小程序

1. 小程序的框架的核心是一个响应的数据绑定系统。整个系统分为两块视图层（View）和逻辑层（App Service）。框架可以让数据与视图非常简单地保持同步。当做数据修改的时候，只需要在逻辑层修改数据，视图层就会做相应的更新。（视图层、逻辑层）
2. 框架管理了整个小程序的页面路由，可以做到页面间的无缝切换，并给以页面完整的生命周期。开发者需要做的只是将页面的数据，方法，生命周期函数注册进框架中，其他的一切复杂的操作都交由框架处理。
3. 小程序包含一个描述整体程序的 app 和多个描述各自页面的 page。一个小程序主体部分由三个文件组成，必须放在项目的根目录，如下：（一个 app、多个 page）

- app.js：小程序逻辑　　（逻辑、公共设置、公共样式表）
- app.json：小程序公共设置
- app.wxss：小程序公共样式表

4. 一个小程序页面由四个文件组成，分别是：（小程序页面、四个文件）

- js：页面逻辑
- wxml：页面结构，类似 HTML 文件　　（wxml、HTML 文件）
- wxss：页面样式表，类似 CSS 文件　　（wxss、CSS 文件）
- json：页面配置

5. app.json 文件用来对微信小程序进行全局配置，决定页面文件的路径、窗口表现、设置网络超时时间、设置多 tab 等。例如：（app.json、全局配置）

```
{
  "pages":[
    "pages/index/index",
    "pages/logs/logs"
  ]
}
```

小程序由 index 页面和 logs 页面组成。

6. 小程序开发框架的逻辑层由 JavaScript 编写。在 JavaScript 的基础上，我们做了一些修改，以方便地开发小程序。（逻辑层、Javascript 编写）

- 增加 App 和 Page 方法，进行程序和页面的注册。
- 增加 getApp 和 getCurrentPages 方法，分别用来获取 App 实例和当前页面栈。
- 提供丰富的 API，如微信用户数据，扫一扫，支付等微信特有能力。

- 每个页面有独立的作用域，并提供模块化能力。
- 由于框架并非运行在浏览器中，所以 JavaScript 在 web 中一些能力都无法使用，如 document，window 等。
- 开发者写的所有代码最终将会打包成一份 JavaScript，并在小程序启动的时候运行，直到小程序销毁。类似 ServiceWorker，所以逻辑层也称之为 App Service。

7. App() 函数用来注册一个小程序。接受一个 object 参数，其指定小程序的生命周期函数等。（APP、注册小程序）

```
App({
  onLaunch: function(options) {
    // Do something initial when launch.
  },
  onShow: function(options) {
      // Do something when show.
  },
  onHide: function() {
      // Do something when hide.
  },
  onError: function(msg) {
    console.log(msg)
  },
  globalData: 'I am global data'
})
```

8. getApp() 函数可以用来获取到小程序实例。（getApp()、获取、实例）

```
var appInstance = getApp()
console.log(appInstance.globalData) // I am global data
```

9. Page() 函数用来注册一个页面。接受一个 object 参数，其指定页面的初始数据、生命周期函数、事件处理函数等。（Page()、注册页面）

```
Page({
  data: {
    text: 'init data',
    array: [{msg: '1'}, {msg: '2'}]
  }
})
```

10. getCurrentPages() 函数用于获取当前页面栈的实例，以数组形式按栈的顺序给出，第一个元素为首页，最后一个元素为当前页面。（getCurrentPages()、获取、当前页面栈）

11. 在 JavaScript 文件中声明的变量和函数只在该文件中有效；不同的文件中可以声明相同名字的变量和函数，不会互相影响。通过全局函数 getApp() 可以获取全局的应用实例，如果需要全局的数据可以在 App() 中设置，如：（变量和函数、该文件有效、全局数据、App()中设置）

```
// app.js
App({
  globalData: 1
})
// a.js
// The localValue can only be used in file a.js.
var localValue = 'a'
// Get the app instance.
var app = getApp()
// Get the global data and change it.
app.globalData++
// b.js
// You can redefine localValue in file b.js, without interference with the localValue in a.js.
var localValue = 'b'
// If a.js it run before b.js, now the globalData shoule be 2.
console.log(getApp().globalData)
```

12. 可以将一些公共的代码抽离成为一个单独的 js 文件，作为一个模块。模块只有通过 module.exports 或者 exports 才能对外暴露接口。（module.exports、exports、暴露接口）

```
// common.js
function sayHello(name) {
  console.log(`Hello ${name} !`)
}
function sayGoodbye(name) {
  console.log(`Goodbye ${name} !`)
}

module.exports.sayHello = sayHello
exports.sayGoodbye = sayGoodbye
```

在需要使用这些模块的文件中，使用 require(path) 将公共代码引入：（require(path)、引入）

```
var common = require('common.js')
Page({
  helloMINA: function() {
    common.sayHello('MINA')
  },
  goodbyeMINA: function() {
```

```
    common.sayGoodbye('MINA')
  }
})
```

13. 框架的视图层由 WXML 与 WXSS 编写，由组件来进行展示。将逻辑层的数据反应成视图，同时将视图层的事件发送给逻辑层。

- WXML(WeiXin Markup language) 用于描述页面的结构。类似 HTML 文件。
- WXS(WeiXin Script) 是小程序的一套脚本语言，结合 WXML，可以构建出页面的结构。
- WXSS(WeiXin Style Sheet) 用于描述页面的样式。类似 CSS 文件。
- 组件(Component)是视图的基本组成单元。

14. 数据绑定使用 Mustache 语法（双大括号）将变量包起来，可以作用于：（双大括号、变量）

- 内容

```
<view> {{ message }} </view>
```

```
Page({
  data: {
    message: 'Hello MINA!'
  }
})
```

- 组件属性(需要在双引号之内)

```
<view id="item-{{id}}"> </view>
```

```
Page({
  data: {
    id: 0
  }
})
```

- 控制属性(需要在双引号之内)

```
<view wx:if="{{condition}}"> </view>
```

```
Page({
  data: {
    condition: true
  }
})
```

- 关键字(需要在双引号之内)

```
<checkbox checked="{{false}}"> </checkbox>
```

15. 可以在 {{}} 内进行简单的运算，支持的有如下几种方式：（{{}}、简单运算）

- 三元运算
- 算数运算
- 逻辑判断
- 字符串运算
- 数据路径运算

16. 也可以在 Mustache 内直接进行组合，构成新的对象或者数组：（Mustache 内、组合）

- 数组

```
<view wx:for="{{[zero, 1, 2, 3, 4]}}"> {{item}} </view>
```

```
Page({
  data: {
    zero: 0
  }
})
```

- 对象

```
<template is="objectCombine" data="{{for: a, bar: b}}"></template>
```

```
Page({
  data: {
    a: 1,
    b: 2
  }
```

```
})
```

17. wx:for="{{list}}"用来循环数组。默认数组的当前项的下标变量名默认为 index，数组当前项的变量名默认为 item。（wx:for="{{list}}"、循环数组）

- 使用 wx:for-item 可以指定数组当前元素的变量名。（wx:for-item 、变量名）
- 使用 wx:for-index 可以指定数组当前下标的变量名：（wx:for-index、下标）

```
<view wx:for="{{[1, 2, 3, 4, 5, 6, 7, 8, 9]}}" wx:for-item="i">
  <view wx:for="{{[1, 2, 3, 4, 5, 6, 7, 8, 9]}}" wx:for-item="j">
    <view wx:if="{{i <= j}}">
      {{i}} * {{j}} = {{i * j}}
    </view>
  </view>
</view>
```

18. 如果列表中项目的位置会动态改变或者有新的项目添加到列表中，并且希望列表中的项目保持自己的特征和状态（如 <input/> 中的输入内容，<switch/> 的选中状态），需要使用 wx:key 来指定列表中项目的唯一的标识符。（wx:key、指定、标识符）

```
<switch wx:for="{{objectArray}}" wx:key="unique" style="display: block;"> {{item.id}} </switch>
<button bindtap="switch"> Switch </button>
<button bindtap="addToFront"> Add to the front </button>

<switch wx:for="{{numberArray}}" wx:key="*this" style="display: block;"> {{item}} </switch>
<button bindtap="addNumberToFront"> Add to the front </button>
```

19. 在框架中，使用 wx:if="{{condition}}" 来判断是否需要渲染该代码块，类似其他语言的 if 语句：（wx:if、if 语句）

```
<view wx:if="{{condition}}"> True </view>
```

也可以用 wx:elif 和 wx:else 来添加一个 else 块：

```
<view wx:if="{{length > 5}}"> 1 </view>
<view wx:elif="{{length > 2}}"> 2 </view>
<view wx:else> 3 </view>
```

20. 因为 wx:if 是一个控制属性，需要将它添加到一个标签上。如果要一次性判断多个组件标签，可以使用一个 <block/> 标签将多个组件包装起来，并在上边使用 wx:if 控制属性。（wx:if、多个属性、block 标签、包装起来）

```
<block wx:if="{{true}}">
  <view> view1 </view>
  <view> view2 </view>
</block>
```

<block/> 并不是一个组件，它仅仅是一个包装元素，不会在页面中做任何渲染，只接受控制属性。

21. 类似 block wx:if，也可以将 wx:for 用在<block/>标签上，以渲染一个包含多节点的结构块。例如：

```
<block wx:for="{{[1, 2, 3]}}">
  <view> {{index}}: </view>
  <view> {{item}} </view>
</block>
```

22. wxml 提供模板（template），可以在模板中定义代码片段，然后在不同的地方调用。使用<template/>定义模块，使用 name 属性，作为模板的名字：（<template>、定义模板、name 属性、模块名字）

```
<!--
  index: int
  msg: string
  time: string
-->
<template name="msgItem">
  <view>
    <text> {{index}}: {{msg}} </text>
    <text> Time: {{time}} </text>
  </view>
</template>
```

23. 使用 is 属性声明需要的使用的模板，然后将模板所需要的 data 传入，如：（<template>、is、声明、需要使用）

```
<template is="msgItem" data="{{...item}}"/>
```

```
Page({
  data: {
    item: {
      index: 0,
      msg: 'this is a template',
      time: '2016-09-15'
```

```
    }
  }
})
```

24. is 属性可以使用 Mustache 语法，来动态决定具体需要渲染哪个模板：（is 属性、Mustache 语法）

```
<template name="odd">
  <view> odd </view>
</template>
<template name="even">
  <view> even </view>
</template>

<block wx:for="{{[1, 2, 3, 4, 5]}}">
    <template is="{{item % 2 == 0 ? 'even' : 'odd'}}"/>
</block>
```

25. 事件的使用方式：

- 在组件中绑定一个事件处理函数：（组件中绑定）

```
<view id="tapTest" data-hi="WeChat" bindtap="tapName"> Click me! </view>
```

- 在相应的 Page 定义中写上相应的事件处理函数，参数是 event：（Page 中写上、事件处理函数）

```
Page({
  tapName: function(event) {
    console.log(event)
  }
})
```

