的最长递增数组，这里就是{2,4,5}或者{1,4,5}

第四步：写代码-----既然突破口找到了，代码就不是问题了

代码

```
/**
    *建桥问题
    **/
#include <iostream>
#include <cstring>
using namespace std;

/**
    *建立建桥索引数组
    *a          : 原始数组a
```

```

    *b                : 原始数组b
    *c                : 建立的索引数组
    *length : 数组长度
    */
void build(const int* a, const int* b, int* c, int length){
    for(int i=0; i<length; i++){
        for(int j=0; j<length; j++){
            if(a[i]==b[j]){
                c[i]=j+1;//我们建立数组时以1为开始
                break;
            }
        }
    }
}

/**
    *s2                : 原始数组s2
    *rt                : 求出的最大递增子序列
    *length : 数组长度
    */
void printResult(int* s2, int* rt, int length){
    int j = 0;
    for(int i=0; i<length; i++)
    {
        if(rt[i]==0) break;
        j = s2[rt[i]-1];
        cout<<"B"<<j<<"--A"<<j<<" ";
    }
    cout<<endl;
}

/**
    *a                : 原始数组
    *length : 数组长度
    *sub                : 求出的最大递增子序列
    */
void longestIncreaseSub(int* a, int length, int* sub){
```

```
int n = length;
int *t = new int[n];
int *p = new int[n];
memset(p,0,sizeof(int));
t[0]=1;
for(int i=1; i<n; i++){
    t[i]=1;
    for(int j=0; j<i; j++){
        if(a[j]<a[i] && t[j]>t[i]-1){
            t[i]=t[j]+1;
            p[i]=j;
        }
    }
}
int m=0,k=0;
for (int i=1;i<=n;i++)
{
    if (m<t[i])
    {
        m = t[i];
        k = i;
    }
}
while(m>=0){
    sub[m-1] = a[k];
    m--;
    k = p[k];
}

delete[] t;
delete[] p;
p = NULL;
t = NULL;
}

int main(){
    int a[] = {1,2,5,4,3};
    int b[] = {2,1,3,5,4};
```

```
int length = sizeof(a)/sizeof(int);  
//索引数组  
int* c = new int[length];  
//存储最终结果  
int* sub = new int[length];  
memset(sub,0,sizeof(int));//初始化sub  
memset(sub,0,sizeof(int));//初始化sub  
  
build(a,b,c,length);  
longestIncreaseSub(c,length, sub);  
printResult(b,sub, length);  
  
delete[] c;  
delete[] sub;  
c = NULL;  
sub = NULL;  
return 0;  
}
```

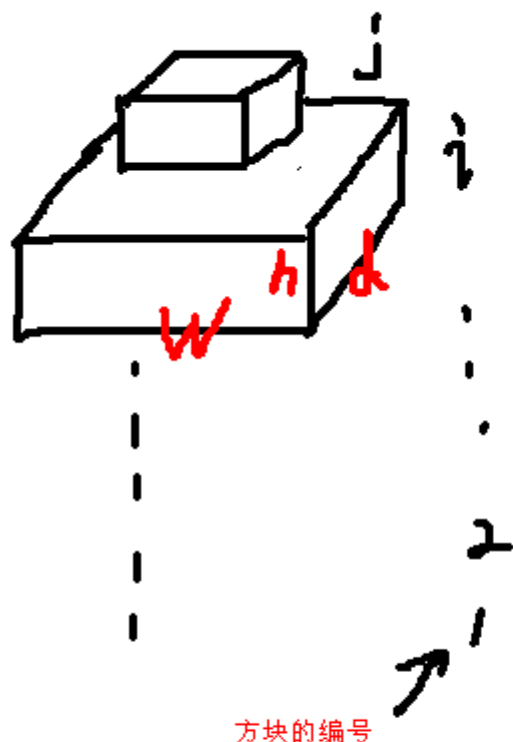
b.叠箱子问题

问题描述：

- 1.一排有许多不同的箱子，长宽高不一样
- 2.你需要把他叠放的尽量的高.但是箱子的摆放必须满足大的在下面,小的在上面的原则
- 3.箱子可以任意旋转（这就意味着你要用一个箱子的时候，旋转到合适位置）

问题分析：

具体分析见图：



1. 题目要求:

- 下面的盒子要比上面的大 ($w[i] * d[i] > w[j] * d[j]$)
就是下面的表面积比上面的大
- 给定的盒子叠的尽量高
- 盒子可以旋转, 数量不限

对于盒子可以旋转, 那么盒子就可以任何一面朝上, 那么一个盒子经过旋转可以变成三个盒子

2. 题目抽象

- 首先满足第一个要求, 就必须比较表面积, $w * d$, 就是看谁是否能在上或者下, 必须比较 $w * d$.
- 给定盒子尽量高, 设前 i 个最优序列的高度为 $H(i)$, 那么...

$$H(j) = \max_{\substack{i < j \\ w_i > w_j \\ d_i > d_j}} \{H(i)\} + h_j$$

$$w_i > w_j \rightarrow S_i > S_j$$

$$d_i > d_j$$

这样就得到了第 j 个盒子的高度

- 对任意给定的 n 个盒子, 从中按序选出 k 个, 使其高度最高就抽象成求最大递增子序列的问题了

注1: 盒子要比上面的大--准确的说, 是 $w_i > w_j \ \&\& \ d_i > d_j$, 单独使用 $S_i > S_j$ 不准确, 因为可能很长但很窄, 不和题目要求

为了满足要求, 我们人为规定, $w \leq d$ (输入时确定或者后来处理), 这样就不用担心这个问题了, 使用 $S_i > S_j$ 就满足要求。

注2: 记住盒子可以旋转, 意味着每个盒子可以有三个面可以使用, 旋转数量不限。

注3: 本体关键是建立模型, 列出动态规划的方程

代码:

```
/**
 *
 *叠箱子问题
 */

#include <iostream>
#include <vector>
#include <cstring>
#include <algorithm>
using namespace std;
//定义一个箱子结构体
struct Box{
    int h;//高
    int w;//宽
    int d;//长
};

//排序比较器
bool boxCompare(const Box& a, const Box&b){
    return a.d*a.w > b.d*b.w;//降序排序
}

//最高的堆叠高度算法
int highestBox(const vector<Box>& b){
    //初始判断
    if(b.size()<=0) return 0;
    //初始化一个vector，长度是原来的三倍，因为每个箱子都有三种翻转方式，每种当成一个新的箱子
    vector<Box> boxs(b.size()*3);
    //箱子的处理，使所有的箱子都是w<=d,每个箱子都翻转两次
    for(int i=0; i<b.size(); i++){
        //下面是一个箱子的三种翻转情况
        boxs[i*3+0].h = b[i].h;
        boxs[i*3+0].w = b[i].w < b[i].d ? b[i].w : b[i].d;
        boxs[i*3+0].d = b[i].w < b[i].d ? b[i].d : b[i].w;
```

```
        boxs[i*3+1].h = b[i].h;
        boxs[i*3+1].w = b[i].h < b[i].d ? b[i].h : b[i].d;
        boxs[i*3+1].d = b[i].h < b[i].d ? b[i].d : b[i].h;

        boxs[i*3+2].h = b[i].h;
        boxs[i*3+2].w = b[i].w < b[i].h ? b[i].w : b[i].h;
        boxs[i*3+2].d = b[i].w < b[i].h ? b[i].h : b[i].w;
    }
    //排序(不是boxCompare ( ))
    sort(boxs.begin(), boxs.end(), boxCompare);

    //最长递增子序列问题
    vector<int> m(b.size()*3);
    m[0] = boxs[0].h;
    for(int i=0; i<boxs.size(); i++){
        for(int j=0; j<i; j++){
            if ( (boxs[i].w <= boxs[j].w) && (boxs[i].d <= boxs[j].d) && (n
                m[i] = m[j]+boxs[i].h;
            }
        }
    }
    int mm = *max_element(m.begin(),m.end());
    return mm;
}

int main(){
    vector<Box> box(3);
    box[0].h = 2;
    box[0].w = 3;
    box[0].d = 4;
    box[1].h = 2;
    box[1].w = 3;
    box[1].d = 1;
    box[2].h = 5;
    box[2].w = 3;
    box[2].d = 4;
    cout<<highestBox(box)<<endl;
```

```
    return 0;  
}
```

4.10 Josephus问题

发表时间: 2012-06-21 关键字: Josephus, 指针

1.算法描述

简单的游戏：有N个人坐成一圈，编号1-N、从编号为1的人开始传递热马铃薯。M次传递之后，持有马铃薯的人退出游戏，圈缩小，然后游戏从退出人下面的人开始，继续进行，最后留下的人获胜。如，M=0，N=5则参加游戏的人依次退出，5号获胜。M=1,N=5则退出顺序是2,4,1,5.

2.算法分析

该算法使用一个**没有头指针的循环链表完成**，移动的过程计数，如果计数为M，则将其移除并从下一位继续开始，否则继续。过程非常简单，主要考察指针的运用。当然还可以使用数组实现，不过移动所花费开销非常大！不推荐使用数组。

该算法的复杂度为 $O(n)$,直接对链表操作。

3.代码

```
#include <iostream>
#include <ctime>
using namespace std;
const int LNE = 100000;

struct Node{
    Node *next;
    int data;
};

//创建一个没有头结点的链表
Node* createList(int *a, int n){
    Node* first = new Node;
    first->data = a[0];
    Node* t = first;
    for(int i=1; i<n; i++){
```

```
        Node* r = new Node;
        r->data = a[i];
        t->next = r;
        t = r;
    }
    t->next = first;
    return first;
}

//N个人，传递M次
int josephus(Node *l, int M, int N){
    if(M<0 || N < 1) return -1;
    if(M >= N) M %= N;                                     //处理M
    int count = 0;                                          //用于计数
    Node *p = l, *t;
    //当M为0的时候处理方式，依次处理，最后一个人即是胜利者
    if(M == 0){
        while(1){
            if(p->next == l) return p->data;
            t = p;
            //cout<<p->data<<" ";
            p = p->next;
            delete t;
        }
    }else{
        //循环到只剩最后一个人
        while(N != 1){
            //计数到M之前的位置
            if(count == M-1){
                t = p->next;
                //只剩最后元素，退出
                if(t == p){
                    break;
                }else{                                     //没有到
                    p->next = t->next;
                    //cout<<t->data<<" ";
                    delete t;
                }
            }
            count++;
            p = p->next;
        }
    }
    return p->data;
}
```

```
                N --;                                //总人数
            }
            count = -1;
        }
        p = p->next;
        count ++;                                    //计数器
    }
}

return p->data;
}

int main(){
    int *a = new int[LNE];
    for(int i=0; i<LNE; i++){
        a[i] = i+1;
    }

    int k;
    Node *f = createList(a, LNE);
    cout<<"输入传递次数：";
    cin>>k;

    time_t b = time(NULL);
    cout<<endl;
    cout<<"胜利者是："<<josephus(f, k, LNE)<<endl;
    time_t e = time(NULL);
    cout<<"耗时："<<e-b<<endl;
    delete []a;
    return 0;
}
```

4.11 后缀表达式的值

发表时间: 2012-06-27 关键字: 后缀表达式, 异常处理, 栈

1. 算法描述

计算后缀表达式的值

2. 事例

如：(2+3) *5--->后缀表达式：23+5*，或者523+*

在计算机中不能直接处理算术表达式，我们就转换为后缀表达式利用栈来解决这个问题

3. 思想

利用数据结构栈

a. 后缀表达式依次入栈，如果遇到操作符，就将栈顶两个元素出栈，计算结果在入栈。

b. 循环进行，直到栈中只有一个元素，就是结果

4. 算法

异常处理类：

```
#ifndef DSEXCEPTIONS_H
#define DSEXCEPTIONS_H
//#include <exception>

class StackOverflowException{ //public: exception
public:
    const char* what() const throw()
    {
        return "stack over flow exception, the size<=0 !\n";
    }
};
```



```
        }  
};  
  
#endif
```

栈实现：

```
#ifndef STACK_H  
#define STACK_H  
#include <iostream>  
#include "dsexceptions.h"  
  
template<class T>  
struct Node{  
    Node<T>* next;  
    T data;  
};  
  
template<class T>  
class stack{  
public:  
    stack(){  
        first = new Node<T>;  
        first->next = NULL;  
        end = first;  
        length = 0;  
    }  
  
    stack(T a[], int n){  
        first = new Node<T>;  
        length = n;  
        Node<T> *r = first;
```

```
        for(int i=0; i<n; i++){
            Node<T> *t = new Node<T>;
            t->data = a[i];
            t->next = r;
            r = t;
        }
        end = r;
    }

    ~stack(){
        freeStack();
    }

    T pop(){
        T data;
        if(length != 0){
            Node<T>* r = end;
            end = end->next;
            data = r->data;
            delete r;
            length--;
        }else{
            throw StackOverFlowException();
        }
        return data;
    }

    void push(T x){
        Node<T> *r = new Node<T>;
        r->data = x;
        r->next = end;
        end = r;
        length++;
    }

    T top() const{
        if(length == 0)
```

```
        throw StackOverFlowException();
        return end->data;
    }

    bool empty() const{
        if(length == 0) return true;
        return false;
    }

    int size() const{
        return length;
    }

    void printStack() const{
        std::cout<<"[";
        Node<T> *r = end;
        while(r != first){
            std::cout<<r->data<<" ";
            r = r->next;
        }
        std::cout<<"]"<<std::endl;
    }

private:
    Node<T>* first;//栈底
    Node<T>* end;//栈顶
    int length;
    void freeStack(){
        Node<T> *r = end, *t;
        while(r != first){
            t = r;
            r = r->next;
            delete t;
        }
        delete r;
    }
};
```

```
#endif
```

测试：

```
#include <iostream>
#include <cmath>
#include "stack.h"
using namespace std;

//判断是否是操作符
bool isOper(char ch){
    if((ch == '+') || (ch == '-') || (ch == '*') || (ch == '/') || (ch == '%') || (ch == '^'))
        return true;
    return false;
}

//计算表达式的值
int compute(int left, int right, char oper){
    int result;
    switch (oper)
    {
    case '+':
        result = left + right;
        break;
    case '-':
        result = left - right;
        break;
    case '*':
        result = left * right;
        break;
```

```
case '/':
    if (right == 0)
    {
        throw "除数不能为0!";
    }
    else
        result = left / right;
    break;
case '%':
    if (right == 0)
    {
        throw "除数不能为0!";
    }
    else
        result = left % right;
    break;
case '^':
    if ((left == 0) || (right == 0))
    {
        throw "操作数不能为0!";
    }
    else
        result = pow(right, left);
    break;
}
return result;
}

//获取左右操作数
void getOperands(stack<int> &t,int &left, int &right){
    if(t.empty())
        throw "操作符太多!";
    else
        right = t.pop();

    if(t.empty())
        throw "操作符太多!";
    else
```

```
        left = t.pop();
    }

int main(){
    //char a[] = {'3','2','+','5','*','3','2','*','-'};
    char a[] = {'3','2','5','+','*'};
    int len = sizeof(a)/sizeof(char);

    stack<char> s(a, len);
    s.printStack();
    s.push('8');
    s.printStack();
    s.pop();
    s.printStack();
    cout<<s.top()<<endl;
    cout<<s.empty()<<endl;
    cout<<s.size()<<endl;

    cout<<"后缀表达式计算"<<endl;
    stack<int> t;
    int left, right;
    for(int i=0; i<len; i++){
        try{
            if(!isOper(a[i])){
                t.push(a[i]-'0');
            }else{
                getOperands(t, left, right);
                t.push(compute(left, right, a[i]));
            }
        }catch(char const* s){
            cout<<"抛出异常："<<s<<endl;
        }
    }

    cout<<"计算后缀表达式的值："<<endl;
    try{
        cout<<t.top()<<endl;
    }
```

```
    }catch(StackOverFlowException &e){  
        cout<<"\n异常："<<e.what();  
    }  
    t.printStack();  
    return 0;  
}
```

5.说明

a.栈结构：

这里栈使用链表实现，好处是大小动态增长。

b.异常处理（我是自定义异常，没有继承exception）

两个参考网址：

<http://blog.csdn.net/btwsmile/article/details/6685549>

http://tech.ddvip.com/2006-12/116514811312868_2.html

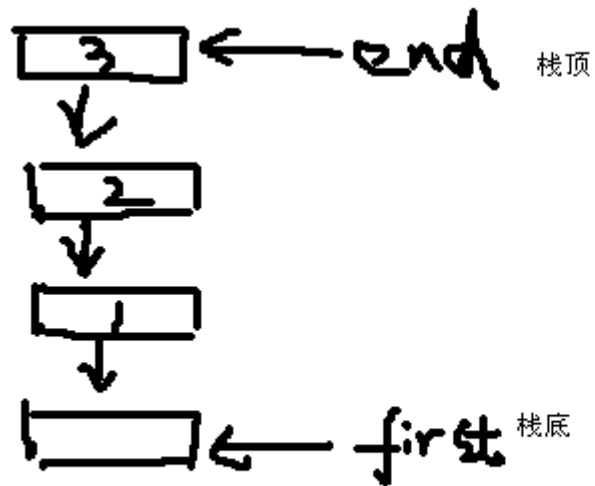
异常处理在c++中是一个非常好的特性，我们要充分利用。

在里面用到的地方：1.栈检测，检测是否达到栈底，尤其是pop(), top()操作

在使用时，要用try..catch捕获，防止异常发生。

2.是判断输入的合法性，看是否操作符过多等问题，如果是则要捕获异常

栈结构：



4.12 中缀表达式转换为后缀

发表时间: 2012-06-28 关键字: 栈, 中缀表达式, 后缀表达式

1. 算法描述

例如 $a+b*c$ 这是常见的中缀表达式，但是为了方便计算，在计算机中常要转换为后缀表达式 $abc*+$ 的形式，那如何转换呢？

用到的关键数据结构: 栈

转换的关键原则：

1. 优先级判断：关键是比较运算符的优先级, 谁的优先级高, 谁就出现在前面上面的表达式中, 有括号的时候括号优先级最高, *

/次之, +-最后. 在上面的表达式中+的优先级不如*的高, 因此, 在后缀表达式中*出现在+前面,

2. 操作数处理：遇到操作数的时候总是直接输出, 不做任何比较

3. 括号处理：遇到左括号总是直接入栈, 遇到右括号的时候总是弹栈, 一直弹到遇到一个左括号

4. 优先级处理：遇到操作符的时候就先将这个操作符和它前面的操作符比较优先级

a. 高于前面(栈顶)的优先级, 先将它压栈;

b. 低于或等于前面的操作符的优先级, 就把前面的优先级比它高的或相等的顺序弹出来, 一直弹到遇到优先级比它还低的或者到了栈顶, 然后在压栈.

2. 优先级

enum{FLAG = -1, // FLAG为栈顶标志, 优先级最低, 这样做的目的在于任何操作符都能够压栈 (**4. 优先级处理中有说明**)

L_BRACKET = 0, // 次与上面, 目的在于所有操作符都能在 (之后压栈 (**3. 括号处理**)

PLUS = 1, //然后其他就按常规规定

MINUS = 1,

MULTIPLY = 2,

DIVISON = 2,

PERSENT = 2,

POWER = 2};

分别为左括号,加减乘除的优先级定义,这儿有一个 FLAG = -1.是做什么咧?

假如分析上面的4点就会发现,有一些特例,比如第一个操作符入栈之前要跟前面的操作符比较优先级,

但是前面还没有操作符,就只好当做一个特例特别处理,先判断是否栈为空,然后操作,假如我们先将 一个标志符号压入栈,

并让它的优先级低于其他所有的操作符的优先级,这样它就永远不会被弹出, 而且消除了特例的判断,这是技巧

另外注意,把左括号的优先级定义的很低,这也是有道理的.因为我们总是当遇到右括号的时候才把左括号弹出来..

3.判断伪代码：

```
for(遍历所有字符){
```

```
    if(数字) 输出；
```

```
else{
```

```
    if(左括号) 入栈;
```

```
    else if(右括号) 弹栈,一直弹到遇到一个左括号；
```

```
else{
```

```
    判断操作符优先级；
```

```
    if(栈顶优先级小于当前优先级){
```

压栈；

}else{

将小于当前优先级的所有操作符出栈，然后入栈；

}

}

}

}

4.代码

```
#include <iostream>
#include "stack.h"
#include <cstring>
using namespace std;

//定义操作符优先级
enum{ FLAG = -1, // FLAG为栈顶标志,优先级最低,此时根据规则,则所有操作符都可以入栈
      L_BRACKET = 0, //左括号优先级最低
      PLUS = 1,
      MINUS = 1,
      MULTIPLY = 2,
      DIVISON = 2,
      PERSENT = 2,
      POWER = 2;
};

//获取操作符优先级
int getPri(char ch){
    switch(ch){
        case '+':
```

```
        return PLUS;
    case '-':
        return MINUS;
    case '*':
        return MULTIPLY;
    case '/':
        return DIVISON;
    case '%':
        return PERSENT;
    case '^':
        return POWER;
    case '(':
        return L_BRACKET;
    case '#':
        return FLAG;
}

//判断是否是操作符
bool isOper(char ch){
    if((ch == '+') || (ch == '-') || (ch == '*') || (ch == '/') || (ch == '%') || (ch == '^'))
        return true;
    return false;
}

//判断是否是括号
bool isBracket(char ch){
    if(ch == '(' || ch == ')')
        return true;
    else
        return false;
}

int main(){
    //string infix = "10/(2+3)"; (1+2)/2+(3+5)*7; "2*3/5-(2*3)"
    string infix = "(1+2+3)/2/3+(3+5)*7";
    cout<<"中缀表达式:"<<infix<<endl;
```

```
stack<char> opt; //栈
opt.push('#');
int len = infix.length();
int flag = 0;
char* result = new char[len]; //存储结果

for (int i = 0; i < len; i++)
{
    char ch = infix.at(i);
    //1.判断是否是操作数，如果是则直接打印，否则继续判断
    if(!isOper(ch) && !isBracket(ch)){
        result[flag++] = ch;
    }
    //2.如果是操作符，则判断是哪种类型，若是(则直接入栈，其他判断优先级决定出入栈
    else{
        if(ch == '('){
            opt.push(ch);
        }else if(ch == ')'){ //b.如果是)则，
            while(opt.top() != '('){
                result[flag++] = opt.pop();
            }
            opt.pop();
        }else{
            char t = opt.top(); //首先，看当前
            int pre = getPri(t);
            int cur = getPri(ch);
            //A.操作符比较规则1：高于栈顶的优先级,先将它压栈
            //(这里用了一个技巧，当栈为空此时栈底是'#'优先级最低-1，其次是'('为0
            if(cur > pre){
                opt.push(ch);
            }
            //B.操作符比较规则2：小于等于栈顶的优先级,把前面的优先级比它高的或相等
            else{
                result[flag++] = opt.pop();
                pre = getPri(opt.top());
            }
        }
    }
}
```

```
        while(pre >= cur){
            result[flag++] = opt.pop();
            pre = getPri(opt.top());
        }
        opt.push(ch);
    }

}

char ch = opt.top(); // 表达式扫描完后把栈中剩余的操作符全部输出
while (ch != '#')
{
    result[flag++] = ch;
    opt.pop();
    ch = opt.top();
}
//输出结果
cout<<"后缀表达式:";
for(int i=0; i<flag; i++){
    cout<<result[i];
}
cout<<endl;
delete[] result;
return 0;
}
```

5.

注1：栈stack.h和后缀表达式的计算见<http://hao3100590.iteye.com/blog/1569122>

谢谢！

4.13 栈的各种实现

发表时间: 2012-06-28 关键字: 栈, 数组, 双栈, 单链表

1.算法描述

- a.实现二个栈，在一个数组里面，除非没有任何空间剩余，否则不能有溢出声明
- b.实现一个没有头尾结点的栈（单链表）
- c.实现带有头结点的栈（单链表）

2.双栈

对于双栈，我们还可以添加resize()方法，当空间满了重新自动分配空间（new），就是将原来的两个栈，拷贝到新建立的数组上面去

a.dsexceptions.h

```
#ifndef DSEXCEPTIONS_H
#define DSEXCEPTIONS_H
//栈溢出异常
class StackOverFlowException{ //public: exception
public:
    const char* what() const throw()
    {
        return "stack over flow exception, the size<=0 !\n";
    }
};
//数组越界异常
class ArrayOverFlowException{
public:
    const char* what() const throw()
    {
        return "array over flow exception, the size over array length !\n";
    }
}
```

```
};  
//栈索引异常 ( 栈序号只能是1,2 )  
class ErrorIndexException{  
    public:  
        const char* what() const throw()  
        {  
            return  "the stack index only be 1 and 2 !\n";  
        }  
};  
  
#endif
```

b.stack2.h

```
/**  
    *实现二个栈，在一个数组里面，除非没有任何空间剩余，否则不能有溢出声明  
    **/  
#ifndef STACK2_H  
#define STACK2_H  
#include <iostream>  
#include "dsexceptions.h"  
  
enum{  
    LEFT = 1,           //左栈  
    RIGHT = 2           //右栈  
};  
  
template<class T>  
class stack2{  
    public:  
        stack2(int len){  
            array = new T[len];  
            leftTop = -1;  
            rightTop = len;  
            max = len;  
        }  
};
```



```
    }

    ~stack2(){
        delete [] array;
    }

    //出栈, flag为栈编号 ( 1为栈1, 2为栈2 )
    T pop(int flag){
        if(flag == LEFT){
            if(leftTop == -1)
                throw StackOverflowException();
            return array[leftTop--];
        }else if(flag == RIGHT){
            if(rightTop == max)
                throw StackOverflowException();
            return array[rightTop++];
        }else{
            throw ErrorIndexException();
        }
    }

    void push(int flag, T x){
        if(leftTop == rightTop)
            throw ArrayOverflowException();
        if(flag == LEFT){
            array[++leftTop] = x;
        }else if(flag == RIGHT){
            array[--rightTop] = x;
        }else{
            throw ErrorIndexException();
        }
    }

    T top(int flag) const{
        if(flag == LEFT){
            if(leftTop == -1)
                throw StackOverflowException();
```

```
        return array[leftTop];
    }else if(flag == RIGHT){
        if(rightTop == max)
            throw StackOverflowException();
        return array[rightTop];
    }else{
        throw ErrorIndexException();
    }
}

bool empty(int flag) const{
    if(flag == LEFT){
        if(leftTop == -1)
            return true;
    }else if(flag == RIGHT){
        if(rightTop == max)
            return true;
    }else{
        throw ErrorIndexException();
    }
    return false;
}

void printStack(int flag) const{
    if(flag == LEFT){
        std::cout<<"[";
        for(int i=leftTop; i>-1; i--){
            std::cout<<array[i]<<" ";
        }
        std::cout<<"]"<<std::endl;
    }else if(flag == RIGHT){
        std::cout<<"[";
        for(int i=rightTop; i<max; i++){
            std::cout<<array[i]<<" ";
        }
        std::cout<<"]"<<std::endl;
    }else{

```

```
        throw ErrorIndexException();
    }
}

int size(int flag) const{
    if(flag == LEFT){
        return leftTop+1;
    }else if(flag == RIGHT){
        return max-rightTop;
    }else{
        throw ErrorIndexException();
    }
}

private:
    int leftTop;           //左边数组栈顶
    int rightTop;          //右边数组栈顶
    int max;               //数组长度
    T* array;              //数组
};

#endif
```

c.main.cpp

```
#include <iostream>
#include "stack2.h"
using namespace std;

int main(){
    try{
        stack2<int> s(10);
        for(int i=0; i<4; i++){
            s.push(1, i);
            s.push(2, i);
        }
    }
}
```

```
    }

    cout<<"stack1:"<<endl;
    s.printStack(1);
    cout<<s.top(1)<<endl;
    cout<<s.pop(1)<<endl;
    cout<<s.size(1)<<endl;
    cout<<s.empty(1)<<endl;
    s.push(1, 9);
    s.push(1, 9);
    s.push(1, 9);
    cout<<s.size(1)<<endl;
    s.printStack(1);

    cout<<"stack2:"<<endl;
    s.printStack(2);
    cout<<s.top(2)<<endl;
    for(int i=0; i<4; i++)
        s.pop(2);
    cout<<s.size(2)<<endl;
    cout<<s.empty(2)<<endl;
    s.printStack(2);

}catch(StackOverFlowException &e){
    cout<<e.what();
}catch(ArrayOverFlowException &e){
    cout<<e.what();
}catch(ErrorIndexException &e){
    cout<<e.what();
}
return 0;
}
```

3.单链表实现没有头尾结点的栈

stack.h

```
#ifndef STACK_H
#define STACK_H
#include <iostream>
#include "dexceptions.h"

template<class T>
struct Node{
    Node<T>* next;
    T data;
};

template<class T>
class stack{
public:
    stack(){
        first = new Node<T>;
        first->next = NULL;
        length = 0;
    }

    stack(T a[], int n){
        first = new Node<T>;
        first->next = NULL;
        length = n;
        first->data = a[0];
        for(int i=1; i<n; i++){
            Node<T> *t = new Node<T>;
            t->data = a[i];
            t->next = first;
            first = t;
        }
    }
};
```

```
        }
    }

    ~stack(){
        freeStack();
    }

    T pop(){
        T data;
        if(length != 0){
            Node<T> *r = first;
            data = r->data;
            if(length == 1)           //当只剩下一个节点的时候不删除,保留方便下次
                first->data = 0;
            else{
                first = first->next;
                delete r;
            }
            length--;
        }else{
            throw StackOverFlowException();
        }
        return data;
    }

    void push(T x){
        if(length == 0) first->data = x;
        else{
            Node<T> *r = new Node<T>;
            r->data = x;
            r->next = first;
            first = r;
        }
        length++;
    }

    T top() const{
```

```
        if(length == 0)
            throw StackOverFlowException();
        return first->data;
    }

    bool empty() const{
        if(length == 0) return true;
        return false;
    }

    int size() const{
        return length;
    }

    void printStack() const{
        std::cout<<"[";
        Node<T> *r = first;
        while(r){
            if(length == 0) break; //当没有元素的时候，first
            std::cout<<r->data<<" ";
            r = r->next;
        }
        std::cout<<"]"<<std::endl;
    }

private:
    Node<T>* first;//栈顶(指向栈顶)
    int length;
    void freeStack(){
        Node<T> *r = first, *t;
        while(r){
            t = r;
            r = r->next;
            delete t;
        }
    }
};
```

```
#endif
```

main.cpp

```
#include <iostream>
#include "stack.h"
using namespace std;

int main(){
    stack<int> k;
    k.push(2);
    cout<<k.size()<<endl;
    k.pop();
    k.printStack();

    int a[] = {1,2,3,4,5,6};
    stack<int> s(a, 6);
    s.printStack();
    s.push(7);
    s.printStack();
    cout<<"size:"<<s.size()<<endl;
    s.pop();
    s.printStack();
    try{
        for(int i=0; i<6; i++){
            s.pop();
        }
        s.push(7);
        s.push(8);
        s.printStack();
        cout<<"size:"<<s.size()<<endl;
        s.pop();
        s.pop();
    }
```



```
        s.pop();
        s.printStack();
    }catch(StackOverFlowException &e){
        cout<<e.what();
    }

    return 0;
}
```

4.带有头结点的栈

stack.h

```
#ifndef STACK_H
#define STACK_H
#include <iostream>
#include "dsexceptions.h"

template<class T>
struct Node{
    Node<T>* next;
    T data;
};

template<class T>
class stack{
public:
    stack(){
        first = new Node<T>;
        first->next = NULL;
        end = first;
        length = 0;
    }
};
```

```
stack(T a[], int n){
    first = new Node<T>;
    length = n;
    Node<T> *r = first;
    for(int i=0; i<n; i++){
        Node<T> *t = new Node<T>;
        t->data = a[i];
        t->next = r;
        r = t;
    }
    end = r;
}

~stack(){
    freeStack();
}

T pop(){
    T data;
    if(length != 0){
        Node<T>* r = end;
        end = end->next;
        data = r->data;
        delete r;
        length--;
    }else{
        throw StackOverFlowException();
    }
    return data;
}

void push(T x){
    Node<T> *r = new Node<T>;
    r->data = x;
    r->next = end;
    end = r;
    length++;
}
```

```
    }

    T top() const{
        if(length == 0)
            throw StackOverFlowException();
        return end->data;
    }

    bool empty() const{
        if(length == 0) return true;
        return false;
    }

    int size() const{
        return length;
    }

    void printStack() const{
        std::cout<<"[";
        Node<T> *r = end;
        while(r != first){
            std::cout<<r->data<<" ";
            r = r->next;
        }
        std::cout<<"]"<<std::endl;
    }

private:
    Node<T>* first;//栈底(不存储元素)
    Node<T>* end;//栈顶 (指向栈顶)
    int length;
    void freeStack(){
        Node<T> *r = end, *t;
        while(r != first){
            t = r;
            r = r->next;
            delete t;
        }
    }
}
```

```
        }  
        delete r;  
    }  
};  
  
#endif
```

4.14 多种队列的实现

发表时间: 2012-06-29 关键字: 队列, 循环队列, 双端队列, 链表

1.算法描述

a.数据结构与算法 (Mark Allen Weiss) 3.28双端队列的实现,在队列的两端都可以进行插入和删除工作, 每种操作复杂度 $O(1)$.

b.没有头结点和尾结点的队列实现

c.循环数组的队列实现

2.算法实现

a.由于有复杂度的限制, 和两端插入删除, 故而使用数组是不适合的, 必须使用链表, 我这里使用的是双向链表, 在两端操作的复杂度就一样的, 非常方便

b.没有头尾结点实现队列, 关键就是注意空队列的判断, 在空队列情况下的插入删除操作。

c.循环数组就是在尾部循环的时候操作。

3.代码实现

a.双端队列

deque.h

```
//双端队列(两头都可以插入删除,故而使用的双向链表)
```

```
#ifndef DEQUE_H
#define DEQUE_H

template<class T>
struct Node{
    Node* next;
```

```
    Node* pre;
    T data;
};

template<class T>
class Deque{
public:
    Deque();
    ~Deque();
    void push(T x);                //插入双端队列的前端
    T pop();                       //删除前端元素并
    T top() const;                //前端元素
    void inject(T x);             //插入双端队列的尾端
    T eject();                    //从双端队列尾端删除并返
    T last() const;              //尾端元素
    int size() const;
    void printDeque() const;
private:
    Node<T> *first;               //双端队列的前端
    Node<T> *end;                //双端队列的尾端
    int length;                  //双端队列长度
    void freeDeque();
};

#endif
```

deque.cpp

```
#include <iostream>
#include "deque.h"
#include "dsexceptions.h"

template<class T>
Deque<T>::Deque(){
```

```
        first = new Node<T>;
        end = new Node<T>;
        first->next = end;
        first->pre = NULL;
        end->next = NULL;
        end->pre = first;
        length = 0;
    }

template<class T>
Deque<T>::~~Deque(){
    freeDeque();
}

//插入双端队列的前端
template<class T>
void Deque<T>::push(T x){
    Node<T> *r = first->next;
    Node<T> *t = new Node<T>;
    t->data = x;
    first->next = t;
    t->pre = first;
    t->next = r;
    r->pre = t;
    length++;
}

//删除前端元素并返回
template<class T>
T Deque<T>::pop(){
    if(first->next == end)
        throw DequeOverflowException();
    Node<T> *r = first->next;
    T data = r->data;
    first->next = r->next;
    r->next->pre = first;
    delete r;
}
```

```
        length--;\n        return data;\n    }\n\n    //前端元素\n    template<class T>\n    T Deque<T>::top() const{\n        if(first->next == end)\n            throw DequeOverflowException();\n        return first->next->data;\n    }\n\n    //插入双端队列的尾端\n    template<class T>\n    void Deque<T>::inject(T x){\n        Node<T> *r = new Node<T>;\n        r->next = NULL;\n        r->pre = end;\n        end->data = x;\n        end->next = r;\n        end = r;\n        length++;\n    }\n\n    //从双端队列尾端删除并返回\n    template<class T>\n    T Deque<T>::eject(){\n        if(first->next == end)\n            throw DequeOverflowException();\n\n        Node<T> *r = end->pre;\n        int data = r->data;\n        end->pre = r->pre;\n        r->pre->next = end;\n        delete r;\n        length--;\n        return data;\n    }\n}
```



```
//尾端元素
template<class T>
T Deque<T>::last() const{
    if(first->next == end)
        throw DequeOverflowException();
    return end->pre->data;
}

template<class T>
int Deque<T>::size() const{
    return length;
}

template<class T>
void Deque<T>::printDeque() const{
    std::cout<<"[";
    Node<T> *r = first->next;
    while(r != end){
        std::cout<<r->data<<" ";
        r = r->next;
    }
    std::cout<<"]"<<std::endl;
}

template<class T>
void Deque<T>::freeDeque(){
    Node<T> *r = first->next,*t;
    while(r != end){
        t = r;
        r = r->next;
        delete t;
    }
    delete first;
    delete end;
}
```

dsexceptions.h

```
#ifndef DSEXCEPTIONS_H
#define DSEXCEPTIONS_H

class DequeOverflowException{
public:
    const char* what() const throw()
    {
        return "deque over flow exception, the size<=0 !\n";
    }
};

#endif
```

main.cpp

```
#include <iostream>
#include "deque.cpp"
using namespace std;

int main(){
    Deque<int> d;
    cout<<"前端操作："<<endl;
    d.push(1);
    d.push(2);
    d.push(3);
    d.printDeque();
    cout<<d.size()<<endl;
    cout<<d.top()<<endl;
```

```
d.pop();
cout<<d.top()<<endl;

cout<<"后端操作："<<endl;
d.inject(5);
d.inject(6);
d.inject(7);
d.printDeque();
cout<<d.size()<<endl;
cout<<d.last()<<endl;
d.eject();
cout<<d.last()<<endl;
d.printDeque();
}
```

b.无头尾结点实现

queue.h

```
//链表实现队列，不含有头尾结点
#ifndef QUEUE_H
#define QUEUE_H

template<class T>
struct Node{
    Node* next;
    T data;
};

template<class T>
class Queue{
public:
    Queue();
    Queue(T a[], int n);
```

```
    ~Queue();  
    bool empty() const;           //对是否为空  
    void enQueue(T x);           //进队  
    T deQueue();                 //出队  
    T queueFront() const;        //返回队头元素  
    T queueRear() const;         //返回队尾元素  
    int size() const;            //队列大小  
    void printQueue() const;     //打印队列  
private:  
    Node<T>* front;              //队头，允许删除的一端  
    Node<T>* rear;              //队尾，允许插入的一端  
    int length;                 //队长度  
    void freeQueue();           //释放队列  
};  
  
#endif
```

queue.cpp

```
#include <iostream>  
#include "queue.h"  
#include "dsexceptions.h"  
  
template<class T>  
Queue<T>::Queue(){  
    front = new Node<T>;  
    front->next = NULL;  
    rear = front;  
    length = 0;  
}  
  
template<class T>  
Queue<T>::Queue(T a[], int n){  
    if(n == 0) Queue();  
    else{
```

```
        front = new Node<T>;
        front->data = a[0];
        front->next = NULL;
        rear = front;
        for(int i=1; i<n; i++){
            Node<T>* r = new Node<T>;
            r->data = a[i];
            rear->next = r;
            rear = r;
        }
        rear->next = NULL;
        length = n;
    }
}

template<class T>
Queue<T>::~~Queue(){
    freeQueue();
}

template<class T>
bool Queue<T>::empty() const{
    if(length == 0)
        return true;
    return false;
}

template<class T>
void Queue<T>::enqueue(T x){
    //注意不能使用rear == front判断，这样当队列为空，插入的元素始终在第一个位置
    if(length == 0) front->data = x; //不含有队头队尾，故而都要存储元素
    else{
        Node<T>* r = new Node<T>;
        r->data = x;
        r->next = NULL;
        rear->next = r;
        rear = r;
    }
}
```

```
    }
    length++;
}

template<class T>
T Queue<T>::deQueue(){
    T data;
    if(length == 0){
        throw QueueEmptyException();
    }else{
        data = front->data;
        //当为1的时候front==rear,我们不能将其删除,要保留front rear
        if(length == 1){
            front->data = 0;
        }else{
            Node<T>* r = front;
            front = r->next;
            delete r;
        }
        length--;
    }
    return data;
}

template<class T>
T Queue<T>::queueFront() const{
    if(length == 0){
        throw QueueEmptyException();
    }else{
        return front->data;
    }
}

template<class T>
T Queue<T>::queueRear() const{
    if(length == 0){
        throw QueueEmptyException();
    }
```

```
        }else{
            return rear->data;
        }
    }

template<class T>
int Queue<T>::size() const{
    return length;
}

template<class T>
void Queue<T>::printQueue() const{
    Node<T>* r = front;
    std::cout<<"[";
    while(r){
        //当front==rear的时候还有可能是空，此时没有删除front,rear
        if(length == 0) break;
        std::cout<<r->data<<" ";
        r = r->next;
    }
    std::cout<<"]"<<std::endl;
}

template<class T>
void Queue<T>::freeQueue(){
    Node<T>* r = front, *t;
    while(r){
        t = r;
        r = r->next;
        delete t;
    }
}
```

esexceptions.h

```
#ifndef DSEXCEPTIONS_H
#define DSEXCEPTIONS_H

class QueueEmptyException{
public:
    const char* what() const throw()
    {
        return "queue empty exception, the size<=0 !\n";
    }
};

#endif
```

main.cpp

```
#include <iostream>
#include "queue.cpp"
using namespace std;

int main(){
    try{
        Queue<int> q;
        cout<<q.empty()<<endl;
        cout<<q.size()<<endl;
        q.printQueue();
        q.enqueue(1);
        q.enqueue(2);
        cout<<q.size()<<endl;
        cout<<"出队"<<q.dequeue()<<endl;
        cout<<q.size()<<endl;
        q.printQueue();
        cout<<"出队"<<q.dequeue()<<endl;
        cout<<q.size()<<endl;
        //q.dequeue();
    }
```



```
int a[] = {1,2,3,4,5};
Queue<int> p(a, 5);
cout<<p.empty()<<endl;
cout<<p.size()<<endl;
p.printQueue();
p.dequeue();
p.dequeue();
p.dequeue();
p.enqueue(6);
p.printQueue();
cout<<p.size()<<endl;
cout<<"出队"<<p.dequeue()<<endl;
cout<<p.size()<<endl;
p.printQueue();

}catch(QueueEmptyException &e){
    cout<<e.what();
}
return 0;
}
```

c.循环队列实现

queue.h

```
//链表实现队列，不含有头尾结点
#ifndef QUEUE_H
#define QUEUE_H

template<class T>
class Queue{
public:
```

```
    Queue(int n);
    Queue(T a[], int n, int length);
    ~Queue();
    bool empty() const;           //对是否为空
    void enqueue(T x);           //进队
    T dequeue();                  //出队
    T queueFront() const;        //返回队头元素
    T queueRear() const;         //返回队尾元素
    int size() const;            //队列大小
    void printQueue() const;      //打印队列
private:
    int front;                   //队头，允许删除的一端
    int rear;                    //队尾，允许插入
    int max;                     //队列固定长度
    int length;                  //队长度
    T* array;                    //队列数组
};

#endif
```

queue.cpp

```
#include <iostream>
#include "queue.h"
#include "dsexceptions.h"

template<class T>
Queue<T>::Queue(int n){
    array = new T[n];
    length = 0;
    max = n;
    rear = front = -1;
}
```

```
template<class T>
Queue<T>::Queue(T a[], int n, int len){
    if(n>= len)
        throw QueueOverFlowException();
    array = new T[len];
    for(int i=0; i<n; i++)
        array[i] = a[i];
    length = n;
    max = len;
    front = 0;
    rear = n-1;
}

template<class T>
Queue<T>::~~Queue(){
    delete[] array;
}

template<class T>
bool Queue<T>::empty() const{
    if(length == 0)
        return true;
    return false;
}

template<class T>
void Queue<T>::enqueue(T x){
    if(length == max)
        throw QueueOverFlowException();
    if(length == 0){
        rear++;
        if(rear > max-1) rear %= max;
        array[rear] = x;
        front++; //此时队不空，front和rear指向同一个位置
    }
    else{
        rear ++;
    }
}
```

```
        if(rear > max-1) rear %= max;
        array[rear] = x;
    }
    length++;
}

template<class T>
T Queue<T>::deQueue(){
    T data;
    if(length == 0){
        throw QueueEmptyException();
    }else{
        data = array[front];
        if(length == 1) front = rear = -1; //将front , rear置为-1表示为空
        front++;
        if(front > max-1) front %= max;
        length--;
    }
    return data;
}

template<class T>
T Queue<T>::queueFront() const{
    if(length == 0){
        throw QueueEmptyException();
    }else{
        return array[front];
    }
}

template<class T>
T Queue<T>::queueRear() const{
    if(length == 0){
        throw QueueEmptyException();
    }else{
        return array[rear];
    }
}
```

```
}

template<class T>
int Queue<T>::size() const{
    return length;
}

template<class T>
void Queue<T>::printQueue() const{
    int a = front, b = length;
    std::cout<<"[";
    while(b){
        b--;
        std::cout<<array[a++]<<" ";
        if(a > max-1) a %= max;
    }
    std::cout<<"]"<<std::endl;
}
```

dsexceptions.h

```
#ifndef DSEXCEPTIONS_H
#define DSEXCEPTIONS_H

class QueueEmptyException{
public:
    const char* what() const throw()
    {
        return "queue empty exception, the size<=0 !\n";
    }
};

class QueueOverFlowException{
public:
```

```
        const char* what() const throw()
    {
        return "queue over flow exception, the size over the capacity !\n";
    }
};

#endif
```

main.cpp

```
#include <iostream>
#include "queue.cpp"
using namespace std;

int main(){
    try{
        int a[] = {1,2,3,4,5};
        Queue<int> p(a, 5, 7);
        cout<<p.empty()<<endl;
        cout<<p.size()<<endl;
        p.printQueue();
        p.deQueue();
        p.enqueue(6);
        p.enqueue(7);
        p.enqueue(8);
        //p.enqueue(9); //入队超出capacity
        p.printQueue();
        cout<<p.size()<<endl;
        cout<<"出队"<<p.deQueue()<<endl;
        cout<<p.size()<<endl;
        p.printQueue();

    }catch(QueueEmptyException &e){
        cout<<e.what();
    }
```

```
    }catch(QueueOverflowException &e){  
        cout<<e.what();  
    }  
    return 0;  
}
```

