# 最近总结

### 1.get和set，getter，setter区别

get的方法都是用来取值

set是用来赋值的

setter方法，是对象引用调用setter方法，并传入参数。

getter方法，对象调用getter方法，获取对象属性的值。

### 2.request和会话区别

request范围较小一些，只是一个请求。

request对象的生命周期是针对一个客户端的一次请求，当请求完毕之后，request里边的内容也将被释放点 。

当结果输出之后，request就结束了

会话是用户打开浏览器，点击多个超链接,访问Web服务器上多个资源,然后关闭浏览器,整个过程称之为一次会话

session的生命周期也是针对一个客户端 但是却是在别人设置的会话周期内(一般是20-30分钟) session里边的内容将一直存在 即便关闭了这个客户端浏览器 session也不一定会马上释放掉的

### 3.request/response/session/application的底层用的什么数据结构？

response不知道其他事map

### 常见的表单元素

|  |  |
| --- | --- |
| 文本框 | text |
| 密码框 | password |
| 隐藏域 | hidden |
| 文本域 | textarea |
| 普通按钮 | button |
| 提交按钮 | submit |
| 重置按钮 | reset |
| 图片按钮 | image |
| 单选框 | radio |
| 复选框 | checkbox |
| 下拉框 |  |

### 4.spring?

#### IOC

**Ioc—Inversion of Control，即“控制反转”，不是什么技术，而是一种设计思想。**在Java开发中，**Ioc意味着将你设计好的对象交给容器控制，而不是传统的在你的对象内部直接控制。**

**就是说创建对象的控制权进行转移，以前创建对象的主动权和创建时机是由自己把控的，而现在这种权力转移到第三方**，比如转移交给了IoC容器，它就是一个专门用来创建对象的工厂，你要什么对象，它就给你什么对象，有了 IoC容器，依赖关系就变了，原先的依赖关系就没了，它们都依赖IoC容器了，通过IoC容器来建立它们之间的关系。

### 传统的方式：fig:

### IoC/DI的容器后：fig:

#### AOP

* AOP称为面向切面编程，在程序开发中主要用来解决一些系统层面上的问题，比如日志，事务，权限等待，Struts2的拦截器设计就是基于AOP的思想，是个比较经典的例子。
* AOP的基本概念
* (1) Aspect(切面):通常是一个类，里面可以定义切入点和通知
* (2) JointPoint(连接点):
* AOP的思想:在原有功能不变的情况下，增强了功能
* 可以理解为一个拦截器框架，但是这个拦截器会非常武断，如果它拦截一个类，那么它就会拦截这个类中的所有方法。如对一个目标列的代理，增强了目标类的所有方法。

1. **按照自己的说法说你对Spring的理解**

* [谈谈对Spring IOC的理解](https://www.cnblogs.com/xdp-gacl/p/4249939.html)

#### Spring的注入用什么方式？

* **DI—Dependency Injection，即“依赖注入”**：**组件之间依赖关系**由容器在运行期决定，形象的说，即**由容器动态的将某个依赖关系注入到组件之中**。**依赖注入的目的并非为软件系统带来更多功能，而是为了提升组件重用的频率，并为系统搭建一个灵活、可扩展的平台。**通过依赖注入机制，我们只需要通过简单的配置，而无需任何代码就可指定目标需要的资源，完成自身的业务逻辑，而不需要关心具体的资源来自何处，由谁实现。

●**谁依赖于谁：**当然是**应用程序依赖于IoC容器**；

●**为什么需要依赖：**应用程序需要IoC容器来提供对象需要的外部资源；

●**谁注入谁：**很明显是**IoC容器注入应用程序某个对象，应用程序依赖的对象**；

**●注入了什么：**就是**注入某个对象所需要的外部资源（包括对象、资源、常量数据）**。

#### **IoC是什么？**

* Ioc—Inversion of Control，即“控制反转”，不是什么技术，而是一种设计思想。\*\*在Java开发中，\*\*Ioc意味着将你设计好的对象交给容器控制，而不是传统的在你的对象内部直接控制。
* **就是说创建对象的控制权进行转移，以前创建对象的主动权和创建时机是由自己把控的，而现在这种权力转移到第三方**，比如转移交给了IoC容器，它就是一个专门用来创建对象的工厂，你要什么对象，它就给你什么对象，有了 IoC容器，依赖关系就变了，原先的依赖关系就没了，它们都依赖IoC容器了，通过IoC容器来建立它们之间的关系。

#### 谁控制谁，控制什么？

* \*\*谁控制谁？当然是IoC 容器控制了对象；\*\*  
    
  \*\*控制什么？那就是主要控制了外部资源获取（不只是对象包括比如文件等）\*\*]

