# 周会学习记录：

销售问题：

当我们公司的东西出去展会，是放现有的卖的最火的还是最新的。

答案：我们展会的目的是为了吸引与会者的眼球，让他们觉得有机会合作，是一次智慧与机会的撞击，所以我们应该放新的东西，

模板简历：<http://download.csdn.net/download/qazwsxpcm/10002613>。

Java基础：<http://blog.csdn.net/jackfrued/article/details/44921941>。

大牛文档：

http://blog.csdn.net/zuoxiaolong8810/article/details/51482550

其它的相关的文章

* http://blog.csdn.net/nuaazhaofeng/article/details/63707235
* http://blog.csdn.net/v123411739/article/details/71437307

《深入理解计算机系统》，《tcp/ip详解 卷一、二、三》，《数据结构与算法》

《深入理解计算机系统》比较适合一直从事Java Web开发和APP后端开发工作的人群。《tcp/ip详解 卷一、二、三》比较适合做网络编程的人群，比如你使用netty去开发的话，那么就要对TCP/IP有更深入的了解。而《数据结构与算法》这本书，则比较适合做计算研究工作的人，比如刚才提到的分布式计算

---------------------

作者：左潇龙

来源：CSDN

原文：https://blog.csdn.net/zuoxiaolong8810/article/details/51482550?utm\_source=copy

版权声明：本文为博主原创文章，转载请附上博文链接！

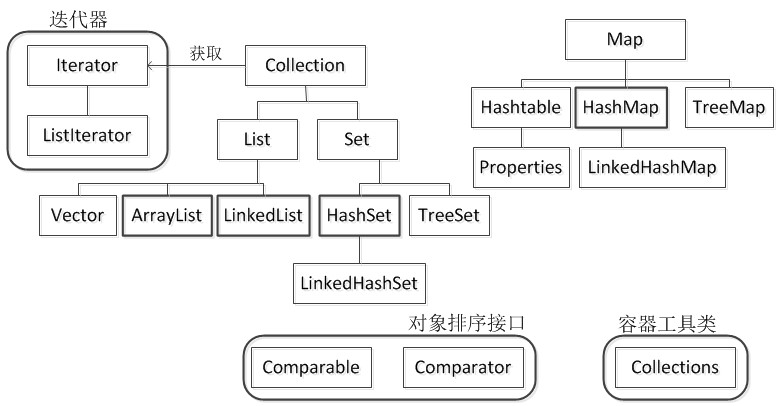
第一年《Java编程思想》

1-2《大话设计模式》《重构 改善既有代码的设计》，《effective java》

2-3《深入理解Java虚拟机》《Java并发编程实战》

# Java集合框架和泛型机制

JDK API中专门设计用来存储其他对象的类，一般称为对象容器类，简称容器类，这组类和接口的设计结构也被统称为集合框架（Collection Framework）。集合框架中容器类的关系如下图示：



Set的三个常用实现类:

**HashSet**类是Set接口实现类之一，使用较广泛，它不保存元素的加入顺序。**HashSet**类根据元素的哈希码进行存放，取出时也可以根据哈希码快速找到。

**LinkedHashSet**类根据元素的**哈希码**进行存放，同时用**链表记录元素加入的顺**序。**通过链表来存储对象，一般插入和删除效率较高，检索效率相对较低。**

**TreeSet**类使用红黑树结构对加入的元素进行**排序**存放，通过TreeSet构造方法来获取TreeSet对象。

**List的三个常用实现类：**

**ArrayList**类扩展AbstractList并执行List接口，ArrayList类支持可随需要而增长的动态数组。ArrayList类对于使用索引取出元素有较高的效率，它可以使用索引来快速定位对象。但元素做删除或插入速度较慢，因为使用了数组，需要移动后面的元素已调整索引顺序。

**LinkedList**类扩展AbstractSequentialList并执行List接口。它提供了一个链接列表数据结构，LinkedList容器类通过连接指针来关联前后两个元素。LinkedList是使用双向链表实现的容器，所以针对频繁的插入和删除元素使用LinkedList类效率较高，它适合实现栈和队列。