4.15 二叉树基本操作大全

发表时间: 2012-07-03 关键字: 二叉树, 操作, 创建, 插入, 遍历

1. 二叉树的基本操作

这里我有一个疑问：

在使用构造函数的时候，传参数的问题？

开始我是这么理解的-----只使用指针（其实指针本身就是一个地址，相当于引用，也会改变root建立起二叉树），而2指针的引用，相当于就是对记录了指针的地址，采用了二次引用，其实是没有必要的，一次就够了。但是实际上用的时候并不是这样？根本不能建立二叉树，原因是因为开始指针指向的是一个不确定的位置？然后我又实验了，分配空间先，但是还是不行？所以不知道为什么一定要使用双重指针或指针引用？不知哪位高人能解答之，谢谢了

就是这样void creatTree(Node<T>* root);

```
//声明类BiTree及定义结构BiNode,文件名为bitree.h
#ifndef BITREE_H
#define BITREE_H
template <class T>
struct Node    //二叉树的结点结构
{
    T data;
    Node<T> *lchild, *rchild;
};
//定义队列或栈空间大小
const int SIZE = 100;

template <class T>
class BiTree
{
public:
```



```
BiTree( ); //构造函数，初始化一棵二叉树，其前序序列由键盘输入(输入序列是前序遍历顺序)
BiTree(T* a, int n); //构造函数，输入序列是层序序列(非递归建立)
~BiTree(); //析构函数，释放二叉链表中各结点的存储空间
Node<T>* getRoot(); //获得指向根结点的指针
void preOrder(Node<T> *root); //前序遍历二叉树
void inOrder(Node<T> *root); //中序遍历二叉树
void postOrder(Node<T> *root); //后序遍历二叉树
void leverOrder(Node<T> *root); //层序遍历二叉树
void nrPreOrder(Node<T> *root); //前序遍历二叉树(非递归)
void nrInOrder(Node<T> *root); //中序遍历二叉树(非递归)
void nrPostOrder(Node<T> *root); //后序遍历二叉树(非递归)
```

//基本操作

```
Node<T>* searchNode(T x); //寻找值为x的节点
int biTreeDepth(Node<T>* root);
Node<T>* getParent(Node<T>* root, T x); //如果存在值为data的节点，返回其父节点
Node<T>* getLeftSibling(Node<T>* root, T x); //获取左兄弟
Node<T>* getRightSibling(Node<T>* root, T x); //获取右兄弟
int leafCount(Node<T>* root);
int nodeCount(Node<T>* root);
void deleteSibTree(Node<T>* root, T x); //删除节点x为根节点的子树
```

private:

```
Node<T> *root; //指向根结点的头指针
//1.使用双指针(指向指针的地址，这样就不再是拷贝，相当于一个引用)
//void creatTree(Node<T> **root); //有参构造函数调用(根地址的地址，也可指针的引用)
```

//2.使用引用(别名，可以直接对root操作)

```
void creatTree(Node<T>* &root);
```

//3.使用返回

```
//Node<T>* creatTree();
```

//4.只使用指针(其实指针本身就是一个地址，相当于引用，也会改变root建立起二叉树)

//而2指针的引用，相当于就是对记录了指针的地址，采用了二次引用，其实是没有必要的，一次就够了

//但是实际上用的时候并不是这样？根本不能建立二叉树，原因是因为开始指针指向的是一个不确定的位置？

//然后我又实验了，分配空间先，但是还是不行？所以不知道为什么一定要使用双重指针或指针引用？不知哪位高手

```
//void creatTree(Node<T>* root);
```

```
void createTree(Node<T>* &root, T* a, int n); //非递归建立二叉树
```

```
void search(Node<T>* root, T x, Node<T>* &p);          //搜索
void releaseTree(Node<T> *root);    //析构函数调用
};
#endif
```

2.基本操作实现

```
#include <iostream>
#include "bitree.h"
using namespace std;

//构造函数，初始化一棵二叉树，其前序序列由键盘输入
template <class T>
BiTree<T>::BiTree( ){
    //creatTree(&root); //1.双指针（指向指针的地址，故而使用取地址操作符，取出指针root的地址）
    creatTree(root);          //2.root的引用
    //root = creatTree();    //3.返回(实际上是this->root = createTree())
}

template <class T>
BiTree<T>::BiTree(T* a, int n){
    createTree(root, a, n);
}

//析构函数，释放二叉链表中各结点的存储空间
template <class T>
BiTree<T>::~~BiTree(){
    releaseTree(root);
}

//获得指向根结点的指针
template <class T>
Node<T>* BiTree<T>::getRoot(){
    return root;
}
```

//前序遍历二叉树

```
template <class T>
void BiTree<T>::preOrder(Node<T> *root){
    if(root){
        cout<<root->data<<" ";
        preOrder(root->lchild);
        preOrder(root->rchild);
    }
}
```

//中序遍历二叉树

```
template <class T>
void BiTree<T>::inOrder(Node<T> *root){
    if(root){
        inOrder(root->lchild);
        cout<<root->data<<" ";
        inOrder(root->rchild);
    }
}
```

//后序遍历二叉树

```
template <class T>
void BiTree<T>::postOrder(Node<T> *root){
    if(root){
        postOrder(root->lchild);
        postOrder(root->rchild);
        cout<<root->data<<" ";
    }
}
```

//层序遍历二叉树

```
template <class T>
void BiTree<T>::levelOrder(Node<T> *root){
    int front = 0;
    int rear = 0; //采用顺序队列，并假定不会发生上溢
    Node<T>* queue[SIZE]; //存放结点指针的数组
```

```
Node<T>* t; //临时结点指针
if(root == NULL) return;
else{
    queue[rear++] = root;
    while(front != rear){
        t = queue[front++];
        if(t) cout<<t->data<<" ";
        if(t->lchild) queue[rear++] = t->lchild;
        if(t->rchild) queue[rear++] = t->rchild;
    }
}

//层序遍历二叉树(利用循环队列)
/*
template <class T>
void BiTree<T>::levelOrder(Node<T> *root){
    int front = 0;
    int rear = 0; //采用顺序队列，并假定不会发生上溢
    Node<T>* queue[SIZE]; //存放结点指针的数组
    Node<T>* t; //临时结点指针
    if(root == NULL) return;
    else{
        queue[rear] = root;
        rear = (rear+1)%SIZE;
        while(front != rear){
            t = queue[front];
            front = (front+1)%SIZE;
            if(t) cout<<t->data<<" ";
            if(t->lchild){ queue[rear] = t->lchild; rear = (rear+1)%SIZE;}
            if(t->rchild){ queue[rear] = t->rchild; rear = (rear+1)%SIZE;}
        }
    }
}

}*/

//-----建立树的三种参数设置方法-----
```

```
//有参构造函数调用(前序递归建立树，输入顺序也必须是前序)
/*
template <class T>
void BiTree<T>::creatTree(Node<T> **root){
    T ch;
    cout<<"请输入创建一棵二叉树的结点数据"<<endl;
    cin>>ch;
    if(ch == "#") *root = NULL;
    else{
        *root = (Node<T>*)new Node<T>; // *root = new Node<T>;不过前面写法更好
        (*root)->data = ch;
        creatTree(&(*root)->lchild);
        creatTree(&(*root)->rchild);
    }
}
*/

/*
template <class T>
Node<T>* BiTree<T>::creatTree(){
    Node<T>* root;
    T ch;
    cout<<"请输入创建一棵二叉树的结点数据"<<endl;
    cin>>ch;
    if(ch == "#") root = NULL;
    else{
        root = new Node<T>;
        root->data = ch;
        root->lchild = creatTree();
        root->rchild = creatTree();
    }
    return root;
}
*/

template <class T>
```

```
void BiTree<T>::creatTree(Node<T>* &root){
    T ch;
    cout<<"请输入创建一棵二叉树的结点数据"<<endl;
    cin>>ch;
    if(ch == "#") root = NULL;
    else{
        root = new Node<T>;
        root->data = ch;
        creatTree(root->lchild);
        creatTree(root->rchild);
    }
}

template <class T>
void BiTree<T>::createTree(Node<T>* &root, T* s, int n){
    Node<T>* queue[SIZE];
    Node<T> *p, *m;
    int rear = 0, front = 0;
    if(s[0] == "#"){
        root = NULL;
    }else{
        root = new Node<T>;
        root->data = s[0];
        queue[rear++] = root;
    }
    int i=1;//注意不管出不出栈，或者是空节点，i都要往前，出一次栈对应i前进2
    while(front != rear && i<n)
    {
        p = queue[front++];
        if(s[i] != "#"){
            //出栈后遍历后面连续两个左右节点
            m = new Node<T>;
            m->data = s[i++];
            p->lchild = m;
            queue[rear++] = m;
            if(s[i] != "#"){ //右
                m = new Node<T>;
```

```
        m->data = s[i++];
        p->rchild = m;
        queue[rear++] = m;
    }else{
        i++;
        p->rchild = NULL;
    }
}
}else{
    i++;
    p->lchild = NULL;
    if(s[i] == "#"){
        p->rchild = NULL;
        i++;
    }else{
        m = new Node<T>;
        m->data = s[i++];
        p->rchild = m;
        queue[rear++] = m;
    }
}
}
//cout<<"----"<<i<<endl;
}
}

//析构函数调用
template <class T>
void BiTree<T>::releaseTree(Node<T> *root){
    if(root){
        //递归删除，先释放左右结点在根结点,不能是先根在左右
        releaseTree(root->lchild);
        releaseTree(root->rchild);
        delete root;
    }
}
```

```
//前序遍历二叉树(非递归)
//它是在进栈的时候输出
template <class T>
void BiTree<T>::nrPreOrder(Node<T> *root){
    Node<T>* stack[SIZE];
    int top = 0;

    Node<T>* t = root;
    while(t || top != 0){
        //一直遍历左孩子，直到左孩子为空
        while(t){
            cout<<t->data<<" ";
            stack[top++] = t;
            t = t->lchild;
        }
        //如果左孩子为空了，然后出栈，遍历右结点
        if(top != 0){
            t = stack[--top];
            t = t->rchild;
        }
    }
}
```

```
//中序遍历二叉树(非递归)
//它是在出栈的时候输出
template <class T>
void BiTree<T>::nrInOrder(Node<T> *root){
    Node<T>* stack[SIZE];
    int top = 0;

    Node<T>* t = root;
    while(t || top != 0){
        //一直遍历左孩子，直到左孩子为空
        while(t){
            stack[top++] = t;
            t = t->lchild;
        }
    }
```



```
        //如果左孩子为空了，然后出栈，遍历右结点
        if(top != 0){
            t = stack[--top];
            cout<<t->data<<" ";
            t = t->rchild;
        }
    }
}

//后序遍历二叉树(非递归)
//需要设置标志位，
template <class T>
void BiTree<T>::nrPostOrder(Node<T> *root){
    Node<T>* stack[SIZE]; //存储树结点
    int flag[SIZE];        //标记结点(0标记第一次出栈(未出)，1标记第二次出栈(可出))
    int top = 0;
    Node<T>* t = root;
    while(t || top != 0){
        while(t){
            stack[top] = t;
            flag[top] = 0;
            top++;
            t = t->lchild;
        }
        //可以出栈
        while(top != 0 && flag[top-1] == 1){
            t = stack[--top];
            cout<<t->data<<" ";
        }

        if(top != 0){ //第一次出栈(实际上并未出栈，只是标志位变为0)
            t = stack[top-1];
            flag[top-1] = 1;
            t = t->rchild;
        }else{
            t = NULL; //这是结束条件，当top==0,在置t为空才能结束整个循环
        }
    }
}
```

```
    }  
}  
  
//搜索节点x  
template <class T>  
void BiTree<T>::search(Node<T>* root, T x, Node<T>* &p){  
    if(root){  
        if(root->data == x){ p = root; return; }  
        search(root->lchild, x, p);  
        search(root->rchild, x, p);  
    }  
}
```

```
//寻找值为data的节点  
template <class T>  
Node<T>* BiTree<T>::searchNode(T x){  
    Node<T>* r = NULL;  
    if(root){  
        search(root, x, r);  
    }  
    return r;  
}
```

```
//二叉树深度  
template <class T>  
int BiTree<T>::biTreeDepth(Node<T>* root){  
    int hl, hr;  
    if(root){  
        hl = biTreeDepth(root->lchild);  
        hr = biTreeDepth(root->rchild);  
        return (hl>hr?hl:hr)+1;  
    }  
    return 0;  
}
```

```
//如果存在值为data的节点，返回其父节点
```

```
template <class T>
Node<T>* BiTree<T>::getParent(Node<T>* root, T x){
    Node<T> *q;
    if(!root) return NULL;
    else{
        //如果找到则返回
        if((root->lchild && root->lchild->data==x) || (root->rchild && root->rchild->data==x))
            return root;
        //如果递归左子树没找到，到右子树上找(把递归当成普通的操作一样)
    }
    //如果左子树找到了返回双亲
    if(q = getParent(root->lchild, x)) return q;
    //如果左子树没找到，则到右子树找
    else if(q = getParent(root->rchild, x)) return q;
    //如果左右都没找到则返回空
    else return NULL;
}

}

//获取左兄弟
template <class T>
Node<T>* BiTree<T>::getLeftSibling(Node<T>* root, T x){
    if(!root) return NULL;
    else{
        if(root->lchild && root->lchild->data==x){//如果左子树是x，则必然不存在左兄弟
            return NULL;
        }

        if(root->rchild && root->rchild->data==x){//如果右子树是x，看是否存在左子树，存在则返回左子树
            return root->lchild;
        }

        return getLeftSibling(root->lchild, x);
        return getLeftSibling(root->rchild, x);
    }
}

}
```

//获取右兄弟

```
template <class T>
```

```
Node<T>* BiTree<T>::getRightSibling(Node<T>* root, T x){
```

```
    if(!root) return NULL;
```

```
    else{
```

```
        if(root->rchild && root->rchild->data==x){
```

```
            return NULL;
```

```
        }
```

```
        if(root->lchild && root->lchild->data==x){//获取右兄弟，则只需要关心左节点，如果左
```

```
            return root->rchild;
```

```
        }
```

```
        return getRightSibling(root->lchild, x);
```

```
        return getRightSibling(root->rchild, x);
```

```
    }
```

```
}
```

//叶子个数

```
template <class T>
```

```
int BiTree<T>::leafCount(Node<T>* root){
```

```
    if(!root) return 0;
```

```
    else if(!root->lchild && !root->rchild){//如果是叶子节点，返回1
```

```
        return 1;
```

```
    }else{//如果左右子树不空，则递归遍历左右子树，结果就是左右子树叶子节点的和(递归加)
```

```
        return leafCount(root->lchild) + leafCount(root->rchild);
```

```
    }
```

```
}
```

//节点个数

```
template <class T>
```

```
int BiTree<T>::nodeCount(Node<T>* root){
```

```
    if(!root) return 0;
```

```
    else{//递归调用加
```

```
        //节点个数=左子树节点个数+右子树节点个数+1
```

```
        return nodeCount(root->lchild) + nodeCount(root->rchild) + 1;
```

```
    }  
}  
  
template <class T>  
void BiTree<T>::deleteSibTree(Node<T>* root, T x){  
    Node<T>* p;  
    if(root){  
        if(root->lchild && root->lchild->data==x){  
            p = root->lchild;  
            root->lchild = NULL;  
            //递归删除p的左右子树  
            releaseTree(p->lchild);  
            releaseTree(p->rchild);  
            delete p;  
        }else if(root->rchild && root->rchild->data==x){  
            p = root->rchild;  
            root->rchild = NULL;  
            //递归删除p的左右子树  
            releaseTree(p->lchild);  
            releaseTree(p->rchild);  
            delete p;  
        }else{//递归查找左右子树  
            deleteSibTree(root->lchild, x);  
            deleteSibTree(root->rchild, x);  
        }  
    }  
}
```

3.测试程序

```
//二叉树的主函数，文件名为bitreemain.cpp  
#include<iostream>  
#include<string>  
#include"bitree.cpp"
```

```
using namespace std;

int main()
{
    //注：BiTree<string> bt在输入的时候，必须是按照前序遍历的顺序输入节点，因为递归建立树的过程就
    //BiTree<string> bt;
    //string a[] = {"a", "b", "c", "#", "d", "#", "#", "#", "#", "#", "#"};
    //string a[] = {"a", "b", "c", "d", "#", "#", "#", "#", "#"};
    string a[] = {"a", "b", "c", "#", "d", "e", "f", "g", "#", "#", "#", "#", "#", "#", "#"};
    BiTree<string> bt(a, 15); //创建一棵树(按层序输入)
    Node<string>* root = bt.getRoot( ); //获取指向根结点的指针

    cout<<"-----前序遍历----- "<<endl;
    bt.preOrder(root);
    cout<<endl;
    cout<<"-----中序遍历----- "<<endl;
    bt.inOrder(root);
    cout<<endl;
    cout<<"-----后序遍历----- "<<endl;
    bt.postOrder(root);
    cout<<endl;
    cout<<"-----层序遍历----- "<<endl;
    bt.laverOrder(root);
    cout<<endl;

    cout<<"-----非递归前序遍历----- "<<endl;
    bt.nrPreOrder(root);
    cout<<endl;
    cout<<"-----非递归中序遍历----- "<<endl;
    bt.nrInOrder(root);
    cout<<endl;
    cout<<"-----非递归后序遍历----- "<<endl;
    bt.nrPostOrder(root);

    cout<<"\n-----基本操作-----"<<endl;
    Node<string>* t;
```

```
cout<<"搜索b:";
t = bt.searchNode("b");
if(t) cout<<t->data<<endl;
else cout<<"无"<<endl;
if(t->lchild) cout<<"L:"<<t->lchild->data<<endl;
if(t->rchild) cout<<"R:"<<t->rchild->data<<endl;

cout<<"a的双亲:";
t = bt.getParent(root, "a");
if(t) cout<<t->data<<endl;
    else cout<<"无"<<endl;

cout<<"b的双亲:";
t = bt.getParent(root, "b");
if(t) cout<<t->data<<endl;
    else cout<<"无"<<endl;

cout<<"b的左兄弟:";
t = bt.getLeftSibling(root, "b");
if(t) cout<<t->data<<endl;
    else cout<<"无"<<endl;

cout<<"b的右兄弟:";
t = bt.getRightSibling(root, "b");
if(t) cout<<t->data<<endl;
    else cout<<"无"<<endl;

cout<<"c的左兄弟:";
t = bt.getLeftSibling(root, "c");
if(t) cout<<t->data<<endl;
    else cout<<"无"<<endl;

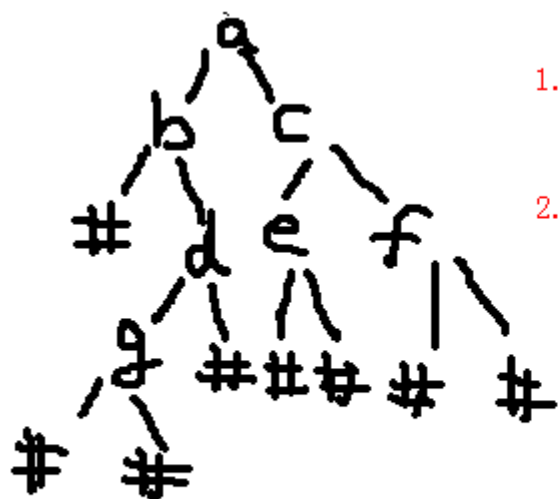
cout<<"c的右兄弟:";
t = bt.getRightSibling(root, "c");
if(t) cout<<t->data<<endl;
    else cout<<"无"<<endl;
```

```
cout<<"二叉树深度："<<bt.biTreeDepth(root)<<endl;
cout<<"叶子个数："<<bt.leafCount(root)<<endl;
cout<<"节点个数："<<bt.nodeCount(root)<<endl;

cout<<"删除节点为d的子树"<<endl;
bt.deleteSibTree(root, "d");
cout<<"-----前序遍历----- "<<endl;
bt.preOrder(root);
cout<<endl;
cout<<"-----中序遍历----- "<<endl;
bt.inOrder(root);
cout<<endl;
return 0;
}
```

4.总结

每个操作的一些注意事项都在代码中进行了说明，主要注意一下，建立二叉树的时候，如何输入节点？见下图：



1. 前序递归建树输入: `ab#dg###ce##f##`
必须按照前序遍历顺序输入, 否则建树就不正确, 这点注意! 叶子节点必须左右都以#代替来建立二叉树
2. 层序建树的输入: `abc#defg#####`
层序建立也一样, 先画出右图所示的树, 然后在输入

注: 有一个小技巧看输入是否正确, 就是#的个数是节点个数+1!

4.16 线索二叉树

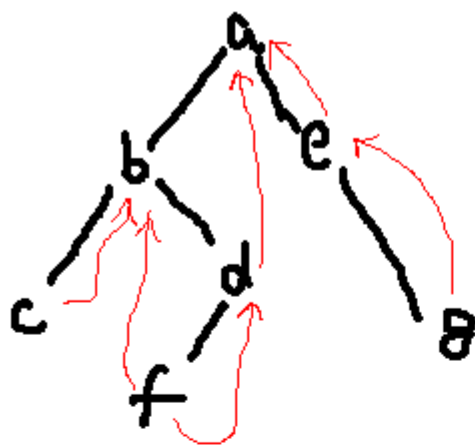
发表时间: 2012-07-04 关键字: 线索二叉树, 遍历, 线索化

1. 算法描述

就是简单的线索二叉树的建立, 遍历, 查找等基本操作, 具体什么是线索二叉树, 百度一下!

2. 算法说明

具体说明有四点: 如下图



建树利用前序:
abc##df###e#g##

中序线索二叉树:

1. 建立之后, 每个节点多出了rflag, lflag用于标记该节点是否有线索。如果没有孩子结点就存线索, lchild指向前序, rchild指向后续结点。
2. 建立索引的好处:
 - a. 避免使用递归遍历操作, 也不再需要栈, 提高了遍历二叉树的效率和节约空间。
 - b. 在未线索化之前, n个结点的二叉链表中有n+1个空指针域, 这样浪费了空间, 没有有效利用, 如果进行线索化之后, 并没有增加多少空间(rflag, lflag), 但是却利用了空指针域, 提高了空间利用率。
3. 注意: 第一个结点没有前驱, 最后一个没有后继结点。
4. 在遍历的过程中判断是否有左右结点, 不能使用root->lchild != NULL, root->rchild != NULL, 因为已经线索化, 除了第一个和最后一个结点, 其余的都不是空, 只能通过判断rflag, lflag来确定是否有孩子还是线。

注：尤其要注意第4点，我在写的时候就没注意这个问题，结果遍历的时候出现了无限循环，找了半天才找到！主要是建立二叉树判断的惯性思维，故而容易出现错误！

3.代码实现

头文件

```
#ifndef BITHRTREE_H
#define BITHRTREE_H

enum flag{CHILD, THREAD}; //枚举类型，枚举常量Child=0，Thread=1
template<class T>
struct Node{ //二叉线索树的结点结构
    Node<T> *rchild, *lchild;
    int lflag, rflag;
    T data;
};

template<class T>
class BiThrTree{
public:
    BiThrTree();
    ~BiThrTree();
    Node<T>* getRoot( ); //获取根结点
    Node<T>* next(Node<T>* p); //查找结点p的中序后继结点
    void inOrder(Node<T>* root); //中序遍历线索链表
    Node<T>* inPreNode(Node<T>* p); //查找结点p的中序前驱结点
    Node<T>* searchNode(T x); //查找结点x

private:
    Node<T>* root; //指向线索链表的头指针
    void creatThrTree(Node<T>* &root); //创建二叉树
    void biThrTree(Node<T>* &root, Node<T>* &pre); //二叉树的线索化(中序线索二叉树)
    void release(Node<T>* root); //析构函数调用
};
```

```
};  
  
#endif
```

bithrtree.cpp

标//<-----notice----->就是容易出错的地方！！

```
#include <iostream>  
#include "bithrtree.h"  
using namespace std;  
  
template<class T>  
BiThrTree<T>::BiThrTree(){  
    Node<T>* pre = NULL;  
    creatThrTree(root);  
    biThrTree(root, pre);  
}  
  
template<class T>  
BiThrTree<T>::~~BiThrTree(){  
    release(root);  
}  
  
//获取根结点  
template<class T>  
Node<T>* BiThrTree<T>::getRoot( ){  
    return root;  
}  
  
//查找结点p的中序后继结点  
template<class T>  
Node<T>* BiThrTree<T>::next(Node<T>* p){  
    Node<T>* t;
```

```
//如果有右子树，则后继就是右子树的第一个结点最左节点，如果第一个结点没有左子树，则就是右子树第-
if(p->rflag == CHILD){ //<-----notice----->
    t = p->rchild;
    while(t->lflag == CHILD){//即存在左子树.注：不能使用t->lchild判断，因为二叉树已经线
        t = t->lchild;
    }
    return t;
}else{
    return p->rchild;
}
}
```

//中序遍历线索链表

```
template<class T>
```

```
void BiThrTree<T>::inOrder(Node<T>* root){
```

```
    if(root){
```

```
        //先找到最左的结点
```

```
        while(root->lflag == CHILD){//不能是root->lchild<-----notice-----
```

```
            root = root->lchild;
```

```
        }
```

```
        //开始中序遍历
```

```
        do{
```

```
            cout<<root->data<<" ";
```

```
            root = next(root);
```

```
        }while(root);
```

```
    }
```

```
}
```

//查找结点p的中序前驱结点

```
template<class T>
```

```
Node<T>* BiThrTree<T>::inPreNode(Node<T>* p){
```

```
    Node<T>* t;
```

```
    if(p->lflag == THREAD){//<-----notice----->
```

```
        return p->lchild;
```

```
    }else{
```

```
        //如果p有左孩子，那么左孩子的最右结点就是其前驱
```

```
        t = p->lchild;
```

```
        while(t->rflag == CHILD){//有右孩子(不要用t->rchild)<-----notice----->
            t = t->rchild;
        }
        return t;
    }
}

//查找结点x
template<class T>
Node<T>* BiThrTree<T>::searchNode(T x){
    Node<T>* t = NULL;
    if(root){
        //先找到最左的结点
        while(root->lflag == CHILD){//<-----notice----->
            root = root->lchild;
        }
        //开始中序遍历
        do{
            if(root->data == x){
                t = root;
                break;
            }
            root = next(root);
        }while(root);
    }
    return t;
}

//创建二叉树
template<class T>
void BiThrTree<T>::creatThrTree(Node<T>* &root){
    T ch;
    cout<<"请输入创建一棵二叉树的结点数据"<<endl;
    cin>>ch;
    if(ch == "#") root = NULL;
    else{
        root = new Node<T>;
    }
}
```

```
        root->data = ch;
        creatThrTree(root->lchild);
        creatThrTree(root->rchild);
    }
}

//二叉树的线索化(中序线索二叉树)
template<class T>
void BiThrTree<T>::biThrTree(Node<T>* &root, Node<T>* &pre){
    if(root){
        biThrTree(root->lchild, pre);

        //<-----notice----->
        //左孩子处理 ( lflag和前驱 )
        if(!root->lchild){
            root->lflag =  THREAD;
            root->lchild = pre;
        }else{
            root->lflag =  CHILD;
        }
        //右孩子处理(rflag不能处理后序, 因为还没遍历到)
        if(!root->rchild){
            root->rflag =  THREAD;
        }else{
            root->rflag =  CHILD;
        }
        //pre的后续 ( rflag之前已经处理, 如果设置为了索引, 设置右索引为root )
        if(pre){
            if(pre->rflag ==  THREAD) pre->rchild = root;
        }
        //<-----notice----->

        //记录前驱
        pre = root;
        biThrTree(root->rchild, pre);
    }
}
```

```
//析构函数调用
template<class T>
void BiThrTree<T>::release(Node<T>* root){
    if(root){
        release(root->lchild);
        release(root->rchild);
        delete root;
    }
}
```

main.cpp

```
#include <iostream>
#include <string>
#include "bithrtree.cpp"
using namespace std;

int main(){
    BiThrTree<string> bt;
    Node<string>* t = bt.getRoot();

    cout<<"-----中序遍历-----"<<endl;
    bt.inOrder(t);
    cout<<endl;

    Node<string>* r;
    cout<<"查找结点b:";
    r = bt.searchNode("b");
    if(r) cout<<r->data<<endl;
    else cout<<"无"<<endl;
```



```
cout<<"b的前驱:";
r = bt.inPreNode(r);
if(r) cout<<r->data<<endl;
else cout<<"无"<<endl;

cout<<"a的前驱:";
r = bt.searchNode("a");
r = bt.inPreNode(r);
if(r) cout<<r->data<<endl;
else cout<<"无"<<endl;

cout<<"d的前驱:";
r = bt.searchNode("d");
r = bt.inPreNode(r);
if(r) cout<<r->data<<endl;
else cout<<"无"<<endl;

return 0;
}
```

4.总结

基本上只要明白了二叉树的基本原理，线索二叉树就非常的简单，就多了一个线索化的过程，简化了搜索前序和后继结点的时间，提高了效率！

如果有什么不对的地方，欢迎指正！

4.17 构造哈夫曼树

发表时间: 2012-07-04 关键字: 哈夫曼树, huffman

1.算法说明

就是建造哈夫曼树，从而使得构造出的树带权路径长度最小

2.步骤

输入叶子结点个数n；

创建长度为 $2*n-1$ 的数组并初始化；

while($i < n$) 循环输入n个叶子结点的权值；

while($n-1$ 次循环建立树){

在parent == -1的元素中查找权最小的两个结点；

合并两个叶子结点，并加入新结点到数组；

}

3.代码

```
//构造haffman树
#include <iostream>
using namespace std;
const int MAX = 10000;
struct Node{
    int weight;           //权值
    int parent;           //双亲
    int lchild;
    int rchild;
};
```

```
//创建一个haffman树
void createHaffman(Node* &a, int n){
    int m1,m2, x1, x2;//m1,m2是最小的两个值，x1,x2是他们的位置
    //n个结点，只需要n-1次就可以构造好
    for(int i=0; i<n-1; i++){
        m1 = m2 = MAX;
        x1 = x2 = 0;
        //查找最小值，在查找到最小的两个值后，构造新的节点，并加入到a中（数组的n+i个节点之后）
        for(int j=0; j<n+i; j++){
            //首先必须满足，还没有双亲的孤立节点,查找m1,m2都是最小
            if(a[j].parent == -1 && a[j].weight < m1){
                m1 = a[j].weight;
                x1 = j;
            }else if(a[j].parent == -1 && a[j].weight < m2){
                m2 = a[j].weight;
                x2 = j;
            }
        }
        //新的节点存入n+i,并设置x1, x2的双亲
        a[x1].parent = n+i;
        a[x2].parent = n+i;
        a[n+i].weight = a[x1].weight + a[x2].weight;
        a[n+i].parent = -1;
        a[n+i].lchild = x1;
        a[n+i].rchild = x2;
    }
}

//测试得到最小和次小的值
void test(){
    int a[] = {3,4,7,0,79,9,12,1,4};
    int m = MAX,k = MAX;
    for(int i=0; i<9; i++){
        if(a[i]<m){
            m = a[i];
        }else if(a[i]<k){
```

```
        k = a[i];
    }

}

cout<<m<<" "<<k<<endl;
}

int main(){
    int n;
    cout<<"输入叶子节点个数：";
    cin>>n;
    Node* a = new Node[2*n - 1];
    for(int i=0; i<2*n-1; i++){//初始化
        a[i].weight = 0;
        a[i].parent = -1;
        a[i].lchild = -1;
        a[i].rchild = -1;
    }

    cout<<"输入前n个叶子结点的权值"<<endl;
    for(int i=0; i<n; i++){
        cin>>a[i].weight;
    }

    cout<<"输出构造好的haffman树"<<endl;
    createHaffman(a, n);
    for(int i=0; i<2*n-1; i++){
        cout<<"["<<a[i].weight<<","<<a[i].parent<<","<<a[i].lchild<<","<<a[i].rchild<<
    }

    delete[] a;
    return 0;
}
```

4.18 B-树

发表时间: 2012-07-04 关键字: B树, 插入, 删除, 提升

1.B-树的概念

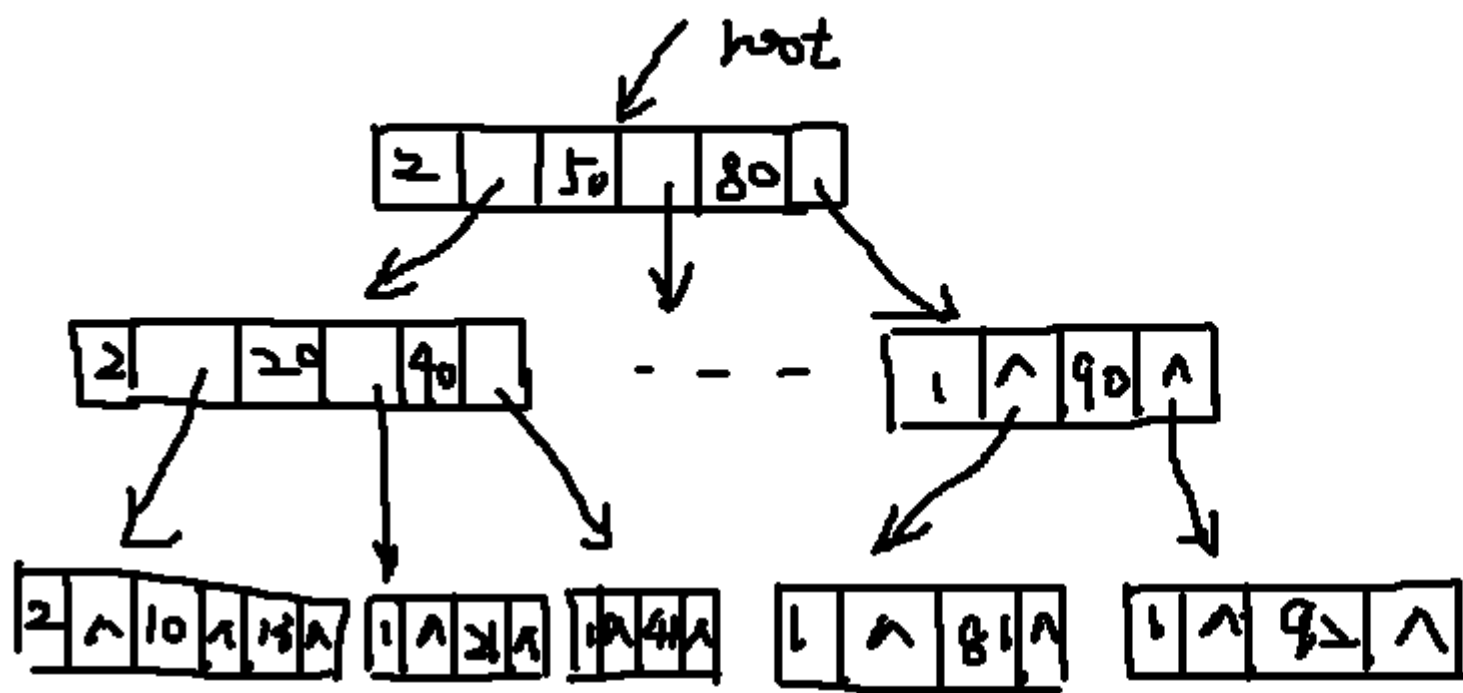
是一种多路搜索树，适合在磁盘等直接存取设备上组织动态的查找表，可能部分数据不在内存中。它作为索引文件的一种重要存储结构（数据库索引）

对于 m 阶（ $m \geq 3$ ）B-tree,满足如下特性：

- 1) 树中每个节点至多有 m 个节点
- 2) 根节点子树个数在：2--- m （根非叶子节点）
- 3) 非根节点子树个数在： $m/2$ (向上取整)--- m 。
- 4) 排列规则：所有叶节点在同一层，按照递增次序排列。

由于节点关键字个数比子树个数少1，故而关键字个数满足：

- 1) 根节点关键字个数在：1--- $m-1$ （根非叶子节点）
- 2) 非根节点 关键字 个数在： $m/2$ (向上取整)-1--- $m-1$ 。



关键字个数 n	指针域 P_0	关键字 K_1	指针域 P_1	...
--------------	-----------	-----------	-----------	-----

关键字 $K_i < K_{i+1}$, 递增排列; 指针域个数比关键字个数多1— $n+1$

2.B-tree的搜索特性

- 1) 关键字集合分布在整颗树中;
- 2) 任何一个关键字出现且只出现在一个结点中;
- 3) 搜索有可能在非叶子结点结束;

- 4) .其搜索性能等价于在关键字全集内做一次二分查找；
- 5) .自动层次控制；

由于限制了除根结点以外的非叶子结点，至少含有 $M/2$ 个儿子，确保了结点的至少利用率，其最底搜索性能为： $O(\log n)$

n 个关键字的 m 阶B-tree的最大深度 $\leq 1 + \log_{\lfloor m/2 \rfloor}(n+1)/2$ (其中 $\lfloor \cdot \rfloor$ 代表是底为 $m/2$ 的下界)