26. 事件绑定的写法同组件的属性，以 key、value 的形式。（事件绑定、bind 或 catch、value、函数）

- key 以 bind 或 catch 开头，然后跟上事件的类型，如 bindtap、catchtouchstart。自基础库版本 1.5.0 起，bind 和 catch 后可以紧跟一个冒号，其含义不变，如 bind:tap、catch:touchstart。
- value 是一个字符串，需要在对应的 Page 中定义同名的函数。不然当触发事件的时候会报错。

27. bind 事件绑定不会阻止冒泡事件向上冒泡，catch 事件绑定可以阻止冒泡事件向上冒泡。如在下边这个例子中，点击 inner view 会先后调用 handleTap3 和 handleTap2：（catch、阻止冒泡）

```
<view id="outer" bindtap="handleTap1">
  outer view
  <view id="middle" catchtap="handleTap2">
    middle view
    <view id="inner" bindtap="handleTap3">
      inner view
    </view>
  </view>
</view>
```

28. WXML 提供两种文件引用方式 import 和 include：（文件引用、import 和 include）

(1) import 可以在该文件中使用目标文件定义的 template，如：在 item.wxml 中定义了一个叫 item 的 template：

```
<!-- item.wxml -->
<template name="item">
  <text>{{text}}</text>
</template>
```

在 index.wxml 中引用了 item.wxml，就可以使用 item 模板：

```
<import src="item.wxml"/>
<template is="item" data="{{text: 'forbar'}}"/>
```

import 有作用域的概念，即只会 import 目标文件中定义的 template，而不会 import 目标文件 import 的 template。（import、不会）

(2) include 可以将目标文件除了 <template/> <wxs/> 外的整个代码引入，相当于是拷贝到 include 位置，如：

```
<!-- index.wxml -->
<include src="header.wxml"/>
<view> body </view>
<include src="footer.wxml"/>
<!-- header.wxml -->
<view> header </view>
<!-- footer.wxml -->
<view> footer </view>
```

29. WXS 代码可以编写在 wxml 文件中的 <wxs> 标签内，或以 .wxs 为后缀名的文件内。（WXS 代码、<wxs>标签或.wxs 后缀）

30. 每一个 .wxs 文件和 <wxs> 标签都是一个单独的模块。每个模块都有自己独立的作用域。即在一个模块里面定义的变量与函数，默认为私有的，对其他模块不可见。一个模块要想对外暴露其内部的私有变量与函数，只能通过 module.exports 实现。（ .wxs 文件和 <wxs> 标签、单独的模块）
31. 每个 wxs 模块均有一个内置的 module 对象。（wxs 模块、module 对象）

```
// /pages/tools.wxs

var foo = "'hello world' from tools.wxs";
var bar = function (d) {
  return d;
}
module.exports = {
  FOO: foo,
  bar: bar,
};
module.exports.msg = "some msg";
```

```
<!-- page/index/index.wxml -->

<wxs src="./../tools.wxs" module="tools" />
<view> {{tools.msg}} </view>
<view> {{tools.bar(tools.FOO)}} </view>
```

页面输出：

```
some msg
'hello world' from tools.wxs
```

32. 在.wxs 模块中引用其他 wxs 文件模块，可以使用 require 函数。（引用 wxs 文件、require 函数）
33. <wxs> 标签属性：

- module：当前<wxs> 标签的模块名。必填字段。
- src：引用.wxs 文件的相对路径。仅当本标签为单闭合标签或标签的内容为空时有效。

34. 小程序的使用 var 关键字来定义变量：

```
var foo = 1;
var bar = "hello world";
var i; // i === undefined
```

35. WXS 主要有 3 种注释的方法。（wxs、3 种注释）

- //：定义单行注释
- /* */：定义多行注释
- /*：结尾注释。即从 /* 开始往后的所有 WXS 代码均被注释

36. WXS 使用===代表全等：（===、全等）

```
console.log(30 === a + b);
```

37. 在 WXS 中，可以使用以下格式的 if 语句：（if 语句）

- if (expression) statement
- if (expression) statement1 else statement2
- if ... else if ... else statementN

38. 小程序还支持 switch 语句、for 语句和 while 语句，使用语法基本和 c++差不多。（switch 语句、for 语句、while 语句）

39. WXS 语言目前共有以下几种数据类型：

- number： 数值
- string： 字符串
- boolean：布尔值
- object：对象
- function：函数
- array : 数组
- date：日期
- regexp：正则

40. console.log 方法用于在 console 窗口输出信息。它可以接受多个参数，将它们的结果连接起来输出。（console.log 方法、输出信息）

41. 数据类型的判断可以使用 constructor 属性。（数据类型判断、 constructor 属性）

```
var number = 10;
console.log( "Number" === number.constructor );
```

42. WXSS 用来决定 WXML 的组件应该怎么显示。为了适应广大的前端开发者，WXSS 具有 CSS 大部分特性。同时为了更适合开发微信小程序，WXSS 对 CSS 进行了扩充以及修改。

43. 使用@import 语句可以导入外联样式表，@import 后跟需要导入的外联样式表的相对路径，用;表示语句结束。（@import、导入、外联样式表）

```
/** common.wxss **/
.small-p {
  padding:5px;
}
```

```
/** app.wxss **/
@import "common.wxss";
.middle-p {
  padding:15px;
}
```

44. 框架组件上支持使用 style、class 属性来控制组件的样式。（style、class 属性、控制组件样式）

- style：静态的样式统一写到 class 中。style 接收动态的样式，在运行时会进行解析，请尽量避免将静态的样式写进 style 中，以免影响渲染速度。

```
<view style="color:{{color}};" />
```

- class：用于指定样式规则，其属性值是样式规则中类选择器名(样式类名)的集合，样式类名不需要带上.，样式类名之间用空格分隔。

```
<view class="normal_view" />
```

45. 定义在 app.wxss 中的样式为全局样式，作用于每一个页面。在 page 的 wxss 文件中定义的样式为局部样式，只作用在对应的页面，并会覆盖 app.wxss 中相同的选择器。（app.wxss、全局样式）

46. 类似于页面，一个自定义组件由 json、wxml、wxss、js 4 个文件组成。编写一个自定义组件：

(1) 首先需要在 json 文件中进行自定义组件声明（将 component 字段设为 true 可这一组文件设为自定义组件）：

```
{
```

```
  "component": true
}
```

(2) 实现界面排版（wxml 文件夹）和样式（wxss 文件）。

```
<!-- 这是自定义组件的内部 WXML 结构 -->
<view class="inner">
  {{innerText}}
</view>
<slot></slot>
```

```
/* 这里的样式只应用于这个自定义组件 */
.inner {
  color: red;
}
```

(3) 在 js 文件中实现属性定义以及组件事件：

```
Component({
  properties: {
    // 这里定义了 innerText 属性，属性值可以在组件使用时指定
    innerText: {
      type: String,
      value: 'default value',
    }
  },
  data: {
    // 这里是一些组件内部数据
    someData: {}
  },
  methods: {
    // 这里是一个自定义方法
    customMethod: function(){}
  }
})
```

47. 使用自定义组件：

(1) 在 component.json 中声明要引用自定义组件

```
{
  "usingComponents": {
    "component-tag-name": "path/to/the/custom/component"
  }
```

```
}
```

(2) 然后就可以在 component.wxml 布局中使用组件

```
<view>
  <!-- 以下是对一个自定义组件的引用 -->
  <component-tag-name inner-text="Some text"></component-tag-name>
</view>
```

48. 在组件 wxss 中不应使用 ID 选择器、属性选择器和标签名选择器。（组件、不使用、ID 选择器、属性和标签选择器）

49. 在自定义组件的 js 文件中，需要使用 Component() 来注册组件，并提供组件的属性定义、内部数据和自定义方法。（Component()、注册组件）

```
Component({
  properties: {
    // 这里定义了 innerText 属性，属性值可以在组件使用时指定
    innerText: {
      type: String,
      value: 'default value',
    }
  },
  data: {
    // 这里是一些组件内部数据
    someData: {}
  },
  methods: {
    // 这里是一个自定义方法
    customMethod: function(){}
  }
})
```

50. 使用已注册的自定义组件前，首先要在页面的 json 文件中进行引用声明。此时需要提供每个自定义组件的标签名和对应的自定义组件文件路径：（使用自定义组件、引用声明）

```
{
  "usingComponents": {
    "component-tag-name": "path/to/the/custom/component"
  }
}
```

51. 在组件模板中可以提供一个 <slot> 节点，用于承载组件引用时提供的子节点。（<slot> 节点）

```
<!-- 组件模板 -->
<view class="wrapper">
  <view>这里是组件的内部节点</view>
  <slot></slot>
</view>
```

```
<!-- 引用组件的页面模版 -->
<view>
  <component-tag-name>
    <!-- 这部分内容将被放置在组件 <slot> 的位置上 -->
    <view>这里是插入到组件 slot 中的内容</view>
  </component-tag-name>
</view>
```

52. 默认情况下，一个组件的 wxml 中只能有一个 slot。需要使用多 slot 时，可以在组件 js 中声明启用。（多个 slot、声明启用）

```
Component({
  options: {
    multipleSlots: true // 在组件定义时的选项中启用多 slot 支持
  },
  properties: { /* ... */ },
  methods: { /* ... */ }
})
```

此时，可以在这个组件的 wxml 中使用多个 slot，以不同的 name 来区分。

```
<!-- 组件模板 -->
<view class="wrapper">
  <slot name="before"></slot>
  <view>这里是组件的内部细节</view>
  <slot name="after"></slot>
</view>
```

53. 编写组件样式时，需要注意以下几点：

- 组件和引用组件的页面不能使用 id 选择器（#a）、属性选择器（[a]）和标签名选择器，请改用 class 选择器。
- 组件和引用组件的页面中使用后代选择器（.a .b）在一些极端情况下会有非预期的表现，如遇，请避免使用。

- 子元素选择器（.a>.b）只能用于 view 组件与其子节点之间，用于其他组件可能导致非预期的情况。
- 继承样式，如 font 、 color ，会从组件外继承到组件内。
- 除继承样式外， app.wxss 中的样式、组件所在页面的的样式对自定义组件无效。

54. Component 构造器可用于定义组件，调用 Component 构造器时可以指定组件的属性、数据、方法等。
55. 事件系统是组件间交互的主要形式。自定义组件可以触发任意的事件，引用组件的页面可以监听这些事件。监听自定义组件事件的方法与监听基础组件事件的方法完全一致。
56. behaviors 是用于组件间代码共享的特性。（behaviors 、共享的特性）
57. 每个 behavior 可以包含一组属性、数据、生命周期函数和方法，组件引用它时，它的属性、数据和方法会被合并到组件中，生命周期函数也会在对应时机被调用。每个组件可以引用多个 behavior 。 behavior 也可以引用其他 behavior 。behavior 需要使用 Behavior() 构造器定义。（定义 behavior、Behavior() ）

```
// my-behavior.js
module.exports = Behavior({
  behaviors: [],
  properties: {
    myBehaviorProperty: {
      type: String
    }
  },
  data: {
    myBehaviorData: {}
  },
  attached: function(){},
  methods: {
    myBehaviorMethod: function(){}
  }
})
```