**Vector**类提供了实现可增长数组的功能，Vector类大多数操作与ArrayList类相同，区别在于Vector类是线程同步的。

**Java集合中不能存放基本类型数据，而只能存放对象的引用。**

**主要从两方面来选择容器：**

**（1）存放要求**

无序：Set，不能重复

## Set:注重独一无二的性质,该体系集合可以知道某物是否已近存在于集合中,不会存储重复的元素

有序：List，允许重复

“key-value”对：Map

**（2）读写数据效率**

Hash：两者都最高。

Array：读快改慢。

Linked：读慢改快。

Tree：加入元素可排序使用。

**数据结构——表（线性表和链表）需要关注数据在内存中的存储形式**。

**线性表**：在内存中分配一块**连续的存储空间**。在线性表中访问数据元素是很快的，但当改变线性表长度的时候（往中间添加数据或删除数据），其会在移动元素上花费大量的时间。例如数组Array，或ArrayList都是数据结构中线性表的实现。

**链表**：存储的链式的，不强迫数据是在一片连续的内存空间，可以是分散存储的，这样表中的每个元素除了包括元素的值外，还要包括一些额外的信息。查找时间比线性表慢。

**备注：通过链表来存储对象，一般插入和删除效率较高，检索效率相对较低。**

**2、泛型**

应用背景：存入容器的对象在取出时需要强制转换类型，因为对象加入容器时都被转化为Object类型，而取出时又要转成实际类型。

在Java中向下类型转换时容易出现ClassCastException的异常，这时应当尽量避免。有什么办法可以让装入容器中的**数据保存自己的类型而不被转化为Object对象**呢，这就需要用到JDK5.0支持的新功能——Java的泛型。

定义：**泛型只是编译时的概念，是供编译器进行语法检查用的。**所谓泛型，就是在定义（类型的定义，方法的定义，形式参数的定义，成员变量的定义等等）的时候，指定它为通用类型，也就是数据类型可以是任意的类型，如List<?> list = null，具体调用时，要将通用类型转换成指定的类型。泛型提高了大型程序的类型安全和可维护性。

目的：

* 努力将运行时异常转换成编译时的问题，减少运行时异常数量（提高了编译器的能力）。
* 解决模版编程的问题。

泛型允许在定义类、接口和方法时使类型（类、接口）成为参数，声明的类型参数在使用时用具体的类型替换。

　　泛型主要应用在集合框架中。

　　1、为什么使用泛型

* 提高程序的类型安全；泛型的使用让编译器可以验证类型假设。
* 有助于避免（强制）转型，使得编译器能够在编译时发现转型错误而不用等到运行时
* 可以实现通用算法

　　2、泛型类

　　泛型类的定义和声明：

　　类名之后通过<>指定一个或者多个类型参数的名字，同时还可以对类型参数的取值范围进行限定，多个类型参数之间使用逗号分隔。

public class Matrix<T>{

….

}

　　泛型的使用：

　　定义完类型参数后，可以在该类几乎任意地方（静态块、静态属性、静态方法除外）使用类型参数。注意，父类定义的类型参数不能被子类继承。

/\*实例化泛型类\*/

Matrix<Float> ft=new Matrix<Float>();

　　3、泛型方法

　　方法也可以泛型化，不管定义他们的类是不是泛型化。

　　为什么要使用泛型方法而不是将类型T添加到类定义中呢?

* 当泛型方法是静态时，不能使用类的类型参数。
* 当T上的类型约束对于方法是局部时，这意味着没有在类的另一个方法签名中使用相同类型的T的约束。泛型方法的类型参数是局部的，可以简化封闭类型的签名。

　　声明泛型方法：

public <T> T ifThenElse(boolean b, T first, T second) {

return b? first : second;

}

　　泛型类是在实例化类时指明泛型的具体类型，泛型方法是在调用方法时指明泛型的具体类型。

　　编译器会允许调用下面的代码时使用类型推理来推断出T的类型，并用实际参数替代T：

String s=ifThenElse(b, ”a”, ”b”);

Integer i=ifThenElse(b, new Integer(1), new Integer(2));

　　4、类型限制

　　有时我们需要限制类型参数的范围。

　　类型通配符（？），Matrix<?>表示任意的泛型类型。

/\*表示add()可以接受任意泛型类型的参数。\*/

public void add(Matrix<?> m);

         这种情况下，add()可以接受的参数类型太宽泛了。开发人员可能希望限制参数的具体类型，例如只希望接受Number及其子类的类型的变量，而不接受Random、Locale等类型的变量。这样就要对通配符有所限制。