3.B-树的诞生

a.就树本身来说

由于在之前我们已经知道了，简单二叉树，排序二叉树，然后到平衡二叉树，对树的结构一步步的进行了改进！那么它为什么要改进？目的就是为了找到最优的数据存储方式，通过这样存储数据，使得查找，插入等基本的数据操作变得非常快捷，方便，节约时间和空间！在每次改进都有很大的进步。但是对于这些二叉树，最大的局限是在内存操作级别效率是很高的，但是也会随着高度的增加，效率逐渐降低！时间也许就会花在遍历查找！尤其是数据量非常之大，如百万级的数据，你想想要是建立一个二叉树，其高度至少是 $\log(M+1)$ ， M 为数据量，如果 $M=100000000$ （1亿），那么高度是多少可想而知！况且这么多数据有时候也不可能全部存入内存！！那么怎么办？就是要寻找一种数据结构，以最低的高度存储最多的数据量！效率要足够高，这样，多路搜索树应运而生！

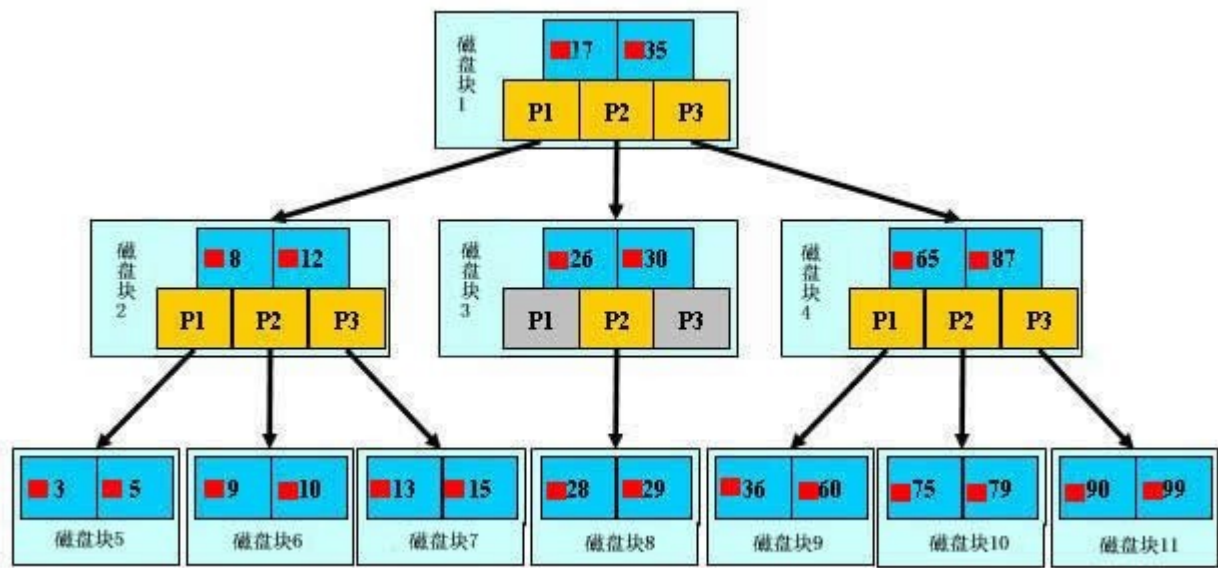
b.就硬件层次来说

其来源起源于提高外部数据的读取速度，因为有时候读取的数据非常之多，不可能全部读入内存，而是部分存于外部存储器，这样进行数据查找等就相当费时

那么，怎么办？就要找一种有效的数据组织方式，使在外存查找数据的时间减到最少！

具体详细说明见：http://blog.csdn.net/v_july_v/article/details/6530142

上面博文中，数据全部都是在外存，是需要什么数据才将其读入内存，故而这就要求，数据组织方式要求树尽量低，这样进行的I/O操作就会降低到最少！如下：



为了简单，这里用少量数据构造一棵3叉树的形式，实际应用中的B树结点中关键字很多的。上面的图中比如根结点，其中17表示一个磁盘文件的文件名；小红方块表示这个17文件内容在硬盘中的存储位置；p1表示指向17左子树的指针。

假如每个盘块可以正好存放一个B树的结点（正好存放2个文件名）。那么一个BTNODE结点就代表一个盘块，而子树指针就是存放另外一个盘块的地址。

下面，咱们来模拟下查找文件29的过程：

- a.根据根结点指针找到文件目录的根磁盘块1，将其中的信息导入内存。【磁盘IO操作 1次】
- b.此时内存中有两个文件名17、35和三个存储其他磁盘页面地址的数据。根据算法我们发现 $17 < 29 < 35$ ，因此我们找到指针p2。
- c.根据p2指针，我们定位到磁盘块3，并将其中的信息导入内存。【磁盘IO操作 2次】
- d.此时内存中有两个文件名26、30和三个存储其他磁盘页面地址的数据。根据算法我们发, $26 < 29 < 30$ ，因此我们找到指针p2。
- e.根据p2指针，我们定位到磁盘块8，并将其中的信息导入内存。【磁盘IO操作 3次】
- f.此时内存中有两个文件名28，29。根据算法我们查找到文件名29，并定位了该文件内存的磁盘地址。

分析上面的过程，发现需要**3次磁盘IO操作和3次内存查找操作**。关于内存中的文件名查找，由于是一个有序表结构，可以利用折半查找提高效率。至于IO操作是影响整个B树查找效率的决定因素。

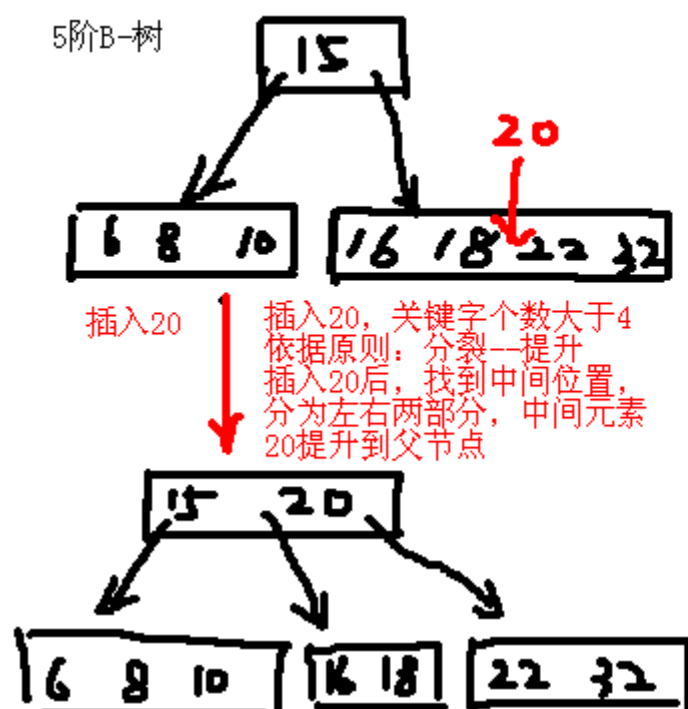
当然，如果我们使用平衡二叉树的磁盘存储结构来进行查找，磁盘4次，最多5次，而且文件越多，B树比平衡二叉树所用的磁盘IO操作次数将越少，效率也越高。

4.B-tree的基本操作

这里有一个关于插入和删除操作的**模拟操作动画**，可以看到插入删除的过程（非常好哦）：<http://slady.net/java/bt/view.php>

a.插入

5阶B-树



插入20

插入20，关键字个数大于4
依据原则：分裂—提升
插入20后，找到中间位置，
分为左右两部分，中间元素
20提升到父节点

说明：

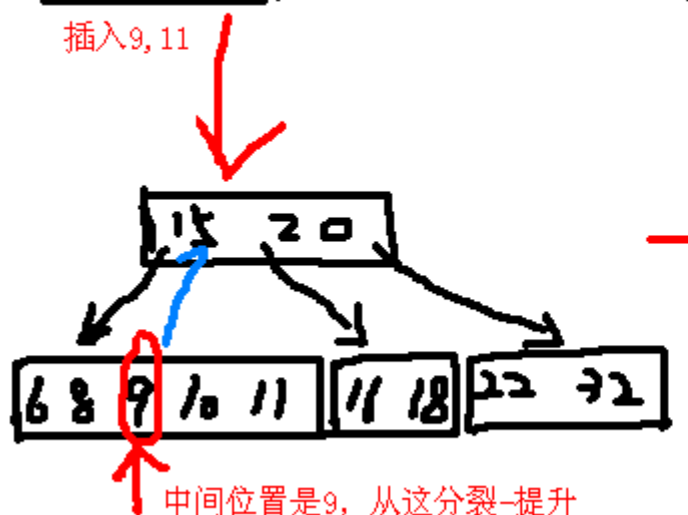
1. 根据B-树特性，得出关键字个数的原则：
关键字个数是节点子树个数-1，故而有：

- a. 根节点（非叶子）关键字个数是 $1 \sim m-1$ 个；
- b. 非根节点关键字个数是 $(m/2(\text{下界})-1) \sim m-1$ 个。

故而对于本题，5阶B-树
根节点关键字个数是：1--4个
非根是：2--4个

如果大于上面的范围就要：分裂—提升

分裂—提升：将 $(m/2(\text{下界})-1)$ 处即中间位置的元素提升，并分成左右两个部分，重复这个过程，直到根节点。



插入9, 11

中间位置是9，从这分裂—提升

2. 删除

3阶B-树

根据前面插入的分析基本原则，要保证其：

根节点关键字个数： $1 \sim m-1$

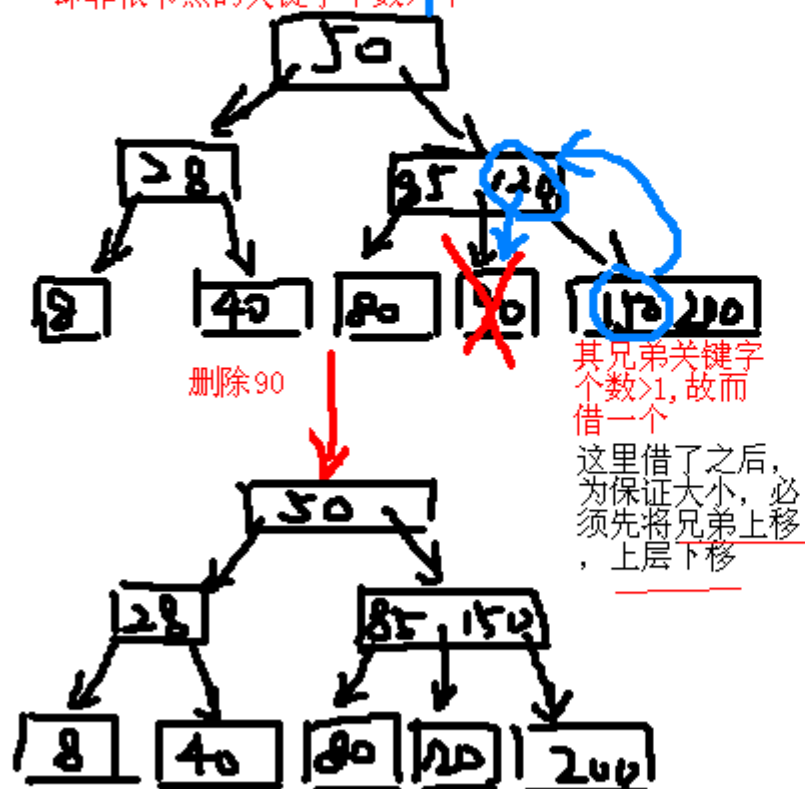
非根节点个数： $1 \sim \lfloor m/2 \rfloor - 1 \sim m-1$

1. 最简单情况

要删除的节点，删除后关键字个数在上述范围，直接删除即可

2. 兄弟货源充足，借！

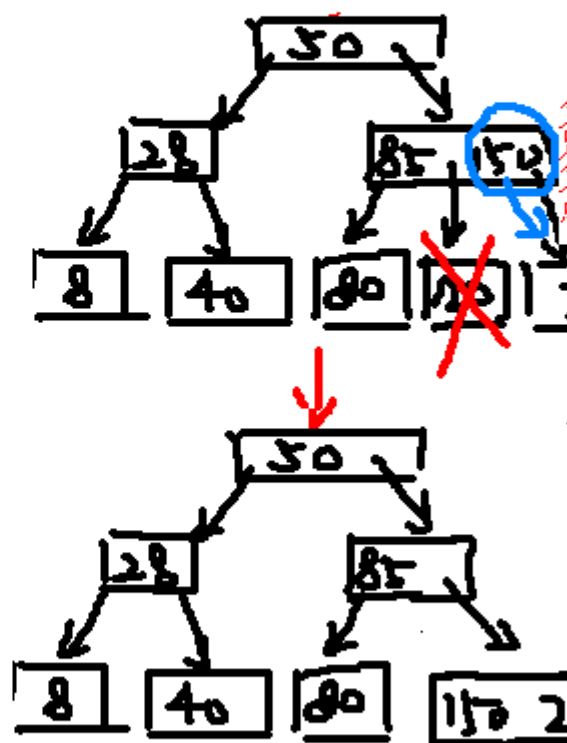
即非根节点的关键字个数 > 1 个



删除思路：

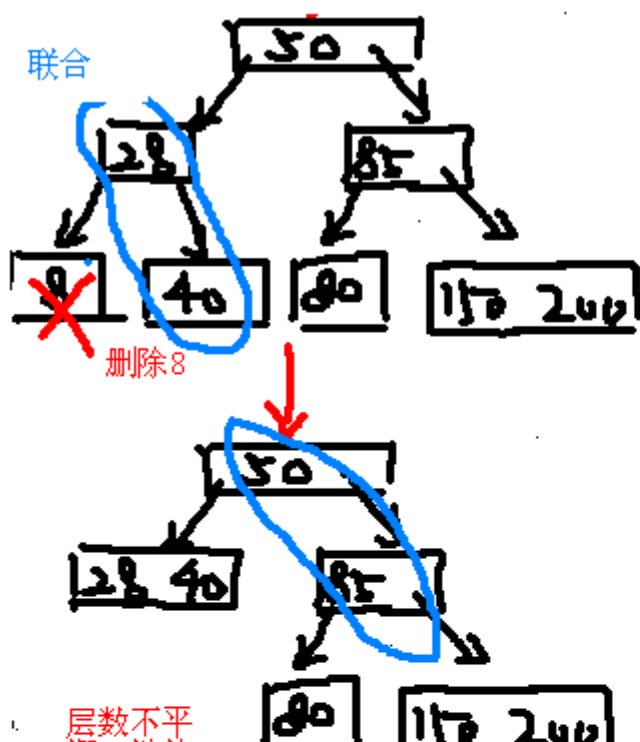
1. 先看兄弟，兄弟可借，借之，
2. 在看父节点，进行合并

3. 兄弟仅能自保 ($n=1$)，借父亲！
兄弟都是 $\lfloor m/2 \rfloor - 1$ 个关键字，无论哪一层父亲是充足的，则借父亲！



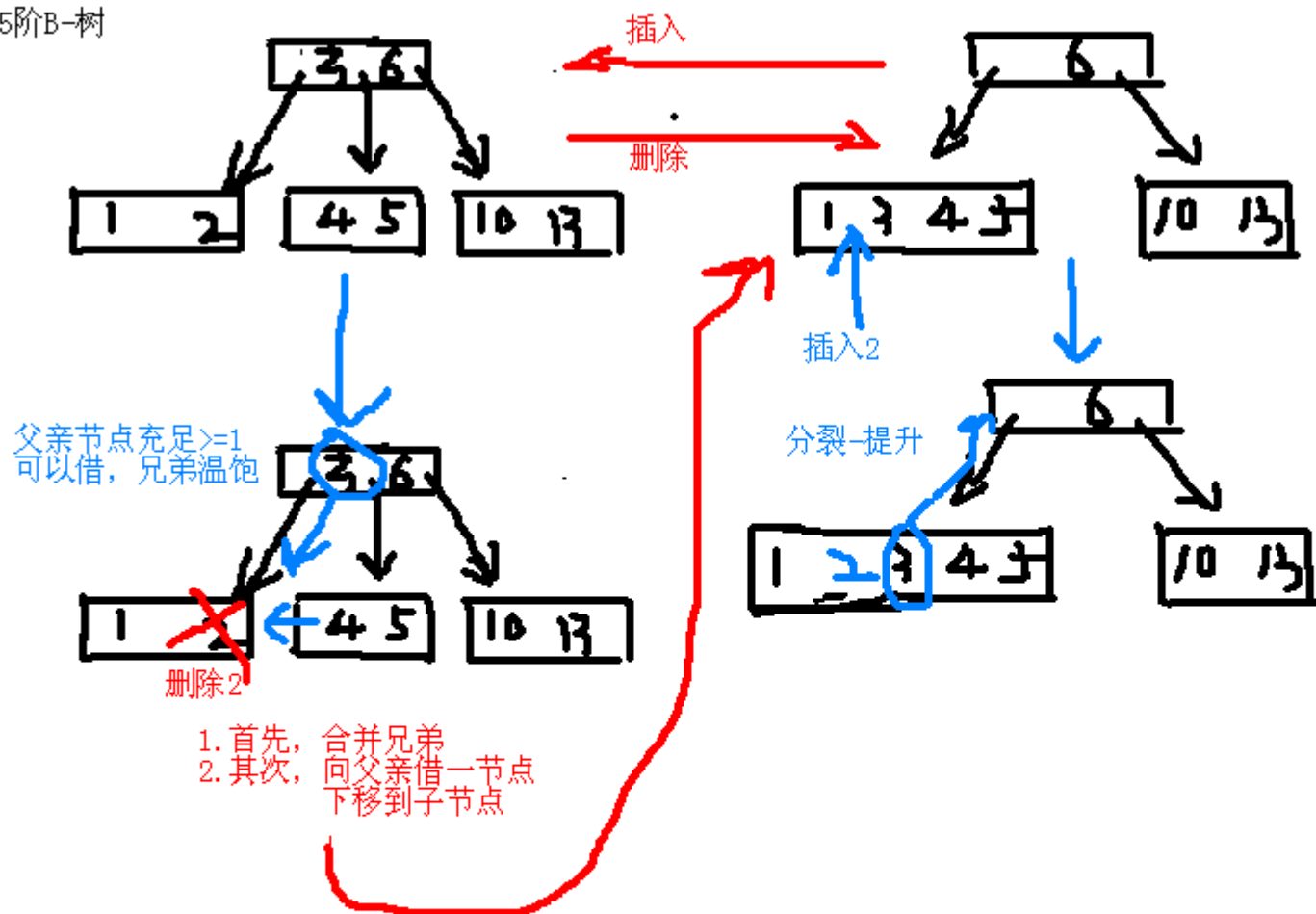
4. 父亲和兄弟都仅能自保 ($n=1$)！只有和兄弟联手抗曹！即合并（或者兄弟和父亲联合）

5. 最后还有一种比较复杂的情况，就是当兄弟和父亲联合的时候，就相对复杂些了，见下面！

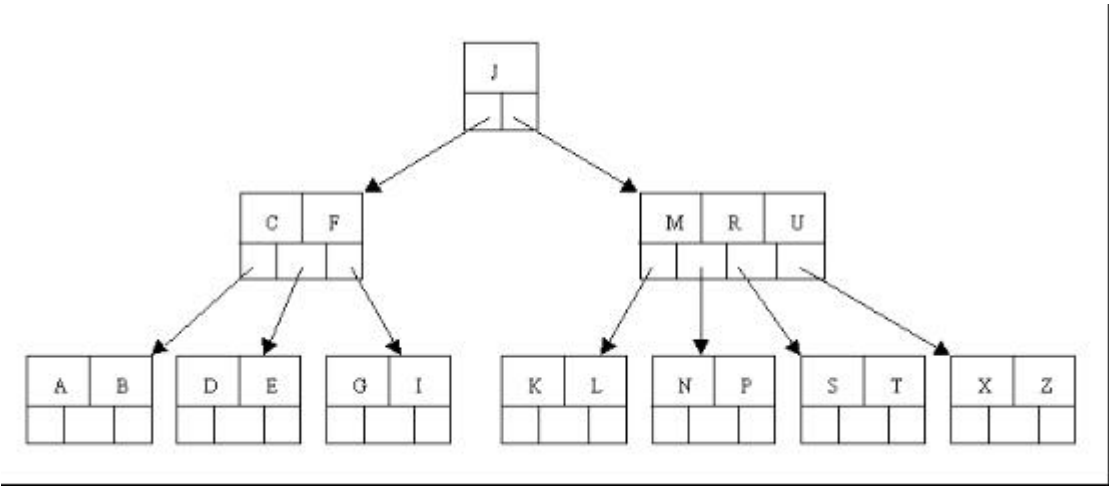


插入与删除的转换-----

5阶B-树

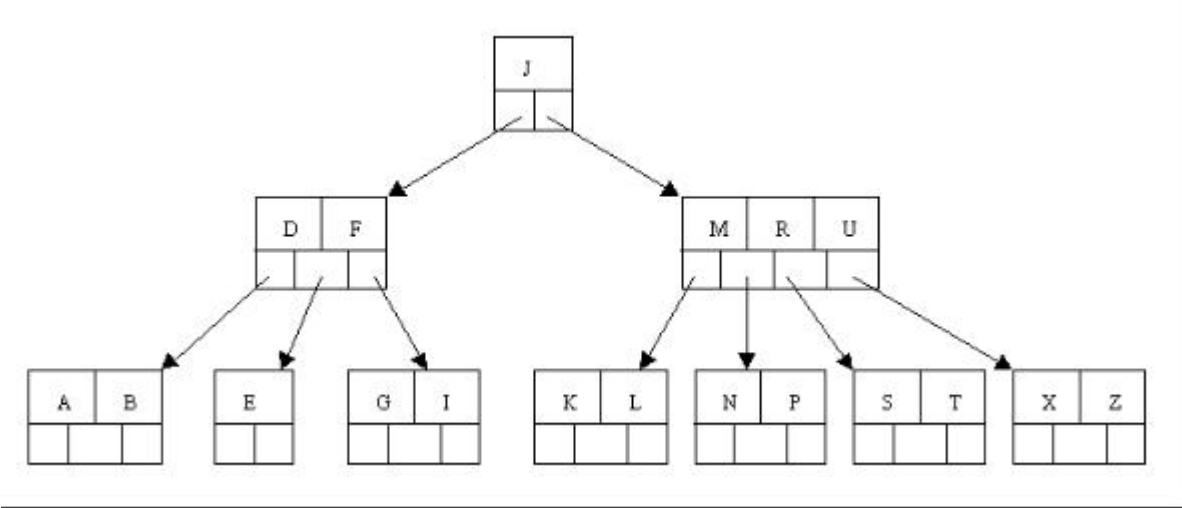


5. 5序B树，那咱们试着删除C

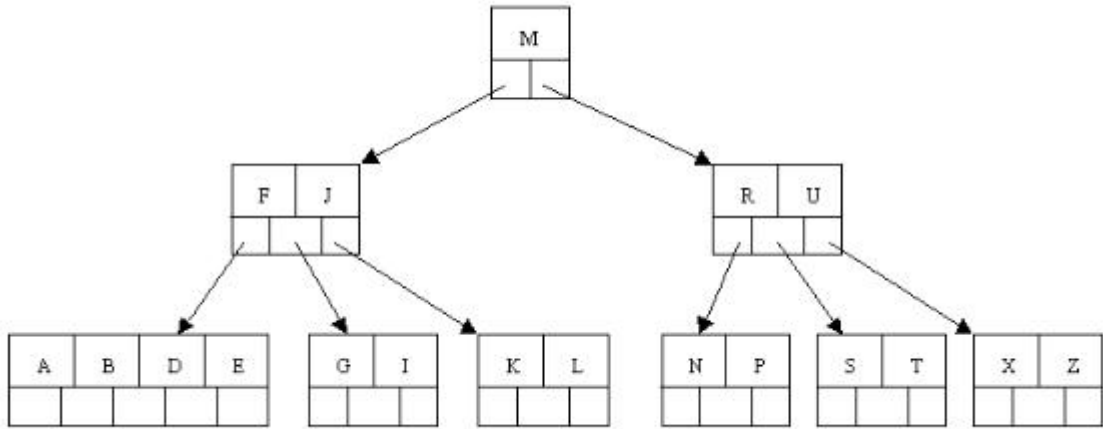


于是将删除元素C的右子结点中的D元素上移到C的位置，但是出现上移元素后，只有一个元素的结点的情况。

又因为含有E的结点，其相邻兄弟结点才刚脱贫（最少元素个数为2），不可能向父节点借元素，所以只能进行合并操作，于是这里将含有A,B的左兄弟结点和含有E的结点进行合并成一个结点。



这样又出现只含有一个元素F结点的情况，这时，其相邻的兄弟结点是丰满的（元素个数为3>最小元素个数2），这样就可以想父结点借元素了，把父结点中的J下移到该结点中，相应的如果结点中J后有元素则前移，然后相邻兄弟结点中的第一个元素（或者最后一个元素）上移到父节点中，后面的元素（或者前面的元素）前移（或者后移）；注意含有K, L的结点以前依附在M的左边，现在变为依附在J的右边。这样每个结点都满足B树结构性质。



从以上操作可看出：除根结点之外的结点（包括叶子结点）的关键字的个数n满足： $(\text{ceil}(m / 2)-1) \leq n \leq m-1$ ，即 $2 \leq n \leq 4$ 。这也佐证了咱们之前的观点。删除操作完。

5.为什么设置树的限制策略？它具体的用途？

第一问，是为了提高空间的利用率，在B-树中要求最低是 $m/2$ (非根)，而在B*中是要求 $2/3m$,进一步提高了利用率，使树进一步“紧凑”。

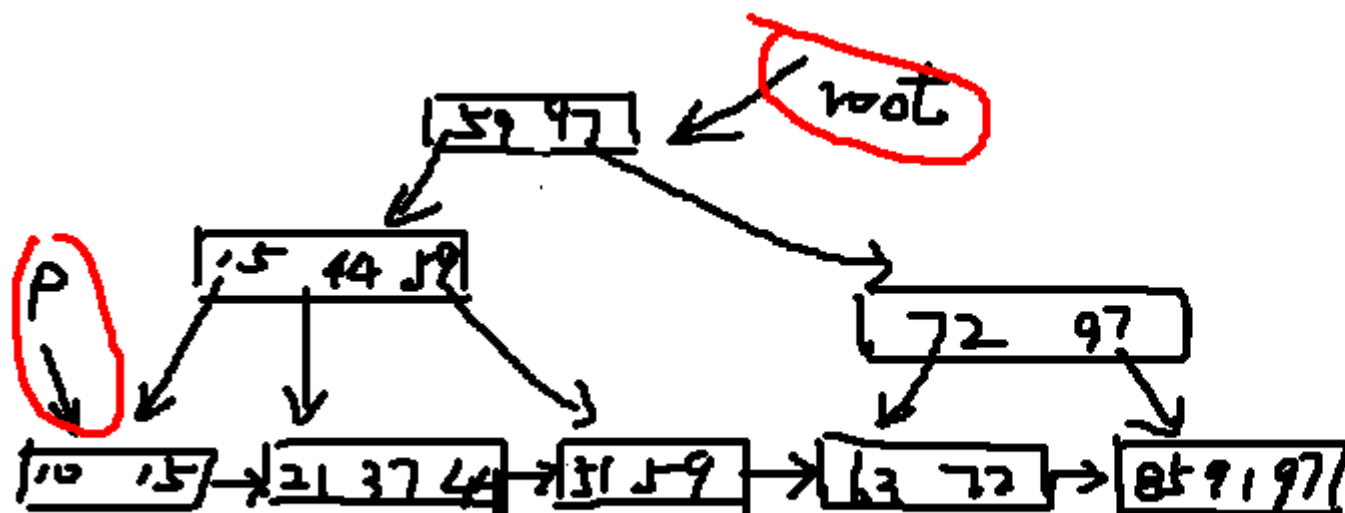
第二问，具体用途就是在文件系统，在磁盘等直接存取设备上组织动态的查找表。

6.扩展

a.B+树：B-树的变种，也是一种多路搜索树。

特点（与B-树的差异）：

- 1) 有n棵子树的节点中包含有n个关键字。
- 2) 所有的叶子节点中包含了全部关键字信息，及指向含这些关键字记录的指针，且叶子节点本身依照关键字大小自小而大顺序链接。
- 3) 所有的非终端节点可以看成是索引部分，节点中仅含有其子树（根节点）中的最大（或最小）的关键字。



有几个明显的特点：

1. 非叶子结点的子树指针与关键字个数相同
2. 所有关键字都在叶子结点出现
3. 所有叶子结点有一个指针p（可以从p开始遍历，也可以由root）
4. 叶子结点之间有链接指针链接
5. 非终端节点含有子树中的最大或最小关键字。

b.B-与B+树哪个优？

引用http://blog.csdn.net/v_july_v/article/details/6530142

B树：有序数组+平衡多叉树；

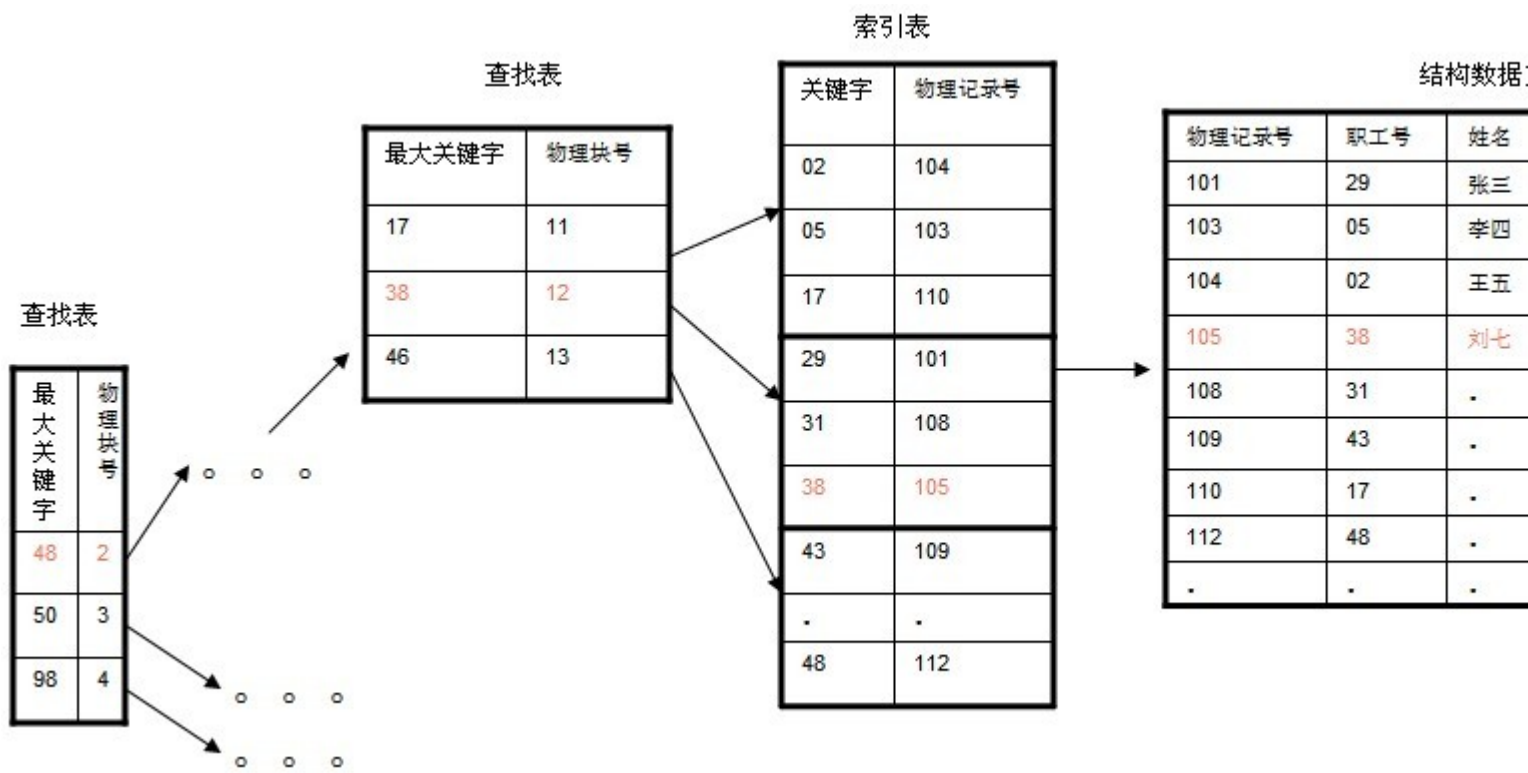
B+树：有序数组链表+平衡多叉树；

B*树：一棵丰满的B+树。

在大规模数据存储的文件系统中，B~tree系列数据结构，起着很重要的作用，对于存储不同的数据，节点相关的信息也是有所不同，这里根据自己的理解，画的一个查找以职工号为关键字，职工号为38的记录的简单示

意图。(这里假设每个物理块容纳3个索引，磁盘的I/O操作的基本单位是块 (block),磁盘访问很费时，采用B+树有效的减少了访问磁盘的次数。)

对于像MySQL，DB2，Oracle等数据库中（数据库中数据实际存于磁盘）的索引结构得有较深入的了解才行，建议去找一些B 树相关的开源代码研究。



走进搜索引擎的作者梁斌老师针对B树、B+树给出了他的意见（为了真实性，特引用其原话，未作任何改动）：“B+树还有一个最大的好处，方便扫库，B树必须用中序遍历的方法按序扫库，而B+树直接从叶子结点挨个扫一遍就完了，B+树支持range-query非常方便，而B树不支持。这是数据库选用B+树的最主要原因。

比如要查 5-10之间的，B+树一把到5这个标记，再一把到10，然后串起来就行了，B树就非常麻烦。B树的好处，就是成功查询特别有利，因为树的高度总体要比B+树矮。不成功的情况下，B树也比B+树稍稍占一点点便宜。

B树比如你的例子中查，17的话，一把就得到结果了，

有很多基于频率的搜索是选用B树，越频繁query的结点越往根上走，前提是需要对query做统计，而且要对key做一些变化。

另外B树也好B+树也好，根或者上面几层因为被反复query，所以这几块基本都在内存中，不会出现读磁盘IO，一般已启动的时候，就会主动换入内存。（利用了内存的缓存机制，预存？）”非常感谢。

Bucket Li：“mysql 底层存储是用B+树实现的，知道为什么么?(上面红色粗体)。内存中B+树是没有优势的，但是一到磁盘，B+树的威力就出来了（主要是range-query功能）”。

我从上面也总结：

1).B+树好处在于要连续访问节点，如从1--10，是连续的，这取决于B+的存储结构，因为B+树的叶子结点都用链接指针连起来了，故而连续访问非常快

而这是B树的弱点，它没有B+这样的存储特点，故而更适合单个查询，不论成不成功，都很快。故而是**B+树用于数据库主要原因**，数据库数据大多数在磁盘中（B与B+差不多），也经常涉及连续访问（B+）！

2).基于频率的搜索，属于单个查询，B树合适

c.为什么说B+-tree比B 树更适合实际应用中操作系统的文件索引和数据库索引？

1) B+-tree的磁盘读写代价更低

B+-tree的内部结点并没有指向关键字具体信息的指针。因此其内部结点相对B 树更小。如果把所有同一内部结点的关键字存放在同一盘块中，那么盘块所能容纳的关键字数量也越多。一次性读入内存中的需要查找的关键字也就越多。相对来说IO读写次数也就降低了。

举个例子，假设磁盘中的一个盘块容纳16bytes，而一个关键字2bytes，一个关键字具体信息指针2bytes。一棵9阶B-tree(一个结点最多8个关键字)的内部结点需要2个盘快。而B+ 树内部结点只需要1个盘快。当需要把内部结点读入内存中的时候，B 树就比B+ 树多一次盘块查找时间(在磁盘中就是盘片旋转的时间)。

2) B+-tree的查询效率更加稳定

由于非终结点并不是最终指向文件内容的结点，而只是叶子结点中关键字的索引。所以任何关键字的查找必须走一条从根结点到叶子结点的路。所有关键字查询的路径长度相同，导致每一个数据的查询效率相当。

4.19 二叉排序树

发表时间: 2012-08-10 关键字: 二叉排序树

1.基本概念

二叉排序树，树的定义就不赘述了，主要就是想说明一下在设计类的过程中需要注意的问题。

a.问题引入？

设计插入，删除等操作的过程中，我们的二叉排序树根节点的指针有可能改变，如删除根结点的指针操作，那么root的指针指向已经不是原来的位置，而是新的位置，怎么样才能返回最新的位置呢？

b.问题解决

在设计类中的函数就可以轻易的解决这个问题，这个在函数设计之初就必须详细的考虑这个问题，这里有两种办法

首先，通过返回值，返回最新的root结点；

```
Node<T>* deleteBST(T x);  
Node<T>* insertBST(T x);
```

这样在使用的过程中就需要这样使用：

```
BiSortTree<int> bst;  
Node<int>* root = bst.getRoot();  
...  
bst.deleteBST(20); //这里假定20是根结点
```

```
bst.printBST(root);    //这里打印出来的肯定不是现在最新的二叉树，因为原来的root已经改变了
正确的使用方法是：
BiSortTree<int> bst;
Node<int>* root = bst.getRoot();
...
root = bst.deleteBST(20); //这里假定20是根结点
bst.printBST(root);
```

其次，可以通过引用，设置函数的参数；

```
void deleteBST(Node<T>* &root, T x);
void insertBST(Node<T>* &root, T x);
```

这里利用了指针引用，这样会在删除，插入操作自动更新root

它的使用很简单，直接使用就是了

c.设计的缺陷

根据面向对象的原则，在上述设计中void deleteBST(Node<T>* &root, T x);它的设计是不合理的，因为root已经暴露在函数外面，为了实现其隐蔽的原则，就必须将root封装起来，不让使用者接触，那么怎样设计了？

```
class BiSortTree{
public:
    ...

    void insertBST(T x){ insertBST(root, x);}
```

```
void deleteBST(T x){ deleteBST(root, x);}

private:

...

void insertBST(Node<T>* &root, T x);

void deleteBST(Node<T>* &root, T x);

Node<T>* &root; //完成隐藏root

};
```

这样就实现了，隐蔽的原则，实现了面向对象的方法，使得使用者不用关心具体的实现，只是直接调用就ok了

2.代码实现

代码中就没有完全按照上述原则实现，所以需要自己改装

a.bisorttree.h

```
//二叉排序树
#ifndef BISORTTREE_H
#define BISORTTREE_H

//数据域，里面存放结点数据
template<class T>
struct Data{
    T data;                //结点数据
    int count; //次数
    //int bf; //平衡因子（对于AVL平衡二叉树是需要的）
};

template<class T>
struct Node{
    Data<T> data;
    Node<T> *rchild, *lchild;
```

```
};

/**
 *注意插入，删除都可能改变根结点，故而可能改变root指向的指针，就要注意设计函数的参数或者返回值
 *例如：void delete(T x);如果删除x是根结点，则删除后访问，就不能在找到root，访问内存出错
 *故而要想删除之后访问不出错，应该设计成
 *1.Node<T>* delete(T x);(利用返回值)返回最新根结点,使用方法Node<T>* root = delete(12); pr
 *2.void delete(Node<T>* &root, T x);(利用函数参数，这样不是很好，破坏封装性，可以设计成priv
 * 它的使用简单，直接就是delete(root, 12); printBST(root);
 */

template<class T>
struct BiSortTree{
public:
    BiSortTree();
    BiSortTree(T a[], int n);
    ~BiSortTree();
    Node<T>* getRoot();

    //-----删除-----
    void deleteBST(Node<T>* &root, T x); //设计方法1
    //返回根结点
    Node<T>* deleteBST(T x); //设计方法2：当删除根结点之后，root改变

    //-----插入-----
    //void insertBST(Node<T>* root, T x); //递归插入,不使用引用是不行的
    void insertBST(Node<T>* &root, T x); //root写成引用，这样root改变就会自动
    //返回根结点（在首次插入之后，其实root就不会改变）
    Node<T>* insertBST(T x); //最好不写成void insertBST(T x)，这样每

    //-----搜索-----
    //返回搜索结点
    Node<T>* searchBST(T x);
    //返回搜索结点
    Node<T>* searchBST(Node<T>* root, T x); //递归搜索
    void printBST(Node<T>* root);
private:
    Node<T>* root;
```

```
        Node<T>* pre;//用于递归插入时，记录前序结点
        void releaseBST(Node<T>* &root);
        void create(Node<T>* &root);
        void deleteNode(Node<T>* &root, Node<T>* r, Node<T>* pre);//删除结点p
};
#endif
```

b.bisorttree.cpp

```
#include <iostream>
#include "bisorttree.h"
using namespace std;

template<class T>
BiSortTree<T>::BiSortTree(){
    pre = NULL;
    root = NULL;
    create(root);
}

template<class T>
BiSortTree<T>::BiSortTree(T* a, int n){
    if(n <= 0) return;
    pre = NULL;
    root = NULL;
    for(int i=0; i<n; i++){
        insertBST(root, a[i]);
        //insertBST(a[i]);
    }
}

template<class T>
BiSortTree<T>::~~BiSortTree(){
    releaseBST(root);
}
```

```
}

template<class T>
Node<T>* BiSortTree<T>::getRoot(){
    return root;
}

//返回根结点
template<class T>
Node<T>* BiSortTree<T>::insertBST(T x){
    Node<T>* r = root;
    if(r == NULL){
        Node<T>* t = new Node<T>;
        (t->data).data = x;
        (t->data).count = 1;
        t->lchild = t->rchild = NULL;
        root = t; //必须设置新的root结点，因为t和root在内存指向位置不同
    }else{
        Node<T>* p = r;
        //查找待插入结点位置
        while(r){
            if((r->data).data == x){
                (r->data).count++;
                return root;
            }
            p = r;
            r = ((r->data).data > x)?r->lchild:r->rchild;
        }
        Node<T>* t = new Node<T>;
        (t->data).data = x;
        (t->data).count = 1;
        t->lchild = t->rchild = NULL;
        if((p->data).data > x){ //插入左
            p->lchild = t;
        }else{
            p->rchild = t;
        }
    }
}
```



```
        }
    }
    return root;
}

//用于检测不用root引用是否会修改root,结果不使用root是不能改变root,必须是指针的引用
template<class T>
void BiSortTree<T>::insertBST(Node<T>* &root, T x){
    if(root){
        pre = root;
        if((root->data).data > x) insertBST(root->lchild, x);
        else if((root->data).data < x) insertBST(root->rchild, x);
        else (root->data).count++;
    }else{
        Node<T>* r = new Node<T>;
        (r->data).data = x;
        (r->data).count = 1;
        r->lchild = r->rchild = NULL;
        if(pre){
            if((pre->data).data > x) pre->lchild = r;
            else pre->rchild = r;
        }else{
            root = r;
        }
    }
}

template<class T>
void BiSortTree<T>::deleteBST(Node<T>* &root, T x){
    if(!root) return;
    Node<T>* pre = NULL, *r = root;
    //查找待插入结点位置(p是r的前序结点)
    while(r){
        if((r->data).data == x){
            if((r->data).count > 1){
                (r->data).count--; return;
            }else{