#### 为何是反转,哪些方面反转了？

* 有反转就有正转，传统应用程序是由我们自己在对象中主动控制去直接获取依赖对象，也就是正转；\*\*而反转则是由容器来帮忙创建及注入依赖对象；\*\*
* **为何是反转？ 因为由容器帮我们查找及注入依赖对象，对象只是被动的接受依赖对象，所以是反转；**
* **哪些方面反转了？依赖对象的获取被反转了。**

#### IOC能做什么？

* IoC 不是一种技术，只是一种思想，一个重要的面向对象编程的法则，它能指导我们如何设计出松耦合、更优良的程序。传统应用程序都是由我们在类内部主动创建依赖对象，从而导致类与类之间高耦合，难于测试；\*\*有了IoC容器后，把创建和查找依赖对象的控制权交给了容器，由容器进行注入组合对象，\*\*所以对象与对象之间是 松散耦合，这样也方便测试，利于功能复用，更重要的是使得程序的整个体系结构变得非常灵活。

#### IOC和DI?

* \*\*IOC的实现方式是DI\*\*，DI又叫依赖注入，依赖注入的实现方式三种实现方式

**1. set注入**

**2.** **构造器注入**

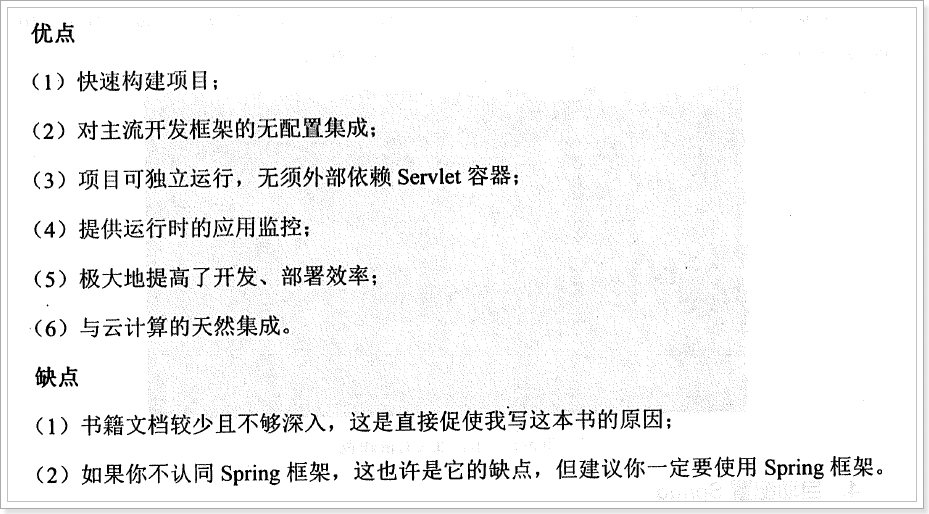
**3.** **接口注入/注解注入** **@Autowrid @Resource**

### 5.SpringBoot？**可以做什么？**

spring boot 是一个服务于框架的框架，服务范围是简化配置文件，一种全新的编程规范，他的产生简化了框架的使用，所谓简化是指简化了Spring众多框架中所需的大量且繁琐的配置文件，比如我们要集成redis,只需要引入redis的依赖,然后在application.properties或者application.yml配置必要的东西就可以了.

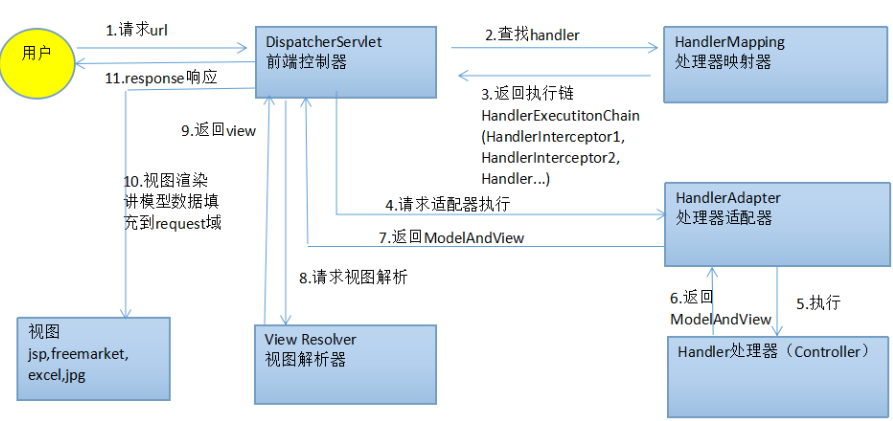
还有就是spring boot就是整合了很多优秀的框架，不用我们自己手动的去写一堆xml配置然后进行配置