         如Matrix类，使用了类型参数T，我们让Number类作为类型上界来限制这个类型参数：

/\*表示Matrix中包含的参数类型是Number及其子类\*/

public class Matrix<T extends Numbers>{…}

         当引入了类型上界后，在使用类型时就可以使用类型上界类（Number）中定义的方法。

         同理我们可以使用Number类作为类型下界：

/\*表示Matrix中包含的参数类型是Number及其父类\*/

public class Matrix<? super Number>{…}

　　5、类型擦除（Type Erasure）

　　泛型是在编译器这个层次实现的，在生成的字节代码中是不包含泛型的类型信息的。

　　使用泛型时加上的类型参数，会被编译器在编译时去掉，这个过程称为类型擦除。也就是说，由泛型附加的类型信息对JVM来说都是不可见的。

　　编译器在编译时尽可能地发现可能出错的地方，但是仍然无法避免在运行时刻出现类型转换异常的情况。

　　类型擦除的过程：

　　首先，找到用来替换类型参数的具体类。如果指定了类型参数的上界的话，则使用上界。

　　然后，把代码中的类型参数都替换成具体类型，同时去掉出现的类型声明，即<>的内容；

 　　理解了类型擦除，就会明白**编译器承担了全部的类型检查工作**。编译器禁止某些泛型的使用方式，是为了确保类型的安全性。

　　很多泛型的特性都与类型擦除有关，如

* 静态变量是被泛型类的所有实例共享的，即泛型不能用于静态变量
* 泛型的类型参数不能用在异常处理的catch语句中，因为异常处理是由JVM在运行时刻进行的。由于类型擦除，JVM已经无法区分源于同一泛型类型的两个不同类型的异常。

　　6、开发泛型类

　　因为类型擦除机制，类型参数并不能用来（在类中）创建对象（比如T t=new T();）或是静态变量的类型。

/\*泛型类\*/

public class Lhist<V>{}

/\*创建泛型类实例\*/

Lhist<Integer>li=new Lhist<Integer>(30);

　　7、泛型的最佳实践

　　使用泛型常见的实践原则：

* 在代码中避免泛型类和原始类型混用，如List<String>和List不应共同使用。
* 使用带通配符(？)的泛型类时，需要明确通配符所代表的一组类型的概念。
* 不能实例化泛型类型变量，然后利用反射的newInstance来创建实例；同样，不能创建一个泛型的数组。
* 尽量不要忽视编译器给出的警告。

# 线程

### 一、线程的概念

**线程--什么是进程**

进程--概念

要解释线程，就必须明白什么是进程。

什么是进程呢？

    进程是指运行中的应用程序，每个进程都有自己独立的地址空间(内存空间)，比如用户点击桌面的IE浏览器，就启动了一个进程，操作系统就会为该进程分配独立的地址空间。当用户再次点击左面的IE浏览器，又启动了一个进程，操作系统将为新的进程分配新的独立的地址空间。目前操作系统都支持多进程。

**要点：**用户每启动一个进程，操作系统就会为该进程分配一个独立的内存空间。

线程--概念

在明白进程后，就比较容易理解线程的概念。

什么是**线程**呢？

    是进程中的一个实体，是被系统独立调度和分派的基本单位，线程自己不拥有系统资源，只拥有一点在运行中必不可少的资源，但它可与同属一个进程的其它线程共享进程所拥有的全部资源。一个线程可以创建和撤消另一个线程，同一进程中的多个线程之间可以并发执行。线程有就绪、阻塞和运行三种基本状态。

**线程**

1、线程是轻量级的进程

2、线程没有独立的地址空间(内存空间)

3、线程是由进程创建的(寄生在进程)

4、一个进程可以拥有多个线程-->这就是我们常说的多线程编程

5、线程有几种状态：

  a、新建状态(new) Thread t = new MyThread();

  b、就绪状态(Runnable)：当调用线程对象的start()方法（t.start();），线程即进入就绪状态。处于就绪状态的线程，只是说明此线程已经做好了准备，随时等待CPU调度执行，并不是说执行了t.start()此线程立即就会执行；

  c、运行状态(Running)：

当CPU开始调度处于就绪状态的线程时，此时线程才得以真正执行，即进入到运行状态。注：就     绪状态是进入到运行状态的唯一入口，也就是说，线程要想进入运行状态执行，首先必须处于就绪状态中；

  d、阻塞状态(Blocked)：

处于运行状态中的线程由于某种原因，暂时放弃对CPU的使用权，停止执行，此时进入阻塞状态，直到其进入到就绪状态，才 有机会再次被CPU调用以进入到运行状态。根据阻塞产生的原因不同，阻塞状态又可以分为三种：