```

```
                break;
            }
        }

        pre = r;
        r = ((r->data).data > x)?r->lchild:r->rchild;
    }
    //没找到
    if(!r) return;
    //如果pre == NULL说明是root结点
    deleteNode(root, r, pre);
}

template<class T>
Node<T>* BiSortTree<T>::deleteBST(T x){
    if(!root) return root;
    Node<T>* pre = NULL, *r = root;
    //查找待删除结点位置(pre是r的前序结点)
    while(r){
        if((r->data).data == x){
            if((r->data).count > 1){
                (r->data).count--; return root;
            }else{
                break;
            }
        }

        pre = r;
        r = ((r->data).data > x)?r->lchild:r->rchild;
    }
    //没找到
    if(!r) return root;
    //如果pre == NULL说明是root结点
    deleteNode(root, r, pre);
    return root;
}
```

```
template<class T>
Node<T>* BiSortTree<T>::searchBST(T x){
    Node<T>* r = root;
    while(r){
        if(r->data.data == x) break;
        r = ((r->data).data > x)?r->lchild:r->rchild;
    }
    //如果没找到，返回空
    return r;
}

template<class T>
Node<T>* BiSortTree<T>::searchBST(Node<T>* root, T x){
    if(!root) return NULL;
    else{
        if(root->data.data == x) return root;
        else if(root->data.data > x) return searchBST(root->lchild, x);
        else return searchBST(root->rchild, x);
    }
}

template<class T>
void BiSortTree<T>::printBST(Node<T>* root){
    if(root){
        cout<<root->data.data<<" ";
        printBST(root->lchild);
        printBST(root->rchild);
    }
}

template<class T>
void BiSortTree<T>::releaseBST(Node<T>* &root){
    if(root){
        releaseBST(root->lchild);
        releaseBST(root->rchild);
    }
}
```

```
        delete root;
    }
}

template<class T>
void BiSortTree<T>::create(Node<T>* &root){
    T ch;
    cout<<"请输入创建一棵二叉排序树的结点数据(输入#结束)："<<endl;
    while(cin>>ch){
        if(ch == '#') break;
        //insertBST(root, ch);
        insertBST(ch);
    }
}

//pre为空，说明删除的是根结点
template<class T>
void BiSortTree<T>::deleteNode(Node<T>* &root, Node<T>* r, Node<T>* pre){
    Node<T>* p;
    if(!r->rchild && !r->lchild){                //如果是叶子结点
        if(pre){
            if(pre->lchild == r){
                pre->lchild = NULL;
            }else{
                pre->rchild = NULL;
            }
        }else{
            root = NULL;
        }
        delete r;
    }else if(r->rchild && r->lchild){                //如果左右子树都有(还有一种处理办法就是序
        p = r;
        //寻找右子树的最左结点
        r = r->rchild;
        while(r->lchild){
            r = r->lchild;
        }
    }
}
```

```
    }
    //将删除结点的左结点接到找到的最左结点之后
    r->lchild = p->lchild;
    //删除结点(如果pre是空,说明删除结点是根结点,不用改变前序结点指针)
    if(pre){
        if(pre->lchild == p) pre->lchild = p->rchild;
        else pre->rchild = p->rchild;
    }else{
        root = p->rchild;
    }
    delete p;
}else if(r->lchild){                //如果只有左子树
    p = r;
    if(pre){
        if(pre->lchild == p) pre->lchild = r->lchild;
        else pre->rchild = r->lchild;
    }else{
        root = r->lchild;
    }
    delete p;
}else{                            //如果只有右子树
    p = r;
    if(pre){
        if(pre->lchild == p) pre->lchild = r->rchild;
        else pre->rchild = r->rchild;
    }else{
        root = r->rchild;
    }
    delete p;
}
}
```

c.main.cpp

```
#include <iostream>
#include "bisorttree.cpp"
using namespace std;

int main(){
    cout<<"-----二叉排序树测试-----"<<endl;
    BiSortTree<int> bst;
    Node<int>* root = bst.getRoot();

    bst.printBST(root);
    cout<<"\n插入23"<<endl;
    bst.insertBST(23);
    //root = bst.insertBST(23);
    //bst.insertBST(24);
    root = bst.insertBST(24); //如果初始的root为空，则必须这样使用，因为root已经改变，如果上面那
    bst.printBST(root);

    /*
    int a[] = {60, 40, 65, 30, 45, 70, 68, 20, 43};
    BiSortTree<int> bst(a, 9);
    Node<int>* root = bst.getRoot();
    if(root == NULL){
        cout<<"---null---"<<endl;
    }
    cout<<"-----遍历结果-----"<<endl;
    bst.printBST(root);
    cout<<"\n插入23"<<endl;
    bst.insertBST(23);
    bst.printBST(root);

    Node<int> *p;
    cout<<"\n搜索23"<<endl;
    p = bst.searchBST(root, 23);
    if(p) cout<<"搜索结果："<<p->data.data<<","count:"<<p->data.count<<endl;
    else cout<<"无"<<endl;

    cout<<"删除叶子23"<<endl;
```

```
bst.deleteBST(root, 23);
bst.printBST(root);

//测试构造函数和几种删除，一个root
cout<<"\n删除叶子结点20"<<endl;
bst.deleteBST(root, 20);
bst.printBST(root);

cout<<"\n删除根结点60"<<endl;
//bst.deleteBST(root, 60); //1.方案1
root = bst.deleteBST(60); //2.方案2，    不获取返回值bst.deleteBST(60);是不正确的使用方法
bst.printBST(root);
*/
return 0;
}
```

4.20 平衡二叉树

发表时间: 2012-08-10 关键字: 平衡二叉树, avl, 递归, 非递归

1.问题描述

什么是平衡二叉树？在此就不在赘述，下面主要就几个关键问题进行分析

2.关键问题

a.AVL树的非递归与递归插入

平衡二叉树的非递归的关键:

1.在寻找插入位置和旋转的时候设置其路径上的平衡因子，这个要特别注意

当然非递归比递归复杂的多，但是对于理解其执行过程很有帮助！

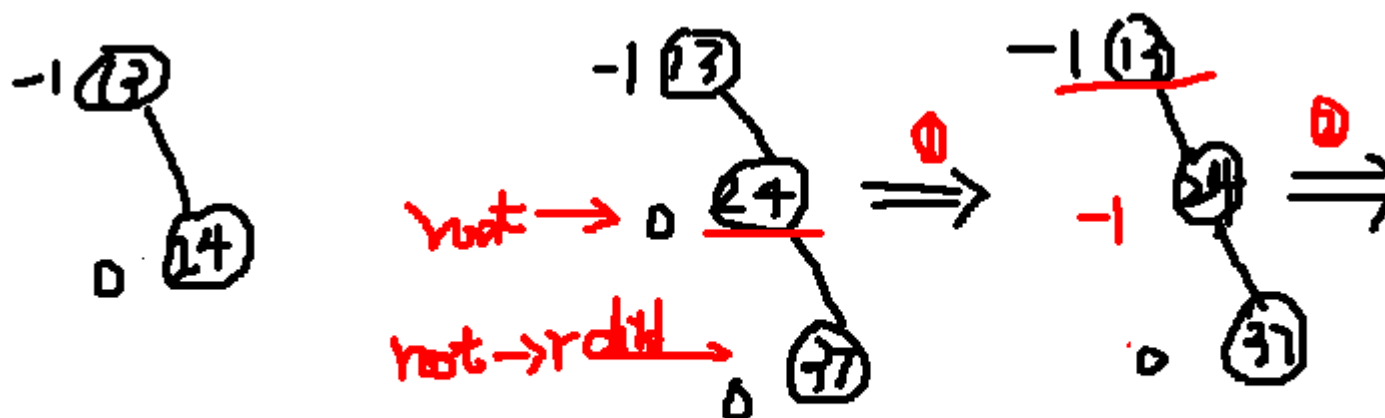
2.是旋转之后可能并没有产生效果（实际上是旋转成功，但是输出的树没有变），因为在修改的时候，并没有注意结点的改变，没有使用指针引用，修改原来树的指针，指向旋转之后的新的根结点。

这个在setBF(Node<T>* r);中分了两情况：**a.旋转结点是根结点; b.旋转结点非根结点，必须手动设置pre前序指针指向。**

3.是左旋和右旋，这里是将LL,LR都归为L。将RR,RL都归为R

其实处理方式还是很简单的，就是改变指针的指向，稍微复杂些的就是设置平衡因子

图示见下：



1. 当插入37后, 37出栈, 开始执行右边代码
执行之前状态是图2, root指向24, 由于右
边插入, 故而平衡因子变为-1; 然后24出
栈, 继续右边代码, -1-->-2并执行旋转。

2. 当当前结点的root子结点root->rchild的平衡指
针为1或者-1的时候 (见数据结构中二叉树平衡旋
转图例中可看出此特点), 下次出栈的root结点就
应该是旋转结点, 因为向右插入其平衡因子都-1,
-1-->-2; 若是左插入, 都加1, 1-->2, 故而root
是旋转结点。

```
if(*h){
    switch(root->data.bf){
        case -1:
            *h = 0;
            rchange(root);
            break;
        case 1:
            *h = 0;
            root->data.bf = 0;
            break;
        case 0:
            root->data.bf = -1;
            break;
    }
}
```

```
if(*h){
    switch(root->data.bf){
        case -1:
            root->data.bf = 0;
            *h = 0;
            break;
        case 1:
            lchange(root);
            *h = 0;
            break;
        case 0:
            root->data.bf = 1;
            break;
    }
}
```

在1或者-

a. 对于LL以B为支点, B上升一个位置, 右子树高度加1, 则平衡因子1-->0, 同理A: 2-->0;

RR也是一样, 平衡因子都变为0.

b. 就是决定平衡因子主要决定于结点C (C的平衡因子只可能为-1, 1, 0), 然后分情况讨论, 画出实图既可以
决定平衡因子。

平衡二叉树非递归插入算法伪代码：

```
if(root是空) 插入根结点；

else{

    看插入结点是否存在；

    if(存在x) 将count++;

    else{

        看插入结点的位置，是否会增加树的高度（通过判断插入位置是否有兄弟）；

        while(结点不空){

            查找插入结点位置，并记录其前序结点，如果插入结点会增加树的高度，则在插入过程中

            增加结点的平衡因子，并记录到需要旋转的结点。

        }

        将新结点插入到指定位置；

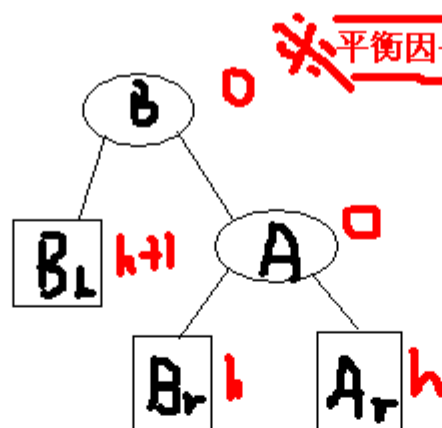
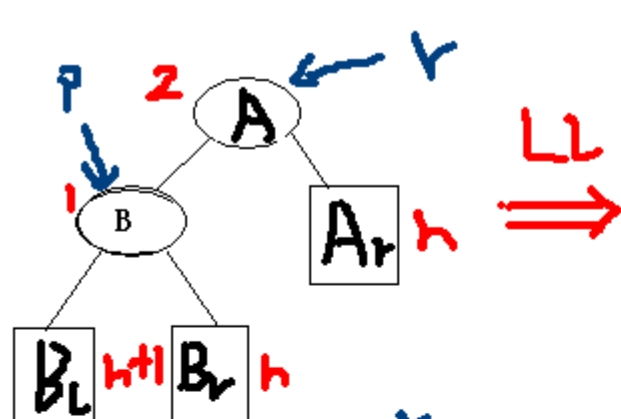
        通过记录的需要旋转的结点，进行LL,RR,LR,RL旋转

    }

}
```

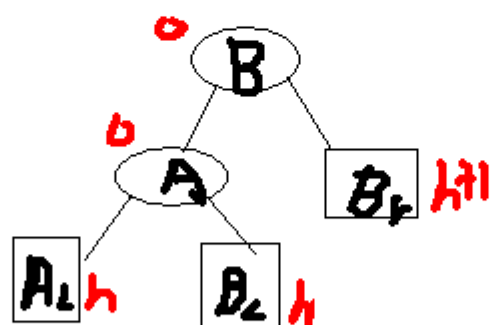
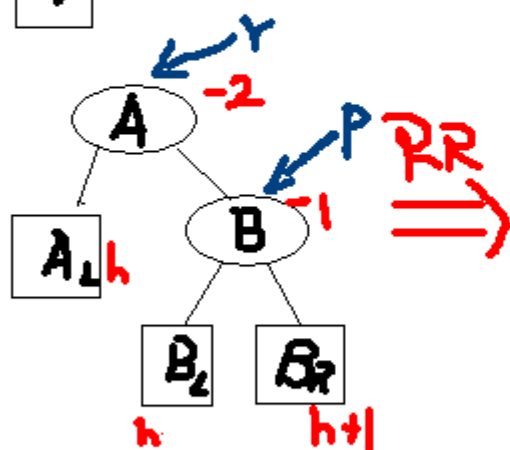
b.平衡因子分析

具体分析如下图：

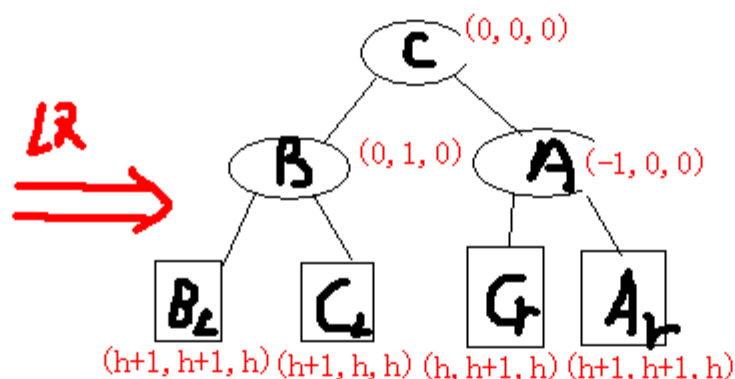
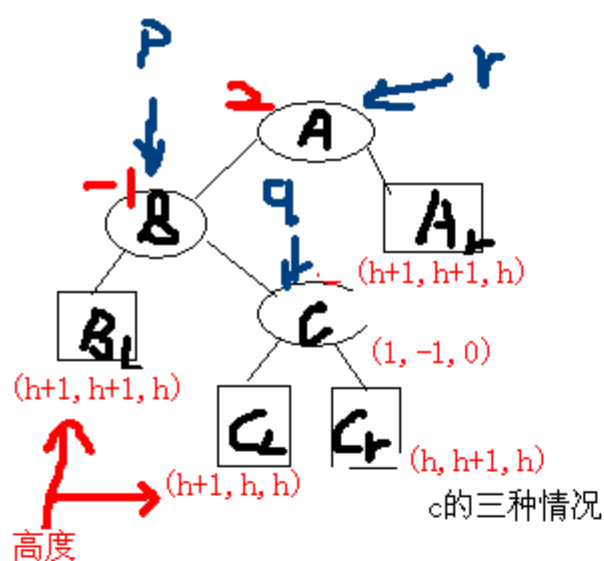


平衡因子=左树高度-右树高度

LL型:
从这个画出了
从中可以看出
和平衡因子的
平衡因子:
A: 2-0 (左右子
B: 1-0

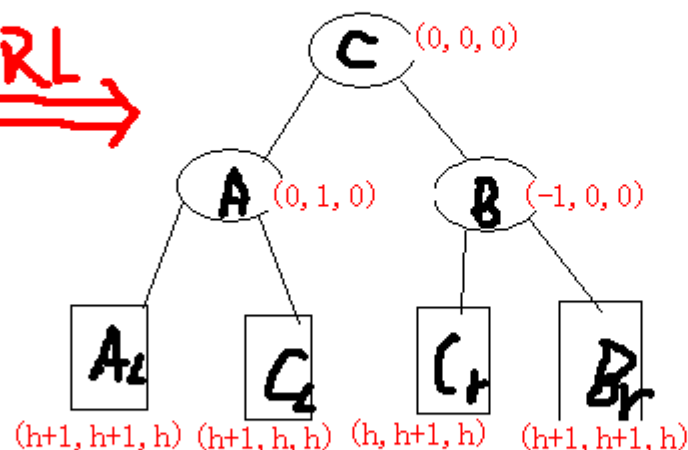
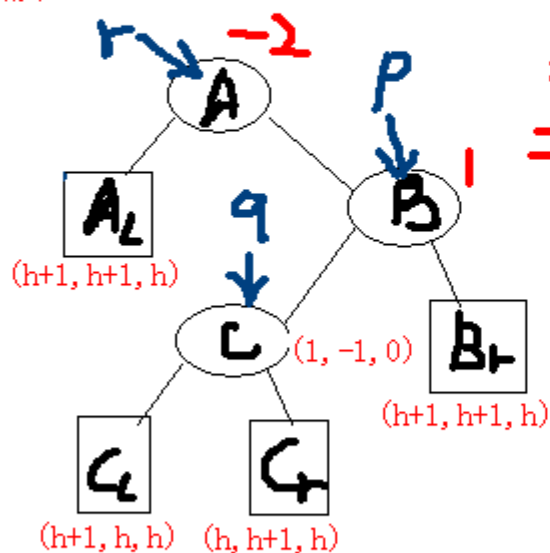


RR型:
平衡因子:
A: h-h=0,
-2-->0
B: (h+2) - (h+1)
-1-->0



LR型:
根据C的三
得整个树
图中已经
(情况1,

注意: 树高度是最高的子树高度
如左子树高度h+2, 右子树高度h
则树高度是h+2, 而不是h+1 !!!



RL型:
和上面类似的
为三种情况

3.代码

avltree.h

```
//平衡二叉树 ( AVL--balanced binary tree )

#ifndef AVLTREE_H
#define AVLTREE_H

template<class T>
struct Data{
    T data;                //结点数据
    int count; //次数
    int bf; //平衡因子 ( 对于AVL平衡二叉树是需要的 )
};

template<class T>
struct Node{
    Data<T> data;
    Node<T> *rchild, *lchild;
};

template<class T>
class AVLTree{
public:
    AVLTree();
    AVLTree(int a[], int n);
    ~AVLTree();
    Node<T>* deleteAVL(T x); //删除结点, 返回最新根结点(删除还没完全实现, 没有恢复平衡特性)
    Node<T>* insertAVL(T x); //插入结点, 返回最新根结点
    void insertAVL(Node<T>* &root, T x, int *h); //递归插入结点, h是树的高度
    Node<T>* searchAVL(T x);
    Node<T>* findMin(); //查找最小值
    Node<T>* findMax(); //查找最大值
    void printAVL();
    bool hasSibling(T x); //x是否有兄弟结点,x可能不存在, 若不存在则在待插入位置若有兄弟
```

```
        int height();
    private:
        Node<T>* root;
        void print(Node<T>* root);
        //AVL的调整策略
        void lChange(Node<T>* &r); //左旋转(LL--LR),r为旋转的根结点 (bf为2, -2)
        void rChange(Node<T>* &r); //右旋转(RR--RL),r为旋转的根结点 (bf为2, -2)
        void create(Node<T>* &root);
        void release(Node<T>* &root);
        void setBF(Node<T>* r); //设置平衡因子,r为旋转的根结点 (bf为2, -2)
        int height(Node<T>* root); //树高度
        //删除相关
        void deleteNode(Node<T>* &root, Node<T>* r, Node<T>* pre); //删除结点r,pre是前缀结点
        void changeBF(Node<T>* &root); //重新设置整个树的平衡因子
        void resetAVL(Node<T>* &root); //重新设置树为平衡树
};

#endif
```

b.avltree.cpp

```
#include <iostream>
#include "avltree.h"
using namespace std;

template<class T>
AVLTree<T>::AVLTree(){
    root = NULL;
    create(root);
}

template<class T>
AVLTree<T>::AVLTree(int a[], int n){
    root = NULL;
```

```
        if(n <= 0) return;
        for(int i=0; i<n; i++){
            insertAVL(a[i]);
        }
    }

template<class T>
AVLTree<T>::~~AVLTree(){
    release(root);
}

//删除结点，返回最新根结点(删除过程中也可能造成树不平衡)
template<class T>
Node<T>* AVLTree<T>::deleteAVL(T x){
    if(!root) return root;
    Node<T>* pre = NULL, *r = root;
    //查找待删除结点位置(pre是r的前序结点)
    while(r){
        if((r->data).data == x){
            if((r->data).count > 1){
                (r->data).count--; return root;
            }else{
                break;
            }
        }

        pre = r;
        r = ((r->data).data > x)?r->lchild:r->rchild;
    }

    //没找到
    if(!r) return root;
    //如果pre == NULL说明是root结点
    deleteNode(root, r, pre);
    //重新设置平衡因子
    changeBF(root);
    //重新设置树为平衡树
    //-----还没做? -----
```

```
//**判断
```

```
//查找，如果找到（已经
```

//看待插入结点是

```
//待插入结点所在位置
```

```
//记录最近需要旋转的根结点
```



```
        Node<T>* p = new Node<T>;
        p->data.data = x;
        p->data.count = 1;
        p->data.bf = 0;
        p->lchild = p->rchild = NULL;
        if((pre->data).data > x){//插入左
            if(hasSib) pre->data.bf++;
            pre->lchild = p;
        }else{
            if(hasSib) pre->data.bf--;
            pre->rchild = p;
        }
        //进行平衡旋转
        if(n) setBF(n);
    }
}
return root;
}
```

/**插入递归算法

*为什么要设置*h，它决定在出栈进入上次状态之后，是否需要设置结点的平衡因子，旋转等操作，还是直

*那什么时候需要在弹栈后，还要继续设置上次状态？例如：见图例

*

*/

```
template<class T>
```

```
void AVLTree<T>::insertAVL(Node<T>* &root, T x, int *h){
```

```
    if(!root){
```

```
        root = new Node<T>;
```

```
        root->data.data = x;
```

```
        root->data.count = 1;
```

```
        root->data.bf = 0;
```

```
        *h = 1;
```

```
        root->lchild = root->rchild = NULL;
```

```
    }else{
```

```
        if(x < root->data.data){
```

```
            insertAVL(root->lchild, x, h);
```

```
            if(*h){//在左子树中插入了新结点
```

```
        switch(root->data.bf){
            case -1:
                root->data.bf = 0;
                *h = 0;
                break;
            case 1:
                lChange(root);
                *h = 0;
                break;
            case 0:
                root->data.bf = 1;
                break;
        }
    }
}
}else if(x > root->data.data){
    insertAVL(root->rchild, x, h);
    if(*h){//在右子树中插入了新结点(往右插入了结点,则平衡因子都要-1)
        //下面是设置上一个结点的状态,旋转等,当*h == 0,则返回上一个结点时候
        switch(root->data.bf){
            case -1:
                *h = 0;
                rChange(root);
                break;
            case 1:
                *h = 0;
                root->data.bf = 0;
                break;
            case 0:
                root->data.bf = -1;
                break;
        }
    }
}
}
root->data.count++;
}
```

```
}

template<class T>
Node<T>* AVLTree<T>::searchAVL(T x){
    Node<T>* r = root;
    while(r){
        if(r->data.data == x) return r;
        r = (r->data.data > x)?r->lchild:r->rchild;
    }
    if(!r) return NULL;
}

template<class T>
void AVLTree<T>::printAVL(){
    print(root);
}

template<class T>
Node<T>* AVLTree<T>::findMin(){
    Node<T>* r = root;
    while(r && r->lchild){
        r = r->lchild;
    }
    return r;
}

template<class T>
Node<T>* AVLTree<T>::findMax(){
    Node<T>* r = root;
    while(r && r->rchild){
        r = r->rchild;
    }
    return r;
}

template<class T>
void AVLTree<T>::print(Node<T>* root){
    if(root){
```

```
        cout<<root->data.data<<"("<<root->data.count<<","<<root->data.bf<<")"<<" ";
        print(root->lchild);
        print(root->rchild);
    }
}
```

//左旋转(LL), r为旋转的根结点

```
template<class T>
```

```
void AVLTree<T>::lChange(Node<T>* &r){
```

```
    Node<T>* p, *q;
```

```
    p = r->lchild;
```

```
    //-----LL型-----
```

```
    if(p->data.bf == 1){
```

```
        cout<<"---LL---"<<endl;
```

```
        r->lchild = p->rchild;
```

```
        p->rchild = r;
```

```
        r->data.bf = 0;
```

```
        p->data.bf = 0;
```

```
        r = p;
```

```
    }
```

```
    //-----LR型----- ( p->data.bf == -1 )
```

```
    else{
```

```
        cout<<"---LR---"<<endl;
```

```
        q = p->rchild;
```

```
        p->rchild = q->lchild;
```

```
        r->lchild = q->rchild;
```

```
        q->lchild = p;
```

```
        q->rchild = r;
```

```
        if(q->data.bf == 1){
```

```
            p->data.bf = 0;
```

```
            r->data.bf = -1;
```

```
        }else if(q->data.bf == -1){
```

```
            p->data.bf = 1;
```

```
            r->data.bf = 0;
```

```
        }else{//q->data.bf == 0
```

```
            p->data.bf = 0;
```

```
            r->data.bf = 0;
```

```
        }
        q->data.bf = 0;
        r = q;
    }
}

//右旋转(RR), r为旋转的根结点
template<class T>
void AVLTree<T>::rChange(Node<T>* &r){
    Node<T>* p, *q;
    p = r->rchild;
    //-----RR型-----
    if(p->data.bf == -1){
        cout<<"---RR---"<<endl;
        r->rchild = p->lchild;
        p->lchild = r;
        r->data.bf = 0;
        p->data.bf = 0;
        r = p;
    }
    //-----RL型(p->data.bf == 1)-----
    else{
        cout<<"---RL---"<<endl;
        q = p->lchild;
        r->rchild = q->lchild;
        p->lchild = q->rchild;
        q->lchild = r;
        q->rchild = p;
        if(q->data.bf == -1){
            r->data.bf = 1;
            p->data.bf = 0;
        }else if(q->data.bf == 1){
            r->data.bf = 0;
            p->data.bf = -1;
        }else if(q->data.bf == 0){
            r->data.bf = 0;
            p->data.bf = 0;
        }
    }
}
```

```
        }
        q->data.bf = 0;
        r = q;
    }
}

template<class T>
void AVLTree<T>::create(Node<T>* &root){
    T ch;
    cout<<"请输入二叉平衡树的结点 ( '#'结束 )" <<endl;
    while(cin>>ch){
        if(ch == '#') break;
        insertAVL(ch);
        //int a = 1;
        //insertAVL(root, ch, &a);
    }
}
```

```
template<class T>
void AVLTree<T>::release(Node<T>* &root){
    if(root){
        release(root->lchild);
        release(root->rchild);
        delete root;
    }
}
```

//r为需要旋转的根结点，旋转过程中旋转的树的高度必然减少，故而需要调节其路径中树的平衡因子

```
template<class T>
void AVLTree<T>::setBF(Node<T>* r){
    Node<T>* p = root, *pre = NULL;
    //查找旋转结点以及前缀,查找过程中设置旋转结点之上的平衡因子
    while(p){
        if(p == r) break;
        pre = p;
        //在旋转结点路径之上的结点的平衡因子要变化（左减右加,左边旋转意味着左子树高度会减1，右
        if(p->data.data > r->data.data){
```

```
        p->data.bf--;
        p = p->lchild;
    }else{
        p->data.bf++;
        p = p->rchild;
    }
}
//如果旋转的是根结点
if(!pre){
    if(r->data.bf == 2) lChange(root);
    else if(r->data.bf == -2) rChange(root);
}
//旋转非根结点，其前缀后的结点改变，故而重新设置指针
else{
    if(r->data.bf == 2){
        lChange(r);
        if(pre->lchild == p) pre->lchild = r;
        else pre->rchild = r;
    }else if(r->data.bf == -2){
        rChange(r);
        if(pre->lchild == p) pre->lchild = r;
        else pre->rchild = r;
    }
}
}

/**
 *x是否有兄弟结点
 *1.若x存在，如果有兄弟返回true，否则false
 *2.若x不存在，则在待插入位置若有兄弟则返回true，否则false
 */
template<class T>
bool AVLTree<T>::hasSibling(T x){
    Node<T>* r = root, *pre = NULL;
    while(r){
        //存在x
        if(r->data.data == x){
```

```
        if(!pre) return false;//root结点
        else if(pre->lchild && pre->rchild) return true;
        else return false;
    }
    pre = r;
    r = (r->data.data > x)?r->lchild:r->rchild;
}
//r为空, 不存在x
if(!r){
    if(pre->lchild || pre->rchild) return true;
    else return false;
}
return false;
}

template<class T>
int AVLTree<T>::height(){
    return height(root);
}

template<class T>
int AVLTree<T>::height(Node<T>* root){
    int hl, hr;
    if(!root) return 0;
    else{
        hl = height(root->lchild);
        hr = height(root->rchild);
        return (hl>hr?hl:hr)+1;
    }
}

template<class T>
void AVLTree<T>::deleteNode(Node<T>* &root, Node<T>* r, Node<T>* pre){
    Node<T>* p;
    if(!r->rchild && !r->lchild){                //如果是叶子结点
        if(pre){
            if(pre->lchild == r){
```



```
        pre->lchild = NULL;
    }else{
        pre->rchild = NULL;
    }
}
}else{
    root = NULL;
}
delete r;
}else if(r->rchild && r->lchild){                //如果左右子树都有
    p = r;
    //寻找右子树的最左结点
    r = r->rchild;
    while(r->lchild){
        r = r->lchild;
    }
    //将删除结点的左结点接到找到的最左结点之后
    r->lchild = p->lchild;
    //删除结点(如果pre是空,说明删除结点是根结点,不用改变前序结点指针)
    if(pre){
        if(pre->lchild == p) pre->lchild = p->rchild;
        else pre->rchild = p->rchild;
    }else{
        root = p->rchild;
    }
    delete p;
}else if(r->lchild){                            //如果只有左子树
    p = r;
    if(pre){
        if(pre->lchild == p) pre->lchild = r->lchild;
        else pre->rchild = r->lchild;
    }else{
        root = r->lchild;
    }
    delete p;
}else{                                          //如果只有右子树
    p = r;
    if(pre){
```

```
        if(pre->lchild == p) pre->lchild = r->rchild;
        else pre->rchild = r->rchild;
    }else{
        root = r->rchild;
    }
    delete p;
}

}

template<class T>
void AVLTree<T>::changeBF(Node<T>* &root){
    int bl, br;
    if(root){
        bl = height(root->lchild);
        br = height(root->rchild);
        root->data.bf = bl - br;
        changeBF(root->lchild);
        changeBF(root->rchild);
    }
}

template<class T>
void AVLTree<T>::resetAVL(Node<T>* &root){

}
```

c.main.cpp

```
#include <iostream>
#include "avltree.cpp"
using namespace std;
```

```
int main(){
    cout<<"-----平衡二叉树的测试-----"<<endl;
    AVLTree<int> avl;
    //int a[] = {29, 17, 13, 8, 34, 12, 6, 28, 32, 36, 30};
    //AVLTree<int> avl(a, 11);
    cout<<"tree:";
    avl.printAVL();

    /*
    Node<int>* r;
    cout<<"\n搜索8:";
    r = avl.searchAVL(8);
    if(r) cout<<r->data.data<<endl;
    else cout<<"无"<<endl;

    cout<<"\n搜索10:";
    r = avl.searchAVL(10);
    if(r) cout<<r->data.data<<endl;
    else cout<<"无"<<endl;

    cout<<"\n最小, 最大值:";
    r = avl.findMin();
    cout<<r->data.data<<",";
    r = avl.findMax();
    cout<<r->data.data<<endl;

    cout<<"\n树高度:"<<avl.height()<<endl;

    cout<<"\n删除17 ( root ) "<<endl;
    r = avl.deleteAVL(17);
    avl.printAVL();
    */
    return 0;
}
```


4.21 B-树实现

发表时间: 2012-08-10

1.什么是B-树？

这个在我的前一篇博客中已经详细的阐释过：

<http://hao3100590.iteye.com/blog/1576846>

具体的了解，好好看看这篇文章就可以了！

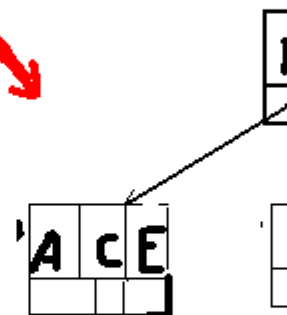
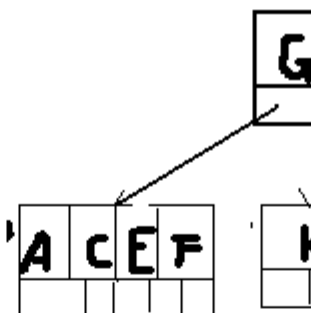
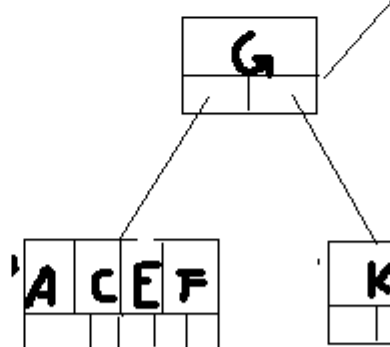
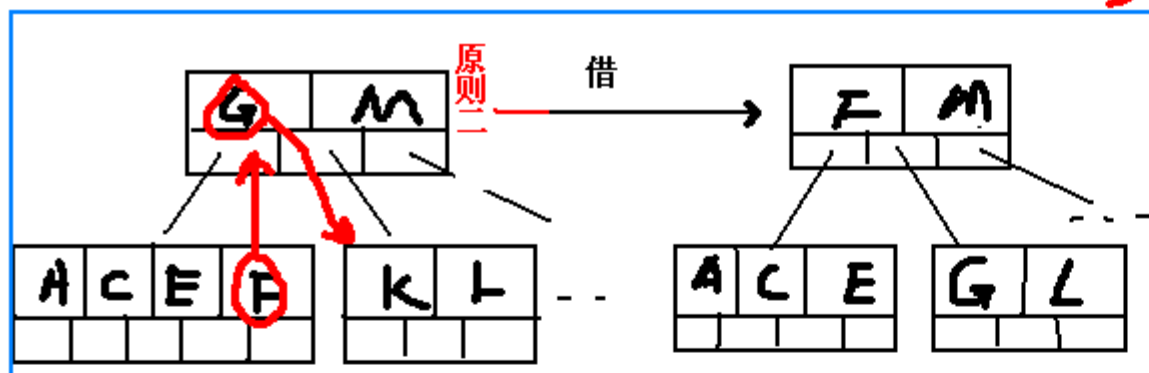
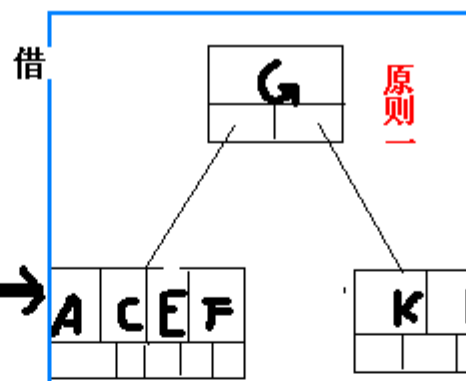
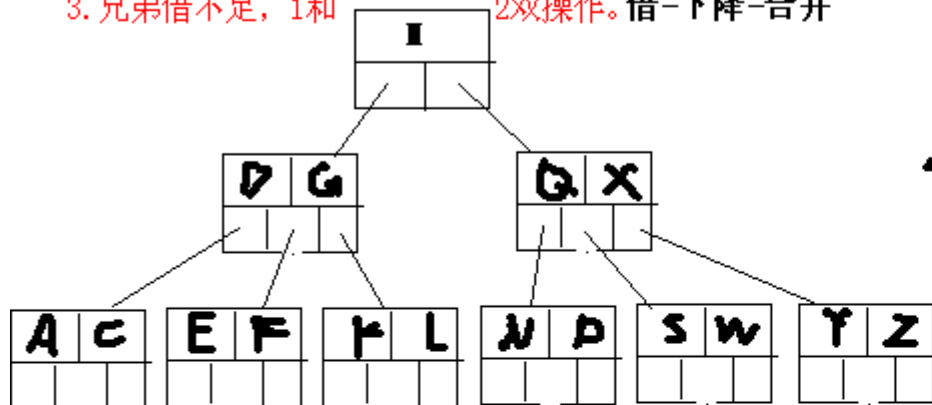
2.实现关键问题分析

a.B-树删除原则

见下图：

B-树删除原则：

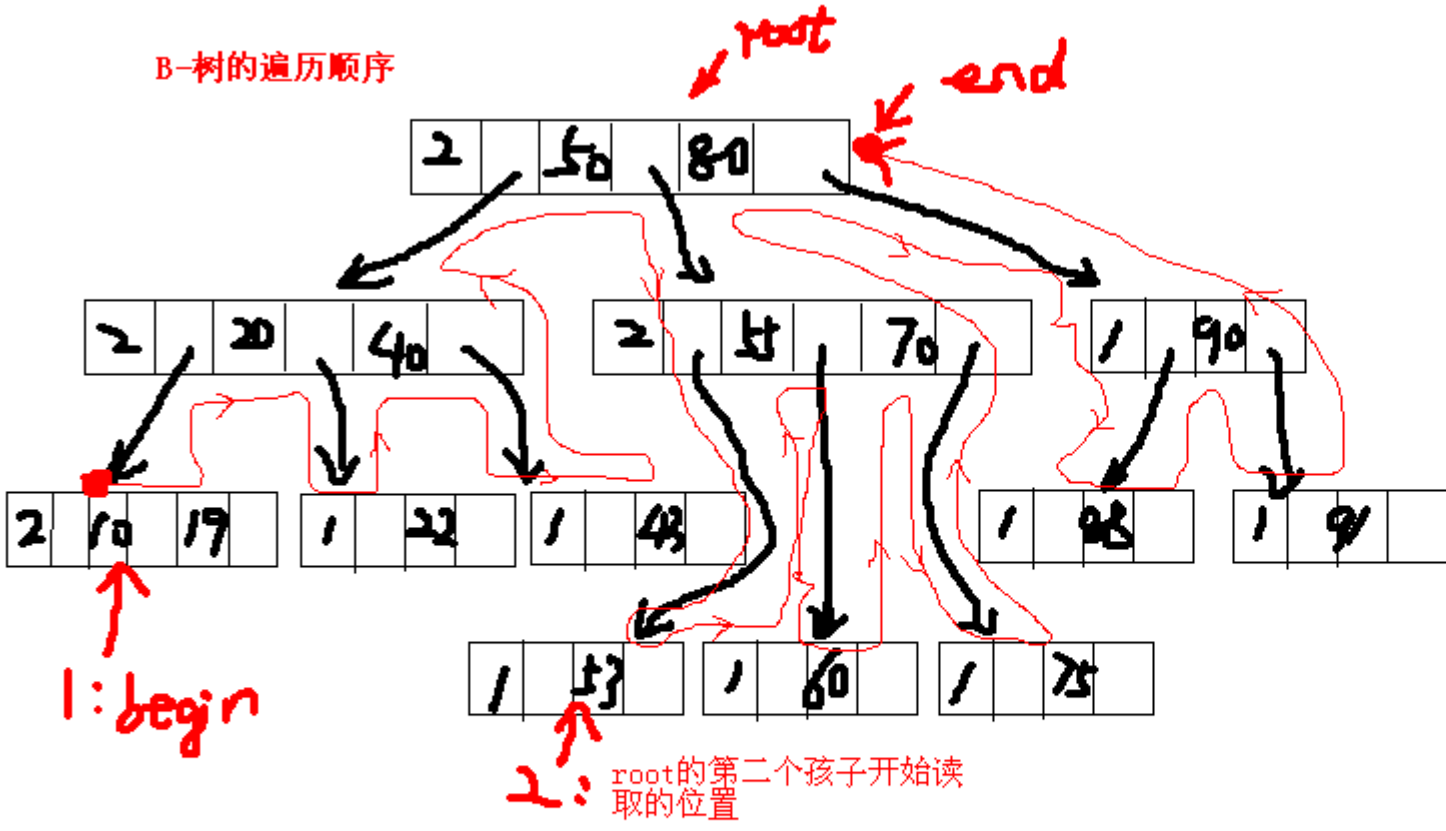
1. 当兄弟可以借：从兄弟借；当是非叶子，儿子可借，则可从儿子借
2. 当兄弟不可借：分裂-提升(插入时的原则)的逆下降-合并
3. 兄弟借不足，1和2双操作。借-下降-合并