组件引用时，在 behaviors 定义段中将它们逐个列出即可。（使用、逐个列出）

```
// my-component.js
var myBehavior = require('my-behavior')
Component({
  behaviors: [myBehavior],
  properties: {
    myProperty: {
      type: String
    }
```

```
  },
  data: {
    myData: {}
  },
  attached: function(){},
  methods: {
    myMethod: function(){}
  }
})
```

58. 自定义组件可以通过引用内置的 behavior 来获得内置组件的一些行为。

```
Component({
  behaviors: ['wx://form-field']
})
```

59. 类似 HTML 的 ul 标签和 li 标签这样的标签关系可以使用 relations 字段定义。（标签关系、relations 字段字义）
60. 有时，自定义组件模版中的一些节点，其对应的自定义组件不是由自定义组件本身确定的，而是自定义组件的调用者确定的。这时可以把这个节点声明为“抽象节点”。使用 componentGenerics 字段定义抽象节点。（抽象节点、componentGenerics 字段定义）
61. 抽象节点可以指定一个默认组件，当具体组件未被指定时，将创建默认组件的实例。默认组件可以在 componentGenerics 字段中指定：

```
{
  "componentGenerics": {
    "selectable": {
      "default": "path/to/default/component"
    }
  }
}
```

62. 插件是对一组 js 接口或自定义组件的封装，用于提供给第三方小程序调用。插件必须嵌入在其他小程序中才能被用户使用。（插件、第三方、调用）
63. 在构建小程序分包项目时，构建会输出一个或多个功能的分包，其中每个分包小程序必定含有一个主包，所谓的主包，即放置默认启动页面/TabBar 页面，以及一些所有分包都需用到公共资源/JS 脚本，而分包则是根据开发者的配置进行划分。（分包）
64. 在小程序启动时，默认会下载主包并启动主包内页面，如果用户需要打开分包内某个页面，客户端会把对应分包下载下来，下载完成后再进行展示。

65. 假设支持分包的小程序目录结构如下：

```
├── app.js
├── app.json
├── app.wxss
├── packageA
│   └── pages
│       ├── cat
│       └── dog
├── packageB
│   └── pages
│       ├── apple
│       └── banana
├── pages
│   ├── index
│   └── logs
└── utils
```

开发者通过在 app.json subPackages 字段声明项目分包结构：

```
{
  "pages":[
    "pages/index",
    "pages/logs"
  ],
  "subPackages": [
    {
      "root": "packageA",
      "pages": [
        "pages/cat",
        "pages/dog"
      ]
    }, {
      "root": "packageB",
      "pages": [
        "pages/apple",
        "pages/banana"
      ]
    }
  ]
}
```

66. 一些异步处理的任务，可以放置于 Worker 中运行，待运行结束后，再把结果返回到小程序主线程。Worker 运行于一个单独的全局上下文与线程中，不能直接调用主线程的方法。Worker 与主线程之间的数据传输，双方使用 Worker.postMessage() 来发送数据，

Worker.onMessage() 来接收数据，传输的数据并不是直接共享，而是被复制的。（异步处理、Worker.postMessage() 发送数据、Worker.onMessage() 接收数据）

Django

1. Django 模板语言中的 url 标签用于简化 url 链接的书写。（url 标签、简化链接）

HTML 模板文件

```
<a href="/article">资讯</a>
```

Urls.py 文件

```
urlpatterns = patterns('',
    (r'^article$','news_index' ),
)
```

如果需要修改，则需要同时修改模板文件和 urls.py 文件。有 url 标签则可以只需要修改 urls.py 文件即可。（url 标签、只需要修改、urls.py）

HTML 模板文件

```
<a href="{%url 'polls:news_index'%}">资讯</a>
```

Urls.py 文件

```
app_name='polls'
urlpatterns = patterns('',
    url(r'^article$','news_index' ， name="news_index"),
)
```

注意 polls:news_index 的用法。如果 html 文件中包含变量，则直接在后面加上变量：（包含变量、后面加上）

原 HTML 模板文件

```
<li><a href="/polls/{{ question.id }}/">{{ question.question_text }}</a></li>
```

修改后的 HTML 模板文件

```
<li><a href="{% url 'polls:detail' question.id %}">{{ question.question_text }}</a></li>
```

2. 在子 urls.py 文件中：

(1) 使用 app_name 可以给应用添加命名空间：（app_name、命名空间）

```
from django.urls import path
```

```
from . import views

app_name = 'polls'
urlpatterns = [
    path('', views.index, name='index'),
    path('<int:question_id>/', views.detail, name='detail'),
    path('<int:question_id>/results/', views.results, name='results'),
    path('<int:question_id>/vote/', views.vote, name='vote'),
]
```

这个命名空间主要用于在 html 模板中：（用于、html 模板）

polls/templates/polls/index.html

```
<li><a href="{% url 'polls:detail' question.id %}">{{ question.question_text }}</a></li>
```

而主 urls.py 中的 include 函数中填的是文件夹名。（include 函数、文件夹名）

(2) 在带具体的视图的 urls.py，需要将视图文件导入的 urls.py 中：（视图文件、导入）

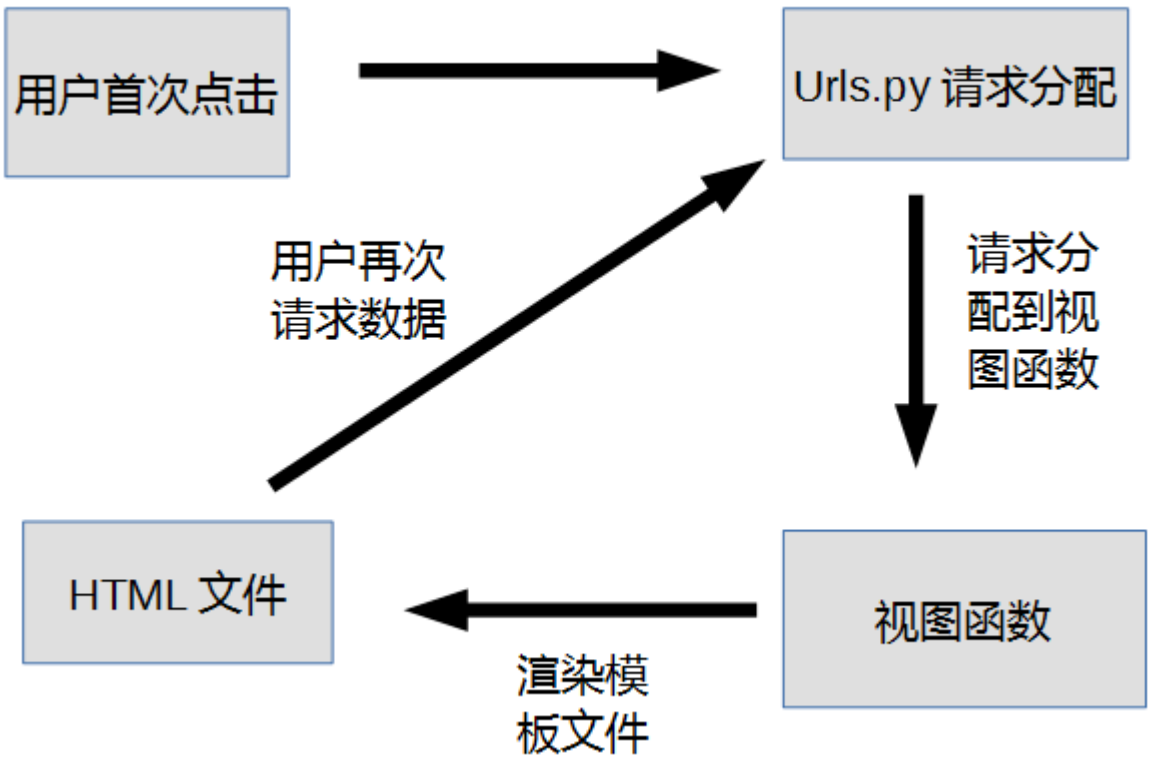
```
from django.urls import path

from . import views

app_name = 'polls'
urlpatterns = [
    path('', views.index, name='index'),
    path('<int:question_id>/', views.detail, name='detail'),
    path('<int:question_id>/results/', views.results, name='results'),
    path('<int:question_id>/vote/', views.vote, name='vote'),
]
```

views 指的是 views.py 文件，导入时省略.py 后缀。（省略.py 后缀）

3. Django 的运行流程如下：



4. 视图函数 render()中有一个可选的参数 context，是一个字典，这个字典的 key 就是 Django 中的变量名，value 就是变量值。

5. url()中使用 include()函数时不能使用类似“^$”格式：（include、不能使用、^$）

```
urlpatterns = [
    url(r'^admin/', admin.site.urls),
    url(r'^$',include('myadmin.urls')),
]
```

而应该直接省略：（直接省略“^$”）

```
urlpatterns = [
    url(r'^admin/', admin.site.urls),
    url(r'',include('myadmin.urls')),
]
```

6. url()中的主页设置不能省略“^$”，否则会出错：（主页、不能省略^$）

```
urlpatterns = [
    url(r'^$',views.index),
    url(r'^blog_login$',views.blog_login),
```

```
]
```

7. 有两种方法来保存静态文件：

- STATIC_URL 方式：该方式把静态文件放置到应用目录之下。
- STATICFILES_DIRS 方式：该方式把所有静态文件统一放置到一个目录里。

8. 静态文件设置如下：

(1) 确保 django.contrib.staticfiles 已经添加到 INSTALLED_APPS。（确保、添加）

(2) 在配置文件中设置 STATIC_URL，例如 STATIC_URL = '/static/'，这是使用 STATIC_URL 方式保存静态文件；或者设置 STATICFILES_DIRS，这是使用 STATICFILES_DIRS 方式来保存静态文件;（设置 STATIC_URL、或设置 STATICFILES_DIRS）

```
STATIC_URL = '/static/'
或者
STATICFILES_DIRS = [
    os.path.join ( BASE_DIR , "static" ),
]
```

(3) 在模板中导入并使用 static 模板标签　　（导入、static 标签）

```
 {% load static %}
<img src = " {% static "my_app/example.jpg" %} " alt = "My image" />
```

9. 与静态文件相关的几个 setting.py 设置：

- STATIC_ROOT：用来保存最终的静态文件。（STATIC_ROOT、保存、静态文件）
- STATICFILES_DIRS：指定了一个工程里面哪个目录存放了与这个工程相关的静态文件。运行 collectstatic 命令，执行下面的命令会将所有的静态文件都拷贝到 STATIC_ROOT 目录下。（STATICFILES_DIRS、哪个目录）

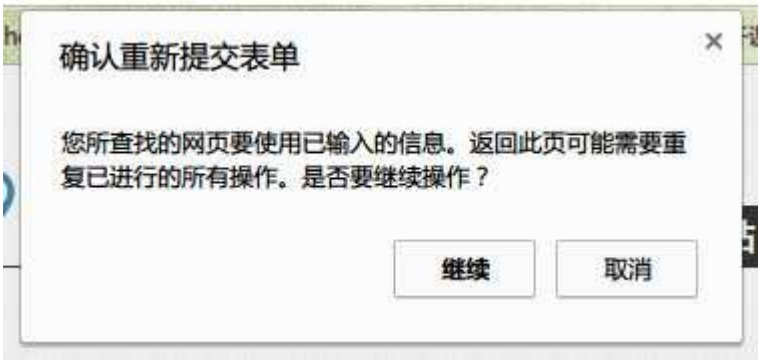
```
$ python manage.py collectstatic
```