**可以做什么？**

**让文件配置变的相当简单、让应用部署变的简单（SpringBoot内置服务器，并装备启动类代码），可以快速开启一个Web容器进行开发。**

### 6.SpringMVC?工作流程？

1. **Model**：承载数据，并对用户提交请求进行计算的模块。其分为两类，一类称为数据承载Bean，一类称为业务处理Bean。所谓数据承载Bean是指实体类，专门承载业务数据的，如Student、User等。而业务处理Bean则是指Service或Dao对象，专门用于处理用户提交请求的。
2. **View**：为用户提供使用界面，与用户直接进行交互
3. **Controller**：用于将用户请求转发给相应的Model进行处理，并将处理Model的计算结果向用户提供相应响应。



### 7.Servlet/Filter的生命周期！！！！！！！！

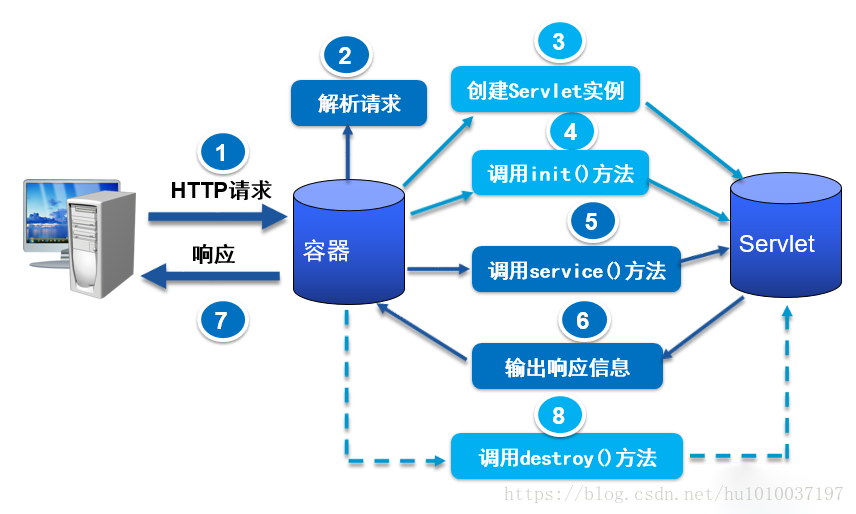
1 第一次请求Servlet的时候，此时会调用Servlet的init方法，初始化这个Servlet，Servlet只需要初始化一次，在当前tomcat不关闭的情况下，就会一直提供服务

2 初始化完成之后，调用Servlet的中方法doGet、doPost方法（根据请求方式自动识别），处理业务

3 业务处理完成，响应数据

4 当处理完某次请求之后，servlet就进入等待状态，等待浏览器的下一次请求

5 当关闭tomcat的时候，自动销毁servlet



### 8.mybatis#号和$符的区别？

* # 相当于对数据加上 双引号 ，$ 相当于直接显示数据
* 1.#将传入的数据都当成一个字符串，会对自动传入的数据加一个双引号。如：order by #user\_id#，如果传入的值是111,那么解析成sql时的值为order by "111", 如果传入的值是id，则解析成的sql为order by "id".
* 2.$将传入的数据直接显示生成在sql中。如：order by $user*id$，如果传入的值是111,那么解析成sql时的值为order by user*id, 如果传入的值是id，则解析成的sql为order by id.
* #方式能够很大程度防止sql注入。
* $方式无法防止Sql注入。
* $方式一般用于传入数据库对象，例如传入表名
* 一般能用#就别用$

### 9.mybatis的编程步骤

1. 创建SqlSessionFactory
2. 通过SqlSessionFactory创建SqlSession
3. 通过sqlSession执行数据库操作
4. 调用session.commit()提交事务
5. 调用session.close()关闭会话

### 10.在学mybatis之前你是怎样连接数据库的的？

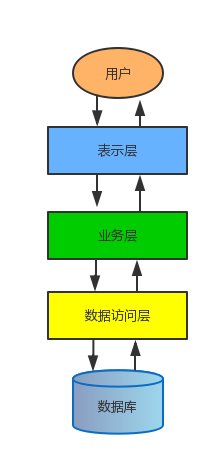
JDBC

### 说一下JDBC操作数据库的流程？

1. Class.forName加载数据库链接驱动
2. DriverManager.getConnection()获取数据库连接对象
3. 根据SQL获取sql会话对象，有两种方式，那两种忘了
4. 执行SQL处理结果
5. 关闭

### 11.三层思想是什么?(Controller,service,dao)他们之间的联系是什么?

1. **表示层**：负责与用户进行交互。
2. **业务逻辑层**：主要是针对具体的问题的操作，也可以理解成对数据层的操作，对数据业务逻辑处理。
3. **数据访问层**：主要是对非原始数据（数据库或者文本文件等存放数据的形式）的操作层，也就是说是对数据库，而不是对数据的操作，具体为业务逻辑层或用户界面层提供数据服务。