当然总结起来，大的方面就3点，具体的细节就没有做多大的说明，在下面具体实现的时候会说明

b.B-树的递归和非递归遍历

对于非递归遍历，比较麻烦，自己需要按照下面图中所示的方式来遍历B-树，输出的关键字大小从小到大排列，但是对于递归却相当简单，这里非递归没有利用栈来实现，所以注意！



c.B-树删除伪代码

见下图：

B-tree删除伪代码：

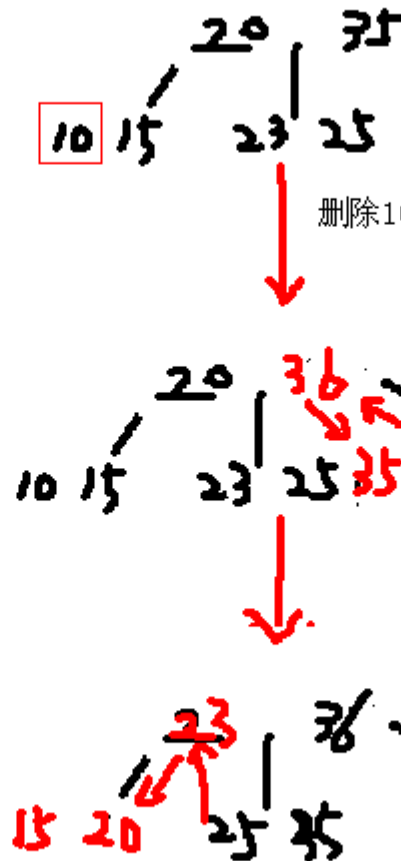
待删除结点为r，其父结点为p

```
if(r为叶子){
    if(r充足) 直接删除指定位置关键字;
    else{
        if(r兄弟可借) 从兄弟借;
        else 下降合并（往往还要递归向上合并）;
    }
} else {
    if(r的儿子可借) 从儿子借;
    else {
        if(r的兄弟不可借) 下降合并;
        else 先从孩子借，然后下降-合并;
    }
}
```

d.具体删除之借详细演示

如下图：

从兄弟或者孩子借



从孩子借，首先要寻找在所有孩子中是否有合适的，然后通过左右移动，实现从孩子中借
注意：如果孩子是非叶节点，还需要修改指针

从兄弟借，和从孩子借一样

3.关键函数说明

```
void insertList(T a[], int n, T x, int *p);
```

```
void deleteList(T *a, int n, int p);
```

```
void insertPoint(Node<T> *a[], int n, Node<T>* p, int pos);
```

```
void deletePoint(Node<T> *a[], int n, int p);
```

上面四个函数是实现插入和删除在Node中的关键字数组和指针数组的插入删除操作（就是数组删除）

下面的函数主要就是对插入和删除的操作

//插入的时候分裂提升结点p

```
Node<T>* divideTree(Node<T>* &root, Node<T>* &p);
```

//修改结点p的所有孩子结点的父节点为p，因为当删除的时候，父亲节点也可能改变

```
void modifyParent(Node<T>* &p);
```

//查询结点p在结点parent中的位置，或者关键字在结点parent中的位置（寻找插入位置或者查找）

```
int position(Node<T>* parent, Node<T>* p, T key);
```

//parent为p的双亲，从兄弟借元素（如果兄弟可借，否则什么都不做），pos是p中待删除元素的位置

```
bool borrowFromSib(Node<T>* &parent, Node<T>* p, int pos);
```

//从parent的儿子中借一个然后删除parent中一个关键字（pos是parent待删除位置）

```
bool borrowFromSon(Node<T>* &parent, int pos);
```

//合并parent中位置在pos处关键字的左右儿子,如果key[pos]是待删除的则设置flag=true

```
void mergeSib(Node<T>* &parent, int pos, bool flag);
```

//删除的时候递归合并操作

```
void recursionMerge(Node<T>* &p);
```

//合并结点q到p中（当删除的时候用）

```
void merge2Node(Node<T>* &p, Node<T>* q);
```

4. 代码实现

a.btree.h

```
//B树 ( B-树 ) 的头文件定义
#ifndef BTREE_H
#define BTREE_H

const int M=20;

template<class T>
struct Node{
    int keyNum;                //the number of key
    T key[M];                  //the array of key
    Node<T>* parent;           //point to parent
    Node<T>* son[M];           //point to sons
    //construct function! very important!!!!!!!!!!!!!!
    Node(){
        parent = 0;
        for(int i=0; i<M; i++){
            key[i] = 0;
            son[i] = 0;
        }
    }
};

template<class T>
class BTree{
public:
    BTree(int m);
    //m:B-树的阶, 长度为n的数组a
    BTree(int m, T a[], int n);
    ~BTree();
    Node<T>* insertBT(T x);
```

```
//void insertBT(Node<T>* &root, T x, int pos); //递归还没做出来
Node<T>* deleteBT(T x);
//x是查询关键字；pos是x在结点中的位置；p为k所在结点的双亲结点，如果没找到，也返回失败的叶子结点
Node<T>* search(T x, int *pos, Node<T>* &p);
int height();
void printBT(); //非递归打印
void printBT(Node<T>* root); //递归打印
Node<T>* getRoot();

private:
    int m; //m阶B-树
    Node<T>* root;
    void create(Node<T>* &root);
    void release(Node<T>* &root);
    void insertList(T a[], int n, T x, int *p);
    void insertPoint(Node<T>* a[], int n, Node<T>* p, int pos); //将节点指针p插入指针
    void deleteList(T *a, int n, int p);
    void deletePoint(Node<T>* a[], int n, int p);
    Node<T>* divideTree(Node<T>* &root, Node<T>* &p);
    void modifyParent(Node<T>* &p);
    //删除需要用的函数
    int position(Node<T>* parent, Node<T>* p, T key);
    bool borrowFromSib(Node<T>* &parent, Node<T>* p, int pos); //parent为p的双
        bool borrowFromSon(Node<T>* &parent, int pos);
        void mergeSib(Node<T>* &parent, int pos, bool flag);
        void recursionMerge(Node<T>* &p);
        void merge2Node(Node<T>* &p, Node<T>* q);
};
#endif
```

b.btree.cpp

```
#include <iostream>
#include "btree.h"
using namespace std;
```

```
template<class T>
BTree<T>::BTree(int m){
    this->m = m;
    root = NULL;
    create(root);
}

template<class T>
BTree<T>::BTree(int m, T a[], int n){
    this->m = m;
    root = NULL;
    if(n<=0) return;
    for(int i=0; i<n; i++){
        insertBT(a[i]);
    }
}

template<class T>
BTree<T>::~~BTree(){
    release(root);
}

template<class T>
Node<T>* BTree<T>::getRoot(){
    return root;
}

/**
 *插入的原则是分裂-提升
 *对于m阶子树的：根（非叶子）关键字个数是 1~m-1；非根关键字个数 m/2的上界-1~m-1
 *一旦超过这个范围就执行分裂
 */

template<class T>
Node<T>* BTree<T>::insertBT(T x){
    Node<T>* r = root;
    Node<T>* p = NULL;
```

```
int pos = 0; //待插入位置
if(!root){
    root = new Node<T>;
    root->keyNum = 1;
    root->parent = NULL;
    //注：结构体key和son都会自己初始化为空
    (root->key)[0] = x;
}else{
    //如果树中还没有x，就插入；否则，什么都不做，返回
    r = search(x, &pos, p);
    if(!r){
        insertList(p->key, p->keyNum, x, &pos);
        p->keyNum++;
    }else{
        return root;
    }

    //查看刚才插入的结点p是否达到上界，如果达到关键字上界，就分裂-提升
    //分裂-提升：将中间的key拿出来，提升到父结点，然后两边的key分成两个单独的结点，直到root
    if(p->keyNum >= m){
        cout<<"分裂-提升"<<endl;
        Node<T>* parent = divideTree(root, p);
        while(parent && parent->keyNum >= m){
            cout<<"继续提升！"<<endl;
            parent = divideTree(root, parent);
        }
    }
}
return root;
}
```

//删除，就依据两个原则：借和下降-合并(注：一下降就可能造成其父节点不足，需递归向上)

```
template<class T>
Node<T>* BTree<T>::deleteBT(T x){
    Node<T>* r = root;
    Node<T>* p = NULL;
    int pos = 0;
```



```
int t = 0;
if(!r) return NULL;
else{
    r = search(x, &pos, p);
    if(!r) return NULL;
    else{
        int low = m%2==0 ? m/2-1 : m/2;
        //1.*****如果待删除的结点是叶子结点*****
        if(!r->son[0]){
            //1.1.-----如果该结点充足，则直接删除结点-----
            if(r->keyNum > low){
                deleteList(r->key, r->keyNum, pos);
                r->keyNum--;
            }
            //1.2.-----如果该结点不足，兄弟可借(向左或者右兄弟借)-----
            else if(p){
                int ps = position(p, r, 0);
                bool b = borrowFromSib(p, r, pos);

                //1.3.-----如果该节点不足，且兄弟不可借，则下降-合并---
                if(!b){
                    //将r和其兄弟合并
                    deleteList(r->key, r->keyNum, pos);
                    r->keyNum--;
                    mergeSib(p, ps, false);
                    if(p->keyNum < low){
                        cout<<"递归合并"<<endl;
                        //删除后，其上层不够,递归合并
                        recursionMerge(p);
                    }
                }
            }
            }else{
                deleteList(r->key, r->keyNum, pos);
                r->keyNum--;
            }
        }
    }
}
//2.*****如果待删除的结点非叶子*****
```

```
        else{
            //int ps = positon(p, r, 0);
            //2.1.如果儿子可借，则从儿子借
            bool b = borrowFromSon(r, pos);
            if(b) cout<<"从儿子借\n";
            //2.2.如果儿子不可借，则看兄弟是否可借
            if(!b){
                //2.2.1.如果兄弟可借，则从兄弟借
                b = borrowFromSib(p, r, pos);
                if(b) cout<<"从兄弟借\n";
                //2.2.2.如果兄弟不可借，下降-合并
                if(!b){
                    mergeSib(r, pos, true);
                    if(r->keyNum < low){
                        //删除后，其上层不够,递归合并
                        recursionMerge(r);
                    }
                }
            }
        }
    }
}

return root;
}

/**
 * x      :待查询关键字；
 * *pos   :x在结点中的位置，如果没有，就是待插入位置
 * *p     :返回x所在结点的双亲结点，如果没找到，也返回失败的当前结点
 * 返回查找到的节点，如果没有返回空
 */

template<class T>
Node<T>* BTree<T>::search(T x, int *pos, Node<T>* &p){
    Node<T>* r = root;
    p = NULL;
    int n, i;
    while(r){
```

```
n = r->keyNum;
i = 0;
//找到指针所在位置或指针
if(x<r->key[0]){//就是第一个儿子指针
    *pos = 0;
    p = r;
    r = r->son[0];
}else{//后面位置
    while(i<n && x>r->key[i]) i++;
    if(x == r->key[i]){
        *pos = i;
        p = r->parent;
        return r;
    }else{
        p = r;
        r = r->son[i];
    }
    *pos = i;
}
}
//如果r是空的话，说明没有找到
return r;
}

template<class T>
int BTree<T>::height(){
    int h = 0;
    Node<T>* r = root;
    while(r){
        h++;
        r = r->son[0];
    }
    return h;
}
```

/**

```
*非递归打印
*根据B-树的特点，遍历顺序是关键字从小到大的顺序遍历
*
*/

template<class T>
void BTree<T>::printBT(){
    Node<T>* r = root;
    bool isLeaf = true;    //是否是最下层结点
    while(r && r->son[0]){ //如果非一层，则找到起始的结点（最左下结点）
        r = r->son[0];
    }

    Node<T>* pre = NULL;    //结点r的遍历前缀结点
    int k = 0;
    while(r){
        if(isLeaf){ //如果是最下层结点，遍历所
            for(int j=0; j<r->keyNum; j++){
                cout<<r->key[j]<<" ";
            }
            pre = r;
            r = r->parent;
        }else{
            int keyNum = r->keyNum;
            for(k=0; k<=keyNum; k++){ //注意这里：儿子指针个数=关键字个数+1
                if(r->son[k] == pre){
                    if(k != keyNum){
                        cout<<r->key[k]<<" ";
                        pre = r;
                        r = r->son[k+1];
                    }
                    break;
                }
            }
        }
    }
    //root的一个儿子结点结束，开始另一个，则先要找到其最下层左边第一个结点开始遍历
    if(pre == root && r->son[0]){
        while(r && r->son[0]){
            r = r->son[0];
        }
    }
}
```

```
        }

    }

    //如果是keyNum+1说明当前结点以及指向了最后儿子结点，遍历完了所有儿子，r之后指
    else if(k == keyNum){
        pre = r;
        r = r->parent;
    }

}

//判断当前结点是不是最下层结点
if(r){
    if(!r->son[0]) isLeaf = true;
    else isLeaf = false;
}

}

}

template<class T>
void BTree<T>::create(Node<T>* &root){
    T ch;
    cout<<"输入B-树数据(#结束)"<<endl;
    while(cin>>ch){
        if(ch == '#') break;
        insertBT(ch);
    }
}

template<class T>
void BTree<T>::insertList(T *a, int n, T x, int *p){
    int i=0;//待插入位置
    int k=n;
    if(x<a[0]){
        i = 0;
    }else{
        while(i<n && x>a[i]) i++;
    }
}
```

```
//插入
while(k != i){
    a[k] = a[k-1];
    k --;
}
a[i] = x;
*p = i;
}

//插入节点指针到son数组
template<class T>
void BTree<T>::insertPoint(Node<T> *a[], int n, Node<T>* p, int pos){
    if(!p) return;
    int k = n;
    //插入
    while(k != pos){
        a[k] = a[k-1];
        k --;
    }
    a[pos] = p;
}

template<class T>
void BTree<T>::deleteList(T *a, int n, int p){
    if(p<0 || p>=n) return;
    int k = p;
    //删除
    while(k != n-1){
        a[k] = a[k+1];
        k ++;
    }
}

template<class T>
void BTree<T>::deletePoint(Node<T> *a[], int n, int p){
    if(p<0 || p>=n) return;
    int k = p;
```

```
//Node<T>* r = a[p];
//删除
while(k != n-1){
    a[k] = a[k+1];
    k ++;
}
//delete r;
}

/**
 * *parent:结点p的父亲结点
 * *p      :待查询的结点p
 * key     :待查询的关键字key
 * 返回p在parent的位置; 或者key在parent的位置; 如果没找到返回-1
 * 注:如果p空,则是按关键字查询, 否则是按结点查询( 优先结点p查询, 不能同时查询两个 )
 */
template<class T>
int BTree<T>::position(Node<T>* parent, Node<T>* p, T key){
    int pos = 0;
    if(!parent) return -1;
    int n = parent->keyNum;
    //按关键字查询
    if(p == NULL){
        while(pos < n && key != parent->key[pos]) pos++;
        if(pos == n) pos = -1;
    }else{
        if(p->parent != parent) throw "输入不合法";
        while(pos <= n && p != parent->son[pos]) pos++;
        if(pos == n+1) pos = -1;
    }
    return pos;
}

/**
 * 分裂-提升结点p (注: 由B-树的特点可以知道, 插入的结点必定是在最低的结点(即没有叶子结点), 增
 * *root   : 根结点
 * *p      : 待分裂结点
```

```
* 返回    : 分裂后的父节点
* 注意    : 在分裂过程一定不要忘了修改结点的父亲
*/
```

```
template<class T>
Node<T>* BTree<T>::divideTree(Node<T>* &root, Node<T>* &p){
    int mid = 0;
    Node<T>* parent = NULL;
    Node<T>* t;
    if(p && p->keyNum >= m){
        mid = (p->keyNum)/2;                //找到结点p关键字的中间位置
        parent = p->parent;                  //找到结点p的父节点，如果存在
        if(!parent){                        //无父节点
            int i=0;
            //新根结点
            Node<T>* r = new Node<T>;
            r = new Node<T>;
            r->keyNum = 1;
            r->parent = NULL;
            r->key[0] = p->key[mid];

            //右结点
            Node<T>* right = new Node<T>;
            right->keyNum = (p->keyNum)-1-mid;
            right->parent = r;
            for(i=0; i<mid; i++){
                right->key[i] = p->key[mid+1+i];
                right->son[i] = p->son[mid+1+i];
            }
            right->son[i] = p->son[mid+1+i];

            //左结点
            Node<T>* left = new Node<T>;
            left->keyNum = mid;
            left->parent = r;
            for(i=0; i<left->keyNum; i++){
                left->key[i] = p->key[i];
                left->son[i] = p->son[i];
            }
        }
    }
}
```



```
    }
    left->son[i] = p->son[i];

    (r->son)[0] = left;
    (r->son)[1] = right;
    modifyParent(left);
    modifyParent(right);
    root = r;
    parent = r;
}else{
    //右结点
    int i;
    Node<T>* right = new Node<T>;
    right->keyNum = (p->keyNum)-1-mid;
    right->parent = parent;
    for(i=0; i<mid; i++){
        right->key[i] = p->key[mid+1+i];
        right->son[i] = p->son[mid+1+i];
    }
    right->son[i] = p->son[mid+1+i];

    //左结点
    Node<T>* left = new Node<T>;
    left->keyNum = mid;
    left->parent = parent;
    for(i=0; i<left->keyNum; i++){
        left->key[i] = p->key[i];
        left->son[i] = p->son[i];
    }
    left->son[i] = p->son[i];

    int pos = 0;
    //修改父节点
    insertList(parent->key, parent->keyNum, p->key[mid], &pos);
    parent->keyNum++;
    //在父的son数组中插入新结点的左右指针
    //注意：首先应该用left替换pos位置的废弃指针——因为它指向的节点提升了，然后
```

```
        parent->son[pos] = left;
        insertPoint(parent->son, parent->keyNum+1, right, pos+1);
        modifyParent(left);
        modifyParent(right);
    }
    delete p;
}
return parent;
}

/**
 *当分裂-提升之后修改结点的父亲
 *p : 已经经过分裂-提升之后的新的双亲
 *需要修改的是p的所有儿子的双亲, 不在是原来的双亲 (已经delete)
 */
template<class T>
void BTree<T>::modifyParent(Node<T>* &p){
    //p存在并且有孩子
    if(p && p->son[0]){
        for(int i=0; i<p->keyNum+1; i++){
            p->son[i]->parent = p;
        }
    }
}

//递归删除
template<class T>
void BTree<T>::release(Node<T>* &root){
    if(root){
        for(int i=0; i<=root->keyNum; i++){
            release(root->son[i]);
        }
        delete root;
    }
}

//递归打印
```

```
template<class T>
void BTree<T>::printBT(Node<T>* root){
    if(root){
        for(int i=0; i<root->keyNum; i++){
            printBT(root->son[i]);
            cout<<root->key[i]<<" ";
        }
        printBT(root->son[root->keyNum]);
    }
}

/**
 * 从兄弟借元素(如果存在，否则什么都不做)
 * parent: p的父节点
 * p      : 有待删除元素的节点
 * ps     : 待删除元素在p中的位置
 *注：parent是p的双亲
*如果存在兄弟可借，则借并返回true，否则false
*/

template<class T>
bool BTree<T>::borrowFromSib(Node<T>* &parent, Node<T>* p, int ps){
    bool flag = false, haveSon = false;
    if(!p) return flag;
    if(p->parent != parent) throw "输入参数有误！";
    Node<T>* r, *s;
    T k;
    int low = m%2==0 ? m/2-1 : m/2;
    int pos = position(parent, p, 0);
    int i = 0, t=0;
    //先找到可借的兄弟
    for(i=0; i<=parent->keyNum; i++){
        r = parent->son[i];
        if(r->keyNum > low){
            flag = true;
            break;
        }
    }
}
```

```
if(r->son[0]) haveSon = true;
if(i <= parent->keyNum){
    if(i>pos){//找到的兄弟位于p的右边
        do{
            k = parent->key[i-1];
            parent->key[i-1] = r->key[0];
            deleteList(r->key, r->keyNum, 0);
            s = r->son[0];
            deletePoint(r->son, r->keyNum+1, 0);
            r->keyNum--;
            i--;
            r = parent->son[i];
            insertList(r->key, r->keyNum, k, &t);
            r->keyNum++;
            insertPoint(r->son, r->keyNum+1, s, r->keyNum);
        }while(r != p);
        //插入的位置都是在尾部，故而不用管t
        if(haveSon){
            mergeSib(p, ps, true);
        }else{
            deleteList(r->key, r->keyNum, ps);
            r->keyNum--;
        }
    }else{
        //位于左边
        do{
            k = parent->key[i];
            parent->key[i] = r->key[r->keyNum-1];
            deleteList(r->key, r->keyNum, r->keyNum-1);
            s = r->son[r->keyNum];
            deletePoint(r->son, r->keyNum+1, r->keyNum);
            r->keyNum--;
            i++;
            r = parent->son[i];
            insertList(r->key, r->keyNum, k, &t);
            r->keyNum++;
            insertPoint(r->son, r->keyNum+1, s, 0);
        }while(r != p);
    }
}
```

```
        //插在待删除位置前(这是肯定满足的, 因为插入的时候都是0位置)
        if(t <= ps) ps++;
        if(haveSon){
            mergeSib(p, ps, true);
        }else{
            deleteList(r->key, r->keyNum, ps);
            r->keyNum--;
        }
    }
}
return flag;
}

/**
 *功能   : 删除父亲结点元素, 如果儿子充足, 则从儿子借
 *parent : 待删除元素所在结点
 *pos     : 待删除元素的位置
 *返回    : 如果找到合适son就执行操作并返回true, 否则false
 */

template<class T>
bool BTree<T>::borrowFromSon(Node<T>* &parent, int pos){
    bool flag = false;
    bool haveSon = false;
    if(!parent || pos<0 || pos>=parent->keyNum) throw "参数有误!";
    if(!parent->son[0]) return flag;
    Node<T>* r, *s;
    T k;
    int low = m%2==0 ? m/2-1 : m/2;
    //寻找合适的儿子
    int i=0,t=0;
    for(i=0; i<=parent->keyNum; i++){
        r = parent->son[i];
        if(r->keyNum > low){
            flag = true;
            break;
        }
    }
}
```

```
if(r->son[0]) haveSon = true;
if(i <= parent->keyNum){
    if(i>pos){//如果儿子在右边
        while(i != pos+1){
            k = parent->key[i-1];
            parent->key[i-1] = r->key[0];
            deleteList(r->key, r->keyNum, 0);
            s = r->son[0];
            deletePoint(r->son, r->keyNum+1, 0);
            r->keyNum--;
            i--;
            r = parent->son[i];
            insertList(r->key, r->keyNum, k, &t);
            r->keyNum++;
            insertPoint(r->son, r->keyNum+1, s, r->keyNum);
        }
        if(haveSon){
            parent->key[pos] = r->key[0];
            deleteList(r->key, r->keyNum, 0);
            s = r->son[0];
            deletePoint(r->son, r->keyNum+1, 0);
            r->keyNum--;
            Node<T>* q = parent->son[pos]->son[parent->son[pos]->keyNum];
            merge2Node(q, s);
        }else{
            parent->key[pos] = r->key[0];
            deleteList(r->key, r->keyNum, 0);
            r->keyNum--;
        }
    }else{
        while(i != pos){
            k = parent->key[i];
            parent->key[i] = r->key[r->keyNum-1];
            deleteList(r->key, r->keyNum, r->keyNum-1);
            s = r->son[r->keyNum];
            deletePoint(r->son, r->keyNum+1, r->keyNum);
            r->keyNum--;
```

```
        i++;
        r = parent->son[i];
        insertList(r->key, r->keyNum, k, &t);
        r->keyNum++;
        insertPoint(r->son, r->keyNum+1, s, 0);
    }
    if(haveSon){
        parent->key[pos] = r->key[r->keyNum-1];
        deleteList(r->key, r->keyNum, r->keyNum-1);
        s = r->son[r->keyNum];
        deletePoint(r->son, r->keyNum+1, r->keyNum);
        r->keyNum--;
        Node<T>* m = parent->son[pos+1]->son[0];
        merge2Node(s, m);
        parent->son[pos+1]->son[0] = s;
        modifyParent(parent->son[pos+1]->son[0]);
    }else{
        parent->key[pos] = r->key[r->keyNum-1];
        deleteList(r->key, r->keyNum, r->keyNum-1);
        r->keyNum--;
    }
}

}

return flag;
}

/**
 *将q合并到p之后
 */

template<class T>
void BTree<T>::merge2Node(Node<T>* &p, Node<T>* q){
    if(!p || !q) return;
    int t, i=0;
    for(i=0; i<q->keyNum; i++){
        insertList(p->key, p->keyNum, q->key[i], &t);
        p->keyNum++;
        insertPoint(p->son, p->keyNum+1, q->son[i], p->keyNum);
    }
}
```

```
    }
    insertPoint(p->son, p->keyNum+1, q->son[i], p->keyNum+1);
    delete q;
}

/**
 * 合并parent所在pos位置的左右孩子，并将key[pos]下降到合并结点中(下降-合并)
 * 注：下降之后，并且删除了父结点中的key，它在合并结点中
 * parent：待合并孩子的双亲结点
 * pos：在parent中孩子指针位置，pos位置的孩子和其左或右孩子是待合并的
 * ***如果key[pos]是待删除的则设置flag=true
 */
template<class T>
void BTree<T>::mergeSib(Node<T>* &parent, int pos, bool flag){
    if(!parent || pos<0 || pos>parent->keyNum) return;
    if(!parent->son[0]) throw "参数有误!";
    //将r和其兄弟合并
    int t=0;
    int pos1,pos2;
    if(pos == parent->keyNum){
        pos1 = pos-1;
        pos2 = pos;
    }else{
        pos1 = pos;
        pos2 = pos+1;
    }
    Node<T>* r = parent->son[pos1];
    Node<T> *x = parent->son[pos2];
    if(x){
        //将父结点放入左儿子(下降)
        insertList(r->key, r->keyNum, parent->key[pos1], &t);
        r->keyNum++;

        //将右兄弟中的关键字和指针都放到左儿子中(合并)
        int i=0;
        while(x->keyNum != 0){
            insertPoint(r->son, r->keyNum+1, x->son[i++], r->keyNum);
        }
    }
}
```



```
        insertList(r->key, r->keyNum, x->key[0], &t);
        r->keyNum++;
        deleteList(x->key, x->keyNum, 0);
        x->keyNum--;
    }
    insertPoint(r->son, r->keyNum+1, x->son[i], r->keyNum);
    delete x;

    //如果flag是true, 则要删除下降的key
    if(flag){
        int pp = position(r, NULL, parent->key[pos1]);
        if(!r->son[0]){
            deleteList(r->key, r->keyNum, pp);
            r->keyNum--;
        }else{
            mergeSib(r, pp, true);
        }
    }
}
```

```
//删除父节点的关键字和指针
deleteList(parent->key, parent->keyNum, pos1);
deletePoint(parent->son, parent->keyNum+1, pos2);
parent->keyNum--;
```

```
if(parent->keyNum == 0){
    if(parent == root) root = r;
    parent = r;
    parent->parent = NULL;
}
modifyParent(r);
```

```
}
```

```
}
```

//递归合并, 如果p不够就要向上合并

```
template<class T>
```

```
void BTree<T>::recursionMerge(Node<T>* &p){
```

```
    int low = m%2==0 ? m/2-1 : m/2;
```

```
if(p && p->keyNum<low){
    //看是否可以从兄弟借,若可借,则借,否则下降-合并
    if(!borrowFromSib(p->parent, p, -1)){
        cout<<"不可借"<<endl;
        //下降-合并
        int i = position(p->parent, p, 0);
        mergeSib(p->parent, i, false);
        recursionMerge(p->parent);
    }
}

}

/**
 * 递归插入
 * *p   : 暂存插入位置的节点
 * *pos : 暂存插入的位置

template<class T>
void BTree<T>::insertBT(Node<T>* &r, T x, int pos){
    if(!r){
        r = new Node<T>;
        r->keyNum = 1;
        r->parent = NULL;
        (r->key)[0] = x;
        return;
    }else{
        pos = 0;
        //在r上搜索插入位置
        if(x<r->key[0]){
            pos = 0;
        }else{//后面位置
            while(pos<r->keyNum && x>r->key[pos]) pos++;
            if(x == r->key[pos]) return;
        }
        //是否递归插入,如果在叶子结点,就找到了插入位置,否则递归插入
        if(!r->son[pos]){//找到了插入的节点
            cout<<"插入结点"<<endl;
```

```
        insertList(r->key, r->keyNum, x, &pos);
        r->keyNum++;
        if(r->keyNum >= m){
            cout<<"分裂-提升"<<endl;
            //divideTree(root, r);
            Node<T>* parent = divideTree(root, r);
            while(parent && parent->keyNum >= m){
                cout<<"继续提升!"<<endl;
                parent = divideTree(root, parent);
            }
        }
        return;
    }else{//没有找到插入结点,递归插入
        insertBT(r->son[pos], x, pos);
    }
}

}

}

*/
```

c.main.cpp

```
#include <iostream>
#include "btree.cpp"
using namespace std;

//注意35是 '#' ,故而输入的时候注意
int main(){
    //int a[] = {23, 40, 12, 50, 80, 42, 6, 7};//, 47, 90, 95
    //int a[] = {23,50,80,100,60,55,70,85,87,57,20,89,90,110,52,59,53, 111,114,24,25,26};//
    int a[] = {23,50,80,100,60,55,70,85,87,57,20,89,90,110,52,59,53,21};//
    BTree<int> bt(5, a, 18);
    //BTree<int> bt(3);

    cout<<"非递归打印"<<endl;
```

```
bt.printBT();

cout<<"\n递归打印"<<endl;
Node<int>* root = bt.getRoot();
bt.printBT(root);

cout<<"\n高度:"<<bt.height()<<endl;

cout<<"\n搜索"<<endl;
Node<int>* p;
int pos = 0;
p = bt.search(100, &pos, p);
if(p){
    for(int i=0; i<p->keyNum; i++) cout<<p->key[i]<<" ";
    cout<<"\n位置"<<pos<<" "<<p->keyNum<<endl;
}

/*
cout<<"\n删除叶子，并且充足"<<endl;
bt.deleteBT(54);
bt.printBT();

cout<<"\n删除叶子，叶子不足，兄弟可借（借在左或右）"<<endl;
bt.deleteBT(23);
bt.printBT();

cout<<"\n删除叶子，叶子不足，兄弟不可借"<<endl;
root = bt.deleteBT(70);
bt.printBT();
cout<<endl;
bt.printBT(root);

cout<<endl;
cout<<root->keyNum<<endl;
for(int i=0; i<root->keyNum; i++){
    cout<<root->key[i]<<" ";
}
```

```
cout<<endl;
for(int i=0; i<=root->keyNum; i++){
    cout<<root->son[i]->keyNum<<endl;
    for(int j=0; j<root->son[i]->keyNum; j++){
        cout<<root->son[i]->key[j]<<" ";
    }
    cout<<endl;
}
cout<<endl;
*/

/*
cout<<"\n删除非叶子，儿子可借（借在左或右）"<<endl;
root = bt.deleteBT(55);
cout<<root->key[0]<<endl;
for(int i=0; i<root->keyNum+1; i++){
    for(int j=0; j<root->son[i]->keyNum; j++){
        cout<<root->son[i]->key[j]<<" ";
    }
    cout<<endl;
}
bt.printBT(root);
cout<<endl;
bt.printBT();
p = bt.search(53, &pos, p);
//p = p->son[p->keyNum];
cout<<p->keyNum<<endl;
for(int i=0; i<p->keyNum+1; i++){
    for(int j=0; j<p->son[i]->keyNum; j++){
        cout<<p->son[i]->key[j]<<" ";
    }
    cout<<endl;
}

cout<<"\n删除非叶子，儿子不可借，兄弟不可借"<<endl;
root = bt.deleteBT(85);
```

```
bt.printBT(root);  
cout<<"\nroot:"<<root->keyNum<<endl;  
bt.printBT();  
  
cout<<"\n删除非叶子，儿子不可借，兄弟可借（借在左或右）"<<endl;  
bt.deleteBT(85);  
bt.printBT();  
*/  
return 0;  
}
```

4.22 红黑树的插入总结

发表时间: 2012-08-10 关键字: 红黑树

1.红黑树

这个在july的博客中有详尽的说明，我就不再赘述了

http://blog.csdn.net/v_JULY_v/article/details/6105630

2.红黑树的插入

插入见下图：

注：当前插入结点是N，叔叔是U，祖父是G

case1:新结点N是根

case2:新结点N的父P是黑色

下面的情况都有：父结点P是红色（因为插入的也是红色，两个红色就破坏了规则4，故而需要重新着色），祖父是黑色（如果祖父是红色，而父亲也是红色，明显也违背了规则4，故而插入之前就不是红黑树，所以祖父必是黑色），故而下面三种情况就不写了。

case3:叔叔结点U是红色

case4:叔叔结点U为黑色或不存在，新结点N是右孩子

case5:叔叔结点U为黑色或不存在，新结点N是左孩子

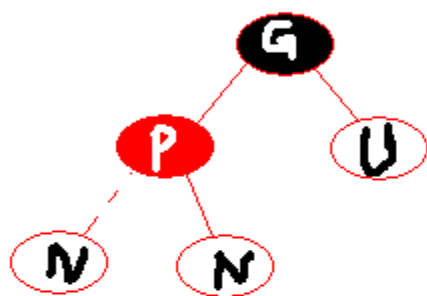
父为红
祖父为黑

五条规则：

1. 每个结点要么红，要么黑；
2. 根为黑色；
3. 叶子即空节点（NULL）；
4. 若一个节点是红色，那么它的父节点必须是黑色；
5. 对每个节点，从该节点到叶子节点的所有路径上，黑色节点的数量必须相同。

1. 对于插入和删除，主要那么左边五种情况主要
2. 对于删除，尤其是5，着色，那就意味着经过，那就要想办法平衡，了恢复性质5。
3. 朝着上面的说法来看待已在模拟插入或者删除case1, case2也会明白。

4. 对于插入，如果有父亲新着色，因为它破坏了规



N可能是左也可能是右

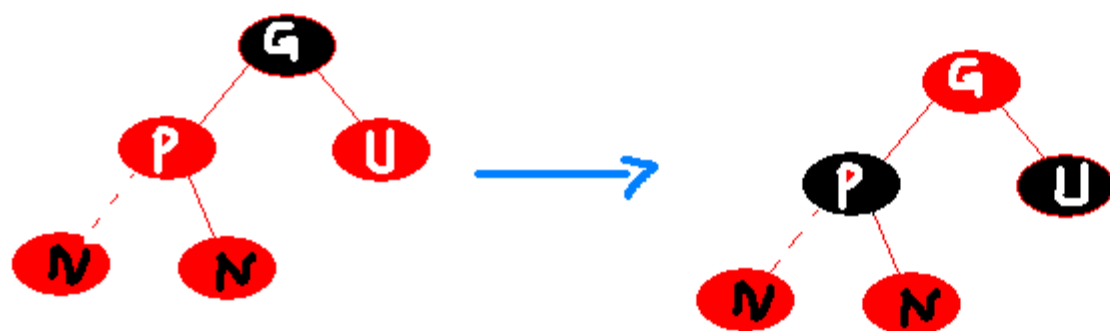
case1:新结点N是根



case2:新结点N的父P是黑色



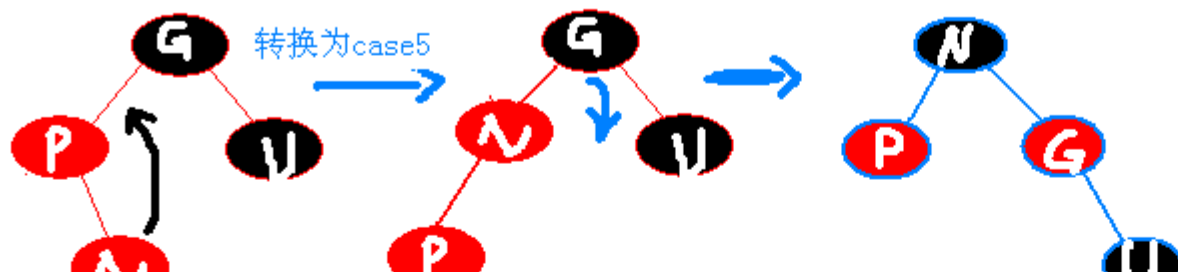
case3:叔叔结点N是红色（P为红色，G为黑色）



case3:将祖父G变黑，这样黑色结点的数量不变

注意：如果G是根

case4:叔叔结点N为黑色或不存在，N为右孩子（P为红色，G为黑色）



case4:首先，通过旋转，然后，经

注：在转换先消除了

即红色结点右子树上

对策

case1:将结点变黑即可

case2:不做任何改变，因为没有改变任何规则

4.23 set和map的简单实现

发表时间: 2012-08-10 关键字: set, map, 实现

1.Set的简单实现

set是利用二叉查找树来实现的,而且为了查找方便,添加了另外两个指针,一个指向下一个最小结点,一个指向上一个最大结点。

iset.h

```
//利用二叉查找树实现set
#ifndef ISET_H
#define ISET_H

template<class T>
struct Node{
    T data;
    Node<T> *rchild, *lchild; //左右孩子
    Node<T> *parent;           //父结点
    Node<T> *next, *pre;       //下一个最小和上一个最大的结点
    Node(){
        rchild = lchild = parent = 0;
        next = pre = 0;
    }
};

template<class T>
class ISet{
public:
    ISet();
    ISet(T a[], int n);
    ~ISet();
    int size();
    bool isEmpty();
    bool contains(T t);
    T* toArray();
};
```

```
void add(T t);
void remove(T t);
void addAll(T a[], int n);
void removeAll();
void clear();
void print();

//下面实现迭代器
typedef Node<T>* iterator;
typedef const Node<T>* const_iterator;
iterator Iterator();
const_iterator Iterator() const;
iterator next();
const_iterator next() const;
iterator pre();
const_iterator pre() const;
bool hasNext();
bool hasPre();
//还可以操作符重载, ++, --, +实现通用的迭代器
private:
    Node<T>* root;
    Node<T> *head, *tail;           //头(最小)和尾(最大)
    void clear(Node<T>* &root);
    int size(Node<T>* root);
    void print(Node<T>* root);
    void toArray(Node<T>* root, T* &a, int *i);
    void deleteNode(Node<T>* &root, Node<T>* r, Node<T>* pre); //删除结点p
    void preSet(Node<T>* t);
    void nextSet(Node<T>* t);
};
#endif
```