在开发的过程中，如果没有指定 STATIC_ROOT 并运行 collectstatic，则即使修改了静态文件，Django 也无法载入，使用的依然是旧版文件。（没有指定 STATIC_ROOT、无法载入）

- STATIC_URL：指定静态目录的 URL，也就是渲染之后 HTML 中静态文件的前缀。（STATIC_URL、指定 URL）

10. Django 模板的 extends 会完全覆盖掉父模板的内容。（extends、覆盖）

11. form 提交后刷新会导致再次提交数据，同时跳出一个重复提交的对话框：



表单提交后对网页进行重定向可以去掉这个对话框，再使用其他方法避免表单重复提交。

12. Django 模型中的字段有个 choices 参数，这个属性可以提供备选数据。如果一个字段设置了这个属性，在模版中如果我要显示这个字段，那么 django 模版系统就会将它默认解析为一个下拉菜单，这样对于一个静态的下拉菜单式很方便的。（choices 属性、备选数据）

```
from django.db import models
class Person(models.Model):
    GENDER_CHOICES = (
        (u'M', u'Male'),
        (u'F', u'Female'),
    )
    name = models.CharField(max_length=60)
    gender = models.CharField(max_length=2, choices=GENDER_CHOICES)
```

使用方式如下：

```
>>> p = Person(name="Fred Flinstone", gender="M")
>>> p.save()
>>> p.gender
u'M'
>>> p.get_gender_display()
u'Male'
```

choices 属性的被选数据类似于字典，字段名返回的是键值，get_xxx_display()返回的是字典值。（字段名、键值、get_xxx_display()、字典值）

13. 字段模型中有 auto_now 和 auto_now_add 两个参数：

- auto_now 无论是创建和修改对象，字段值都会改变。（auto_now、修改、值改变）
- auto_now_add 为创建时的时间，更新对象时不会有变动。（auto_now_add、创建时间）

```
class article(models.Model):
    STATUS_CHOICES = (
        ('d', 'part'),
        ('p', 'Published'),
```

```
)
title = models.CharField('标题', max_length=100)
body = models.TextField('正文')

created_time = models.DateTimeField('创建时间', auto_now_add=True)
# auto_now_add : 创建时间戳，不会被覆盖

last_modified_time = models.DateTimeField('修改时间', auto_now=True)
# auto_now: 自动将当前时间覆盖之前时间
```

14. Model 元数据就是：不是一个字段的任何数据。比如排序选项，admin 选项等等。Django 内置的元数据如下：（元数据、不是字段的数据）

(1) abstract：如果 abstract = True，就表示模型是抽象基类。（abstract、抽象基类）

(2) app_label：这个选项只在一种情况下使用，就是你的模型类不在默认的应用程序包下的 models.py 文件中，这时候你需要指定你这个模型类属于哪个应用程序的。比如你在其他地方写了一个模型类，而这个模型类是属于 myapp 的，那么你这是需要指定为：（app_label、指定、模型所属应用程序）

```
app_label='myapp'
```

(3) db_table：用于指定自定义数据库表名的。Django 有一套默认的按照一定规则生成数据模型对应的数据库表名，如果你想使用自定义的表名，就通过这个属性指定，比如：（db_table、自定义表名）

```
table_name='my_owner_table'
```

(4) db_tablespace：当前模型所使用的数据库表空间的名字。默认值是项目设置中的 DEFAULT_TABLESPACE，如果它存在的话。如果后端并不支持表空间，这个选项可以忽略。（db_tablespace、表空间名字）

(5) get_latest_by：模型中某个可排序的字段的名称，比如 DateField、DateTimeField 或者 IntegerField。它指定了 Manager 的 latest()和 earliest()中使用的默认字段。（get_latest_by、排序字段、latest()和 earliest()中使用）

```
get_latest_by = "order_date"
```

(6) managed：默认为 True，意思是 Django 在 migrate 命令中创建合适的数据表，并且会在 flush 管理命令中移除它们。换句话说，Django 会管理这些数据表的生命周期。（manage、为 true、Django 管理）

(7) order_with_respect_to：按照给定的字段把这个对象标记为“可排序的”。这一属性通常用到关联对象上面，使它在父对象中有序。比如，如果 Answer 和 Question 相关联，一个问题有至少一个答案，并且答案的顺序非常重要，你可以这样做：
（order_with_respect_to、可排序的、关联对象）

```
class Question(models.Model):
    text = models.TextField()
    # ...

class Answer(models.Model):
    question = models.ForeignKey(Question)
    # ...

    class Meta:
        order_with_respect_to = 'question'
```

当 order_with_respect_to 设置之后，模型会提供两个用于设置和获取关联对象顺序的方法：get_RELATED_order() 和 set_RELATED_order()，其中 RELATED 是小写的模型名称。例如，假设一个 Question 对象有很多相关联的 Answer 对象，返回的列表中含有与之相关联 Answer 对象的主键：（提供、get_RELATED_order() 、set_RELATED_order()）

```
>>> question = Question.objects.get(id=1)
>>> question.get_answer_order()
[1, 2, 3]
```

与 Question 对象相关联的 Answer 对象的顺序，可以通过传入一格包含 Answer 主键的列表来设置：

```
>>> question.set_answer_order([3, 1, 2])
```

(8) ordering：这个字段是告诉 Django 模型对象返回的记录结果集是按照哪个字段排序的。比如下面的代码：（ordeing、排序字段）

```
ordering=['order_date']
# 按订单升序排列
ordering=['-order_date']
```

需要注意的是：不论你使用了多少个字段排序，admin 只使用第一个字段。

(9) permissions：设置创建对象时权限表中额外的权限。每个模型都会自动创建增加、删除和修改权限。这个例子指定了一种额外的权限 can_deliver_pizzas：（permissions、添加、额外的权限）

```
permissions = (("can_deliver_pizzas", "Can deliver pizzas"),)
```

(10) default_permissions：默认数据库权限，默认为('add', 'change', 'delete')。你可以自定义这个列表，比如，如果你的应用不需要默认权限中的任何一项，可以把它设置成空列表。在模型被 migrate 命令创建之前，这个属性必须被指定，以防一些遗漏的属性被创建。（default_permissions、默认权限）

15. 每个 QuerySet 都包含一个缓存来最小化对数据库的访问，下面是一个示例：（QuerySet、缓存、赋值给变量）

```
# 下面代码会访问两次数据库
print [blog.title for blog in Blog.objects.all()]
print [blog.content for blog in Blog.objects.all()]

# 下面代码只会访问一次数据库
blogs = Blog.objects.all()
print [blog.title for blog in blogs]
print [blog.content for blog in blogs]
```

- 在一个新的 QuerySet 中，缓存为空。当首次对 QuerySet 的所有实例进行求值时，会将查询结果保存到 QuerySet 的缓冲中。当再访问该 QuerySet 时，会直接从缓冲中取数据。
- 如果只对 QuerySet 的部分实例（query_set[5], query_set[0:10]）进行求值，首先会到 QuerySet 的缓冲中查找是否已经缓存了这些实例，如果有就使用缓存值，如果没有就查询数据库，但是不会将查询结果保存到缓冲中。（部分实例求值、没有、查找、不缓存）

16. QuerySet 是惰性加载的，创建查询集不会访问数据库，只有查询集需要求值时，才会真正运行这个查询。在下面的例子中只有执行 print q 才会真正的去查询数据库。下面是文档中给出的几种会对查询集求值的情况：（QuerySet、惰性加载、创建查询值、不访问）

- 迭代：在首次迭代查询集时会执行数据库查询
- 切片(限制查询集)：对查询集执行切片操作时，指定 step 参数
- 序列化／缓存
- repr：对查询集调用 repr 函数
- len：对查询集调用 len 函数
- list: 对查询集调用 list() 方法强制求值
- bool：测试一个查询集的布尔值，例如使用 bool(), or, and 或者 if 语句都将导致查询集的求值

➢ “创建查询集”指的应该是将一个查询集赋值给一个变量，但还没具体使用的情况。（创建查询集、赋值给变量、未使用）

17. 数据库相关：

- 主键：唯一标识一条记录，不能有重复，不允许为空。（主键、不为空）
- 外键：表的外键是另一表的主键，外键是可以有重复的，可以是空值。外键用来和其他表建立联系用。
- 索引：该字段没有重复值，但可以有一个空值。索引用来提高查询排序的速度

18. select_related()：用于优化关联数据库的查询，适用于一对一字段（OneToOneField）和外键字段（ForeignKey）。对 QuerySet 使用 select_related()函数后，Django 会一次性获、获取相应外键对应的对象，从而在之后需要的时候不必再查询数据库了。例如，假设有以下表，记录各个人的故乡、居住地、以及到过的城市：（select_related()、一次性获取、对应的对象）

```
from django.db import models

class Province(models.Model):
    name = models.CharField(max_length=10)
    def __unicode__(self):
        return self.name

class City(models.Model):
    name = models.CharField(max_length=5)
    province = models.ForeignKey(Province)
    def __unicode__(self):
        return self.name

class Person(models.Model):
    firstname  = models.CharField(max_length=10)
    lastname   = models.CharField(max_length=10)
    visitation = models.ManyToManyField(City, related_name = "visitor")
    hometown   = models.ForeignKey(City, related_name = "birth")
    living     = models.ForeignKey(City, related_name = "citizen")
    def __unicode__(self):
        return self.firstname + self.lastname
```

如果不使用 select_related()：

```
>>> citys = City.objects.all()
>>> for c in citys:
...   print c.province
```

ORM 处理时会一次一次自动查询和 city 表关联的 Province 表，所以这里效率低下，4 个 city 对象要查询 5 次。

使用 select_related()可以一次性获取相应外键对应的对象，用法如下：（select_related()、一次性获取、外键）

```
>>> citys = City.objects.select_related().all()
>>> for c in citys:
...   print c.province
...
```

19. prefetch_related()和 select_related()的设计目的很相似，都是为了减少 SQL 查询的数量，但是实现的方式不一样。后者是通过 JOIN 语句，在 SQL 查询内解决问题。该函数适用于多对多字段和一对多字段。（prefetch_related()、类似 select_related()）

```
>>> zhangs = Person.objects.prefetch_related('visitation').get(firstname=u"张",lastname=u"三")
>>> for city in zhangs.visitation.all() :
...   print city
...
```

20. 在关系模型中，注意添加模型数据和添加关联关系是不一样的。关联管理器的 add()是添加关联关系，而 create()是添加模型数据。（添加模型数据、添加关联关系、不一样）
21. 所谓的添加关联关系，实际上就是在从表上添加外键数据。（添加关联、添加外键）
22. ForeignKey 和 ManyToManyField 也有 blank、null 等字段通用的参数：

```
user = models.ForeignKey(
    User,
    models.SET_NULL,
    blank=True,
    null=True,
)
```