1.等待阻塞：运行状态中的线程执行wait()方法，使本线程进入到等待阻塞状态；

2.同步阻塞 -- 线程在获取synchronized同步锁失败(因为锁被其它线程所占用)，它会进入同步阻塞状态；

3.其他阻塞 -- 通过调用线程的sleep()或join()或发出了I/O请求时，线程会进入到阻塞状态。当sleep()状态超时、join()等待线程终止或者超时、或者I/O处理完毕时，线程重新转入就绪状态。

  e、死亡状态(Dead)：线程执行完了或者因异常退出了run()方法，该线程结束生命周期。

线程有什么用处

java程序中流传一句话，不会使用线程就别跟别人说自己学过java。目前绝大部分应用程序都会涉及到多并发的问题。只要应用程序涉及到并发，就离不开多线程编程。

线程--如何使用

**在java中一个类要当作线程来使用有两种方法**。

**1、继承Thread类，并重写run函数**

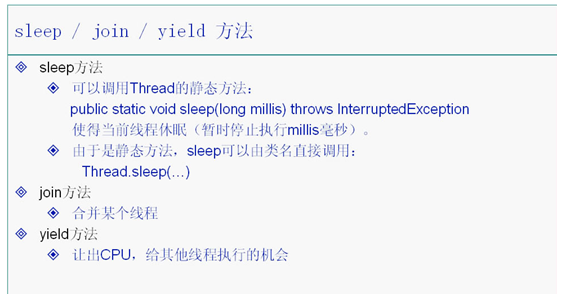
**2、实现Runnable接口，并重写run函数**

因为java是单继承的，在某些情况下一个类可能已经继承了某个父类，这时在用继承Thread类方法来创建线程显然不可能java设计者们提供了另外一个方式创建线程，就是通过实现Runnable接口来创建线程。

线程理解：线程是一个程序里面不同的执行路径



Sleep/join/yield方法介绍



# Gson 和 fastjson

## 1.json-lib

json-lib最开始的也是应用最广泛的json解析工具，json-lib 不好的地方确实是依赖于很多第三方包，  
包括commons-beanutils.jar，commons-collections-3.2.jar，commons-lang-2.6.jar，commons-logging-1.1.1.jar，ezmorph-1.0.6.jar，  
对于复杂类型的转换，json-lib对于json转换成bean还有缺陷，比如一个类里面会出现另一个类的list或者map集合，json-lib从json到bean的转换就会出现问题。  
json-lib在功能和性能上面都不能满足现在互联网化的需求。

## 2.开源的Jackson

相比json-lib框架，Jackson所依赖的jar包较少，简单易用并且性能也要相对高些。  
而且Jackson社区相对比较活跃，更新速度也比较快。  
Jackson对于复杂类型的json转换bean会出现问题，一些集合Map，List的转换出现问题。  
Jackson对于复杂类型的bean转换Json，转换的json格式不是标准的Json格式

## 3.Google的Gson

Gson是目前功能最全的Json解析神器，Gson当初是为因应Google公司内部需求而由Google自行研发而来，  
但自从在2008年五月公开发布第一版后已被许多公司或用户应用。  
Gson的应用主要为toJson与fromJson两个转换函数，无依赖，不需要例外额外的jar，能够直接跑在JDK上。  
而在使用这种对象转换之前需先创建好对象的类型以及其成员才能成功的将JSON字符串成功转换成相对应的对象。  
类里面只要有get和set方法，Gson完全可以将复杂类型的json到bean或bean到json的转换，是JSON解析的神器。  
Gson在功能上面无可挑剔，但是性能上面比FastJson有所差距。

## 4.阿里巴巴的FastJson

Fastjson是一个Java语言编写的高性能的JSON处理器,由阿里巴巴公司开发。  
无依赖，不需要例外额外的jar，能够直接跑在JDK上。  
FastJson在复杂类型的Bean转换Json上会出现一些问题，可能会出现引用的类型，导致Json转换出错，需要制定引用。  
FastJson采用独创的算法，将parse的速度提升到极致，超过所有json库。  
  
综上4种Json技术的比较，在项目选型的时候可以使用Google的Gson和阿里巴巴的FastJson两种并行使用，  
如果只是功能要求，没有性能要求，可以使用google的Gson，