iset.cpp

```
#include <iostream>
#include "iset.h"
using namespace std;

template<class T>
ISet<T>::ISet(){
    head = tail = root = NULL;
}

template<class T>
ISet<T>::ISet(T a[], int n){
    head = tail = root = NULL;
    addAll(a, n);
}

template<class T>
ISet<T>::~~ISet(){
    clear();
}

template<class T>
int ISet<T>::size(){
    return size(root);
}

template<class T>
bool ISet<T>::isEmpty(){
    if(!root) return true;
    return false;
}

template<class T>
bool ISet<T>::contains(T t){
    Node<T>* r = root;
    while(r){
        if(r->data == t) break;
    }
}
```

```
        r = (r->data > t)?r->lchild:r->rchild;

    }
    //如果没找到，返回false
    if(!r) return false;
    return true;
}

template<class T>
T* ISet<T>::toArray(){
    T *a = new T[size()];
    int i = 0;
    toArray(root, a, &i);
    return a;
}

template<class T>
void ISet<T>::add(T x){
    Node<T>* r = root, *m = NULL, *n = NULL;
    if(r == NULL){
        Node<T>* t = new Node<T>;
        t->data = x;
        root = t; //必须设置新的root结点，因为t和root在内存指向位置不同
    }else{
        Node<T>* p = r;
        //查找待插入结点位置
        while(r){
            if(r->data == x){
                return;
            }
            p = r;
            if(r->data > x){
                n = r;
                r = r->lchild;
            }else{
                m = r;
                r = r->rchild;
            }
        }
    }
}
```

```
    }
    Node<T>* t = new Node<T>;
    t->data = x;
    if(p->data > x){//插入左
        p->lchild = t;
        p->pre = t;
    }else{
        p->rchild = t;
        p->next = t;
    }
    t->parent = p;
    preSet(t);
    nextSet(t);
    if(m && m!=p) nextSet(m);
    if(n && n!=p) preSet(n);
}
}

template<class T>
void ISet<T>::remove(T x){
    if(!root) return;
    Node<T>* pre = NULL, *r = root;
    //查找待删除结点位置(pre是r的前序结点)
    while(r){
        if(r->data == x){
            break;
        }

        pre = r;
        r = (r->data > x)?r->lchild:r->rchild;
    }
    //没找到
    if(!r){ cout<<"没有待删除的值："<<x<<endl; return;}
    //如果pre == NULL说明是root结点
    deleteNode(root, r, pre);
}
```

```
template<class T>
void ISet<T>::addAll(T a[], int n){
    if(n <= 0) return;
    for(int i=0; i<n; i++){
        add(a[i]);
    }
}

template<class T>
void ISet<T>::removeAll(){
    clear();
}

template<class T>
void ISet<T>::clear(){
    clear(root);
    //注：当clear之后，root也被delete，故而重新置为NULL
    root = NULL;
}

template<class T>
void ISet<T>::print(){
    print(root);
}

template<class T>
void ISet<T>::print(Node<T>* root){
    if(root){
        cout<<root->data<<" ";
        print(root->lchild);
        print(root->rchild);
    }
}

template<class T>
void ISet<T>::clear(Node<T>* &root){
    if(root){
```

```
        clear(root->lchild);
        clear(root->rchild);
        delete root;
    }
}

template<class T>
int ISet<T>::size(Node<T>* root){
    if(!root) return 0;
    else{
        return size(root->lchild)+size(root->rchild)+1;
    }
}

template<class T>
void ISet<T>::toArray(Node<T>* root, T* &a, int *i){
    if(root){
        a[(*i)++] = root->data;
        toArray(root->lchild, a, i);
        toArray(root->rchild, a, i);
    }
}

//r为待删除结点，pre为r的双亲
template<class T>
void ISet<T>::deleteNode(Node<T>* &root, Node<T>* r, Node<T>* pre){
    Node<T>* p;
    //先修改待删除r的pre,next结点指针
    if(r->pre){
        r->pre->next = r->next;
    }
    if(r->next){
        r->next->pre = r->pre;
    }
    if(!r->rchild && !r->lchild){ //如果是叶子结点
        if(pre){
            if(pre->lchild == r){
```



```
        pre->lchild = NULL;
    }else{
        pre->rchild = NULL;
    }
}
}else{
    root = NULL;
}
delete r;
}else if(r->rchild && r->lchild){ //如果左右子树都有(还有一种处理办法就是F
    p = r;
    //寻找右子树的最左结点
    r = r->rchild;
    while(r->lchild){
        r = r->lchild;
    }
    //将删除结点的左结点接到找到的最左结点之后
    r->lchild = p->lchild;
    r->lchild->parent = r;
    //删除结点(如果pre是空,说明删除结点是根结点,不用改变前序结点指针)
    if(pre){
        if(pre->lchild == p){
            pre->lchild = p->rchild;
        }else{
            pre->rchild = p->rchild;
        }
        p->rchild->parent = pre;
    }else{
        p->rchild->parent = NULL;
        root = p->rchild;
    }
    delete p;
}else if(r->lchild){ //如果只有左子树
    p = r;
    if(pre){
        if(pre->lchild == p) pre->lchild = r->lchild;
        else pre->rchild = r->lchild;
        r->lchild->parent = pre;
    }
```

```
        }else{
            r->lchild->parent = NULL;
            root = r->lchild;
        }

        delete p;
    }else{                                     //如果只有右子树
        p = r;
        if(pre){
            if(pre->lchild == p) pre->lchild = r->rchild;
            else pre->rchild = r->rchild;
            r->rchild->parent = pre;
        }else{
            r->rchild->parent = NULL;
            root = r->rchild;
        }
        delete p;
    }
}

template<class T>
void ISet<T>::preSet(Node<T>* t){
    if(!t) return;
    Node<T>* p;
    //如果t有左孩子，则pre就是左孩子的最右结点
    p = t->lchild;
    if(p){
        while(p->rchild){
            p = p->rchild;
        }
        t->pre = p;
    }else{//p是空
        p = t;
        if(!t->parent){
            t->pre = NULL;
        }else if(t->parent->lchild == t){//若p为左孩子，则回朔，直到遇到结点是右孩子
            t->pre = t->parent->pre;
        }
    }
}
```

```
        while(p->parent){
            if(p->parent->rchild == p) break;
            p = p->parent;
        }
        t->pre = p->parent;
    }else{
        t->pre = t->parent;
    }
}
}
```

```
template<class T>
void ISet<T>::nextSet(Node<T>* t){
    if(!t) return;
    Node<T>* p;
    //如果t有右孩子，则next就是有孩子的最左结点
    p = t->rchild;
    if(p){
        while(p->lchild){
            p = p->lchild;
        }
        t->next = p;
    }else{
        p = t;
        if(!t->parent){
            t->next = NULL;
        }else if(t->parent->rchild == t){
            while(p->parent){
                if(p->parent->lchild == p) break;
                p = p->parent;
            }
            t->next = p->parent;
        }else{
            t->next = p->parent;
        }
    }
}
```

```
template<class T>
typename ISet<T>::iterator ISet<T>::Iterator(){
    if(!root) return NULL;
    Node<T>* p = root;
    while(p->lchild){
        p = p->lchild;
    }
    head = p;
    p = root;
    while(p->rchild){
        p = p->rchild;
    }
    tail = p;
    return head;
}

template<class T>
typename ISet<T>::const_iterator ISet<T>::Iterator() const{
    if(!root) return NULL;
    Node<T>* p = root;
    while(p->lchild){
        p = p->lchild;
    }
    head = p;
    p = root;
    while(p->rchild){
        p = p->rchild;
    }
    tail = p;
    return head;
}

template<class T>
typename ISet<T>::iterator ISet<T>::next(){
    Node<T>* t;
    if(head){
```

```
        t = head;
        head = head->next;
        return t;
    }
    return NULL;
}

template<class T>
typename ISet<T>::const_iterator ISet<T>::next() const{
    Node<T>* t;
    if(head){
        t = head;
        head = head->next;
        return t;
    }
    return NULL;
}

template<class T>
typename ISet<T>::iterator ISet<T>::pre(){
    Node<T>* t;
    if(tail){
        t = tail;
        tail = tail->pre;
        return t;
    }
    return NULL;
}

template<class T>
typename ISet<T>::const_iterator ISet<T>::pre() const{
    Node<T>* t;
    if(tail){
        t = tail;
        tail = tail->pre;
        return t;
    }
}
```

```
        return NULL;
    }

template<class T>
bool ISet<T>::hasNext(){
    if(head) return true;
    return false;
}

template<class T>
bool ISet<T>::hasPre(){
    if(tail) return true;
    return false;
}
```

main.cpp

```
#include <iostream>
#include "iset.cpp"
using namespace std;

int main(){
    int a[] = {19, 38,1,41,39,54,6,3};
    ISet<int> s(a, 8);
    s.print();
    cout<<endl;
    /*
    int n = s.size();
    cout<<"size:"<<n<<endl;
    cout<<"isEmpty:"<<s.isEmpty()<<endl;
    cout<<"contains 4:"<<s.contains(4)<<endl;
    cout<<"contains 90:"<<s.contains(90)<<endl;

    int *k = new int[n];
```

```
k = s.toArray();
for(int i=0; i<n; i++){ cout<<k[i]<<" ";}
cout<<endl;
delete k;

cout<<"添加5\n";
s.add(5);
cout<<"size:"<<s.size()<<endl;
s.print();

cout<<"\n添加10\n";
s.add(10);
cout<<"size:"<<s.size()<<endl;
s.print();

cout<<"\n删除10\n";
s.remove(10);
s.print();

cout<<"\n添加三个集合(一个重合)";
int m[] = {4,12,6};
s.addAll(m, 3);
cout<<"\nsize:"<<s.size()<<endl;
s.print();

s.removeAll();
cout<<"\nsize:"<<s.size()<<endl;
s.print();
*/

ISet<int>::iterator ite = s.Iterator();
while(s.hasNext()){
    cout<<s.next()->data<<" ";
}

s.remove(41);
```

```
        cout<<endl;
        ite = s.Iterator();
        while(s.hasNext()){
            cout<<s.next()->data<<" ";
        }

        s.add(10);

        cout<<endl;
        while(s.hasPre()){
            cout<<s.pre()->data<<" ";
        }

        cout<<endl;
        return 0;
    }
}
```

2.Map的简单实现

map也是利用二叉树实现

imap.h

```
//二叉排序树
#ifndef IMAP_H
#define IMAP_H

//数据域，里面存放结点数据
template<class K, class V>
struct Data{
    K key; //结点数据
```



```
        V value;
};

template<class K, class V>
struct Node{
    Data<K, V> data;
    Node<K, V> *rchild, *lchild;
    Node(){
        rchild = lchild = 0;
    }
};

template<class K, class V>
class IMap{
public:
    IMap();
    ~IMap();
    int size();
    bool isEmpty();
    bool containsKey(K key);
    bool containsValue(V value);
    V get(K key);
    void put(K key, V value); //若插入的值已经存在，则更新value
    V remove(K key);
    void clear();
    void print();
private:
    Node<K, V>* root;
    Node<K, V>* pre;//用于递归插入时，记录前序结点
    void print(Node<K, V>* root);
    void clear(Node<K, V>* &root);
    int size(Node<K, V>* root);
    void containsValue(Node<K, V>* root, V value, bool *b);
    void deleteNode(Node<K, V>* &root, Node<K, V>* r, Node<K, V>* pre);//删除结点p
};

#endif
```

imap.cpp

```
#include <iostream>
#include "imap.h"
using namespace std;

template<class K, class V>
IMap<K, V>::IMap(){
    pre = NULL;
    root = NULL;
}

template<class K, class V>
IMap<K, V>::~~IMap(){
    clear();
}

template<class K, class V>
int IMap<K, V>::size(){
    return size(root);
}

template<class K, class V>
bool IMap<K, V>::isEmpty(){
    if(!root) return true;
    return false;
}

template<class K, class V>
bool IMap<K, V>::containsKey(K key){
    Node<K, V>* r = root;
    while(r){
        if(r->data.key == key) break;
        r = (r->data.key > key)?r->lchild:r->rchild;
    }
}
```

```
//如果没找到，返回false
if(!r) return false;
return true;
}

template<class K, class V>
bool IMap<K, V>::containsValue(V value){
    bool b = false;
    containsValue(root, value, &b);
    return b;
}

template<class K, class V>
V IMap<K, V>::get(K key){
    Node<K, V>* r = root;
    while(r){
        if(r->data.key == key) break;
        r = (r->data.key > key)?r->lchild:r->rchild;
    }
    if(!r) return 0;
    return r->data.value;
}

template<class K, class V>
void IMap<K, V>::put(K key, V value){
    cout<<"插入<<key<<", "<<value<<">\n";
    Node<K, V>* r = root;
    if(r == NULL){
        Node<K, V>* t = new Node<K, V>;
        t->data.key = key;
        t->data.value = value;
        t->lchild = t->rchild = NULL;
        root = t;//必须设置新的root结点，因为t和root在内存指向位置不同
    }else{
        Node<K, V>* p = r;
        //查找待插入结点位置
        while(r){
```

```
        //如果已经存在，则更新value
        if(r->data.key == key){
            r->data.value = value;
            return;
        }
        p = r;
        r = (r->data.key > key)?r->lchild:r->rchild;
    }
    Node<K, V>* t = new Node<K, V>;
    t->data.key = key;
    t->data.value = value;
    t->lchild = t->rchild = NULL;
    if((p->data).key > key){//插入左
        p->lchild = t;
    }else{
        p->rchild = t;
    }
}

}

template<class K, class V>
V IMap<K, V>::remove(K key){
    if(!root) return 0;
    Node<K, V>* pre = NULL, *r = root;
    //查找待删除结点位置(pre是r的前序结点)
    while(r){
        if(r->data.key == key){
            break;
        }

        pre = r;
        r = (r->data.key > key)?r->lchild:r->rchild;
    }
    //没找到
    if(!r) return 0;
    //如果pre == NULL说明是root结点
    deleteNode(root, r, pre);
}
```

```
        return r->data.value;
    }

template<class K, class V>
void IMap<K, V>::clear(){
    clear(root);
    root = NULL;
}

template<class K, class V>
void IMap<K, V>::print(){
    print(root);
}

template<class K, class V>
void IMap<K, V>::print(Node<K, V>* root){
    if(root){
        cout<<root->data.value<<" ";
        print(root->lchild);
        print(root->rchild);
    }
}

template<class K, class V>
void IMap<K, V>::clear(Node<K, V>* &root){
    if(root){
        clear(root->lchild);
        clear(root->rchild);
        delete root;
    }
}

template<class K, class V>
int IMap<K, V>::size(Node<K, V>* root){
    if(!root) return 0;
    else{
        return size(root->lchild)+size(root->rchild)+1;
    }
}
```

```
    }
}

template<class K, class V>
void IMap<K, V>::containsValue(Node<K, V>* root, V value, bool *b){
    if(root){
        if(root->data.value == value){
            *b = true;
            return;
        }
        containsValue(root->lchild, value, b);
        containsValue(root->rchild, value, b);
    }
}

template<class K, class V>
void IMap<K, V>::deleteNode(Node<K, V>* &root, Node<K, V>* r, Node<K, V>* pre){
    Node<K, V>* p;
    if(!r->rchild && !r->lchild){                //如果是叶子结点
        if(pre){
            if(pre->lchild == r){
                pre->lchild = NULL;
            }else{
                pre->rchild = NULL;
            }
        }else{
            root = NULL;
        }
        delete r;
    }else if(r->rchild && r->lchild){            //如果左右子树都有(还有一种处理办法就是F
        p = r;
        //寻找右子树的最左结点
        r = r->rchild;
        while(r->lchild){
            r = r->lchild;
        }
        //将删除结点的左结点接到找到的最左结点之后
```

```
    r->lchild = p->lchild;
    //删除结点(如果pre是空,说明删除结点是根结点,不用改变前序结点指针)
    if(pre){
        if(pre->lchild == p) pre->lchild = p->rchild;
        else pre->rchild = p->rchild;
    }else{
        root = p->rchild;
    }
    delete p;
}
else if(r->lchild){ //如果只有左子树
    p = r;
    if(pre){
        if(pre->lchild == p) pre->lchild = r->lchild;
        else pre->rchild = r->lchild;
    }else{
        root = r->lchild;
    }
    delete p;
}
else{ //如果只有右子树
    p = r;
    if(pre){
        if(pre->lchild == p) pre->lchild = r->rchild;
        else pre->rchild = r->rchild;
    }else{
        root = r->rchild;
    }
    delete p;
}
}
```

main.cpp

```
#include <iostream>
#include "imap.cpp"
using namespace std;
```

```
int main(){
    IMap<int, int> map;
    cout<<"当前map : "<<map.isEmpty()<<endl;
    cout<<"当前map长度 : "<<map.size()<<endl;
    map.print();

    int key[] = {0,1,2,3,4,5,6,7,8,9};
    int value[] = {21,34,32,34,56,6,78,76,111,13};
    for(int i=0; i<10; i++){
        map.put(key[i], value[i]);
    }
    cout<<"当前map长度 : "<<map.size()<<endl;
    map.print();
    cout<<endl;

    bool b = map.containsKey(5);
    if(b) cout<<"包含键5 : "<<map.get(5)<<endl;
    else cout<<"不包含键5\n";

    b = map.containsKey(12);
    if(b) cout<<"包含键12 : "<<map.get(12)<<endl;
    else cout<<"不包含键12\n";

    b = map.containsValue(111);
    if(b) cout<<"包含值111 : "<<111<<endl;
    else cout<<"不包含值111\n";

    b = map.containsValue(211);
    if(b) cout<<"包含值211 : "<<211<<endl;
    else cout<<"不包含值211\n";

    cout<<"移除6\n";
    map.remove(6);
    cout<<"当前map长度 : "<<map.size()<<endl;
    map.print();
    cout<<endl;
```



```
    cout<<"移除90\n";  
    map.remove(90);  
    map.print();  
    cout<<endl;  
  
    return 0;  
}
```

4.24 二叉堆的实现

发表时间: 2012-08-12 关键字: 堆, 二叉树, 满二叉树

1.堆的概念

这里只需要注意两点：

a.堆的存储方式：就是顺序存储在数组中，在二叉树中表现为满二叉树

b.堆的用处：用于排序，查找最大最小都非常方便

2.堆的实现

heapexception.h

```
#ifndef HEAPEXCEPTION_H
#define HEAPEXCEPTION_H

class ArrayOverflowException{
public:
    const char* what() const throw()
    {
        return "array over flow exception!\n";
    }
};

class ParametersErrorException{
public:
    const char* what() const throw()
    {
        return "input error parameters exception!\n";
    }
};

class UnderFlowException{
```

```
public:
    const char* what() const throw()
    {
        return "the size out of array exception(position <= 0)!\n";
    }
};
#endif
```

heap.h

```
//二叉堆,是一颗被完全填满的二叉树,完全二叉树,故而可以使用顺序存储在数组中
#ifndef HEAP_H
#define HEAP_H

template<class T>
class BinaryHeap{
public:
    BinaryHeap( );
    BinaryHeap( const T* items, const int size);
    ~BinaryHeap();

    bool isEmpty() const;
    const T& findMin() const;

    void insert( const T x);//注意:插入过程中,如果满了,则要resize
    void deleteMin();
    void deleteMin( T& minItem);
    void makeEmpty();
private:
    int currentSize;//the number of elements in heap
    int capacity;
    void resize(int capacity);
    T* array;//the heap array
```

```
        void buildHeap();  
        void percolateDown(int hole); //将结点下滤，即从顶至叶子的调整过程  
};  
#endif
```

heap.cpp

```
#include <iostream>  
#include "heap.h"  
#include "heapexception.h"  
using namespace std;  
  
template<class T>  
BinaryHeap<T>::BinaryHeap(){  
    capacity = 100;  
    array = new T[capacity];  
    currentSize = 0;  
}  
  
template<class T>  
BinaryHeap<T>::BinaryHeap( const T* items, const int size){  
    if(size <= 0) throw ParametersErrorException();  
    currentSize = size;  
    capacity = size+10;  
    array = new T[capacity];  
    for(int i=0; i<size; i++){  
        //注意：这里是从1开始存储，主要是为了计算的方便，如1的左右孩子就是1*2,1*2+1,从0就不行  
        array[i+1] = items[i];  
    }  
    buildHeap();  
}  
  
template<class T>  
BinaryHeap<T>::~~BinaryHeap(){  
    delete[] array;
```

```
}

template<class T>
bool BinaryHeap<T>::isEmpty() const{
    if(currentSize == 0) return true;
    return false;
}

template<class T>
const T& BinaryHeap<T>::findMin() const{
    if(currentSize == 0) throw UnderFlowException();
    return array[1];
}

template<class T>
void BinaryHeap<T>::insert( const T x){//注意：插入过程中，如果满了，则要resize
    if(currentSize == capacity-1){
        resize(capacity);
    }
    //在最后建立一个空的位置，如果满足条件则上浮
    int hole = ++currentSize;
    for(; hole>1 && x < array[hole/2]; hole/=2){
        array[hole] = array[hole/2];
    }
    array[hole] = x;
}

template<class T>
void BinaryHeap<T>::deleteMin(){
    if(currentSize == 0) throw UnderFlowException();
    //将当前最新的交换到最后，在进行一次下滤
    array[1] = array[currentSize--];
    percolateDown(1);
}

template<class T>
```

```
void BinaryHeap<T>::deleteMin( T& minItem){
    if(currentSize == 0) throw UnderFlowException();
    minItem = array[1];
    array[1] = array[currentSize--];
    percolateDown(1);
}

template<class T>
void BinaryHeap<T>::makeEmpty(){
    currentSize = 0;
}

template<class T>
void BinaryHeap<T>::buildHeap(){
    cout<<"build heap\n";
    //建堆的过程就是从低到顶的将结点下滤
    for(int i=currentSize/2; i>0; i--){
        percolateDown(i);
    }
}

//将结点下滤，即从顶至叶子的调整过程(小根堆)
template<class T>
void BinaryHeap<T>::percolateDown(int hole){
    int child;
    T tmp = array[hole];
    for(; hole*2 <= currentSize; hole = child){
        child = hole*2;
        if(child != currentSize && array[child+1] < array[child]){//右子树
            child++;
        }
        //如果当前比hole小，交换
        if(array[child]<tmp){
            array[hole] = array[child];
        }else{
            break;
        }
    }
}
```

```
    }  
    //hole是最后交换的位置  
    array[hole] = tmp;  
}  
  
template<class T>  
void BinaryHeap<T>::resize(int capacity){  
    T* t = array;  
    capacity = currentSize*2 + 1;  
    array = new T[capacity];  
    for(int i=0; i<currentSize; i++){  
        array[i+1] = t[i];  
    }  
    delete t;  
}
```

main.cpp

```
#include <iostream>  
#include "heapexception.h"  
#include "heap.cpp"  
using namespace std;  
  
int main(){  
    int a[] = {59, 48, 75, 98, 86, 23, 37, 59};  
    try{  
        BinaryHeap<int> bh(a, 8);  
        int min;  
        cout<<"-----\n";  
        cout<<bh.isEmpty()<<endl;  
        cout<<"min:"<<bh.findMin()<<endl;  
        bh.deleteMin(min);  
        cout<<"min:"<<bh.findMin()<<endl;  
  
        cout<<"insert"<<endl;
```

```
        bh.insert(10);
        cout<<"min:"<<bh.findMin()<<endl;
        bh.makeEmpty();
        bh.findMin();
    }catch(ArrayOverFlowException e){
        cout<<e.what()<<endl;
    }catch(ParametersErrorException e){
        cout<<e.what()<<endl;
    }catch(UnderFlowException e){
        cout<<e.what()<<endl;
    }

    return 0;
}
```


4.25 KMP算法解析

发表时间: 2012-08-12 关键字: KMP, next分析

一.理论基础

1.什么是kmp算法

同BF算法一样，就是串的模式匹配算法。

前面已经学过，我想都应该明白BF算法，就是用一种最直观的方式进行模式匹配。

优点：非常容易理解，是我们常用的思维方式来编程；

缺点：效率比较低，在匹配不成功的时候，回朔做了许多无用功；

从而根据其缺点，KMP算法就在回朔的时候做了工作，减少其无用功，那么怎么去减少回朔的工作呢？

下面举例说明：

例如：

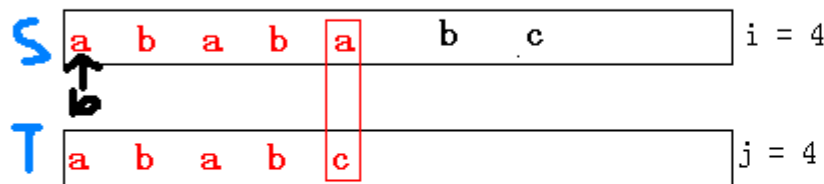
s = abababc

t = ababc

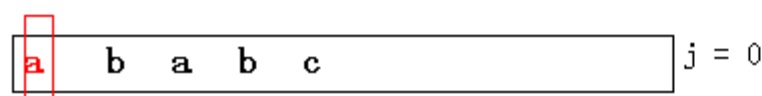
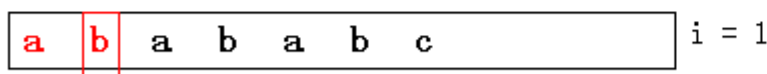
其匹配过程如下图：

S: a b a b a b c
T: a b a b c

1. 第一次匹配失败情况:



↓ 回溯



这个就是匹配失败后的BF处理办法
将i重新设置上次开始位置的下一个位置 $i++$
将j置为0

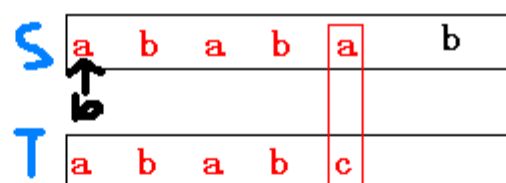
总结:

从这个过程我们能看出什么呢?

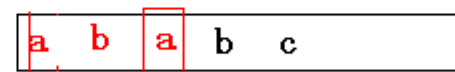
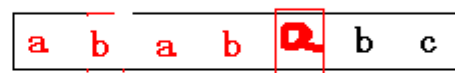
首先, 回溯就完全重新开始了S曾经遍历过的, 这个是其重大的缺陷, 那我们该怎么上次遍历过的元素呢? 好, 这个有点像动态规划, 记住上次遍历的特点, 动态选择T的回溯到0, 就像人一样, 虽然这次尝试失败了, 而是可以在路上站起来一样!

其次, 针对右边比较过的S, 红色为遍历过的时候S是a, 而前面S的部分中我们看出i=2, 位置是不是一样呢, 也是ab, 故而我们就不接从j=2位置开始继续比较! 这样就减少了是因为我们记住了其遍历过的特点, 然后T

改造过后的匹配情况:



↓ 回溯



看了对照情况, 是不是感觉效率相差是很大的, 我想很明显的, 提高了很大的效率
如果S的长度为m, T的长度为n, 则:
BF: $O(m*n)$
KMP: $O(m+n)$

从图中我们看到了具体的变换过程, 就是在回溯的过程中做了许多工作, S的i不需要回溯到1, 只需要T回溯就可以了, 那么怎么知道T回溯到什么位置才好呢? 而从上面的图中我们也会发现, 其实回溯跟S串是没有关系的, i压根就没变, 变的是T的j, 故而需要回溯的是T--待匹配模式串, 所有回溯的位置是由T串的特点决定的!! 当然, 这就是问题2, 求next数组。

2. 为什么要求next数组

什么是next数组, 简而言之, 就是当不匹配的时候, 从当前j的位置需要回溯的位置k

如上面的图中，当 $j=4$ 的时候，回朔的位置是2，故而 $next[4]=2$ ，这就是 $next$ 的价值所在：决定在位置 j 不匹配的时候，回退到位置 $next[j]$ 。

3.next如何求之理论基础

既然求 $next$ 完全是T的事，那么我这里就不像那些书上那样从S去分析，直接切入正题从T分析！书上有时候说的反而有碍思维，要是完全照书，我就不写这些了！

首先实例分析：

T: a b a b c 求next数组?

求T的next数组, 实质就是一个内部的模式匹配问题!

1. 一个递归定义问题

$$t_0 t_1 \cdots t_{k-1} = t_{j-k+1} t_{j-k+2} \cdots t_{j-1}$$

如果在T中存在这样特点的子串存在, 则有 $\text{next}[j]=k$
且:

1. 如果有多个满足条件, k是最大的
2. $1 < k < j$

2. next函数定义

$$\text{next}(j) = \begin{cases} -1 & (j=0) \\ \max\{k \mid 1 < k < j, t_0 t_1 \cdots t_{k-1} = t_{j-k+1} t_{j-k+2} \cdots t_{j-1}\} & \text{其它} \end{cases}$$

3. 结合实例理解

这里先以上面作为例子:

j: 0 1 2 4 5
T: a b a b c



j=0: next[0]=-1;

j: 0 1 2 4 5
T: a b a b c



j=1: next[1]=0
属于其他情况

j: 0 1 2 4
T: a b a b



j=2: next[2]=0
 $t_0 \neq t_1$, 属于其他

j: 0 1 2 4 5
T: a b a b c



j=3: next[3]=1
存在 $t_0 = t_2$ (属于2)
 $k-1=0 \rightarrow k=1$

j: 0 1 2 4 5
T: a b a b c



j=4: next[4]=2
存在 $t_0 t_1 = t_2 t_3$
 $k-1=1 \rightarrow k=2$

故而得出: $\text{next}[] = \{-1, 0, 0, 1, 2\}$; 我们直观的来看, next[3]=1, 则当j=3不匹配的时候, j回

4. 分析的重点

$$t_0 t_1 \cdots t_{k-1} = t_{j-k+1} t_{j-k+2} \cdots t_{j-1}$$

上面的实例中, 从图3, 4都是根据它得出的
要看明白他的特点, 已经为什么要这样做?

1. 当前正在匹配的位置是j, 且j无法匹配
2. 查找的是, 从t开始位置到小于j的某个位置k-1 ($k < j$) 的串a, 和以j-1为结束向前长度为k的串b, 且串a, b是相同的!
3. 根据2, 我们就得出了 $\text{next}[j]=k$
4. 为什么找找这样的两个串? 就是为了找回退的位置k, 如abababc, 当到j=4的时候不匹配了, 但是找到了串abababc, $t_0 t_1 = t_2 t_3$ ($k=2, j=4$) 是符合上述要求的串, 从而k=2. 直观的看就是, 既然先前已经匹配了有abab, 那么我就不用从j=0在开始了, 直接从j=2开始, 因为j=2开始也是ab, 和j=0开始是一样的ab, 所以就减少了匹配次数。

最后, 在实例

2. 代码

```
#include <iostream>
using namespace std;

int strLen(const char *s){
    if(!s) throw "串不能为空!\n";
    int i=0;
    while(*s != '\0'){
        i++;
        s++;
    }
    return i;
}

/**
 *求next数组值
 *求模式t的next值并存入next数组中
 *t :待求模式数组
 *n :模式数组长度
 *next : next数组
 */
void getNext(const char* t, int n, int* next){
    int i=0, j=-1;
    next[0] = -1;
    while(i < n){
        //如果j==-1, 则回退到了开始位置(只有当开始的时候j才为-1, only one,以后至少都是0)
        //如果相等, 匹配, 故而向前, 此时i的next的值就是j, 且存的是最大的j
        if(j == -1 || t[j] == t[i]){
            i++;
            j++;
            next[i]=j;
        }else{//如果不等或j为其他, 则j回到next[j]位置继续匹配
            j = next[j];
        }
    }
}
```

```
        }  
    }  
}  
  
//1.Brute-Force算法(BF算法)  
//子串定位 (p若属于s, 返回串p在s中的位置, 否则返回0)  
int strIndex(const char* s, const char* p){  
    if(!s || !p) throw "串不能为空!\n";  
    int i=0, j=0, r = 1;  
    int sl = strlen(s);  
    int pl = strlen(p);  
    if(sl < pl) throw "参数异常, 串长度!\n";  
    while(i<sl && j<pl){  
        if(s[i] == p[j]){  
            i++; j++;  
        }else{//恢复开始的位置的下一个位置  
            i = i-j+1;  
            j = 0;  
        }  
    }  
    if(j >= pl) r = i-pl+1;  
    else r = -1;  
    return r;  
}
```

//2.KMP算法

//具体的讲解, 见博文

```
int strIndex_kmp(const char* s, const char* p){  
    if(!s || !p) throw "串不能为空!\n";  
    int i=0, j=0, r = 1;  
    int sl = strlen(s);  
    int pl = strlen(p);  
    if(sl < pl) throw "参数异常, 串长度!\n";  
  
    //求next  
    int *next = new int[pl];
```

```
getNext(p, pl, next);
cout<<"next数组:";
for(int j=0; j<pl; j++){
    cout<<next[j]<<" ";
}
cout<<endl;

while(i<sl && j<pl){
    if(j==-1 || s[i] == p[j]){
        i++; j++;
    }else{//恢复到next[j]
        j = next[j];
    }
}
if(j >= pl) r = i-pl+1;
else r = -1;
delete next;
return r;
}

int main(){
    //char a[] = {'a','c','a','b','a','a','b','a','a','b','c','a','c','a','a','b','c','\0'};
    //char b[] = {'a','b','a','a','b','c','\0'};
    char a[] = {'a','b','a','b','a','b','c','\0'};
    char b[] = {'a','b','a','b','c','\0'};
    try{
        cout<<"BF算法："<<strIndex(a ,b)<<endl;
        cout<<"KMP算法："<<strIndex_kmp(a, b)<<endl;

    }catch(const char* s){
        cout<<s<<endl;
    }
}
```


4.26 常见字符串操作大全

发表时间: 2012-08-12 关键字: 字符串操作

1.常见的字符串操作

如计算长度，求子串等都是考验基本功的，现在基本操作进行实现，如下

2.基本实现

mchar.h

```
//串的基本操作(注意：里面的所有操作要求串必须以'\0'结束)

#ifndef MCHAR_H
#define MCHAR_H

//注意：这里的i都是位置，从1开始；而数组中对应于0，从0开始

class MChar{
public:
    int strLen(const char *s);                //串长度(在输入串的时候，必须以'\0'结束)
    char* strCpy(char* s,const char* p);    //串拷贝,p拷贝给s，原来的值被覆盖
    char* strCat(char* s, const char* p);    //串连接，将p连接到s后面
    char* subStr(char* s, int i, int len);    //求从第i个位置开始的长度为len的子串
    int strCmp(const char* s, const char* p);                //串比较
    int strIndex(const char* s, const char* p);              //子串定位 (p若属于s)
    void strInsert(char* s, int i, const char* p);           //将串p插入到串s的第i个字符开始的位
    void strDelete(char* s, int i, int len);                 //删除串s中从第i个字符开始的长度为len的子串
    int findStr(const char* s, int ch);                      //查找为ch的字符
    const char *toUpper(char *s);                           //将所有小写字母转换为大写字母
    void itoaTest(int num, char str[]);                      //整数转换为字符串
    int atoiTest(char s[]);                                  //字符串转换为整数
    bool isPalindrome(char *s);                             //判断是否为回文串
    void printStr(char* s);

};
```

```
#endif
```

mchar.cpp

```
#include <iostream>
#include <cstring>
#include <assert.h>
#include "mchar.h"
using namespace std;

//串长度(在输入串的时候,必须以'\0'结束)
int MChar::strLen(const char *s){
    if(!s) throw "串不能为空!\n";
    int i=0;
    while(*s != '\0'){
        i++;
        s++;
    }
    return i;
}

//串拷贝,p拷贝给s,原来的值被覆盖
char* MChar::strCpy(char* s,const char* p){
    if(!s || !p) throw "串不能为空!\n";
    char *address = s;
    //改进1,简洁
    //while(*p != '\0') *s++ = *p++;
    while((*s++ = *p++) != '\0');
    //改进2:去掉它*s = '\0';
    //改进3:增加返回值,返回最新的s地址,方便链式操作
    return address;
}
```

//串连接，将p连接到s后面

```
char* MChar::strCat(char* s, const char* p){
    if(!s || !p) throw "串不能为空!\n";
    while(*s != '\0') *s++;
    while(*p != '\0') *s++ = *p++;
    *s = '\0';
    return s;
}
```

//求从第i个位置开始的长度为len的子串

```
char* MChar::subStr(char* s, int i, int len){
    if(!s) throw "串不能为空!\n";
    int l = strLen(s);
    if(i<1 || i>l || i+len-2>l) throw "参数越界异常!\n";
    int j=1;
    char* t = s;
    while(j != i && *s != '\0'){
        j++;
        *s++;
    }
    while(*s != '\0' && len > 0)
    {
        len--;
        *t++ = *s++;
    }
    *t = '\0';
    return t;
}
```

//串比较 (s==p, return=0; s>p, return 1; else -1)

```
int MChar::strCmp(const char* s, const char* p){
    if(!s || !p) throw "串不能为空!\n";
    while(*s != '\0'){
        if(*s++ == *p++) continue;
        else if(*s++ > *p++) return 1;
        else return -1;
    }
}
```

```
    if(*s == '\\0' && *p == '\\0') return 0;
    else return -1;
}
```

//将串p插入到串s的第i个字符开始的位置上,假设s的长度足够长,不会溢出

```
void MChar::strInsert(char* s, int i, const char* p){
    if(!s && !p) throw "串不能为空!\\n";
    int len = strLen(s);
    int len2 = strLen(p);
    if(i<1 || i>len+1) throw "参数越界异常!\\n";
    int j=0;
    //不是j=len-1开始,因为要把'\\0'也移动到后面
    for(j=len; j>=i-1; j--){
        s[len2+j] = s[j];
    }
    for(j=0; j<len2; j++){
        s[i+j-1] = p[j];
    }
}
```

//删除串s中从第i个字符开始的长度为len的子串

```
void MChar::strDelete(char* s, int i, int len){
    if(!s) throw "串不能为空!\\n";
    int l = strLen(s);
    if(i<1 || i>l || i+len-1> l) throw "参数越界异常!\\n";
    for(int j=i+len-1; j<=l; j++){
        s[j-len] = s[j];
    }
}
```

//查找为ch的字符

```
int MChar::findStr(const char* s, int ch){
    if(!s) throw "串不能为空!\\n";
    int i=1;
    while(*s != '\\0'){
        if(*s == (char)ch) return i;
        s++;
    }
}
```

```
        i++;
    }
    return 0;
}

//子串定位 (p若属于s, 返回串p在s中的位置, 否则返回0)
int MChar::strIndex(const char* s, const char* p){
    if(!s || !p) throw "串不能为空!\n";
    int i=0, j=0, r = 1;
    int sl = strLen(s);
    int pl = strLen(p);
    if(sl < pl) throw "参数异常, 串长度!\n";
    while(i<sl && j<pl){
        if(s[i] == p[j]){
            i++; j++;
        }else{//恢复开始的位置的下一个位置
            i = i-j+1;
            j = 0;
        }
    }
    if(j >= pl) r = i-pl+1;
    else r = -1;
    return r;
}

void MChar::printStr(char* s){
    if(!s) throw "串不能为空!\n";
    while(*s != '\0') cout<<*s++<<" ";
    cout<<endl;
}

const char* MChar::toUpper(char *s){
    if(!s) throw "串不能为空!\n";
    int l = 'a' - 'A';
    cout<<l<<endl;
    while(*s != '\0'){
        if(*s>= 'a' && *s<='z') *s = *s - l;
    }
}
```

```
        s++;
    }
    return s;
}

//整数转换为字符串
void MChar::itoaTest(int num, char str[]){
    int n = num, i, k;
    char tmp[20]; //假设不会发生溢出
    if(num < 0) n = -n;
    for(k=0; n>0; k++){
        i = n % 10;
        tmp[k] = i + '0'; // '0'-->48, '0'+1==49-->'1'
        n = n / 10;
    }
    //反向
    if(num < 0) tmp[k] = '-';
    else k--;
    for(i=0; k >= 0; i++){
        str[i] = tmp[k--];
    }
    str[i] = '\0';
}

//字符串转换成整数atoi函数(关键是注意，特殊字符的判断，前面的+-等)
/**注意四点：
    *1.整数以谁开头， '+' , '-' 或者空格，要判断；
    *2.如果输入字符串是指针，首先判断是否为空；
    *3.输入字符串可能含非数字字符，如果含果断结束；
    *4.最后得到的整数特别大，可能溢出
    */
int MChar::atoiTest(char s[]){
    int sign = 1, tmp = 0, i = 0;
    //判断前面是否有空格，回车等
    while(' '==s[i]||'\t'==s[i]) i++;
    //符号判断
    if(s[i]=='-'){
```

```
        i++;
        sign = -1;
    }else if(s[i]=='+'){
        i++;
        sign = 1;
    }
    while(s[i] != '\0'){
        //对于非数字，直接结束
        if(s[i] < '0' || s[i] > '9') throw "含有非数字字符!";
//或者使用断言代替throw，不过throw更好，他能提供具体原因，这个可以自己指定，方便找到原因
//assert(s[i] >= '0'); assert(s[i]<= '9');
        tmp = tmp*10 + s[i]-'0';
        //处理溢出
        if(tmp < 0) throw "整数过大溢出!";
//assert(tmp >= 0);
        i++;
    }
    return sign*tmp;
}

//判断输入的是一个回文字符串
bool MChar::isPalindrome(char *s){
    char input[100];
    strcpy(input, s);
    int length = strlen(input);
    int begin = 0,end = length-1;
    while(begin<end)
    {
        if(s[begin]==s[end]){
            begin++;
            end--;
        }else{
            break;
        }
    }
    if(begin<end) return false;
```

```
    else return true;
}
```

main.cpp

```
#include <iostream>
#include <cstring>
#include "mchar.h"
using namespace std;

int main(){
    MChar mc;
    char a[] = {'h','e','l','l','o','j','l','l','k','\0','o','o','o','o','o','o','o','o','o','o','c'};
    char b[] = {'g','m','\0'};
    char c[] = {'l','L','\0','t','K','\0'};

    try{
        /*
        mc.printStr(a);
        cout<<"串长度："<<mc.strLen(a)<<endl;
        cout<<"串拷贝："
        char* s = new char[6];
        mc.strCpy(s, c);
        mc.printStr(s);
        delete s;

        cout<<"\n串连接："
        mc.strCat(a, b);
        mc.printStr(a);

        cout<<"\n子串截取："
        char* m = mc.subStr(a, 1, 4);
        if(m) mc.printStr(a);

        cout<<"\n串比较："
```



```
cout<<mc.strCmp(c, a)<<endl;

cout<<"串插入："；
mc.strInsert(a, 7, b);
mc.printStr(a);

cout<<"串删除："；
mc.strDelete(a, 5, 7);
mc.printStr(a);

cout<<(int)'e'<<endl;
cout<<"查找字符："；
cout<<mc.findStr(a, 101)<<endl;

cout<<"子串位置："；
cout<<mc.strIndex(a, c)<<endl;

cout<<"大写转换："；
mc.toUpper(c);
mc.printStr(c);

cout<<"整数转换为字符串："；
char* t = new char[20];
mc.itoaTest(324344445, t);
mc.printStr(t);
cout<<(char)(3+'0')<<" "<<(int)'0'<<endl;//输出3 48

cout<<"字符串转换为整数："；
char t[] = {' ', '-', '3', '4', '5', '2', '9', '\\0'};
cout<<mc.atoiTest(t)<<endl;
cout<<'3'-'0'<<endl;
*/

cout<<"回文字符串判读："；
```

```
        char s[20] ="1234554321";
        if(mc.isPalindrome(s))
        {
            cout<<"True"<<endl;
        }else{
            cout<<"Fasle"<<endl;
        }
    }catch(char const *s){
        cout<<s<<endl;
    }

    return 0;
}
```