23. ForeignKey 和 ManyToManyField 参数：

(1) ForeignKey：

- on_delete：设置外键关联的对象被删除时，从表执行哪些操作。（on_delete、被删除、执行的操作）
- related_name：用于关联对象反向引用模型的名称。默认为表名的小写，也就是正向、反向添加关联关系中用到的那个从表的对象：

```
host_obj.admin_set.add(*admin_obj)
```

这是反向添加关联关系，如果 related_name 为 myadmin，则中间的 admin 应该改为 myadmin。

- related_query_name：反向关联查询名。用于从目标模型反向过滤模型对象的名称。

- to_field：默认情况下，外键都是关联到被关联对象的主键上（一般为 id）。如果指定这个参数，可以关联到指定的字段上，但是该字段必须具有 unique=True 属性，也就是具有唯一属性。

(2) ManyToManyField

- related_name：参考外键的相同参数。
- related_query_name：参考外键的相同参数。
- through：定义中间表。

24. 对于关系模型而言，根据 ForeignKey 和 ManyToManyField 所在的模型类不同，添加关联关系可以分为正向和反向。注意，这里不是添加模型数据，而是添加关联关系。假设有以下表：（添加关联、正向、反向）

```
# coding:utf-8
from __future__ import unicode_literals
from django.db import models
# Create your models here.
class Host(models.Model):
    ip = models.CharField(max_length=32)
    port = models.IntegerField()

class Admin(models.Model):
    username = models.CharField(max_length=32)
    # 创建多对多字段
    host = models.ManyToManyField(Host)
```

- 正向操作：关系字段是在 Admin 表里定义的，通过 Admin 表来添加关系就是正向操作。

```
host_obj = models.Host.objects.all()
admin_obj = models.Admin.objects.get(username='tom')
admin_obj.host.add(*host_obj)
```

- 反向操作：关系字段是在 Admin 表里定义的，却通过 Host 表来添加关系就是反向操作。

```
host_obj = models.Host.objects.get(ip='4.4.4.4')
admin_obj = models.Admin.objects.filter(id__lt=2)
host_obj.admin_set.add(*admin_obj)
```

通过_set 来实现反向添加操作。admin_set 中的 admin 是 Admin 表名的小写。（_set、反向）

- 主表：有主键的表。（主表、主键）

从表：以主表的主键作为外键的表，也就是有 ForeignKey 和 ManyToManyField 那个表。（从表、外键）

- 正向操作的范式为“主表对象.从表小写.操作函数”，而反向操作的范式为“从表对象.主表小写_set.操作函数”。（正向、从表对象.主表小写.操作函数、反向、主表对象.从表小写_set.操作函数）

25. 添加关联关系应该是添加外键值。（添加关联关系、外键值）
26. 一对一关系典型的例子是 Django 自带 auth 模块的 User 用户表，如果你想在自己的项目里创建用户模型，又想方便的使用 Django 的认证功能，那么一个比较好的方案就是在你的用户模型里，使用一对一关系，添加一个与 auth 模块 User 模型的关联字段。（一对一、User 表）
27. 在多对一模型中，外键要定义在‘多’的一方！（外键、多的一方）
28. "关联管理器"是在一对多或者多对多的关联上下文中使用的管理器。（关联管理器）

- add()：把指定的模型对象添加到关联对象集中。（add()、添加、关联对象）

```
>>> b = Blog.objects.get(id=1)
>>> e = Entry.objects.get(id=234)
>>> b.entry_set.add(e) # Associates Entry e with Blog b.
```

- create()：创建一个新的对象，保存对象，并将它添加到关联对象集之中。返回新创建的对象。（create()、创建对象、添加到关联关系）

```
>>> b = Blog.objects.get(id=1)
>>> e = b.entry_set.create(
...     headline='Hello',
...     body_text='Hi',
...     pub_date=datetime.date(2005, 1, 1)
... )
```

- remove()：从关联对象集中移除执行的模型对象。（remove()、移除）

```
>>> b = Blog.objects.get(id=1)
>>> e = Entry.objects.get(id=234)
>>> b.entry_set.remove(e) #
```

- clear()：从关联对象集中移除一切对象。（clear()、移除一切对象）

```
>>> b = Blog.objects.get(id=1)
>>> b.entry_set.clear()
```

29. 使用 reverse()函数对 url 进行逆向解释时，对视图函数和视图类的使用方式各不相同：（reverse()、使用、各不相同）

- 视图函数：传入 reverse()的参数是一个函数，不带引号：（视图函数、参数、函数）

```
return HttpResponseRedirect(reverse('manage'))
```

- 视图类：传入 reverse()的参数是一个字符串，是 url 中定义的 name：（视图类、参数、字符串、name）

url 文件中：

```
urlpatterns = [
    url(r'^manage$',views.manage.as_view(),name='djManage'),

]
```

reverse()使用：

```
return HttpResponseRedirect(reverse('djManage'))
```

30. 类视图的 get()和 post()需要两个参数：（get()、post()、两个参数）

```
class myadmin(View):
    def get(self,request):
        return render(request,'my-admin.html')

class mylogin(View):
    @csrf_exempt
    def post(self,request):
    ……
```

注意，类视图的装饰器不能用的类上，而应该用在类里面的函数上。（类视图、装饰器、用在函数）

31. @login_required 装饰器不能用在类视图上，类视图可以使用 is_authenticated 来实现登录验证：（类视图、is_authenticated、登录验证）

```
class writearticle(View):
    def get(self,request):
        if not request.user.is_authenticated():
            return HttpResponseRedirect('my-admin')
        return render(request,'writearticle.html')
```

32. python 的正则表达式函数有以下几个：

(1) search()：若 string 中包含 pattern 子串，则返回 MatchObject 对象，否则返回 None，注意，如果 string 中存在多个 pattern 子串，只返回第一个。（search()、返回 MatchObject 对象）

(2) match()：只检测是否从首字符开始匹配，如果是，则匹配成功，返回 MatchObject 对象，如果首字符不匹配，则返回失败。若要完全匹配，pattern 要以$结尾。（match()、只匹配开始）

(3) findall()：返回 string 中所有与 pattern 相匹配的全部字串，返回形式为数组。（findall()、全部匹配、数组）

(4) finditer()：返回 string 中所有与 pattern 相匹配的全部字串，返回形式为迭代器。（finditer()、全部匹配、迭代器）

33. MatchObject 对象部分成员：

(1) group()、groups 或 group(index)都用于获取匹配结果。（group、获取匹配结果）

(2) start()和 end()用于返回正则表达式开始和结束的索引。注意，类似“<title>(.*?)</title>”这样的正则表达式，start()返回的是字符“<”的前一个索引，而 end()返回的是字符“>”的索引。而不是括号中匹配内容的索引。（start()、end()、开始和结束索引、不是匹配的索引）

(3) span()返回一个由 start()和 end()的返回值组成的元组。（span()、start()和 end()的元组）

(4) pos 属性和 endpos 属性返回的是整个字符串开始索引和结果索引。（pos、endpos、整个字符串、索引）

34. 字符串替换函数有 replace()和 re.sub()，re.sub()有多种重载形式，故 re.sub()相比 replace()功能更强大。（替换、replace()和 re.sub()、re.sub()、更强大）

35. 字符串替换除了利用 replace()和 re.sub()之外，还可以利用字符串连接的形式实现：（替换、连接、实现）

(1) 利用正则表达式和 MatchObject 对象找到匹配字符串的位置

(2) 使用切片语法将匹配字符串前后的内容以及新内容连接起来

36. 计算机程序新特性的作用有以下几点：（作用、新功能、简化流程、解决 BUG）

- 添加新功能。
- 简化现有程序的流程。
- 解决 BUG。

37. Django 表单类 form 用于简化 html 表单数据输入流程。（form 类、简化输入流程）

- 注意，form 并不是用于显示输出的，而是简化输入的。也就是简化 html 表单创建、post 提交数据到保存数据到 model 的这个过程。也就是说，要使用 django form，html 的 form 就应该由 django form 来创建，而不是用户创建。

38. Django 表单的实例都有一个内置的 is_valid()方法，用来验证接收的数据是否合法。如果所有数据都合法，那么该方法将返回 True，并将所有的表单数据转存到它的一个叫做 cleaned_data 的属性中，该属性是以个字典类型数据。（is_valid()、验证数据是否合法）

```
# views.py

from django.shortcuts import render
from django.http import HttpResponseRedirect

from .forms import NameForm

def get_name(request):
    # 如果 form 通过 POST 方法发送数据
    if request.method == 'POST':
        # 接受 request.POST 参数构造 form 类的实例
        form = NameForm(request.POST)
        # 验证数据是否合法
        if form.is_valid():
            # 处理 form.cleaned_data 中的数据
            # ...
            # 重定向到一个新的 URL
            return HttpResponseRedirect('/thanks/')

    # 如果是通过 GET 方法请求数据，返回一个空的表单
    else:
        form = NameForm()

return render(request, 'name.html', {'form': form})
```

- 对于 GET 方法请求页面时，返回空的表单，让用户可以填入数据；
- 对于 POST 方法，接收表单数据，并验证；
- 如果数据合法，按照正常业务逻辑继续执行下去；
- 如果不合法，返回一个包含先前数据的表单给前端页面，方便用户修改。

39. 使用 form 的时候，post 请求的那个页面的 get()请求需要返回一个空的表单：（使用 form、请求的页面、返回一个空表单）

```
class news(DetailView):
    model = article
    template_name = 'news.html'
```

```
    context_object_name = "article"
    pk_url_kwarg = 'pk'

    def get(self, request, *args, **kwargs):
        self.object = self.get_object()
        context = self.get_context_data(object=self.object)
        context['myform']=comForm()
        return self.render_to_response(context)
```

40. 通过表单的 is_bound 属性可以获知一个表单到底是绑定了数据，还是一个空表。（is_bound、是否绑定数据）

41. cleaned_data 属性保存着表单的数据，它是一个字典类型：（cleaned_data、表单数据）

```
from django.core.mail import send_mail

if form.is_valid():
    subject = form.cleaned_data['subject']
    message = form.cleaned_data['message']
    sender = form.cleaned_data['sender']
    cc_myself = form.cleaned_data['cc_myself']

    recipients = ['info@example.com']
    if cc_myself:
        recipients.append(sender)

    send_mail(subject, message, sender, recipients)
    return HttpResponseRedirect('/thanks/')
```

42. {{ form }}模板变量可以根据定义字段自动生成 html 标签：

- {{ form.as_table }} 将表单渲染成一个表格元素，每个输入框作为一个<tr>标签（as_table、表格数据）
- {{ form.as_p }} 将表单的每个输入框包裹在一个<p>标签内 tags （as_p、包裹在、p 标签内）
- {{ form.as_ul }} 将表单渲染成一个列表元素，每个输入框作为一个<li>标签