# Vue study

## 模板语法：

Vue.js 使用了基于 HTML 的模板语法，允许开发者声明式地将 DOM 绑定至底层 Vue 实例的数据。所有 Vue.js 的模板都是合法的 HTML ，所以能被遵循规范的浏览器和 HTML 解析器解析。

在底层的实现上，Vue 将模板编译成虚拟 DOM 渲染函数。结合响应系统，Vue 能够智能地计算出最少需要重新渲染多少组件，并把 DOM 操作次数减到最少。

如果你熟悉虚拟 DOM 并且偏爱 JavaScript 的原始力量，你也可以不用模板，**[直接写渲染 (render) 函数](https://cn.vuejs.org/v2/guide/render-function.html)**，使用可选的 JSX 语法。

渲染函数就是JavaScript完全编写能力

## 计算属性

例子：

|  |
| --- |
| <div id="example">  <p>Original message: "{{ message }}"</p>  <p>Computed reversed message: "{{ reversedMessage }}"</p> </div> |

|  |
| --- |
| var vm = new Vue({  el: '#example',  data: {  message: 'Hello'  },  computed: {  // 计算属性的 getter  reversedMessage: function () {  // `this` 指向 vm 实例  return this.message.split('').reverse().join('')  }  } }) |

这里我们声明了一个计算属性 reversedMessage。我们提供的函数将用作属性 vm.reversedMessage 的 getter 函数：

## 计算属性 vs 方法

例子：

|  |
| --- |
| <p>Reversed message: "{{ reversedMessage() }}"</p> |

|  |
| --- |
| // 在组件中 methods: {  reversedMessage: function () {  return this.message.split('').reverse().join('')  } } |

**对比不同点：**

**计算属性是基于它们的依赖进行缓存的**。计算属性只有在它的相关依赖发生改变时才会重新求值。这就意味着只要 message 还没有发生改变，多次访问 reversedMessage 计算属性会立即返回之前的计算结果，而不必再次执行函数。

相比之下，每当触发重新渲染时，调用方法将**总会**再次执行函数

**为什么需要缓存：**

我们为什么需要缓存？假设我们有一个性能开销比较大的计算属性 **A**，它需要遍历一个巨大的数组并做大量的计算。然后我们可能有其他的计算属性依赖于 **A** 。如果没有缓存，我们将不可避免的多次执行 **A** 的 getter！如果你不希望有缓存，请用方法来替代。

## 计算属性 vs 侦听属性

Watch侦听例子：

|  |
| --- |
| <div id="demo">{{ fullName }}</div> |

|  |
| --- |
| var vm = new Vue({  el: '#demo',  data: {  firstName: 'Foo',  lastName: 'Bar',  fullName: 'Foo Bar'  },  watch: {  firstName: function (val) {  this.fullName = val + ' ' + this.lastName  },  lastName: function (val) {  this.fullName = this.firstName + ' ' + val  }  } }) |

上面代码必须要侦听相关属性的变化在进行相应的赋值转换

计算属性例子：

var vm = new Vue({  
 el: '#demo',  
 data: {  
 firstName: 'Foo',  
 lastName: 'Bar'  
 },  
 computed: {  
 fullName: function () {  
 return this.firstName + ' ' + this.lastName  
 }  
 }  
})

## 侦听器

什么是侦听器：

当数据变化时执行异步或开销较大的操作时，使用侦听器。

例子：

|  |
| --- |
| <div id="watch-example">  <p>  Ask a yes/no question:  <input v-model="question">  </p>  <p>{{ answer }}</p> </div> |

|  |
| --- |
| <!-- 因为 AJAX 库和通用工具的生态已经相当丰富，Vue 核心代码没有重复 --> <!-- 提供这些功能以保持精简。这也可以让你自由选择自己更熟悉的工具。 --> <script src="https://cdn.jsdelivr.net/npm/axios@0.12.0/dist/axios.min.js"></script> <script src="https://cdn.jsdelivr.net/npm/lodash@4.13.1/lodash.min.js"></script> <script> var watchExampleVM = new Vue({  el: '#watch-example',  data: {  question: '',  answer: 'I cannot give you an answer until you ask a question!'  },  watch: {    // 如果 `question` 发生改变，这个函数就会运行  question: function (newQuestion, oldQuestion) {  this.answer = 'Waiting for you to stop typing...'  this.debouncedGetAnswer()  }  },  created: function () {    // `\_.debounce` 是一个通过 Lodash 限制操作频率的函数。    // 在这个例子中，我们希望限制访问 yesno.wtf/api 的频率    // AJAX 请求直到用户输入完毕才会发出。想要了解更多关于  // `\_.debounce` 函数 (及其近亲 `\_.throttle`) 的知识，  // 请参考：https://lodash.com/docs#debounce  this.debouncedGetAnswer = \_.debounce(this.getAnswer, 500)  },  methods: {  getAnswer: function () {  if (this.question.indexOf('?') === -1) {  this.answer = 'Questions usually contain a question mark. ;-)'  return  }  this.answer = 'Thinking...'  var vm = this  axios.get('https://yesno.wtf/api')  .then(function (response) {  vm.answer = \_.capitalize(response.data.answer)  })  .catch(function (error) {  vm.answer = 'Error! Could not reach the API. ' + error  })  }  } }) </script> |