3.几个操作

```
#include <iostream>
#include <cstring>
using namespace std;

bool isDigit(const char ch){
    if(ch >= '0' && ch <= '9') return true;
    return false;
}

//计算连续数字的个数，如ak123x5612393837?3043gef435,将其存入数组a中a[] = {123,56,123,43,43}
int countInt(const string s, string* result){
    int count = 0;
    int i=0,temp=11;
    string str = "";
    const char* c = s.c_str();
    while(i < s.size()){
        char a = c[i];
        //-----循环处理连续数字字符串-----
        //判读是不是数字，如果是数字则要循环判段是否顺序递增或递减
        //转换为char数组
```

```
while(isDigit(a)){
    int k = a-'0';
    if(str.size() == 0){
        str.append(sizeof(char),a);
    }else if(k == temp+1 || k == temp-1){ //2.如果不是第一个,而且是下一个数
        str.append(sizeof(char),a);
    }else if(str.size() == 1){
        //restart again
        str = "";
        str.append(sizeof(char),a);
    }else{
        *result++ = str;
        count++;
        //restart again
        str = "";
        str.append(sizeof(char),a);
    }
    a = c[++i];
    if(!isDigit(a)){
        if(str.size() > 1){
            *result++ = str;
            count++;
            str = "";
            temp = 11;
        }
        break;
    }
    temp = k;
}
i++;
}
return count;
}
```

//求最长重复子串

```
int longestString(const string str){
    const char* s = str.c_str();
```

```
int n = str.size();
int index = 0, max = 0;
int length = 1, i=0,start=0;
while(i<n-1){
    if(s[i] == s[i+1]){
        length++;
    }else{
        if(max < length){
            max = length;
            index = start;
        }
        length = 1;
        start = i;
    }
    i++;
}
cout<<"最长子串长度为："<<max<<"，位置："<<index+1<<endl;
return max;
}

bool isSign(const char ch){
    if(ch == ',' || ch == '.' || ch == '?' || ch == '!' || ch == '"' || ch == '\'' || ch ==
return false;
}

bool isSpicial(const char ch){
    if((ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z') || (ch >= '0' && ch <= '9')) re
return true;
}

// (文章的处理) 将字符串中的特殊字符用空格代替 (假设现在只含有, . ? ' ! - ) , 若是多个空格, 则用一个空格代替
void clearString(string str, char* result){
    const char* s = str.c_str();
    int len = str.size(), temp = 0;
    for(int i=0; i<len; i++){
        if(!isSpicial(s[i])){ //如果是字母或数字不处理
            *result++ = s[i];
        }
    }
}
```

```
        }else{
            //如果是特殊字符则跳过
            while(i<len && isSpicial(s[i])){
                i++;
            }
            //特殊字符以空格取代
            *result++ = ' ';
            i--; //减去一个，因为for循环开始循环之前要i++
        }
    }
}
```

```
int main(){
    /*
    string s = "ak123x5612d393837?3043gef435";
    string* a = new string[s.size()];
    int count = countInt(s,a);
    cout<<count<<endl;
    for(int i=0; i<count; i++){
        cout<<a[i]<<" ";
    }
    cout<<endl;
    delete[] a;

    //最长子串
    string m = "4444fdasfdrekkkkkkkkfdeeesfs";
    longestString(m);
    */
    string k = "What is it? she asked   joyfully. Scarlett! 43 44  55ffg  She was:  Bitterl";
    char* result = new char[k.size()];
    clearString(k, result);
    for(int i=0; i<k.size(); i++){
        cout<<result[i];
    }
    cout<<endl;
    delete[] result;
```

```
    return 0;  
}
```

4.27 排序方法总结

发表时间: 2012-08-12 关键字: 排序

这里面包含了所有常见的排序操作

1.性能等比较分析

排序方法	时间复杂度			空间复杂度
	平均情况	最坏情况	最好情况	
直接插入排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$
希尔排序	$O(n \log n)$ 到 $O(n^2)$	$O(n^2)$	$O(n^{\frac{1}{3}})$	$O(1)$
冒泡排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$
快速排序	$O(n \log n)$	$O(n^2)$	$O(n \log n)$	$O(\log n)$ 到 $O(n)$
直接选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
堆排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$
归并排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$
基数排序	$O(d(n+rd))$	$O(d(n+rd))$	$O(d(n+rd))$	$O(rd)$

排序方法	最好情况	最坏情况	平均情况
直接插入排序	$O(n)$	$O(n^2)$	$O(n^2)$
起泡排序 0	$O(n^2)$	$O(n^2)$	
简单选择排序	0	$O(n)$	$O(n)$

2.代码实现

sort.h

```
//各种排序方法总结

#ifndef SORT_H
#define SORT_H

template<class T>
class Sort{
public:
    void insertSort(T r[], int n);           //直接顺序排序
    void shellSort(T r[], int n);           //希尔排序
    void bubbleSort(T r[], int n);          //起泡排序

    void quickSort(T r[], int first, int end); //快速排序
    void selectSort(T r[ ], int n);          //简单选择排序

    void heapSort(T r[ ], int n);            //堆排序

    void mergeSort1(T r[ ], T r1[ ], int n ); //归并排序的非递归算法
    void mergeSort2(T r[], T r1[], T r2[],int s, int t); //归并排序的递归算法
private:
    int partition(T r[], int first, int end); //快速排序一次划分
    void sift(T r[], int n, int hole);        //筛选法调整堆
    void merge(T r[], T r1[], int s, int m, int t); //一次归并
    void mergePass(T r[ ], T r1[ ], int n, int h); //一趟归并
};
#endif
```

sort.cpp

```
#include <iostream>
#include "sort.h"
using namespace std;

//直接顺序排序
```



```
template<class T>
void Sort<T>::insertSort(T r[], int n){
    if(n <= 0) return;
    int i, j;
    for(i=2; i<n; i++){
        r[0] = r[i];
        j = i-1;
        while(r[0]<r[j]){
            r[j+1] = r[j];
            j--;
        }
        r[j+1] = r[0];
    }
}

//希尔排序
template<class T>
void Sort<T>::shellSort(T r[], int n){
    if(n <= 0) return;
    int i,j,d;
    //增量d不断变化
    for(int d=n/2; d>0; d = d/2){
        //对于每个增量，都是直接插入排序
        for(i = d+1; i<n; i+=d){
            r[0] = r[i];
            j = i-d;
            while(r[0]<r[j] && j>0){
                r[j+d] = r[j];
                j-=d;
            }
            r[j+d] = r[0];
        }
    }
}

//起泡排序
template<class T>
```

```
void Sort<T>::bubbleSort(T r[], int n){
    T tmp = 0; //用于交换
    int exchanged = 1; //记录是否发生交换
    int bound, pos = n-1; //最后一次交换位置
    while(exchanged){
        bound = pos;
        exchanged = 0;
        for(int j=0; j<bound; j++){
            if(r[j]>r[j+1]){
                tmp = r[j];
                r[j] = r[j+1];
                r[j+1] = tmp;
                pos = j;
                exchanged = 1;
            }
        }
    }
}
```

//快速排序一次划分

```
template<class T>
int Sort<T>::partition(T r[], int first, int end){
    T tmp;
    int i=first, j=end;
    while(i<j){
        while(r[i]<r[j] && i<j) j--;
        if(i<j){
            tmp = r[i];
            r[i] = r[j];
            r[j] = tmp;
            i++;
        }
        while(r[i]<r[j] && i<j) i++;
        if(i<j){
            tmp = r[i];
            r[i] = r[j];
            r[j] = tmp;
        }
    }
}
```

```
        j--;
    }
}
return i;
}

//快速排序
template<class T>
void Sort<T>::quickSort(T r[], int first, int end){
    int k;
    if(first < end){
        k = partition(r, first, end);
        //对左右递归排序
        quickSort(r, first, k-1);
        quickSort(r, k+1, end);
    }
}

//简单选择排序
template<class T>
void Sort<T>::selectSort(T r[ ], int n){
    T tmp;
    int i, pos;
    for(i=0; i<n; i++){
        pos = i;
        for(int j=i+1; j<n; j++){
            if(r[j] < r[pos]) pos = j;
        }
        if(pos != i){
            tmp = r[pos];
            r[pos] = r[i];
            r[i] = tmp;
        }
    }
}

//筛选法调整堆
```

```
template<class T>
void Sort<T>::sift(T r[], int n, int hole){
    int child;
    T tmp = r[hole];
    for(; hole*2 <= n; hole = child){
        child = hole*2;
        if(child != n && r[child+1] < r[child]){//右子树
            child++;
        }
        //如果当前比hole小, 交换
        if(r[child]<tmp){
            r[hole] = r[child];
        }else{
            break;
        }
    }
    //hole是最后交换的位置
    r[hole] = tmp;
}

//堆排序
template<class T>
void Sort<T>::heapSort(T r[ ], int n){
    for(int i=n/2; i>0; i--){
        sift(r, n, i);
    }
    //重复移除第一个元素, 并重建堆
    while(n>0){
        r[0] = r[1];
        r[1] = r[n];
        r[n] = r[0];
        n--;
        sift(r, n, 1);
    }
}

/*
```

```
//这个写的比较容易理解，非常好！！From v_july_v
template<class T>
void Sort<T>::sift(T heap[], int i, int len)
{
    int min_index = -1;
    int left = 2 * i;
    int right = 2 * i + 1;
    T tmp;

    //反正就是找其左右子树小的，然后递归
    if (left <= len && heap[left] < heap[i])
        min_index = left;
    else
        min_index = i;

    if (right <= len && heap[right] < heap[min_index])
        min_index = right;

    if (min_index != i)
    {
        // 交换结点元素
        tmp = heap[i];
        heap[i] = heap[min_index];
        heap[min_index] = tmp;

        sift(heap, min_index, len);
    }
}

// 建立小根堆
template<class T>
void Sort<T>::buildHeap(T heap[], int len)
{
    if (heap == NULL)
        return;

    int index = len / 2;
```

```
    for (int i = index; i >= 1; i--)
        sift(heap, i, len);
}
*/

//一次归并(有序序列r(开始位置分别为s,m的两个序列),将合并结果放入r1)
template<class T>
void Sort<T>::merge(T r[], T r1[], int s, int m, int t){
    int i=s;
    int j=m+1;
    int k=s;
    while (i<=m && j<=t)
    {
        if (r[i]<=r[j])
            r1[k++]=r[i++];           //取r[i]和r[j]中较小者放入r1[k]
        else
            r1[k++]=r[j++];
    }
    if (i<=m)
        while (i<=m)                 //若第一个子序列没处理完,则进行收尾处理
            r1[k++]=r[i++];
    else
        while (j<=t)                 //若第二个子序列没处理完,则进行收尾处理
            r1[k++]=r[j++];
}

//一趟归并
template<class T>
void Sort<T>::mergePass(T r[ ], T r1[ ], int n, int h){
    int i=0;
    int k;
    while (i<=n-2*h)                 //待归并记录至少有两个长度为h的子序列
    {
        merge(r, r1, i, i+h-1, i+2*h-1);
        i+=2*h;
    }
    if (i<n-h)
```

```
        merge(r, r1, i, i+h-1, n);           //待归并序列中有一个长度小于h
    else for (k=i; k<=n; k++)                //待归并序列中只剩一个子序列
        r1[k]=r[k];
}
```

//归并排序的非递归算法

```
template<class T>
void Sort<T>::mergeSort1(T r[ ], T r1[ ], int n ){
    int h=1;
    int i;

    while (h<n)
    {
        mergePass(r, r1, n-1, h);           //归并
        h=2*h;
        mergePass(r1, r, n-1, h);
        h=2*h;
    }
    for(i=0;i<n;i++)
        cout<<r[i]<<" ";
    cout<<"\n";
}
```

//归并排序的递归算法

```
template<class T>
void Sort<T>::mergeSort2(T r[], T r1[], T r2[],int s, int t){
    int m;
    if (s==t)
    {
        r1[s]=r[s];
    }
    else
    {
        m=(s+t)/2;
        mergeSort2(r, r2, r1, s, m);       //归并排序前半个子序列
        mergeSort2(r, r2, r1, m+1, t);     //归并排序后半个子序列
    }
}
```

```
        merge(r2, r1, s, m, t);           //将两个已排序的子序列归并
    }
}
```

main.cpp

```
#include <iostream>
#include "sort.cpp"
using namespace std;

int main(){
    const int numv=11;           //赋值
    int a[]={0,3,56,32,78,5,24,9,64,34,7};           //插入排
    int b[]={0,4,6,23,45,15,10,36,25,79,21};           //希尔排序的时候
    int c[]={38,23,56,2,79,42,93,29,6,5,57};
    int d[]={50,23,45,67,87,14,29,32,44,97,89};
    int e[]={8,6,1,48,37,63,39,74,52,26,49};
    int f[]={0,12,23,45,87,2,6,15,43,26,40};           //堆的第一个位置
    int g[]={13,10,23,45,64,34,24,7,9,3,16};
    int h[]={34,23,54,76,12,13,14,11,78,8,9};
    int g1[numv];
    int h1[numv];
    int h2[numv];

    Sort<int> s;
    int j;
    /*
    cout << "\n直接顺序排序前：" << "\n";
    for(int j=1;j<numv;j++)
        cout<<a[j]<<" ";
    cout << "\n直接顺序排序结果为：" << "\n";
    s.insertSort(a,numv);
    for(int j=1;j<numv;j++)
        cout<<a[j]<<" ";
    */
}
```



```
cout << "\n希尔排序前：" << "\n";
for(j=1;j<numv;j++)
    cout<<b[j]<<" ";
cout << "\n希尔排序结果为：" << "\n";
s.shellSort(b, numv);
for(j=1;j<numv;j++)
    cout<<b[j]<<" ";

cout << "\n起泡排序前：" << "\n";
for(int k=0;k<numv;k++)
    cout<<c[k]<<" ";
cout << "\n起泡排序结果为：" << "\n";
s.bubbleSort(c, numv);
for(int k=0;k<numv;k++)
    cout<<c[k]<<" ";

cout << "\n快速排序前：" << "\n";
for(j=0;j<numv;j++)
    cout<<d[j]<<" ";
cout << "\n快速排序结果为：" << "\n";
s.quickSort(d,0,numv-1);
for(int i=0;i<numv;i++)
    cout<<d[i]<<" ";
cout<<"\n";

cout << "\n简单选择排序前：" << "\n";
for(j=0;j<numv;j++)
    cout<<e[j]<<" ";
cout << "\n简单选择排序结果为：" << "\n";
s.selectSort(e,numv);
for(j=0;j<numv;j++)
    cout<<e[j]<<" ";
```

```
cout << "\n堆排序前：" << "\n";
for(j=0;j<numv;j++)
    cout<<f[j]<<" ";
cout << "\n堆排序结果为：" << "\n";
s.heapSort(f, numv);
for(j=1;j<numv;j++)
    cout<<f[j]<<" ";
*/

cout << "\n归并排序非递归算法前：" << "\n";
for(j=0;j<numv;j++)
    cout<<g[j]<<" ";
cout << "\n归并排序非递归算法的结果为：" << "\n";
s.mergeSort1(g, g1,numv );

cout << "\n归并排序递归算法前：" << "\n";
for(j=0;j<numv;j++)
    cout<<h[j]<<" ";
cout << "\n归并排序递归算法的结果为：" << "\n";
s.mergeSort2(h,h1,h2, 0, numv-1);
for(int i=0; i < numv; i++)
    cout<<h1[i]<<" ";
cout<<"\n";

return 0;
}
```

5.1 指针与复制构造函数

发表时间: 2012-06-03 关键字: 算法, 指针, 复制构造函数

```
struct Node{
    char *name;
    int age;
    Node(char *n="", int a=0){
        name = new char[strlen(n)+1];
        strcpy(name, n);
        age = a;
    }
}
```

下面声明：

```
Node node1("Roger", 20), node2(node1); //or node2 = node1;
```

注意这是对象的复制，按照道理来说，是两个相互独立的对象，赋值是互不影响的
但是实际上不是这样的，看下面赋值：

```
strcpy(node2.name, "Wendy");
node2.age = 30;
```

输出node1.name, node1.age, node2.name, node2.age

Wendy 30 Wendy 20

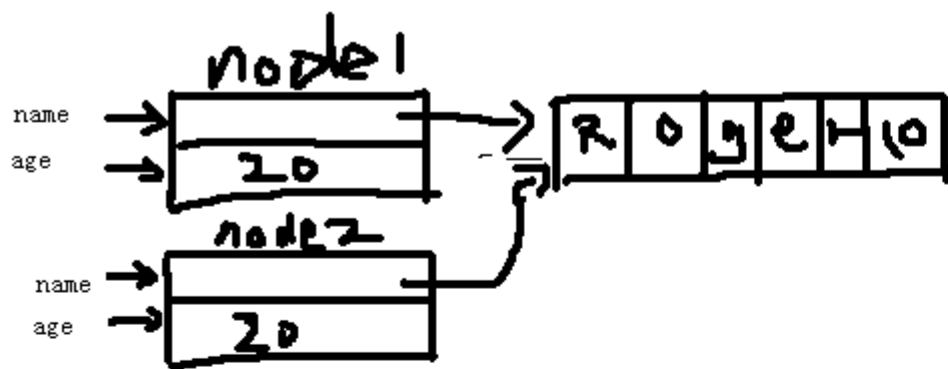
可以看出对象名称是一样的，年龄不一样，why？

在于Node的定义没有提供复制构造函数，编译使用自己生成的构造函数。

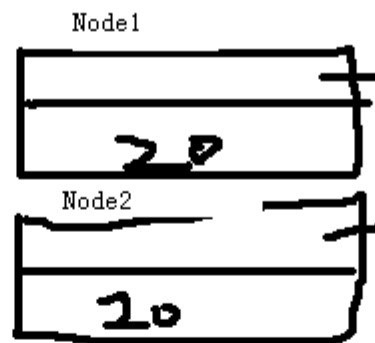
然后默认就是逐个成员进行复制。因为name是一个指针，所以复制构造函数
将node1.name的地址复制给了node2.name,而不是将值拷贝过去。

为了阻止，必须是在拷贝name的时候，重新分配空间并将值拷贝过去，而不是地址。

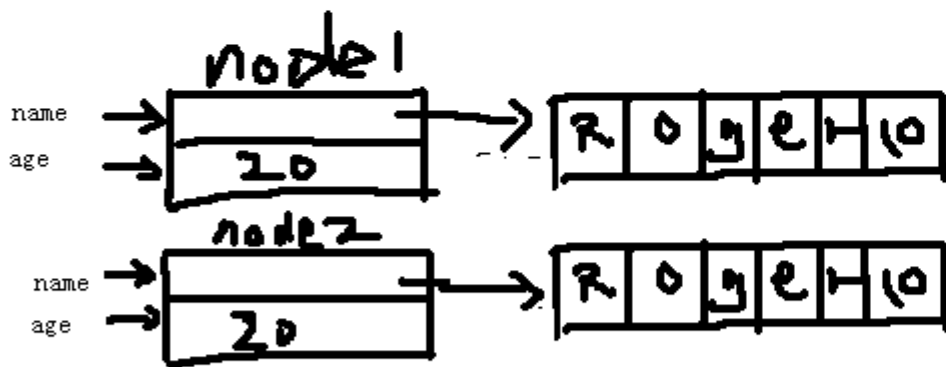
具体的过程如图：复制构造函数.bmp



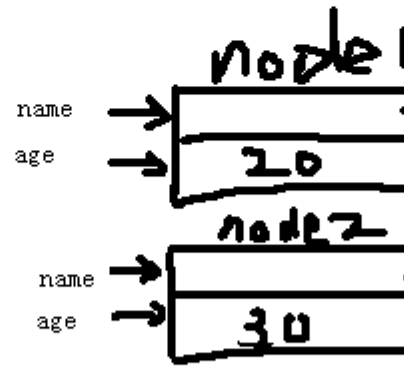
(a)



(b)



(c)



然后写复制构造函数：

```
struct Node{
    char *name;
    int age;
    Node(char *n="", int a=0){
        name = new char[strlen(n)+1];
        strcpy(name, n);
        age = a;
    }
}
```

```
Node(const Node& n){
    name = new char[strlen(n.name)+1];
    strcpy(name, n.name);
    age = n.age;
}
}
```

如果使用`node2 = node1;`同样会有这样的问题，那怎么办，重载操作符

```
Node& operator=(const Node& n){
    if(this != &n){
        if(name!= 0) delete[] name;
        name = new char[strlen(n.name)+1];
        strcpy(name, n.name);
        age = n.age;
    }
    return *this;
}
```

最后在来个析构函数：

```
~Node(){
    if(name!=0) delete[] name;
}
```

5.2 C++动态分配

发表时间: 2012-06-03 关键字: c++, 数组, 动态分配

1.动态分配实例模板：

```
void alloc(int n){  
    //步骤1，分配（放于堆空间中）  
    int *value = new int[n]; //分配数组  
    int *a = new int(4); //分配指针  
  
    //步骤2，初始化（动态分配不会自动初始化）  
    for(int i=0; i<n; i++) value[i]=0;  
  
    //步骤3，释放  
    delete[] value; //释放数组  
    delete a; //与释放数组是不同的  
  
    //步骤4，置空  
    value = NULL;  
    a = NULL;  
}
```

2.动态分配转载

from : <http://hi.baidu.com/xiaomeng008/blog/item/9b7706b0e39d785e08230226.html>

动态内存分配

1.堆内存分配：

C/C++定义了4个内存区间：代码区，全局变量与静态变量区，局部变量区即栈区，动态存储区，即堆（heap）区或自由存储区（free store）。

堆的概念：

通常定义变量（或对象），编译器在编译时都可以根据该变量（或对象）的类型知道所需内存空间的大小，从而系统在适当的时候为他们分配确定的存储空间。这种内存分配称为静态存储分配；

有些操作对象只在程序运行时才能确定，这样编译时就无法为他们预定存储空间，只能在程序运行时，系统根据运行时的要求进行内存分配，这种方法称为动态存储分配。所有动态存储分配都在堆区中进行。

当程序运行到需要一个动态分配的变量或对象时，必须向系统申请取得堆中的一块所需大小的存贮空间，用于存贮该变量或对象。当不再使用该变量或对象时，也就是它的生命结束时，要显式释放它所占用的存贮空间，这样系统就能对该堆空间进行再次分配，做到重复使用有限的资源。

2.堆内存的分配与释放

堆空间申请、释放的方法：

在C++中，申请和释放堆中分配的存贮空间，分别使用new和delete的两个运算符来完成：

指针变量名=new 类型名(初始化式)；

delete 指针名;

例如：

1、int *pi=new int(0);

它与下列代码序列大体等价：

2、int ival=0, *pi=&ival;

区别：pi所指向的变量是由库操作符new()分配的，位于程序的堆区中，并且该对象未命名。

堆空间申请、释放说明：

- (1).new运算符返回的是一个指向所分配类型变量（对象）的指针。对所创建的变量或对象，都是通过该指针来间接操作的，而且动态创建的对象本身没有名字。
- (2).一般定义变量和对象时要用标识符命名，称命名对象，而动态的称无名对象(请注意与栈区中的临时对象的区别，两者完全不同：生命期不同，操作方法不同，临时变量对程序员是透明的)。
- (3).堆区是不会有在分配时做自动初始化的（包括清零），所以必须用初始化式(initializer)来显式初始化。new表达式的操作序列如下：从堆区分配对象，然后用括号中的值初始化该对象。

3.堆空间申请、释放演示：

- (1).用初始化式(initializer)来显式初始化

```
int *pi=new int(0);
```

- (2).当pi生命周期结束时，必须释放pi所指向的目标：

```
delete pi;
```

注意这时释放了pi所指的目标的内存空间，也就是撤销了该目标，称动态内存释放（dynamic memory deallocation），但指针pi本身并没有撤销，它自己仍然存在，该指针所占内存空间并未释放。

下面是关于new 操作的说明

- (1).new运算符返回的是一个指向所分配类型变量（对象）的指针。对所创建的变量或对象，都是通过该指针来间接操作的，而动态创建的对象本身没有名字。
- (2).一般定义变量和对象时要用标识符命名，称命名对象，而动态的称无名对象(请注意与栈区中的临时对象的区别，两者完全不同：生命期不同，操作方法不同，临时变量对程序员是透明的)。
- (3).堆区是不会有在分配时做自动初始化的（包括清零），所以必须用初始化式(initializer)来显式初始化。new表达式的操作序列如下：从堆区分配对象，然后用括号中的值初始化该对象。

4. 在堆中建立动态一维数组

①申请数组空间：

指针变量名=new 类型名[下标表达式];

注意：“下标表达式”不是常量表达式，即它的值不必在编译时确定，可以在运行时确定。

②释放数组空间：

delete []指向该数组的指针变量名;

注意：方括号非常重要的，如果delete语句中少了方括号，因编译器认为该指针是指向数组第一个元素的，会产生回收不彻底的问题（只回收了第一个元素所占空间），加了方括号后就转化为指向数组的指针，回收整个数组。delete []的方括号中不需要填数组元素数，系统自知。即使写了，编译器也忽略。

```
#include <iostream.h>
```

```
#include <string.h>
```

```
void main(){
```

```
    int n;
```

```
    char *pc;
```

```
    cout<<"请输入动态数组的元素个数"<<endl;
```

```
    cin>>n; //n在运行时确定，可输入17
```

```
    pc=new char[n]; //申请17个字符（可装8个汉字和一个结束符）的内存空间
```

```
    strcpy(pc, "堆内存的动态分配"); //
```

```
    cout<<pc<<endl;
```

```
    delete []pc; //释放pc所指向的n个字符的内存空间
```

```
    return ;
```

```
}
```

5. 动态一维数组的说明

① 变量n在编译时没有确定的值，而是在运行中输入，按运行时所需分配堆空间，这一点是动态分配的优点，可克服数组“大开小用”的弊端，在表、排序与查找中的算法，若用动态数组，通用性更佳。一定注意：delete []pc是将n个字符的空间释放，而用delete pc则只释放了一个字符的空间；

② 如果有一个char *pc1，令pc1=p，同样可用delete [] pc1来释放该空间。尽管C++不对数组作边界检查，但在堆空间分配时，对数组分配空间大小是纪录在案的。

③ 没有初始化式（initializer），不可对数组初始化。

6. 指针数组和数组指针

指针类型:

(1) int* ptr; // 指针所指向的类型是int

(2) char* ptr; // 指针所指向的类型是char

(3) int** ptr; // 指针所指向的类型是int*（也就是一个int * 型指针）

(4) int(*ptr)[3]; // 指针所指向的类型是int()[3] // 二维指针的声明

指针数组：(重心在指针，就是存放指针的数组)

一个数组里存放的都是同一个类型的指针，通常我们把他叫做指针数组。

比如 int * a[2]; 它里边放了2个int * 型变量。

```
int * a[2];
```

```
a[0]= new int[3];
```

```
a[1]=new int[3];
```

```
delete a[0];
```

```
delete a[1];
```

注意这里 是一个数组，不能delete []；

数组指针：（重心在数组，就是用指针来表示数组）

一个指向一维或者多维数组的指针.(用数组表示的指针)

```
int * b=new int[10]; 指向一维数组的指针b；
```

注意，这个时候释放空间一定要delete [] ,否则会造成内存泄露， b 就成为了空悬指针

```
int (*b2)[10]=new int[10][10]; 注意，这里的b2指向了一个二维int型数组的首地址.
```

注意：在这里，b2等效于二维数组名，但没有指出其边界，即最高维的元素数量，但是它的最低维数的元素数量必须要指定！就像指向字符的指针，即等效一个字符串,不要把指向字符的指针说成指向字符串的指针。

```
int(*b3) [30] [20]; //三级指针——>指向三维数组的指针；
```

```
int(*b2) [20];    //二级指针；——>指向二维数组的指针；
```

```
b3=new int [1] [20] [30];
```

```
b2=new int [30] [20];
```

删除这两个动态数组可用下式：

```
delete [] b3; //删除（释放）三维数组；
```

```
delete [] b2; //删除（释放）二维数组；
```

在堆中建立动态多维数组

```
new 类型名[下标表达式1] [下标表达式2].....;
```

例如：建立一个动态三维数组

```
float (*cp)[30][20]; //指向一个30行20列数组的指针，指向二维数组的指针
```

```
cp=new float [15] [30] [20]; //建立由15个30*20数组组成的数组；
```

注意：cp等效于三维数组名，但没有指出其边界，即最高维的元素数量，就像指向字符的指针即等效一个字符串,不要把指向字符的指针，说成指向字符串的指针。这与数组的嵌套定义相一致。

5.3 c++中malloc与free

发表时间: 2012-06-04 关键字: malloc, free, 内存分配

from:<http://hi.baidu.com/hayrek/blog/item/4ed2749a5e8307b2c8eaf4c3.html>

c++中malloc与free

一、malloc()和free()的基本概念以及基本用法：

1、函数原型及说明：

`void *malloc(long NumBytes)`：该函数分配了NumBytes个字节，并返回了指向这块内存的指针。如果分配失败，则返回一个空指针（NULL）。

关于分配失败的原因，应该有多种，比如说空间不足就是一种。

`void free(void *FirstByte)`：该函数是将之前用malloc分配的空间还给程序或者是操作系统，也就是释放了这块内存，让它重新得到自由。

2、函数的用法：

其实这两个函数用起来倒不是很难，也就是malloc()之后觉得用够了就甩了它把它给free()了，举个简单例子：

程序代码：

```
// Code...
```

```
char *Ptr = NULL;
```

```
Ptr = (char *)malloc(100 * sizeof(char));
```

```
if (NULL == Ptr)
```

```
{
```

```
exit (1);
```

```
}
```

```
gets(Ptr);
```

```
// code...
```

```
free(Ptr);
```

```
Ptr = NULL;
```

```
// code...
```

就是这样！当然，具体情况要具体分析以及具体解决。比如说，你定义了一个指针，在一个函数里申请了一块内存然后通过函数返回传递给这个指针，那么也许释放这块内存这项工作就应该留给其他函数了。

3、关于函数使用需要注意的一些地方：

A、申请了内存空间后，必须检查是否分配成功。

B、当不需要再使用申请的内存时，记得释放；释放后应该把指向这块内存的指针指向NULL，防止程序后面不小心使用了它。

C、这两个函数应该是配对。如果申请后不释放就是内存泄露；如果无故释放那就是什么也没有做。释放只能一次，如果释放两次及两次以上会

出现错误（释放空指针例外，释放空指针其实也等于啥也没做，所以释放空指针释放多少次都没有问题）。

D、虽然malloc()函数的类型是(void *),任何类型的指针都可以转换成(void *),但是最好还是在前面进行强制类型转换，因为这样可以躲过一

些编译器的检查。

好了！最基础的东西大概这么说！现在进入第二部分：

二、malloc()到底从哪里得来了内存空间：

1、malloc()到底从哪里得到了内存空间？答案是从堆里面获得空间。也就是说函数返回的指针是指向堆里面的一块内存。操作系统中有一个记录空闲内存地址的链表。当操作系统收到程序的申请时，就会遍历该链表，然后就寻找第一个空间大于所申请空间的堆结点，然后就将该结点从空闲结点链表中删除，并将该结点的空间分配给程序。就是这样！

说到这里，不得不另外插入一个小话题，相信大家也知道是什么话题了。什么是堆？说到堆，又忍不住说到了栈！什么是栈？下面就另外开个小部分专门而又简单地说一下这个题外话：

2、什么是堆：堆是大家共有的空间，分全局堆和局部堆。全局堆就是所有没有分配的空间，局部堆就是用户分配的空间。堆在操作系统对进程 初始化的时候分配，运行过程中也可以向系统要额外的堆，但是记得用完了要还给操作系统，要不然就是内存泄漏。

什么是栈：栈是线程独有的，保存其运行状态和局部自动变量的。栈在线程开始的时候初始化，每个线程的栈互相独立。每个函数都有自己的栈，栈被用来在函数之间传递参数。操作系统在切换线程的时候会自动的切换栈，就是切换SS/ESP寄存器。栈空间不需要在高级语言里面显式的分配和释放。

以上的概念描述是标准的描述，不过有个别语句被我删除，不知道因为这样而变得不标准了^_^.

通过上面对概念的描述，可以知道：

栈是由编译器自动分配释放，存放函数的参数值、局部变量的值等。操作方式类似于数据结构中的栈。

堆一般由程序员分配释放，若不释放，程序结束时可能由OS回收。注意这里说是可能，并非一定。所以我想再强调一次，记得要释放！

malloc与free是C++/C语言的标准库函数，new/delete是C++的运算符。它们都可用于申请动态内存和释放内存。

对于非内部数据类型的对象而言，光用maloc/free无法满足动态对象的要求。对象在创建的同时要自动执行构造函数，对象在消亡之前要自动执行析构函数。由于malloc/free是库函数而不是运算符，不在编译器控制权限之内，不能够把执行构造函数和析构函数的任务强加于malloc/free。

因此C++语言需要一个能完成动态内存分配和初始化工作的运算符new，以及一个能完成清理与释放内存工作的运算符delete。注意new/delete不是库函数。

类Obj的函数Initialize模拟了构造函数的功能，函数Destroy模拟了析构函数的功能。函数UseMallocFree中，由于malloc/free不能执行构造函数与析构函数，必须调用成员函数Initialize和Destroy来完成初始化与清除工作。函数UseNewDelete则简单得多。

所以我们不要企图用malloc/free来完成动态对象的内存管理，应该用new/delete。由于内部数据类型的“对象”没有构造与析构的过程，对它们而言malloc/free和new/delete是等价的。

既然new/delete的功能完全覆盖了malloc/free，为什么C++不把malloc/free淘汰出局呢？这是因为C++程序经常要调用C函数，而C程序只能用malloc/free管理动态内存。

如果用free释放“new创建的动态对象”，那么该对象因无法执行析构函数而可能导致程序出错。如果用delete释放“malloc申请的动态内存”，理论上讲程序不会出错，但是该程序的可读性很差。所以new/delete必须配对使用，malloc/free也一样。

5.4 结构体

发表时间: 2012-06-05 关键字: 结构体, 引用, 指针

转载自: <http://www.pconline.com.cn/pcedu/empolder/gj/c/0503/567930.html>

1.结构体定义

简单的来说，结构体就是一个可以包含不同数据类型的一个结构，它是一种可以自己定义的数据类型，它的特点和数组主要有两点不同，首先结构体可以在一个结构中声明不同的数据类型，第二相同结构的结构体变量是可以相互赋值的，而数组是做不到的，因为数组是单一数据类型的数据集合，它本身不是数据类型(而结构体是)，数组名称是常量指针，所以不可以做为左值进行运算，所以数组之间就不能通过数组名称相互复制了，即使数据类型和数组大小完全相同。

定义结构体使用struct修饰符，例如：

```
struct test
{
float a;
int b;
};
```

上面的代码就定义了一个名为test的结构体，它的数据类型就是test，它包含两个成员a和b，成员a的数据类型为浮点型，成员b的数据类型为整型。

2.结构体类型

a.普通结构体

由于结构体本身就是自定义的数据类型，定义结构体变量的方法和定义普通变量的方法一样。

```
test pn1;
```

这样就定义了一test结构体数据类型的结构体变量pn1，结构体成员的访问通过点操作符进行，pn1.a=10 就对结构体变量pn1的成员a进行了赋值操作。

注意:结构体生命的时候本身不占用任何内存空间，只有当你用你定义的结构体类型定义结构体变量的时候计算机才会分配内存。

b.结构指针

结构体，同样是可以定义指针的，那么结构体指针就叫做结构指针。

结构指针通过->符号来访问成员：pn1->a=10;

3.简单示例

```
#include <iostream>
#include <string>
using namespace std;

struct test//定义一个名为test的结构体
{
    int a;//定义结构体成员a
    int b;//定义结构体成员b
```

```
};

void main()
{
    test pn1;//定义结构体变量pn1
    test pn2;//定义结构体变量pn2

    pn2.a=10;//通过成员操作符.给结构体变量pn2中的成员a赋值
    pn2.b=3;//通过成员操作符.给结构体变量pn2中的成员b赋值

    pn1=pn2;//把pn2中所有的成员值复制给具有相同结构的结构体变量pn1
    cout<<pn1.a<<"|"<<pn1.b<<endl;
    cout<<pn2.a<<"|"<<pn2.b<<endl;

    test *point;//定义结构指针

    point=&pn2;//指针指向结构体变量pn2的内存地址
    cout<<pn2.a<<"|"<<pn2.b<<endl;
    point->a=99;//通过结构指针修改结构体变量pn2成员a的值
    cout<<pn2.a<<"|"<<pn2.b<<endl;
    cout<<point->a<<"|"<<point->b<<endl;
    cin.get();
}
```

4.使用结构体的优点

可以清晰的表达数据结构体，如一个人有性别，年龄...等熟悉，如果单独描述就显得很乱，代码也不好看，如果我们定义一个人的结构体就很清晰，所有属性都在里面

```
struct Person{
    int age;
    char sex;
```

```
int num;  
};
```

5.结构体变量是如何作为函数参数进行传递?

```
#include <iostream>  
#include <string>  
using namespace std;  
  
struct test  
{  
    char name[10];  
    float socre;  
};  
  
void print_score(test pn)//以结构变量进行传递  
{  
    cout<<pn.name<<"|"<<pn.socre<<endl;  
}  
  
void print_score(test *pn)//一结构指针作为形参  
{  
    cout<<pn->name<<"|"<<pn->socre<<endl;  
}  
  
void main()  
{  
    test a[2]={{ "marry",88.5},{ "jarck",98.5}};  
    int num = sizeof(a)/sizeof(test);  
    for(int i=0;i<num;i++)  
    {
```

```
        print_score(a[i]); //开辟栈空间，使用了拷贝，从而效率降低
    }
    for(int i=0;i<num;i++)
    {
        print_score(&a[i]); //引用传递，引用相当于变量a的别名，内存地址传递，直接使用相同的内存，故
    }
    cin.get();
}
```

使用很简单，就和普通的变量，指针使用方法相同

但注意：`void print_score(test *pn)`的效率是要高过`void print_score(test pn)`的，因为直接内存操作避免了栈空间开辟结构变量空间需求，节省内存。

下面直接使用引用来传递参数

利用引用传递的好处很多，它的效率和指针相差无几，但引用的操作方式和值传递几乎一样，种种优势都说明善用引用可以做到程序的易读和易操作，

它的优势尤其在结构和大的时候，避免传递结构变量很大的值，节省内存，提高效率。

```
#include <iostream>
#include <string>
using namespace std;

struct test
{
    char name[10];
    float socre;
};

void print_score(test &pn)//以结构变量进行传递，这里是引用传递，上面是指针传递
{
    cout<<pn.name<<"|"<<pn.socre<<endl;
```

```
}

void main()
{
    test a[2]={{"marry",88.5},{"jarck",98.5}};
    int num = sizeof(a)/sizeof(test);
    for(int i=0;i<num;i++)
    {
        print_score(a[i]);
    }
    cin.get();
}
```

6.引用传递效率分析

```
//-----例程1-----

#include <iostream>
#include <string>
using namespace std;

struct test
{
    char name[10];
    float socre;
};

void print_score(test &pn)
{
    cout<<pn.name<<"|"<<pn.socre<<endl;
}

//注意这里面没有使用引用，而是返回
```

```
test get_score()
{
    test pn;
    cin>>pn.name>>pn.socre;
    return pn;
}

void main()
{
    test a[2];
    int num = sizeof(a)/sizeof(test);
    for(int i=0;i<num;i++)
    {
        a[i]=get_score();
    }
    cin.get();
    for(int i=0;i<num;i++)
    {
        print_score(a[i]);
    }
    cin.get();
}

//-----例程2-----

#include <iostream>
#include <string>
using namespace std;

struct test
{
    char name[10];
    float socre;
};

void print_score(test &pn) //引用
{
    cout<<pn.name<<"|"<<pn.socre<<endl;
```



```
}

void get_score(test &pn) //引用
{
    cin>>pn.name>>pn.socre;
}

void main()
{
    test a[2];
    int num = sizeof(a)/sizeof(test);
    for(int i=0;i<num;i++)
    {
        get_score(a[i]);
    }
    cin.get();
    for(int i=0;i<num;i++)
    {
        print_score(a[i]);
    }
    cin.get();
}
```

通过比较两个例子来分析效率

例程2的效率要远高过例程1的原因主要有以下两处：

第一：

例程1中的

```
test get_score()
{
    test pn;
    cin>>pn.name>>pn.socre;
```

```
return pn;  
}
```

调用的时候在内部要在栈空间开辟一个名为pn的结构体变量，程序pn返回的时候又再次在栈内存空间内自动生成了一个临时结构体变量temp，在前面的教程中我们已经说过，它是一个copy，而例程2中的：

```
void get_score(test &pn)  
{  
    cin>>pn.name>>pn.socre;  
}
```

却没有这一过程，不开辟任何新的内存空间，也没有任何临时变量的生成。

第二：

例程1在mian()中，必须对返回的结构体变量进行一次结构体变量与结构体变量直接的相互赋值操作。

```
for(int i=0;i<num;i++)  
{  
    a[i]=get_score();  
}
```

而例程2中由于是通过内存地址直接操作，所以完全没有这一过程，提高了效率。

```
for(int i=0;i<num;i++)
{
    get_score(a[i]);
}
```

函数也是可以返回结构体引用，例子如下：

```
#include <iostream>
#include <string>
using namespace std;

struct test
{
    char name[10];
    float socre;
};

test a;

test &get_score(test &pn) //返回结构体的引用
{
    cin>>pn.name>>pn.socre;
    return pn;
}

void print_score(test &pn)
{
    cout<<pn.name<<"|"<<pn.socre<<endl;
}

void main()
{
    test &sp=get_score(a);
```

```
cin.get();  
cout<<sp.name<<"|"<<sp.socre;  
cin.get();  
}
```