  例如：（as_ul、列表元素）

```
{{ from.as_p }}
```

上面语句将会得到的下列 HTML 表单：

```
<p><label for="id_subject">Subject:</label>
```

```
   <input id="id_subject" type="text" name="subject" maxlength="100" required /></p>
<p><label for="id_message">Message:</label>
   <textarea name="message" id="id_message" required></textarea></p>
<p><label for="id_sender">Sender:</label>
   <input type="email" name="sender" id="id_sender" required /></p>
<p><label for="id_cc_myself">Cc myself:</label>
   <input type="checkbox" name="cc_myself" id="id_cc_myself" /></p>
```

Django 会自动生成 label 和 id，但必须手动提供 table 和 ul 元素。

43. {{ form.field }}中非常有用的属性，这些都是 Django 内置的：

- {{ field.label }}　　字段对应的 label 信息
- {{ field.label_tag }}　自动生成字段的 label 标签，注意与{{ field.label }}的区别。
- {{ field.id_for_label }}　　自定义字段标签的 id
- {{ field.value }}　　当前字段的值，比如一个 Email 字段的值 someone@example.com
- {{ field.html_name }}　　指定字段生成的 input 标签中 name 属性的值
- {{ field.help_text }}　字段的帮助信息
- {{ field.errors }}　　包含错误信息的元素
- {{ field.is_hidden }}　用于判断当前字段是否为隐藏的字段，如果是，返回 True
- {{ field.field }}　　返回字段的参数列表。例如{{ char_field.field.max_length }}

44. 除了使用 form 模板变量自动生成 HTML 表单外，还可以手动渲染。通过{{ form.xxx }}获取每一个字段，然后分别渲染，xxx 指字段名：（可以手动渲染、{{ form.xxx }}）

form 文件：

```
from django import forms

class ContactForm(forms.Form):
    subject = forms.CharField(max_length=100)
    message = forms.CharField(widget=forms.Textarea)
    sender = forms.EmailField()
    cc_myself = forms.BooleanField(required=False)
```

模板文件：

```
{{ form.non_field_errors }}
<div class="fieldWrapper">
    {{ form.subject.errors }}
    <label for="{{ form.subject.id_for_label }}">Email subject:</label>
    {{ form.subject }}
</div>
<div class="fieldWrapper">
```

```
    {{ form.message.errors }}
    <label for="{{ form.message.id_for_label }}">Your message:</label>
    {{ form.message }}
</div>
<div class="fieldWrapper">
    {{ form.sender.errors }}
    <label for="{{ form.sender.id_for_label }}">Your email address:</label>
    {{ form.sender }}
</div>
<div class="fieldWrapper">
    {{ form.cc_myself.errors }}
    <label for="{{ form.cc_myself.id_for_label }}">CC yourself?</label>
    {{ form.cc_myself }}
</div>
```

45. 如果你的页面同时引用了好几个不同的表单模板，那么为了防止冲突，你可以使用 with 参数，给每个表单模板取个别名，如下所示：（with 参数、模板、别名）

```
{% include "form_snippet.html" with form=comment_form %}
```

使用的时候如下：

```
{% for field in comment_form %}
.....
```

46. 使用{{ form.xxx.errors }}模板语法在表单里处理错误信息。对于每一个表单字段的错误，它其实会实际生成一个无序列表，参考下面的样子：（{{ form.xxx.errors }}、错误信息）

```
<ul class="errorlist">
    <li>Sender is required.</li>
</ul>
```

47. Meta 类的 widget 属性可以修改由 form 默认的 html 控件的 css 样式。比如，修改 name 字段的 CharField 对应的控件，生成<textarea>，而不是其默认的<input type="text">，则可以覆盖该字段的小部件。在 ModelForm 中可以这样完成：（widget、修改 css 样式、widgets = {}）

```
 from django.forms import ModelForm , Textarea
from myapp.models import Author

class AuthorForm ( ModelForm ):
    class Meta :
```

```
        model = Author
        fields = ( 'name' , 'title' , 'birth_date' )
        widgets = {
            'name' : Textarea ( attrs = { 'cols' : 80 , 'rows' : 20 }),
        }
```

在 Form 中这样完成：

```
from django import forms

class CommentForm(forms.Form):
    name = forms.CharField()
    url = forms.URLField()
    comment = forms.CharField(widget=forms.Textarea)
```

48. 在创建 Widget 时使用 Widget.attrs 参数添加 CSS 类：（widget.attrs、添加 css 类）

```
class CommentForm(forms.Form):
    name = forms.CharField(widget=forms.TextInput(attrs={'class': 'special'}))
    url = forms.URLField()
    comment = forms.CharField(widget=forms.TextInput(attrs={'size': '40'}))
```

49. meta 的 fields 用于设置哪些 model 中的属性会用于表单之中。

- 将 fields 属性的值设为__all__，表示将映射的模型中的全部字段都添加到表单类中来。（field=’__all__’、全部字段、添加到表单）

```
from django.forms import ModelForm

class AuthorForm(ModelForm):
    class Meta:
        model = Author
        fields = '__all__'
```

- fiels 的 exclude 属性表示排序特定字段，除列出的字段之外的所有字段，添加到表单类中作为表单字段。

50. 默认情况下，当 Django 渲染 form 为实际的 HTML 代码时，不会帮你添加任何的 CSS 样式，也就是说网页上所有的 TextInput 元素的外观是一样的。使用 Widget.attrs 参数可以为 HTML 元素添加实际的 css 样式：（Widget.attrs 参数、添加 css 样式）

```
class CommentForm(forms.Form):
    name = forms.CharField(widget=forms.TextInput(attrs={'class': 'special'}))
    url = forms.URLField()
    comment = forms.CharField(widget=forms.TextInput(attrs={'size': '40'}))
```

51. Session 依赖 Cookie！但与 Cookie 不同的地方在于 Session 将所有的数据都放在服务器端，用户浏览器的 Cookie 中只会保存一个非明文的识别信息，比如哈希值。下面这个简单的视图在用户发表评论后，在 session 中设置一个 has_commented 变量为 True。它不允许用户重复发表评论。（Session 依赖 Cookie）

```
def post_comment(request, new_comment):
    if request.session.get('has_commented', False):
        return HttpResponse("You've already commented.")
    c = comments.Comment(comment=new_comment)
    c.save()
    request.session['has_commented'] = True
    return HttpResponse('Thanks for your comment!')
```

52. django 的 session 框架可以使用多种方式保存数据：（session 保存数据、数据库、缓存、文件、cookie）

- 保存在数据库内
- 保存到缓存
- 保存到文件内
- 保存到 cookie 内

53. 当你设置一个 cookie 时，你无法立刻得到结果，直到浏览器发送下一个请求时才能获得结果。Django 提供一个简单的方法来测试用户的浏览器是否接受 Cookie。只需在一个视图中调用 request.session 的 set_test_cookie()方法，并在随后的视图中调用 test_cookie_worked()获取测试结果（True 或 False)。（设置 cookie、无法立即得到结果、测试 cookie）

54. 当会话中间件启用后，传递给视图 request 参数的 HttpRequest 对象将包含一个 session 属性，这个属性的值是一个类似字典的对象。你可以在视图的任何地方读写 request.session 属性，或者多次编辑使用它。（HttpRequest 对象、session 属性）

(1) flush()：删除当前的会话数据和会话 cookie。经常用在用户退出后，删除会话。

(2) set_test_cookie()：设置一个测试 cookie，用于探测用户浏览器是否支持 cookies。由于 cookie 的工作机制，你只有在下次用户请求的时候才可以测试。

(3) test_cookie_worked()：用户的浏览器接受测试 cookie，返回 True，否则返回 False。你必须在之前先调用 set_test_cookie()方法。

(4) delete_test_cookie()：删除测试 cookie。

(5) set_expiry(value)：设置 cookie 的有效期。可以传递不同类型的参数值：

- 如果值是一个整数，session 将在对应的秒数后失效。例如 request.session.set_expiry(300)将在 300 秒后失效.
- 如果值是一个 datetime 或者 timedelta 对象, 会话将在指定的日期失效
- 如果为 0，在用户关闭浏览器后失效
- 如果为 None，则将使用全局会话失效策略

失效时间从上一次会话被修改的时刻开始计时。

(6) get_expiry_age()：返回多少秒后失效的秒数。对于没有自定义失效时间的会话，这等同于 SESSION_COOKIE_AGE.

(7) get_expiry_date()：和上面的方法类似，只是返回的是日期

(8) get_expire_at_browser_close()：如果用户会话浏览器关闭后就结束，返回 True

(9) clear_expired()：删除已经失效的会话数据。

(10) cycle_key()：创建一个新的会话秘钥用于保持当前的会话数据。django.contrib.auth.login() 会调用这个方法。

55. Django 默认使用 JSON 序列化会话数据。你可以在 SESSION_SERIALIZER 设置中自定义序列化格式，甚至写入警告说明。但是强烈建议你还是使用 JSON，尤其是以 cookie 的方式进行会话时。

56. SESSION_EXPIRE_AT_BROWSER_CLOSE 用于设置 cookie 是否在浏览器关闭后失效，默认设置为 False，也就是说 cookie 保存在用户的浏览器内，直到失效日期，这样用户就不必每次打开浏览器后都要再登录一次。这个设置是一个全局的默认值，可以通过显式地调 request.session 的 set_expiry()方法来覆盖

57. Django 没有提供自动清除失效会话的机制。因此，你必须自己完成这项工作。但是 Django 提供了一个命令 clearsessions 用于清除会话数据，建议你基于这个命令设置一个周期性的自动清除机制，比如 crontab 或者 Windows 的调度任务。（clearsessions、清除失效会话、crontab 调试任务）

58. 使用缓存模式的会话不需要你清理数据，因为缓存系统自己有清理过期数据的机制。使用 cookie 模式的会话也不需要，因为数据都存在用户的浏览器内，不用你帮忙。

59. 在 Django 的认证框架中只有一个用户模型也就是 User 模型，它位于 django.contrib.auth.models。用户模型主要有下面几个字段：

- username
- password
- email

- first_name
- last_name

60. 在 makemigrations 的时候，如果 app 下面有 model，则需要使用命令来生成 app 下的数据库表，否则会出错：（makemigrations、生成 app 下的数据库）

```
Python manage.py makemigrations [app-name]
python manage.py migrate [app-name]
```

app-name 指的是 app 名。

61. setting 中设置了“USE_TZ=true”后，所有的存储和内部处理，甚至包括直接 print 显示全都是 UTC 时间，只有通过模板进行表单输入/渲染输出的时候，才会执行 UTC 本地时间的转换。（USE_TZ=true、模板渲染、转换本地时间）

62. TemplateView 视图类用于显示一个固定的模板，它只需要在视图类中指定模块文件即可：（TemplateView 视图类、显示固定的模板）

```
 # some_app/views.py
from django.views.generic import TemplateView

class AboutView ( TemplateView ):
    template_name = "about.html"
```

63. slug 是 django 提供的用于生成内容合法的可读性强的 url 的手段。以“/posts/13-sui-de-hai-zi”为例，“13-sui-de-hai-zi”就是 slug。（slug、生成合法、可读性强、url）

64. 通用视图类通常只需要指定属性，get()、post()等函数由类继承而来，当需要某些其他的模型的数据等等功能时，才需要重写现有的某些函数。

```
class articleView(ListView):
    model = article
    context_object_name = 'article_list'
    template_name = 'index.html'  # 模板文件
    paginate_by = 2
```