在这个示例中，使用 watch 选项允许我们执行异步操作 (访问一个 API)，限制我们执行该操作的频率，并在我们得到最终结果前，设置中间状态。这些都是计算属性无法做到的。

## Class 与 Style绑定

操作元素的 class 列表和内联样式是数据绑定的一个常见需求。因为它们都是属性，所以我们可以用 v-bind 处理它们：只需要通过表达式计算出字符串结果即可。不过，字符串拼接麻烦且易错。因此，在将 v-bind 用于 class 和 style 时，Vue.js 做了专门的增强。表达式结果的类型除了字符串之外，还可以是对象或数组。

## 条件渲染

例子：

<template v-if="loginType === 'username'">  
 <label>Username</label>  
 <input placeholder="Enter your username">  
</template>  
<template v-else>  
 <label>Email</label>  
 <input placeholder="Enter your email address">  
</template>

那么在上面的代码中切换 loginType 将不会清除用户已经输入的内容。因为两个模板使用了相同的元素，<input> 不会被替换掉——仅仅是替换了它的 placeholder。

**使用key**

<template v-if="loginType === 'username'">  
 <label>Username</label>  
 <input placeholder="Enter your username" key="username-input">  
</template>  
<template v-else>  
 <label>Email</label>  
 <input placeholder="Enter your email address" key="email-input">  
</template>

**这样每次渲染后都会清空**

### v-if 与v-show 对比

注意，v-show 不支持 <template> 元素，也不支持 v-else。

v-if 是“真正”的条件渲染，因为它会确保在切换过程中条件块内的事件监听器和子组件适当地被销毁和重建。

v-if 也是**惰性的**：如果在初始渲染时条件为假，则什么也不做——直到条件第一次变为真时，才会开始渲染条件块。

相比之下，v-show 就简单得多——不管初始条件是什么，元素总是会被渲染，并且只是简单地基于 CSS 进行切换。

一般来说，v-if 有更高的切换开销，而 v-show 有更高的初始渲染开销。因此，如果需要非常频繁地切换，则使用 v-show 较好；如果在运行时条件很少改变，则使用 v-if 较好。

### 事件修饰符：

在事件处理程序中调用 event.preventDefault() 或 event.stopPropagation() 是非常常见的需求。尽管我们可以在方法中轻松实现这点，但更好的方式是：方法只有纯粹的数据逻辑，而不是去处理 DOM 事件细节。

<!-- 阻止单击事件继续传播 -->  
<a v-on:click.stop="doThis"></a>  
  
<!-- 提交事件不再重载页面 -->  
<form v-on:submit.prevent="onSubmit"></form>  
  
<!-- 修饰符可以串联 -->  
<a v-on:click.stop.prevent="doThat"></a>  
  
<!-- 只有修饰符 -->  
<form v-on:submit.prevent></form>  
  
<!-- 添加事件监听器时使用事件捕获模式 -->  
<!-- 即元素自身触发的事件先在此处处理，然后才交由内部元素进行处理 -->  
<div v-on:click.capture="doThis">...</div>  
  
<!-- 只当在 event.target 是当前元素自身时触发处理函数 -->  
<!-- 即事件不是从内部元素触发的 -->  
<div v-on:click.self="doThat">...</div>