调用get_score(a);结束并返回的时候，函数内部没有临时变量的产生，返回直接把全局结构变量a的内存地址赋予结构引用sp

最后提一下指针的引用

定义指针的引用方法如下：

```
void main()  
{  
int a=0;  
int b=10;  
int *p1=&a;  
int *p2=&b;  
int *&pn=p1; //指针的引用，相当于p1的别名  
cout <<pn<<"|"<<*pn<<endl;  
pn=p2; //就是一个指针  
cout <<pn<<"|"<<*pn<<endl;  
cin.get();  
}
```

pn就是一个指向指针的引用，它也可以看做是指针别名，总之使用引用要特别注意它的特性，它的操作是和普通指针一样的，在函数中对全局指针的引用操作要十分小心，避免破坏全局指针！

5.5 动态规划

发表时间: 2012-06-06 关键字: 算法, 动态规划

该文章转载自：http://www.cppblog.com/Fox/archive/2008/05/07/Dynamic_programming.html

非常感谢！

以前在学习非数值算法的时候，曾经了解过**动态规划算法 (Dynamic programming)**，以下是对**Wikipedia**上**动态规划**的翻译，图也是**Wikipedia**上的，仓促行文，不到之处，请方家指正。

这篇文章的术语实在是太多了，所以我在文中加入了少量注释，一律以**粗斜体**注明。

本文的不足之处将**随时修正**，MIT的《**Introduction to Algorithms**》第15章是专门讲动态规划的。

动态规划

在数学与计算机科学领域，**动态规划**用于解决那些可分解为**重复子问题**（overlapping subproblems，**想想递归求阶乘吧**）并具有**最优子结构**（optimal substructure，**想想最短路径算法**）（如下所述）的问题，动态规划比通常算法花费更少时间。

上世纪40年代，**Richard Bellman**最早使用动态规划这一概念表述通过遍历寻找最优决策解问题的求解过程。1953年，Richard Bellman将动态规划**赋予现代意义**，该领域被IEEE纳入系统分析和工程中。为纪念Bellman的贡献，动态规划的核心方程被命名为**贝尔曼方程**，该方程以**递归**形式重申了一个优化问题。

在“动态规划”（dynamic programming）一词中，programming与“计算机编程”（computer programming）中的programming并无关联，而是来自“**数学规划**”（mathematical programming），也称优化。因此，规划是指对生成活动的优化策略。举个例子，编制一场展览的日程可称为规划。在此意义上，规划意味着找到一个可行的活动计划。

• 概述

图1 使用最优子结构寻找最短路径：直线表示边，波状线表示两顶点间的最短路径（路径中其他节点未显示）；粗线表示从起点到终点的最短路径。

不难看出，start到goal的最短路径由start的相邻节点到goal的最短路径及start到其相邻节点的成本决定。

最优子结构即可用来寻找整个问题最优解的子问题的最优解。举例来说，寻找图¹上某顶点到终点的²最短路径，可先计算该顶点所有相邻顶点至终点的最短路径，然后以此来选择最佳整体路径，如图¹所示。

一般而言，最优子结构通过如下三个步骤解决问题：

- a) 将问题分解成较小的子问题；
- b) 通过递归使用这三个步骤求出子问题的最优解；
- c) 使用这些最优解构造初始问题的最优解。

子问题的求解是通过不断划分为更小的子问题实现的，直至我们可以在常数时间内求解。

图² Fibonacci序列的子问题示意图：使用有向无环图（DAG, directed acyclic graph）而非树表示重复子问题的分解。

为什么是DAG而不是树呢？答案就是，如果是树的话，会有很多重复计算，下面有相关的解释。

一个问题可划分为重复子问题是指通过相同的子问题可以解决不同的较大问题。例如，在Fibonacci序列中， $F_3 = F_1 + F_2$ 和 $F_4 = F_2 + F_3$ 都包含计算 F_2 。由于计算 F_5 需要计算 F_3 和 F_4 ，一个比较笨的计算 F_5 的方法可能会重复计算 F_2 两次甚至两次以上。这一点对所有重复子问题都适用：愚蠢的做法可能会为重复计算已经解决的最优子问题的解而浪费时间。

为避免重复计算，可将已经得到的子问题的解保存起来，当我们要解决相同的子问题时，重用即可。该方法即所谓的缓存（memoization，而不是存储memorization，虽然这个词亦适合，姑且这么叫吧，这个单词太难翻译了，简直就是可意会不可言传，其意义是没计算过则计算，计算过则保存）。当我们确信将不会再需要某一解时，可以将其抛弃，以节省空间。在某些情况下，我们甚至可以提前计算出那些将来会用到的子问题的解。

总括而言，动态规划利用：

- 1) 重复子问题
- 2) 最优子结构
- 3) 缓存

动态规划通常采用以下两种方式中的一种两个办法：

自顶向下：将问题划分为若干子问题，求解这些子问题并保存结果以免重复计算。该方法将递归和缓存结合在一起。

自下而上：先行求解所有可能用到的子问题，然后用其构造更大问题的解。该方法在节省堆栈空间和减少函数调用数量上略有优势，但有时想找出给定问题的所有子问题并不那么直观。

为了提高**按名传递**（call-by-name，这一机制与**按需传递**call-by-need相关，复习一下参数传递的各种规则吧，简单说一下，按名传递允许改变实参值）的效率，一些编程语言将函数的返回值“自动”缓存在函数的特定参数集合中。一些语言将这一特性尽可能简化（如Scheme、Common Lisp和Perl），也有一些语言需要进行特殊扩展（如C++，C++中使用的是按值传递和按引用传递，因此C++中本无自动缓存机制，需自行实现，具体实现的一个例子是Automated Memoization in C++）。无论如何，只有**指称透明**（referentially transparent，指称透明是指在程序中使用表达式、函数本身或以其值替换对程序结果没有任何影响）函数才具有这一特性。

• 例子

1. Fibonacci序列

寻找Fibonacci序列中第n个数，基于其数学定义的直接实现：

```
function fib(n)
  if n = 0
    return 0
  else if n = 1
    return 1
  return fib(n-1) + fib(n-2)
```

如果我们调用fib(5)，将产生一棵对于同一值重复计算多次的调用树：

1. fib(5)
2. fib(4) + fib(3)
3. (fib(3) + fib(2)) + (fib(2) + fib(1))
4. ((fib(2) + fib(1)) + (fib(1) + fib(0))) + ((fib(1) + fib(0)) + fib(1))
5. (((fib(1) + fib(0)) + fib(1)) + (fib(1) + fib(0))) + ((fib(1) + fib(0)) + fib(1))

特别是，fib(2)计算了3次。在更大规模的例子中，还有更多fib的值被重复计算，将消耗指数级时间。

现在，假设我们有一个简单的**映射**（map）对象m，为每一个计算过的fib及其返回值建立映射，修改上面的函数fib，使用并不断更新m。新的函数将只需O(n)的时间，而非指数时间：

```
var m := map(0 → 1, 1 → 1)
function fib(n)
  if map m does not contain key n
```

```
m[n] := fib(n-1) + fib(n-2)
return m[n]
```

这一保存已计算出的数值的技术即被称为**缓存**，这儿使用的是**自顶向下**的方法：先将问题划分为若干子问题，然后计算和存储值。

在**自下而上**的方法中，我们先计算较小的fib，然后基于其计算更大的fib。这种方法也只花费线性（ $O(n)$ ）时间，因为它包含一个 $n-1$ 次的循环。然而，这一方法只需要常数（ $O(1)$ ）的空间，相反，**自顶向下**的方法则需要 $O(n)$ 的空间来储存映射关系。

```
function fib(n)
  var previousFib := 0, currentFib := 1
  if n = 0
    return 0
  else if n = 1
    return 1
  repeat n-1 times
    var newFib := previousFib + currentFib
    previousFib := currentFib
    currentFib := newFib
  return currentFib
```

在这两个例子，我们都只计算fib(2)一次，然后用它来计算fib(3)和fib(4)，而不是每次都重新计算。

2. 一种平衡的0-1矩阵

考虑 $n \times n$ 矩阵的赋值问题：只能赋0和1， n 为偶数，使每一行和列均含 $n/2$ 个0及 $n/2$ 个1。例如，当 $n=4$ 时，两种可能的方案是：

+ - - - +	+ - - - +
0 1 0 1	0 0 1 1
1 0 1 0	0 0 1 1
0 1 0 1	1 1 0 0
1 0 1 0	1 1 0 0
+ - - - +	+ - - - +

问：对于给定 n ，共有多少种不同的赋值方案。

至少有三种可能的算法来解决这一问题：**穷举法**（brute force）、**回溯法**（backtracking）及动态规划（dynamic programming）。穷举法列举所有赋值方案，并逐一找出满足平衡条件的方案。由于共有 $C(n, n/2)^n$ 种方案（**在一行中，含 $n/2$ 个0及 $n/2$ 个1的组合数为 $C(n, n/2)$ ，相当于从 n 个位置中选取 $n/2$ 个位置置0，剩下的自然是1**），当 $n=6$ 时，穷举法就已经几乎不可行了。回溯法先将矩阵中部分元素置为0或1，然后检查每一行和列中未被赋值的元素并赋值，使其满足每一行和列中0和1的数量均为 $n/2$ 。回溯法比穷举法更加巧妙一些，但仍需遍历所有解才能确定解的数目，可以看到，当 $n=8$ 时，该题解的数目已经

高达116963796250。动态规划则无需遍历所有解便可确定解的数目（意思是划分子问题后，可有效避免若干子问题的重复计算）。

通过动态规划求解该问题出乎意料的简单。考虑每一行恰含 $n/2$ 个0和 $n/2$ 个1的 $k \times n (1 \leq k \leq n)$ 的子矩阵，函数 f 根据每一行的可能的赋值映射为一个向量，每个向量由 n 个整数对构成。向量每一列对应的一个整数对中的两个整数分别表示该列上该行以下已经放置的0和1的数量。该问题即转化为寻找 $f((n/2, n/2), (n/2, n/2), \dots, (n/2, n/2))$ （具有 n 个参数或者说是一个含 n 个元素的向量）的值。其子问题的构造过程如下：

- 1) 最上面一行（第 k 行）具有 $C(n, n/2)$ 种赋值；
- 2) 根据最上面一行中每一列的赋值情况（为0或1），将其对应整数对中相应的元素值减1；
- 3) 如果任一整数对中的任一元素为负，则该赋值非法，不能成为正确解；
- 4) 否则，完成对 $k \times n$ 的子矩阵中最上面一行的赋值，取 $k = k - 1$ ，计算剩余的 $(k - 1) \times n$ 的子矩阵的赋值；
- 5) 基本情况是一个 $1 \times n$ 的细小的子问题，此时，该子问题的解的数量为0或1，取决于其向量是否是 $n/2$ 个(0, 1)和 $n/2$ 个(1, 0)的排列。

例如，在上面给出的两种方案中，向量序列为：

0

1

0

1

0

0

1

1

1

0

1

0

0

0

1

1

0

1

0

1

1

1

0

0

1

0

1

0

1

1

0

0

0

0

0

0

0

0

0

0

0

1

0

1

1

1

0

0

1

0

1

0

0

0

1

1

0

1

0

1

1

1

0

0

0

1

0

1

1

1

0

0

0

1

0

1

1

1

0

0

0

1

0

1

1

1

0

0

0

1

0

1

1

1

0

0

0

1

0

1

1

1

0

0

动态规划在此的意义在于避免了相同 f 的重复计算，更进一步的，上面着色的两个 f ，虽然对应向量不同，但 f 的值是相同的，想想为什么吧:D。

该问题解的数量（序列a058527在OEIS）是1, 2, 90, 297200, 116963796250, 6736218287430460752, ...

下面的外部链接中包含回溯法的Perl源代码实现，以及动态规划法的MAPLE和C语言的实现。

3. 棋盘

考虑 $n \times n$ 的棋盘及成本函数 $C(i, j)$ ，该函数返回方格 (i, j) 相关的成本。以 5×5 的棋盘为例：

```
5 | 6 7 4 7 8
4 | 7 6 1 1 4
3 | 3 5 7 8 2
2 | 2 6 7 0 2
1 | 7 3 5 6 1
- + - - - -
  | 1 2 3 4 5
```

可以看到：C(1,3)=5

从棋盘的任一方格的第一阶（即行）开始，寻找到达最后一阶的最短路径（使所有经过的方格的成本之和最小），假定只允许向左对角、右对角或垂直移动一格。

```
5 |
4 |
3 |
2 | x x x
1 | o
- + - - - -
  | 1 2 3 4 5
```

该问题展示了最优子结构。即整个问题的全局解依赖于子问题的解。定义函数q(i,j)，令：q(i,j)表示到达方格(i,j)的最低成本。

如果我们可以求出第n阶所有方格的q(i,j)值，取其最小值并逆向该路径即可得到最短路径。

记q(i,j)为方格(i,j)至其下三个方格（(i-1,j-1)、(i-1,j)、(i-1,j+1)）最低成本与c(i,j)之和，例如：

```
5 |
4 | A
3 | B C D
2 |
1 |
- + - - - -
  | 1 2 3 4 5
```

$q(A) = \min(q(B),q(C),q(D)) + c(A)$

定义q(i,j)的一般形式：

$$q(i,j) = \begin{cases} \text{inf.} & j < 1 \text{ or } j > n \\ \text{---} c(i,j) & i = 1 \\ \min(q(i-1,j-1),q(i-1,j),q(i-1,j+1))+c(i,j) & \text{otherwise.} \end{cases}$$

方程的第一行是为了保证递归可以退出（处理边界时只需调用一次递归函数）。第二行是第一阶的取值，作为计算的起点。第三行的递归是算法的重要组成部分，与例子A、B、C、D类似。从该定义我们可以直接给出计算 $q(i,j)$ 的简单的递归代码。在下面的伪代码中， n 表示棋盘的维数， $C(i,j)$ 是成本函数， $\min()$ 返回一组数的最小值：

```
function minCost(i, j)
    if j < 1 or j > n
        return infinity
    else if i = 1
        return c(i,j)
    else
        return min(minCost(i-1,j-1),minCost(i-1,j),minCost(i-1,j+1))+c(i,j)
```

需要指出的是， \minCost 只计算路径成本，并不是最终的实际路径，二者相去不远。与Fibonacci数相似，由于花费大量时间重复计算相同的最短路径，这一方式慢的恐怖。不过，如果采用自下而上法，使用二维数组 $q[i,j]$ 代替函数 \minCost ，将使计算过程快得多。我们为什么要这样做呢？选择保存值显然比使用函数重复计算相同路径要简单的多。

我们还需要知道实际路径。路径问题，我们可以通过另一个前任数组 $p[i,j]$ 解决。这个数组用于描述路径，代码如下：

```
function computeShortestPathArrays()
    for x from 1 to n
        q[1, x] := c(1, x)
    for y from 1 to n
        q[y, 0] := infinity
        q[y, n + 1] := infinity
    for y from 2 to n
        for x from 1 to n
            m := min(q[y-1, x-1], q[y-1, x], q[y-1, x+1])
            q[y, x] := m + c(y, x)
            if m = q[y-1, x-1]
                p[y, x] := -1
            else if m = q[y-1, x]
                p[y, x] := 0
            else
                p[y, x] := 1
```

剩下的求最小值和输出就比较简单了：

```
function computeShortestPath()
    computeShortestPathArrays()
    minIndex := 1
```

```

min := q[n, 1]
for i from 2 to n
  if q[n, i] < min
    minIndex := i
    min := q[n, i]
printPath(n, minIndex)

```

```

function printPath(y, x)
  print(x)
  print("<-")
  if y = 2
    print(x + p[y, x])
  else
    printPath(y-1, x + p[y, x])

```

4. 序列比对

序列比对是动态规划的一个重要应用。序列比对问题通常是使用编辑操作（替换、插入、删除一个要素等）进行序列转换。每次操作对应不同成本，目标是找到编辑序列的最低成本。

可以很自然地想到使用递归解决这个问题，序列A到B的最优编辑通过以下措施之一实现：

插入B的第一个字符，对A和B的剩余序列进行最优比对；

删去A的第一个字符，对A和B进行最优比对；

用B的第一个字符替换A的第一个字符，对A的剩余序列和B进行最优比对。

局部比对可在矩阵中列表表示，单元(i,j)表示A[1..i]到b[1..j]最优比对的成本。单元(i,j)的成本计算可通过累加相邻单元的操作成本并选择最优解实现。至于序列比对的实现算法，参见**Smith-Waterman**和**Needleman-Wunsch**。

对序列比对的话题并不熟悉，更多的话也无从谈起，有熟悉的朋友倒是可以介绍一下。

- 应用动态规划的算法

- 1) 许多**字符串**操作算法如**最长公共子列**、**最长递增子列**、**最长公共子串**；
- 2) 将动态规划用于**图**的树分解，可以有效解决有界**树宽图**的**生成树**等许多与图相关的算法问题；
- 3) 决定是否及如何通过某一特定**上下文无关文法**产生给定字符串的**Cocke-Younger-Kasami** (CYK)算法；
- 4) **计算机国际象棋**中**转换表**和**驳斥表**的使用；
- 5) **Viterbi算法**（用于**隐式马尔可夫模型**）；

- 6) Earley算法（一类图表分析器）；
- 7) Needleman-Wunsch及其他生物信息学中使用的算法，包括序列比对、结构比对、RNA结构预测；
- 8) Levenshtein距离（编辑距离）；
- 9) 弗洛伊德最短路径算法；
- 10) 连锁矩阵乘法次序优化；
- 11) 子集求和、背包问题和分治问题的伪多项式时间算法；
- 12) 计算两个时间序列全局距离的动态时间规整算法；
- 13) 关系型数据库的查询优化的Selinger（又名System R）算法；
- 14) 评价B样条曲线的De Boor算法；
- 15) 用于解决板球运动中中断问题的Duckworth-Lewis方法；
- 16) 价值迭代法求解马尔可夫决策过程；
- 17) 一些图形图像边缘以下的选择方法，如“磁铁”选择工具在Photoshop；
- 18) 间隔调度；
- 19) 自动换行；
- 20) 巡回旅行商问题（又称邮差问题或货担郎问题）；
- 21) 分段最小二乘法；
- 22) 音乐信息检索跟踪。

对于这些算法应用，大多未曾接触，甚至术语翻译的都有问题，鉴于本文主要在于介绍动态规划，所以仓促之中，未及查证。

• 相关

- 1) 贝尔曼方程
- 2) 马尔可夫决策过程
- 3) 贪心算法

• 参考

- Adda, Jerome, and Cooper, Russell, 2003. [Dynamic Economics](#). MIT Press. An accessible introduction to dynamic programming in economics. The link contains sample programs.

- Richard Bellman, 1957, Dynamic Programming, Princeton University Press. Dover paperback edition (2003), ISBN 0486428095.
- Bertsekas, D. P., 2000. Dynamic Programming and Optimal Control, Vols. 1 & 2, 2nd ed. Athena Scientific. ISBN 1-886529-09-4.
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, 2001. Introduction to Algorithms, 2nd ed. MIT Press & McGraw-Hill. ISBN 0-262-03293-7. Especially pp. 323–69.
- Giegerich, R., Meyer, C., and Steffen, P., 2004, "A Discipline of Dynamic Programming over Sequence Data," Science of Computer Programming 51: 215–263.
- Nancy Stokey, and Robert E. Lucas, with Edward Prescott, 1989. Recursive Methods in Economic Dynamics. Harvard Univ. Press.
- S. P. Meyn, 2007. Control Techniques for Complex Networks, Cambridge University Press, 2007.
 - 外部链接
 - Dyna, a declarative programming language for dynamic programming algorithms
 - Wagner, David B., 1995, "Dynamic Programming." An introductory article on dynamic programming in Mathematica.
 - Ohio State University: CIS 680: class notes on dynamic programming, by Eitan M. Gurari
 - A Tutorial on Dynamic programming
 - MIT course on algorithms - Includes a video lecture on DP along with lecture notes -- See lecture 15.
 - More DP Notes
 - King, Ian, 2002 (1987), "A Simple Introduction to Dynamic Programming in Macroeconomic Models." An introduction to dynamic programming as an important tool in economic theory.
 - Dynamic Programming: from novice to advanced A TopCoder.com article by Dumitru on Dynamic Programming
 - Algebraic Dynamic Programming - a formalized framework for dynamic programming, including an entry-level course to DP, University of Bielefeld
 - Dreyfus, Stuart, "Richard Bellman on the birth of Dynamic Programming."
 - Dynamic programming tutorial
 - An Introduction to Dynamic Programming

关于动态规划，这只是一篇译文，后面将根据实际问题具体写点动态规划的应用。

5.6 单链表的面向对象实现

发表时间: 2012-06-20 关键字: 单链表, 链表, 对象化, 节点

1.利用非面向对象定义节点Node

下面是头文件LinkedList.h

```
#ifndef LinkedList_H
#define LinkedList_H

template<class T>
struct Node{
    Node<T>* next;
    T data;
};

template<class T>
class LinkedList{
public:
    LinkedList(); //建立只有头结点
    LinkedList(T a[], int n); //建立有n个元素的单链表
    ~LinkedList(); //析构函数
    int length(); //求单链表的长度
    T get(int i); //取单链表中第i个结点的元
    int locate(T x); //求单链表中值为x的元素序号
    void insertData(int i, T x); //在单链表中第i个位置插入元素值为x的结点
    T deleteData(int i); //在单链表中删除第i个结点
    void printList(); //遍历单链表，按序号依次输出各元素
    Node<T>* getFirst(); //获取头结点
private:
    Node<T>* first; //单链表的头指针
    void release(); //释放链表
};

#endif
```

在这里定义头结点是利用了原来c语言的方式，定义的结构体，这是c++从c继承过来的，现在要改成c++面向对象的方法定义节点！

对头文件的实现LinkedList.cpp就不写了

2.对Node节点的面向对象实现

a.LinkList.h

```
//LinkList.h 声明类LinkList
#ifndef LinkList_H
#define LinkList_H
#include "Node.h"

template <class T>
class LinkList
{
public:
    LinkList( ); //建立只有头结点的空链表
    LinkList(T a[], int n); //建立有n个元素的单链表
    ~LinkList( ); //析构函数
    int Length( ); //求单链表的长度
    T Get(int i); //取单链表中第i个结点的元素值
    int Locate(T x); //求单链表中值为x的元素序号
    void Insert(int i, T x); //在单链表中第i个位置插入元素值为x的结点
    T Delete(int i); //在单链表中删除第i个结点
    void PrintList( ); //遍历单链表，按序号依次输出各元素
private:
    Node<T> *first; //单链表的头指针
};

#endif
```

b.然后在Node中将LinkList.h声明为友元类，这样可以访问Node中的私有成员

Node.h


```
//Node.h 声明类Node
#ifndef Node_H
#define Node_H

template <class T>
class LinkList;      //为是Node类的友元类而声明

template <class T>
class Node
{
public:
    friend class LinkList<T>;    //将LinkList类设为友元类
private:
    T data;
    Node<T> *next;
};

#endif
```

这样就完成了彻底的对象会编程

5.7 C++ 迭代器失效的问题

发表时间: 2012-06-25 关键字: 迭代器, c++

转载自: <http://blog.csdn.net/zhongjiekangping/article/details/5624922>

众所周知当使用一个容器的insert或者erase函数通过迭代器插入或删除元素"可能"会导致迭代器失效, 因此很多建议都是让我们获取insert或者erase返回的迭代器, 以使用重新获取新的有效的迭代器进行正确的操作:

view plaincopy to clipboardprint?

```
iter=vec.insert(iter);
```

```
iter=vec.erase(iter);
```

想想究竟为什么迭代器失效, 原因也不难理解。以vector为例, 当我们插入一个元素时它的预分配空间不够时, 它会重新申请一段新空间, 将原空间上的元素复制到新的空间上去, 然后再把新加入的元素放到新空间的尾部, 以满足vector元素要求连续存储的目的。而后原空间会被系统撤销或征做他用, 于是指向原空间的迭代器就成了类似于“悬垂指针”一样的东西, 指向了一片非法区域。如果使用了这样的迭代器会导致严重的运行时错误就变得很自然了。这也是许多书上叙述vector在insert操作后“可能导致所有迭代器失效”的原因。但是想到这里我不禁想到vector的erase操作的叙述是“会导致指向删除元素和删除元素之后的迭代器失效”。但是明显感觉erase带来失效要比insert来得轻得多。似乎“此失效非彼失效”, 想想似乎也是这样的: erase操作是在原空间上进行的, 假设有一个存有"12345"序列的vector<int>容器原本指向3的迭代器在我删除2之后无非变成指向4了, 我只要注意别用到超过end位置的迭代器不就行了吗?

说了这么多似乎可以归纳一下迭代器失效的类型了:

- 1.由于容器元素整体“迁移”导致存放原容器元素的空间不再有效, 从而使得指向原空间的迭代器失效。
- 2.由于删除元素使得某些元素次序发生变化使得原本指向某元素的迭代器不再指向希望指向的元素。

对于第一种类型没什么好就是的了, 原因应该确定如此了。可对于第二种, 我写了如下的代码

view plaincopy to clipboardprint?

```
vector<int> vec;
```

```
for(int i=0;i<10;i++)
```

```
vec.push_back(i);
```

```
vector<int>::iterator iter =vec.begin()+2;
```

```
vec.erase(iter);//注:这里真的不建议这么写
```

```
cout<<*iter<<endl;
```

```
for(vector<int>::iterator it=vec.begin();it!=vec.end();it++)
cout<<*it<<endl;
```

按照我的猜测尽管我在注释的位置的写法很“危险”，但是我并未涉及到上面总结第一种失效类型的范畴。程序应该还是会如预期的一样删除在vec[2]位置上的2然后输出前移到vec[2]位置上的3，并输出0到10不含2的所有数字，于是vs2008的c++环境下运行一下，竟然一个是个无情的“红叉”。按道理说这种写法是绝对不对出现严重的运行时错误的，难道是猜测得不对吗？

当时被这样的疑问困扰了几几天，有一天突然想起一个关于容器迭代器作参数的例子，在vs2008下运行不了，但是在vc6.0下却可以，有人说是因为08采用了更为严格的类型检测机制。于是将上面的代码放到6.0里，果然得到了预期的效果，看来系统果然是这么处理的。至于08为什么不可以，现在我只能认为是采用了更加安全的检查机制。使得第二种类型的失效后果同样“不可饶恕”。

这样我就又想到假如insert元素时原空间够用的话，是不是也不会产生第一类失效而产生第二类失效呢？

```
view plaincopy to clipboardprint?
vector<int> vec;
for(int i=0;i<10;i++)
    vec.push_back(i);
    cout<<"capacity:"<<vec.capacity()<<endl;//查看预分配空间大小
vector<int>::iterator iter =vec.begin()+2;
vec.insert(iter,100);
cout<<"capacity:"<<vec.capacity()<<endl;
cout<<*iter<<endl;
for(vector<int>::iterator it=vec.begin();it!=vec.end();it++)
cout<<*it<<endl;
```

同样在vc6.0下，证明了假设是对的。但是上面的种种做法只是为了帮助我理解迭代器失效的原因，建议使用insert和erase操作时还是像许多书中介绍的如第一段代码那样的写法，这是一种好的且安全的习惯。总之一句话去相信“insert和erase操作后所有的迭代器都会失效”。

迭代器（iterator）是一个可以对其执行类似指针的操作（如：解除引用（operator*()）和递增（operator++()））的对象，我们可以将它理解成为一个指针。但它又不是我们所谓普通的指针，我们可以称之为广义指针，你可以通过sizeof（vector::iterator）来查看，所占内存并不是4个字节。

首先对于vector而言，添加和删除操作可能使容器的部分或者全部迭代器失效。那为什么迭代器会失效呢？vector元素在内存中是顺序存储，试想：如果当前容器中已经存在了10个元素，现在又要添加一个元素到容器中，但是内存中紧跟在这10个元素后面没有一个空闲空间，而vector的元素必须顺序存储一边索引访问，所以我们不能在内存中随便找个

地方存储这个元素。于是vector必须重新分配存储空间，用来存放原来的元素以及新添加的元素：存放在旧存储空间的元素被复制到新的存储空间里，接着插入新的元素，最后撤销旧的存储空间。这种情况发生，一定会导致vector容器的所有迭代器都失效。

我们看到实现上述所说的分配和撤销内存空间的方式以实现vector的自增长性，效率是极其低下的。为了使vector容器实现快速的内存分配，实际分配的容器会比当前所需的空间多一些，vector容器预留了这些额外的存储区，用来存放新添加的元素，而不需要每次都重新分配新的存储空间。你可以从vector里实现capacity和reserve成员可以看出这种机制。

capacity和size的区别：size是容器当前拥有的元素个数，而capacity则指容器在必须分配新存储空间之前可以存储的元素总数。

vector迭代器的几种失效的情况：1.当插入（push_back）一个元素后，end操作返回的迭代器肯定失效。2.当插入（push_back）一个元素后，capacity返回值与没有插入元素之前相比有改变，则需要重新加载整个容器，此时first和end操作返回的迭代器都会失效。3.当进行删除操作（erase，pop_back）后，指向删除点的迭代器全部失效；指向删除点后面的元素的迭代器也将全部失效。

deque迭代器的失效情况：在C++Primer一书中是这样限定的：1.在deque容器首部或者尾部插入元素不会使得任何迭代器失效。2.在其首部或尾部删除元素则只会使指向被删除元素的迭代器失效。3.在deque容器的任何其他位置的插入和删除操作将使指向该容器元素的所有迭代器失效。但是：我在vs2005测试发现第一条都不满足，不知为何？等以后深入STL以后慢慢的领会吧！

只有list的迭代器好像很少情况下会失效。也许就只是在删除的时候，指向被删除节点的迭代器会失效吧，其他的还没有发现。

5.8 单链表的简单实践

发表时间: 2012-06-27 关键字: 链表, 指针, 插入

1. 算法描述

数据结构与算法分析C++版：3.11

实现一个有序单链表，要求能返回链表大小，打印链表，检测x是否在链表（在则删除，否则添加）

2. 实现

List.h

```
#ifndef LIST_H
#define LIST_H
#include <iostream>

template<class T>
struct Node{
    Node<T>* next;
    T data;
};

template<class T>
class List{
public:
    List(){
        len = 0;
        first = new Node<T>;
        first->next = NULL;
    }
    List(T a[], int n){
        len = 0;
        first = new Node<T>;
        first->next = NULL;
```

```
        for(int i=0; i<n; i++){
            //有序插入元素
            insertData(a[i]);
        }
    }

    ~List(){ release(); };

    //长度
    int length(){
        return len;
    }

    //长度(通过遍历链表)
    int getLength(){
        int length=0;
        Node<T>* r = first->next;
        while(r){
            length++;
            r = r->next;
        }
        return length;
    }

    //打印链表
    void printList(){
        Node<T>* r = first->next;
        std::cout<<"[";
        while(r){
            std::cout<<r->data<<" ";
            r = r->next;
        }
        std::cout<<"]"<<std::endl;
    }

    //检测元素是否存在
    bool isExsit(T x){
```

```
        Node<T> *r = first->next;
        bool b = true;
        while(r && r->data != x){
            r = r->next;
        }
        if(!r){
            b = false;
        }
        return b;
    }

//添加或删除存在元素（存在则删除，不存在则添加）
void findAddOrDelete(T x){
    Node<T> *r = first->next, *p=first, *t;
    while(r && r->data != x){
        p = r;
        r = r->next;
    }
    if(!r){
        insertData(x);
    }else{
        //因为序列是有序的，删除连续相同的x
        while(r && r->data == x){
            t = r;
            p->next = t->next;
            delete t;
            r = p->next;
            len--;
        }
    }
}

//有序插入元素
void insertData(T x){
    Node<T> *t = first->next, *p=first;
    Node<T> *r = new Node<T>;
```

```
        r->data = x;
        r->next = NULL;
        while(t && t->data < x){
            p = t;
            t = t->next;
        }
        p->next = r;
        r->next = t;
        len++;
    }

private:
    Node<T>* first;
    int len;//存储链表长度
    void release(){
        Node<T> *r = first;
        while(r->next){
            Node<T> *t = r->next;
            r->next = t->next;
            delete t;
        }
        first->next = NULL;
    }

};

#endif
```

main.cpp

```
#include <iostream>
#include "list.h"
using namespace std;

int main( ){
```



```
int a[] = {3,11,2,8,4,5,3,6,8,8,8};

List<int> l(a, 11);
cout<<"length:"<<l.length()<<endl;
l.printList();
cout<<l.isExsit(32)<<" "<<l.isExsit(8)<<endl;
l.findAddOrDelete(7);
cout<<"length:"<<l.length()<<endl;
l.printList();
l.findAddOrDelete(8);
cout<<"length:"<<l.length()<<endl;
l.printList();

return 0;
}
```

3.总结

难度没什么，主要是考察链表的基本操作，边界检查，尤其是使用指针的时候尤其注意，指针检查！

4.删除



```
while(r && r->data == x){
    t = r;
    p->next = t->next;
    delete t;
    r = p->next;
    len--;
}
```

5.9 vector的基本实现 (c++)

发表时间: 2012-06-27 关键字: vector, 实现

1.描述

vector的基本操作实现 (包括迭代器)

2.基本操作

3.代码

```
#ifndef DS_EXCEPTIONS_H
#define DS_EXCEPTIONS_H

class UnderflowException { };
class IllegalArgumentException { };
class ArrayIndexOutOfBoundsException { };
class IteratorOutOfBoundsException { };
class IteratorMismatchException { };
class IteratorUninitializedException { };

#endif
```

```
#ifndef VECTOR_H
#define VECTOR_H

#include <cstdio>
#include "dsexceptions.h"

template <typename Object>
```

```
class Vector
{
public:
    explicit Vector( int initSize = 0 )
        : theSize( initSize ), theCapacity( initSize + SPARE_CAPACITY ), currentIndex(0)
        { objects = new Object[ theCapacity ]; }
    Vector( const Vector & rhs ) : objects( NULL )
        { operator=( rhs ); }
    ~Vector( )
        { delete [ ] objects; }

    bool empty( ) const
        { return size( ) == 0; }
    int size( ) const
        { return theSize; }
    int capacity( ) const
        { return theCapacity; }

    //操作符[]重载
    Object & operator[]( int index )
    {
        #ifndef NO_CHECK
        if( index < 0 || index >= size( ) )
            throw ArrayIndexOutOfBoundsException( );
        #endif

        return objects[ index ];
    }

    //const操作符[]重载
    const Object & operator[]( int index ) const
    {
        #ifndef NO_CHECK
        if( index < 0 || index >= size( ) )
            throw ArrayIndexOutOfBoundsException( );
        #endif

        return objects[ index ];
    }

    //操作符=重载，实现对象复制
```

```
const Vector & operator= ( const Vector & rhs )
{
    if( this != &rhs )
    {
        delete [ ] objects;
        theSize = rhs.size( );
        theCapacity = rhs.theCapacity;

        objects = new Object[ capacity( ) ];
        for( int k = 0; k < size( ); k++ )
            objects[ k ] = rhs.objects[ k ];
    }
    return *this;
}

//重新设置vector尺寸 ( 两倍 )
void resize( int newSize )
{
    if( newSize > theCapacity )
        reserve( newSize * 2 );
    theSize = newSize;
}

//设置vector尺寸 ( 修改大小, 并拷贝所有元素到新的数组, 并销毁原来的数组 )
void reserve( int newCapacity )
{
    Object *oldArray = objects;

    //1.重设size,capacity
    int numToCopy = newCapacity < theSize ? newCapacity : theSize;
    //capacity是在size的基础上加一个常数
    newCapacity += SPARE_CAPACITY;

    //2.建立新的数组, 并拷贝元素到新数组
    objects = new Object[ newCapacity ];
    for( int k = 0; k < numToCopy; k++ )
        objects[ k ] = oldArray[ k ];

    theSize = numToCopy;
    theCapacity = newCapacity;

    //删除原来的数组
}
```

```
        delete [ ] oldArray;
    }

    // Stacky stuff
    //放置元素，一旦容量达到最大就重置大小
    void push_back( const Object & x )
    {
        if( theSize == theCapacity )
            reserve( 2 * theCapacity + 1 );
        objects[ theSize++ ] = x;
    }

    //删除最后元素
    void pop_back( )
    {
        if( empty( ) )
            throw UnderflowException( );
        theSize--;
    }

    //最后元素
    const Object & back ( ) const
    {
        if( empty( ) )
            throw UnderflowException( );
        return objects[ theSize - 1 ];
    }
}
```

//为了遍历vector中的元素，实现一个iterator迭代器(迭代器其实就是指向数组的指针，并对指

```
    // Iterator stuff: not bounds checked
    typedef Object * iterator;
    typedef const Object * const_iterator;

    iterator begin( ){
        currentIndex = 0;
        return &objects[ currentIndex ];
    }
}
```

```
const_iterator begin( ) const{
    currentIndex = 0;
    return &objects[ currentIndex ];
}
iterator end( ){
    return &objects[size( )];
}
const_iterator end( ) const{
    return &objects[size( )];
}
iterator next(){
    currentIndex++;
    if( currentIndex < 0 || currentIndex > size( ) )
        throw ArrayIndexOutOfBoundsException( );
    return &objects[currentIndex];
}
//插入操作
void insert(int position, const Object & x ){
    //从这里我们也可以看出，插入操作会导致迭代器失效的原因所在（从新分配了空间，原迭代器变成了悬垂指针）
    if( theSize == theCapacity )
        reserve( 2 * theCapacity + 1 );
    if(position > theSize || position < 0){
        throw ArrayIndexOutOfBoundsException( );
    }else{
        for(int i=theSize-1; i>=position; i--){
            objects[i+1] = objects[i];
        }
        objects[ position ] = x;
        theSize++;
    }
}
//插入（利用迭代器）
iterator insert(iterator position, const Object& x ){
    if( theSize == theCapacity )
        reserve( 2 * theCapacity + 1 );
    if(position<begin() || position>end())
        throw ArrayIndexOutOfBoundsException( );
```

```
    iterator ite = end();
    while(ite != position){
        *ite = *(ite-1);
        ite--;
    }
    *position = x;
    theSize++;
    return begin();
}
```

//擦除从指定位置起数据

```
iterator erase(iterator position){
    if(position<begin() || position>end())
        throw ArrayIndexOutOfBoundsException( );
    while(position != end()){
        theSize--;
    }
    return begin();
}
```

//擦除从指定范围内的数据

```
iterator erase(iterator first, iterator last){
    if((first < begin() || first>end()) || (last < begin() || last>end()) || first>last)
        throw ArrayIndexOutOfBoundsException( );
    if(last == end())        erase(first);
    //1.先将指定范围内的数据覆盖
    int i=1;
    while(first != last+1){
        *first = *(last+i);
        first++;
        theSize--;
        i++;
    }
    //2.将覆盖所用数据剩余部分前移 ( 如1,2,3,4,5,6要删除2,3先用4,5覆盖2,3-->1,4,5,4,5,6-->然后6前移)
    i--;
    while(last <= end()){
        last++;
        *last = *(last+i);
    }
}
```



```
    }
    return begin();
}

bool isEnd(){
    if(currentIndex < 0 || currentIndex >= size( ))
        return true;
    return false;
}

//操作符++重载(前增量, ++a), 注还有后增量的重载差不多, 只是Object & operator++(int a)有参数
Object & operator++()
{
    currentIndex++;
    if( currentIndex < 0 || currentIndex > size( ) )
        throw ArrayIndexOutOfBoundsException( );
    return objects[ currentIndex ];
}

//操作符--重载
Object & operator--()
{
    currentIndex--;
    if( currentIndex < 0 || currentIndex > size( ) )
        throw ArrayIndexOutOfBoundsException( );
    return objects[ currentIndex ];
}

enum { SPARE_CAPACITY = 16 };

private:
    int theSize;
    int theCapacity;
    int currentIndex;//仅用于iterator
    Object * objects;
};

#endif
```

```
#include "vector.h"
#include <iostream>
using namespace std;

int main( )
{
    Vector<int> v;

    for( int i = 0; i < 10; i++ )
        v.push_back( i );

    for( int i = 0; i < 10; i++ )
        cout << v[ i ] << " ";

    //利用操作符--输出
    Vector<int>::iterator t = v.end()-1;//注意end()返回的是最后元素之后的位置
    cout<<endl;
    for(; t >= v.begin(); t--){
        cout<< *t <<" ";
    }

    cout<<endl;
    Vector<int>::iterator k = v.begin();
    while(!v.isEnd()){
        cout<< *k <<" ";
        k = v.next();
    }

    Vector<int>::iterator m = v.begin()+6;
    //m = v.erase(m);//正确用法
    v.erase(m);      //这样是错误的，会使当前迭代器m失效，必须返回新的迭代器，像上面那样用
    cout<<"\nerase:"<<*m<<endl;
```

```
for(m=v.begin(); m != v.end(); ){
    cout<< *m <<" ";
    m = v.next();
}

Vector<int>::iterator mm = v.begin()+2;
mm = v.insert(mm, 5);
cout<<endl;
for(; mm != v.end(); ++mm){
    cout<< *mm <<" ";
}

return 0;
}
```

4.从上面的迭代器实现，可以看出，为了方便指针操作，使用了迭代器封装指针操作，这样使得指针操作变得简单

明白两点：a.迭代器的原理，从上面代码可以看出，就是指针操作

b.迭代器自增原理，当size=capacity的时候，长度会增长一倍

c.size和capacity的区别？为什么用capacity？（使用capacity是为了避免空间浪费，譬如size=20，capacity=30,如果没有capacity

那么大小为21的数据就会使用41个空间的长度（当插入最后一个元素的时候从新分配空间），而有了capacity就不用重新分配）