65. 通用视图类 DetailView 有以下类属性：

(1) model：用于指定视图类对应的模型。

(2) queryset：用于指定视图类使用的数据集。

(3) context_object_name：为 get_queryset 方法返回的 model 列表重新命名，提高可读性。

(4) template_name：指定模板名。

(5) render_to_response： render_to_response 方法返回的响应类。默认为 TemplateResponse。

(6) slug_field：模型中的 slug 的字段名。slug_field 默认为'slug'。

(7) paginate_by:一个整数，指定每页应显示多少个对象。

(8) paginate_orphans：一个整数，指定最后一页可以包含的“overflow”对象的数量。

(9) paginator_class：paginator 类用于分页。默认情况下，使用 django.core.paginator.Paginator。

(10) pk_url_kwarg：在 URLConf 捕获的变量，这个捕获的变量会作为 pk 主键来进行查询。

blog/views.py

```
from django.views.generic import ListView, DetailView

class PostDetailView(DetailView):
    model = Post
    template_name = 'blog/detail.html'
    context_object_name = 'post'

    def get(self, request, *args, **kwargs):
        # 覆写 get 方法的目的是因为每当文章被访问一次，就得将文章阅读量 +1
        # get 方法返回的是一个 HttpResponse 实例
        # 之所以需要先调用父类的 get 方法，是因为只有当 get 方法被调用后，
        # 才有 self.object 属性，其值为 Post 模型实例，即被访问的文章 post
        response = super(PostDetailView, self).get(request, *args, **kwargs)

        # 将文章阅读量 +1
        # 注意 self.object 的值就是被访问的文章 post
        self.object.increase_views()

        return response

    def get_object(self, queryset=None):
        # 覆写 get_object 方法的目的是因为需要对 post 的 body 值进行渲染
        post = super(PostDetailView, self).get_object(queryset=None)
        post.body = markdown.markdown(post.body,
                                      extensions=[
                                          'markdown.extensions.extra',
                                          'markdown.extensions.codehilite',
                                          'markdown.extensions.toc',
                                      ])
        return post
```

```
    def get_context_data(self, **kwargs):
        # 覆写 get_context_data 的目的是因为除了将 post 传递给模板外（DetailView 已经帮我们完成），
        # 还要把评论表单、post 下的评论列表传递给模板。
        context = super(PostDetailView, self).get_context_data(**kwargs)
        form = CommentForm()
        comment_list = self.object.comment_set.all()
        context.update({
            'form': form,
            'comment_list': comment_list
        })
        return context
```

66. 通用视图类 ListView 具有以下属性：

(1) allow_empty：指定是否允许没有对象时返回空页面。如果 False，没有可用的对象时，视图将引发 404 而不是显示一个空页面。默认情况下，这是 True。

(2) context_object_name：指定要在上下文中使用的变量的名称。

(3) http_method_names：该视图接受的 HTTP 方法名称。

(4) model：指定视图将显示数据的模型。

(5) queryset：指定对象的 QuerySet。如果提供，queryset 的值将取代 model 提供的值。

(6) ordering：指定应用于 queryset 的排序方式。

(7) paginate_by：一个整数，指定每页应显示多少个对象。

(8) response_class：render_to_response 方法返回的响应类。默认为 TemplateResponse。

(9) template_name：字符串形式的模板名称。

67. UpdateView 视图类用于更新现有的表单。（UpdateView 视图类、更新表单）

68. 由源码得知，ListView 视图类 paginator 上下文变量叫“page_obj”，而“is_paginated”可以用于判断其中视图中是否有数据。

```
<div class="pagination">
    <span class="page-links">
        {% if page_obj.has_previous %}
            <a href="/index?page={{ page_obj.previous_page_number }}">上一页</a>
        {% endif %}
        {% if page_obj.has_next %}
            <a href="/index?page={{ page_obj.next_page_number }}">下一页</a>
        {% endif %}
        <span class="page-current">
```

```
            第{{ page_obj.number }}页 ， 第{{ page_obj.paginator.num_pages }}页。
        </span>
    </span>
</div>
```

69. 捕获 url 变量时需要加上括号。（捕获 url 变量、加括号）

```
url(r'^manage/(?P<flag>\w{4,20})$',views.manage.as_view()),
url(r'^news/(?P<pk>\d+)$',views.news.as_view(),name='djNews')
```

注意加粗的括号。

70. DetailView 和 ListView 简单用法：

(1) DetailView 最简单用法：

①. 先在视图类中指定几个关键属性：

```
class ArticleDetailView(DetailView):
# Django 有基于类的视图 DetailView,用于显示一个对象的详情页，我们继承它
    model = Article
    # 指定视图获取哪个 model

    template_name = "blog/detail.html"
    # 指定要渲染的模板文件

    context_object_name = "article"
    # 在模板中需要使用的上下文名字

    pk_url_kwarg = 'pk'
    # 这里注意，pk_url_kwarg 用于接收一个来自 url 中的主键，然后会根据这个主键进行查询
    # 我们之前在 urlpatterns 已经捕获 article_id
```

②. 修改 urlconf，捕获 url 中传输的数据库项数据：

```
url(r'^news/(?P<pk>\d+)$',views.news.as_view(),name='djNews')
```

③. 然后，就可以在模板中使用 article 进行布局了。

(2) ListView 简单用法：

①. 先在视图类定义几个关键属性：

```
class articleView(ListView):
    model = article
    context_object_name = 'article_list'
```

```
template_name = 'index.html'  # 模板文件
paginate_by = 2
```

②. 修改 urlconf：

```
url(r'^index$',views.articleView.as_view(),name='djIndex'),
```

③. 使用 article_list 修改 html 模板。

```
{% for article in article_list %}
        <h1 style="font-weight:bold;margin:0;padding:15px 0 0 0;"><a
href="news/{{article.pk}}">{{article.title}}</a></h1>
        <div class="article_abstrace">分类：占位符</div>
        <div class="article_abstrace"> 发表于：{{article.created_time|date:'Y-m-d H:i:s'}}</div>
        <div class="article_abstrace">阅读数：{{article.views}}</div>
        <div class="article_abstrace">点赞数：{{article.likes}}</div>
        <div style="font-size:17px;color: #999;padding:15px 0 15px
0;">{{article.abstract}}</div>
        <hr />
```

注意，article_list 是个查询集。

股票

1. 判断个股价值时，根据同行业同类型其他个股的市盈率来做考量。
2. 股社区白马股标准：过去三年的净资产收益率（摊薄），在 20%以上。
3. 板块联动：指各个相关股之间同涨同跌的联动效应。
4. 市场情绪按种类可分为投机情绪和价投情绪。
5. 投机情绪相关指标：

(1) 打板指：该指描述涨停股（除一字板）第二天的涨跌表现。该指总共四个分指标，分别是平均涨幅、平均跌幅、最高涨幅、最高跌幅。

该指表现了打板族的平均盈利情况，平均涨幅绝对值比平均跌幅绝对值大得越多，盈利情况越好，相应的赚钱效应就越好。

(2) 中证 1000 指数。

(3) 创业板指数。

6. 市场情绪是判断标准的其中一部分，另一部分是抓热点的能力。

7. 庄股（市值管理）判断：

(1) 大股东或控制人是私企。

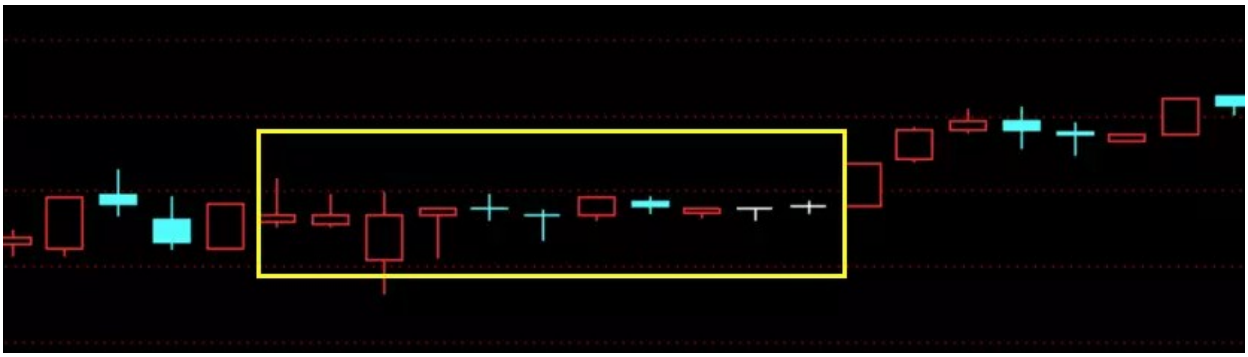
(2) 普遍都是 100 亿市值以下的中小票。

(3) 业绩平庸，无法支撑与股价相匹配的估值。

(4) 日常走势诡异的稳，不随市场波动而波动，当市场出现暴跌时只要没出货就能扛住。即使盘中下跌了，收盘也能拉回来，所以这类股经常有较长的下影线，极少有实体光底的阴线。

(5) 整个主升浪成交量都很少，而且往往是越涨量越小。就算不涨也起码是横盘，一路往下跌的股票基本上可以排除嫌疑。

(6) 不会涨太快，凸凸一波后横向震荡整理。震荡的这些交易日内波动特别稳定。



8. 一致性是指资金对当前股票形势看法之间是否一致。一致则表示所有人都看好或看空，形态通常表现为一字板、快速拉升或急跌；存在分歧则表示一部分看空，一部分看好，形态通常为带量的涨跌之间来回反复、反复烂板、封板。

   注：一致性只适用于非庄股的龙头股。

9. 一支股票反复封板反复烂板，说明这个股票的一致性不高，多空存在严重分歧。如果尾盘板不能封住一段足够的时间以向人表示多方占优，通常第二天会低开。

10. 股性：

(1) 股性好的股票指打板当天不破板，第二天有利润，反之则是股性差。

(2) 股性差的股票通常表现为：

- 首次涨停第二天即高开低走，甚至直接低开埋人。
- 二板常放巨量炸板。

这应该是里面的巨量资金做 T 导致的，这种股吸引不到游资，故没有参与价值。

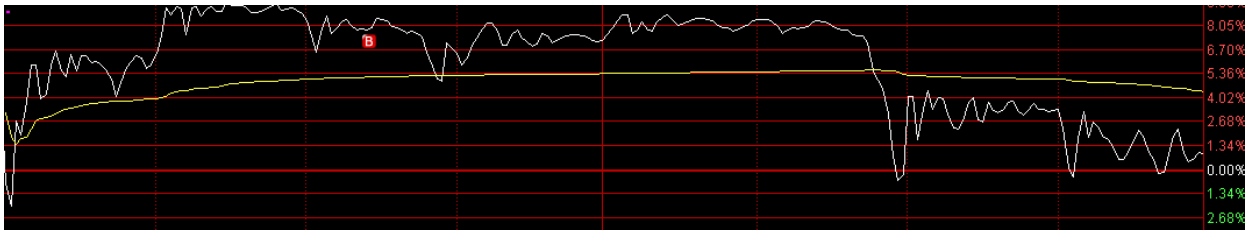
➢ 注意：这里的巨量是相对目标股以前的交易日而言的。

11. 股性越好，能吸引到的游资就越多。

12. 资金的追高意愿指游资对高位股的买卖态度。符合以下条件可称资金的追高意愿低：

- 二板以上换手连板低开或者开盘即向下冲，主力换手龙头都出现冲高回落。
- 中小创和创业板指数冲高回落。
- 烂板率大增。

13. 资金的追高意愿低时，连板股高位震荡，如果不能封板，则尾盘会有跳水的风险，因为原来锁仓的游资选择出货。



14. 龙头的特性：

- 号召性：板块龙头能带动一个板块拉升。
- 可变性：龙头是不稳定的，龙一龙二的排列顺序是可变的。
- 竞争性：某些环境下，不同的板块的龙头之间呈现蹊跷板效应。
- 协同性：某些环境下，不同的板块的龙头之间又是同涨同跌的。