2.14新增

<!-- 点击事件将只会触发一次 -->  
<a v-on:click.once="doThis"></a>

# Redis缓存策略

#### Jedis set存储

1. package com.component.jedis;
2. import java.util.Set;
3. import redis.clients.jedis.Jedis;
4. public class JedisSet extends JedisCommon{
6. private static final String OK\_CODE = "OK";
7. private static final String OK\_MULTI\_CODE = "+OK";
9. /\*\*
10. \* 把一个或多个元素添加到指定集合
11. \*
12. \* @param key
13. \* @param members
14. \* @return
15. \*/
16. public static Long sadd(String key, String members){
17. Jedis jedis = JedisManager.getJedis();
18. Long result ;
19. try {
20. result = jedis.sadd(key, members);
21. }finally{
22. JedisManager.returnResource(jedis);
23. }
24. return result;
25. }
27. /\*\*
28. \* 返回多个集合的差集
29. \* @param keys
30. \* @return
31. \*/
32. public static Set<String> sdiff(String... keys){
33. Jedis jedis = JedisManager.getJedis();
34. Set<String> result ;
35. try {
36. result = jedis.sdiff(keys);
37. }finally{
38. JedisManager.returnResource(jedis);
39. }
40. return result;
41. }
43. /\*\*
44. \* 返回多个集合的交集
45. \*
46. \* @param keys
47. \* @return
48. \*/
49. public static Set<String> sinter(String... keys){
50. Jedis jedis = JedisManager.getJedis();
51. Set<String> result ;
52. try {
53. result = jedis.sinter(keys);
54. }finally{
55. JedisManager.returnResource(jedis);
56. }
57. return result;
58. }
60. /\*\*
61. \* 判断元素是否是集合成员
62. \*
63. \* @param key
64. \* @param member
65. \* @return
66. \*/
67. public static boolean sismember(String key, String member){
68. Jedis jedis = JedisManager.getJedis();
69. boolean result ;
70. try {
71. result = jedis.sismember(key, member);
72. }finally{
73. JedisManager.returnResource(jedis);
74. }
75. return result;
76. }
78. /\*\*
79. \* 返回集合所有成员
80. \*
81. \* @param key
82. \* @return
83. \*/
84. public static Set<String> smembers(String key){
85. Jedis jedis = JedisManager.getJedis();
86. Set<String> result ;
87. try {
88. result = jedis.smembers(key);
89. }finally{
90. JedisManager.returnResource(jedis);
91. }
92. return result;
93. }
95. /\*\*
96. \* 把指定成员从一个集合移动到目标集合，
97. \*  指定成员不存在，不执行任何操作
98. \* @param srckey
99. \* @param dstkey
100. \* @param member
101. \* @return
102. \*/
103. public static Long smove(String srckey, String dstkey, String member){
104. Jedis jedis = JedisManager.getJedis();
105. Long result ;
106. try {
107. result = jedis.smove(srckey, dstkey, member);
108. }finally{
109. JedisManager.returnResource(jedis);
110. }
111. return result;
112. }
114. /\*\*
115. \* 随机移除并返回一个元素
116. \*
117. \* @param key
118. \* @return
119. \*/
120. public static String spop(String key){
121. Jedis jedis = JedisManager.getJedis();
122. String result ;
123. try {
124. result = jedis.spop(key);
125. }finally{
126. JedisManager.returnResource(jedis);
127. }
128. return result;
129. }
131. /\*\*
132. \* 移除一个或多个元素，不存在的  元素会被忽略
133. \*
134. \* @param key
135. \* @param members
136. \* @return
137. \*/
138. public static Long srem(String key, String... members){
139. Jedis jedis = JedisManager.getJedis();
140. Long result ;
141. try {
142. result = jedis.srem(key, members);
143. }finally{
144. JedisManager.returnResource(jedis);
145. }
146. return result;
147. }
149. /\*\*
150. \* 返回指定集合的并集
151. \*
152. \* @param keys
153. \* @return
154. \*/
155. public static Set<String> sunion(String... keys){
156. Jedis jedis = JedisManager.getJedis();
157. Set<String> result ;
158. try {
159. result = jedis.sunion(keys);
160. }finally{
161. JedisManager.returnResource(jedis);
162. }
163. return result;
164. }
166. /\*\*
167. \* 当key不存在时，才放入值,超时时间单位为秒
168. \* @param jedis
169. \* @param key
170. \* @param value
171. \* @return
172. \*/
173. public static boolean set(String key, String value, Long timeOut){
175. Jedis jedis = null;
176. String resp = null;
177. try{
178. jedis = JedisManager.getJedis();
179. resp = jedis.set(key, value, "NX", "EX", timeOut == null ? 1L : timeOut);
180. } finally{
181. JedisManager.returnResource(jedis);
182. }
183. return isStatusOk(resp);
184. }
186. /\*\*
187. \* 判断 返回值是否ok.
188. \*/
189. public static boolean isStatusOk(String status) {
190. return (status != null) && (OK\_CODE.equals(status) || OK\_MULTI\_CODE.equals(status));
191. }
192. }