15. 股票涨跌本身具有相当大的偶然性，投资者本身能做的就是增加利润，减小回撤。

16. 妖板块指背离大盘指数走势，短时间内大涨的板块。任何板块都具有可炒作空间，但大部分板块的上涨是昙花一现，只有少部分板块可成长为妖板块。

17. 加速板是指板块龙头经过换手板后突然连续缩量的涨停板，如一字板，由换手后的游资选择锁仓形成。加速板后的第一次放量如果无法涨停，则由于锁仓游资利润太高，极有可能导致股价高速下跌，这种下跌是相当惨烈的。这种加速板需要一到两个放量板来消化一字板缩量的游资利润，才有参与价值。

18. 一字板又可以称为独食板，因造就一字板的游资利润太高，放量后如果无法稳住涨停，则下跌是异常惨烈的。

19. 加速板第一次完全放量后的成交量通常高于加速板前的放量板，有些甚至是加速板前的放量板的数倍。

20. 一个可参与的的龙头通常具有以下特征：

- 成交量大。
- 无加速板，或前期加速板之后已经经过一到两次充分的换手板消化获利盘。

21. 情绪的平衡点是指多空双方在长期的斗争中双方情绪都处于崩溃边缘的临界点。突破平衡点的关键是在关键时候，在关键的股票中找到关键的薄弱点。突破行为能起多大作用取决于各板块间、个股与板块间的关联关系的强弱，关联性越强，则越容易起作用。这种平衡点大部分时间是不存在的。

22. 情绪的支持点分成上涨支撑点和下跌支撑点，突破支撑点后会导致加速上涨或下跌。

23. 板块轮动是指板块间轮流引领上涨的现象，个股轮动是指某个板块个股间轮流引领上涨的现象。

24. 开盘不买低位股，因为开盘股价处于低位说明该股情绪不乐观。开盘半小时后不买冲高回落股，这种股十有八九是坑爹货。

25. 高开的股票一般有两种走法：

- 冲高回落，一般冲到 9%左右的涨幅后回落。这类票大部分是坑，由多头进攻失利造成，多头失利后，锁仓者变空头，多空失衡，所以这类票大部分会往下掉。基于成本与回撤考虑，这类票不能买。
- 上冲后或回落一点点，或横盘，上冲幅度往往只有一两个点，涨幅也远达不到 9%。这类票才有上涨空间。

26. 基于成本考虑：开盘买情绪高的票，第二天如果情绪依然高，则锁仓。如果情绪不妙，则开盘卖出。卖出股票后超过 10 点当天则空仓，因为 10 点当天大部分情绪高的票都已经涨停了。

27. 形态是指分时图和 K 线随时间形成的形状。形态分好形态和坏形态：

- 好形态是指基于风险和收益的考虑，收益比高、风险低的形态。例如热点题材涨停第二天高开的形态。这里要注意一点，涨停第二天高开如果股价冲到 9%左右后回落，则该形态从好形态变成坏形态。
- 坏形态是指基于风险和收益的考虑，收益低、风险高的形态。例如涨停第二天低开埋人的形态。

28. 市场参与主体：

- 机构：价值投资，资金大，一旦炒作导致股价过高，砸盘压力非常大。
- 游资：快进快出，资金量有限，以炒作来吸引跟风盘为盈利条件。是市场妖股的主要创造者和维护者。
- 散户：资金最少，但人数庞大，市场投机的主力人群，追涨杀跌。导致股价上涨的主力是打板族，维持股价不掉的是低吸族。

这种市场主体的特点也决定了流通市值大的股票很难出现连板股。

29. 充分换手是指一支股票经过半天的交易，换手率已经接近或超过 50%，则这种票拉升的难度会降低。
30. 老妖股集体异动，也就表明现有概念行情接近尾声。
31. 热点题材的形成有两类：

- 一是社会热点新闻。
- 二是资金合力形成二连板，制造题材。一旦二连三连板出现，再加一堆一板，热点题材就形成了。

32. 大盘跳水时冲板相当危险，极有可能吸引不来足够的跟风盘导致冲高回落。
33. 当出现加速板时，开盘后获利高的游资首先出逃，会诱发多米诺骨牌效应，由于获利盘巨大，多军不敢接盘，股价会一路下滑。下滑几天后，由于获利盘已消化完毕，又有反包的可能。
34. 当资金量大到一定程度时，卖出变得困难，经常只卖了一部分股价就大幅下降，这时会有人杀跌。

大势篇

1. A 股是个政策市，涨跌受政策、领导意志的影响非常大。
2. 大盘指数由各个股票数据组成，反映的是股票的总体数据，故任何股票都免不了受大盘指数影响。

3. 当市场中某种悲观情绪达到极致时，只要有一个人做出对市场不利的事情，恐慌情绪就会迅速蔓延，从而导致市场出现严重的亏钱效应。

集合竞价篇

1. 集合竞价对股票的走势有着非常重要的影响，开盘价往往影响着题材的走势。
2. 龙一的开盘价对题材走势有着非常重要的作用。如果龙一当天低开低走，并且全天不拉升，则表明龙一所属的题材未来的走势可能不太好，除非是轮动性的题材。
3. 当题材龙一高开太多时，买入的风险会骤然变大，如果没有相关的市场关注度配合，则下跌的可能性非常高。
4. 开盘涨停的一字板股票破板风险大，特别是当天处于弱市时，破板后打板族当天就可能亏损超过 10 个点。稳妥的涨停板应该具有一定的换手率，而不是一字板。

盘面篇

1. 盘面是整个选股判断是基础，所有选股判断都围绕分时线、K 线组成的盘面数据进行。
2. 选股法有打板、半路板、低吸三大类：

(1) 打板、半路板：围绕盘中强势股进行。

(2) 低吸：围绕高换手龙头进行。

3. 急升、急跌的真实意图难以判断，急升买入可能会在最高点站岗，急跌买入可能买在半山腰。

题材篇

1. 题材分消息型题材和合力型题材：

- 消息型题材由新闻消息影响产生，有明确的消息来源。这种题材往往以一字板开头，很难在开始就参与进去，但参与后得到的利润很高。
- 合力型题材无明确的消息来源，拉板后再由市场赋予题材名称。这种题材在开始时很容易参与，但缺点是很难判断题材的持续性。

2. 一个题材往往只有龙一龙二（即龙头）的风险低，利润高，其余的风险高，利润相对较低。
3. 题材龙头需流通盘小，股性好。

4. 题材龙头开盘价对题材走势相当重要。龙头低开时，往往意味当天行情可能不太好，如果龙头全天不出现放量拉升，则表示该题材行情到此结束。
5. 题材龙一龙二之间往往有个股联动效应，行情好时龙头拉升还会拉动整个板块上涨。
6. 新题材龙头连续放量涨停，则该题材会成为市场热点题材，上涨空间瞬间变大。所以新题材龙头第二天、第三天的盘面情况非常重要。
7. 一个有投机价值的涨停由大游资或游资帮派发动，有板块联动，盘后能得到市场的认可，市场会赋予一个明显的题材。

关注度篇

1. 关注度高的无负面消息的股票，往往上涨空间也大，这也是龙头战法的原理。
2. 高关注度，是造就热点题材的必要但不充分的条件。
3. 高关注度，高开盘价，往往意味着该股当天行情不错。
4. 关注度配合一定的成交量，即使是下跌，反包后的收益也相当大。
5. 接力即筹码交换，是正常上涨必备的条件。左手倒右手是制作虚假筹码交换的手段。当筹码过度集中时，拉升并不难，但同时风险非常大，风险大利润低，一旦筹码全部抛出，股价下跌是非常快的。
6. 龙头的自然涨停由于各方都能参与，筹码不会过度集中，单一游资抛出筹码并不会对股价造成太大的影响。
7. 充分换手后由于获利盘少，股价下跌抛压就少，由此造成的损失比一字板少。

成交量篇

1. 带量的拉升才是真正的拉升，无量拉升往往是诱多。
2. 高换手率、高关注度，也就为游资拉升提供了一定的条件，高换手率下跌时会出现反包。
3. 基于成交量和股价的关系可以对承接力进行分析：

- 量少价跌，则说明承接力很差。
- 高成交横盘，则说明承接力还可以。

4. 加速板是指成交量严重萎缩的涨停板，加速后首次放量如果无法稳住涨停，则下跌的速度是非常快的。所以加速板需要一到两个放量板来消化获利盘。

5. 多空平衡的龙头在上涨或下跌过程中会碰到一个平衡点，这个平衡点是非常重要的一个观测点，一旦放量突破这个平衡点，则后市看好。
6. 加速板由于筹码过度集中，可能会出现左手倒右手的虚假放量板，稳妥的办法是不参与加速板的行情。

风险篇

1. 炒股真正的能力，是在控制回撤的基础上，尽量地增加利润，而不是追求百分比的胜率。
2. 股性差的股票，吸引不到游资，导致风险大，利润低。
3. 加速板放量第二天是风险最大的一天，由于获利盘巨大，抛压大，一旦接力不上，当天亏损就可以达到 10 多点。
4. 当一支股票开盘高开后横盘时，涨停机会较大，但一旦上冲到 8%、9%后失败回落后横盘，则该股风险变大，大概率无法涨停，下个交易日还会低开。
5. 价值涨停是指受某一简单消息刺激，由崇尚价值投资的散户、实力较弱的小游资合力拉出的涨停。这种涨停往往得不到市场的认可，第二天无人接力，但开盘价却受某些急于兑现利润的散户、游资一键按跌停的影响，往往低开或开盘后直线下滑。
6. 急升、急跌的票风险大，拉升失败回落极可能导致山顶站岗，急跌极有可能抄在半山腰，因此合适的方式是待票企稳后再作打算。

声东击西：利用题材个股联运的特点，可以实现声东击西的效果。2018 年 3 月 7 日关注度极高的次新龙头泰永长征和与之有联运效应的龙二江苏租赁低开。泰永长征盘子小，江苏租赁盘子大。午后泰永长征换手率到达一定程序后，出现拉升，带动了江苏租赁拉升。也就是说，用小股资金，能提高大盘股的利润。