**Spring @Transactional的注意事项**

* @Transactional 注解应该只被应用到 public 可见度的方法上。 如果你在 protected、private 或者 package-visible 的方法上使用 @Transactional 注解，它也不会报错， 但是这个被注解的方法将不会展示已配置的事务设置。
* 用 spring 事务管理器，由spring来负责数据库的打开，提交，回滚。默认遇到运行期异常(throw new RuntimeException("注释");)会回滚，即遇到不受检查（unchecked）的异常时回滚；而遇到需要捕获的异常(throw new Exception("注释");)不会回滚，即遇到受检查的异常（就是非运行时抛出的异常，编译器会检查到的异常叫受检查异常或说受检查异常）时，需我们指定方式来让事务回滚 要想所有异常都回滚，要加上 @Transactional( rollbackFor={Exception。class，其它异常}) 。如果让unchecked异常不回滚： @Transactional(notRollbackFor=RunTimeException.class)如下:

1. @Transactional(rollbackFor=Exception.class) //指定回滚，遇到异常Exception时回滚
2. public void methodName()
3. {
4. throw new Exception("注释");
5. }
6. @Transactional(noRollbackFor=Exception.class)//指定不回滚，遇到运行期异常(throw new RuntimeException("注释");)会回滚
7. public ItimDaoImpl getItemDaoImpl()
8. {
9. throw new RuntimeException("注释");
10. }
11. public void methodName()
12. {
13. // 本类的修改方法 1
14. update();
15. // 调用其他类的修改方法
16. otherBean.update();
17. // 本类的修改方法 2
18. update();
19. }
20. /\*other失败了不会影响 本类的修改提交成功
21. 本类update的失败，other也失败
22. \*/

* @Transactional 注解可以被应用于接口定义和接口方法、类定义和类的 public 方法上。然而，请注意仅仅 @Transactional 注解的出现不足于开启事务行为，它仅仅是一种元数据，能够被可以识别 @Transactional 注解和上述的配置适当的具有事务行为的beans所使用。上面的例子中，其实正是 <tx:annotation-driven/>元素的出现 开启了事务行为。
* Spring团队的建议是你在具体的类（或类的方法）上使用 @Transactional 注解，而不要使用在类所要实现的任何接口上。你当然可以在接口上使用 @Transactional 注解，但是这将只能当你设置了基于接口的代理时它才生效。因为注解是不能继承的，这就意味着如果你正在使用基于类的代理时，那么事务的设置将不能被基于类的代理所识别，而且对象也将不会被事务代理所包装（将被确认为严重的）。因此，请接受Spring团队的建议并且在具体的类上使用 @Transactional 注解。
* @Transactional 注解标识的方法，处理过程尽量的简单。尤其是带锁的事务方法，能不放在事务里面的最好不要放在事务里面。可以将常规的数据库查询操作放在事务前面进行，而事务内进行增、删、改、加锁查询等操作。

 这个地方就没必要嵌套事务了吧！

 这样的业务可以放在事务成功之后，发送异步消息更新。

* @Transactional 注解的默认事务管理器bean是“transactionManager”，如果声明为其他名称的事务管理器，需要在方法上添加@Transational("managerName")。
* @Transactional 注解标注的方法中不要出现网络调用、比较耗时的处理程序，因为，事务中数据库连接是不会释放的，如果每个事务的处理时间都非常长，那么宝贵的数据库连接资源将很快被耗尽。

**5**、**Spring @Transactional自我调用中的问题**

# Java io 字节流

套接字：

套接字，是支持[TCP/IP](https://baike.baidu.com/item/TCP%2FIP" \t "_blank)的[网络通信](https://baike.baidu.com/item/%E7%BD%91%E7%BB%9C%E9%80%9A%E4%BF%A1" \t "_blank)的基本操作单元，可以看做是不同主机之间的进程进行双向通信的端点，简单的说就是通信的两方的一种约定，用套接字中的相关函数来完成通信过程。

非常非常简单的举例说明下:Socket=Ip address+ TCP/UDP + port