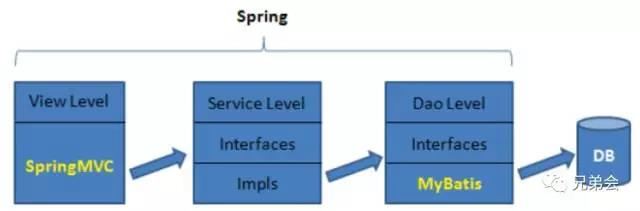
* 

### 12.SSM?SSM与三层架构的关系

###### [MVC，SSM与三层架构的构成及相互关系](https://www.cnblogs.com/fjjchina/p/7099713.html)

SSM主要由Spring，SpringMVC 和 Mybatis三个构成。它们在三层架构中所处的位置是不同的，即它们在三层架构中的功能各不相同，各司其职

1. SpringMVC：作为View层的实现者，完成用户的请求接收功能。SpringMVC的Controller作为整个应用的控制器，完成用户请求的转发及对用户的响应
2. MyBatis：作为 Dao层的实现者，完成对数据库的增、删、改、查功能
3. Spring：以整个应用大管家的身份出现。整个应用中所有的Bean的生命周期行为，均由Spring来管理。即整个应用中所有对象的创建、初始化、销毁，及对象间关联关系的维护，均由Spring进行管理

* 

### 13.数据访问层（dao）与数据库是怎样交互的

用JDBC来实现访问数据库记录可以采用下面的几个步骤：

1、 通过驱动器管理器获取连接接口。

2、 获得Statement或它的子类。

3、 限制Statement中的参数。

4、 执行Statement。

5、 查看返回的行数是否超出范围。

6、 关闭Statement。

7、 处理其它的Statement

8、 关闭连接接口。

### 14.数据库的优化?有哪些?

[八大优化方案](https://blog.csdn.net/liuyanqiangpk/article/details/79827239)

###### 选取最适用的字段属性

###### 使用连接（JOIN）来代替子查询(Sub-Queries)

###### 使用联合(UNION)来代替手动创建的临时表

###### 事务

###### 锁定表

###### 使用外键

###### 使用索引

###### 优化的查询语句

### 15.数据表结构?

数据表是由表名、表中的字段和表的记录三个部分组成的。设计数据表结构就是定义数据表文件名，确定数据表包含哪些字段，各字段的字段名、字段类型、及宽度，并将这些数据输入到计算机当中

### 16.mySql内连接和外链接的区别

* 内连接(笛卡尔积)，也被称为自然连接，内联接使用比较运算符根据每个表共有的列的值匹配两个表中的行

select fieldlist from table1 [inner] join table2 on table1.column = table2.column

* 左向外联接(LEFT JOIN或LEFT OUTER JOIN)的结果集包括
* LEFT OUTER子句中指定的左表的所有行，而不仅仅是联接列所匹配的行。如果\*\*左表的某行在右表中没有匹配行\*\*，则在相关联的结果集行中\*\*右表的所有选择列表列均为空值\*\*
* 右向外联接(RIGHT JOIN 或 RIGHT OUTER JOIN)是左向外联接的反向联接。
* 将返回右表的所有行。如果\*\*右表的某行在左表中没有匹配行\*\*，则将为\*\*左表返回空值。\*\*
* 完整外部联接(FULL JOIN 或 FULL OUTER JOIN)返回左表和右表中的所有行。
* 当某行在另一个表中没有匹配行时，则另一个表的选择列表列包含空值。如果表之间有匹配行，则整个结果集行包含基表的数据值。

a表 id name b表 id age parent\_id

1 张3 1 23 1   
  
 2 李四 2 34 2   
  
 3 王武 3 34 4

a.id同parent\_id 存在关系

1） 内连接 select a.*,b.* from a inner join b on a.id=b.parent\_id

结果是

id name id age parent\_id

1 张3 1 23 1

2 李四 2 34 2

2）左连接 select a.*,b.* from a left join b on a.id=b.parent\_id

结果是

id name id age parent\_id

1 张3 1 23 1

2 李四 2 34 2

3 王武 null

3） 右连接 select a.*,b.* from a right join b on a.id=b.parent\_id

结果是

id name id age parent\_id

1 张3 1 23 1

2 李四 2 34 2

null 3 34 4

4） 完全连接 select a.*,b.* from a full join b on a.id=b.parent\_id

结果是

id name id age parent\_id

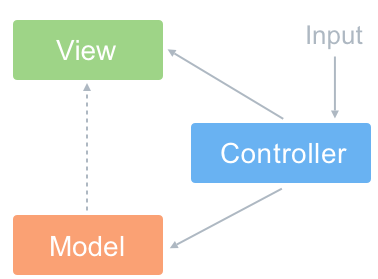
1 张3 1 23 1

2 李四 2 34 2

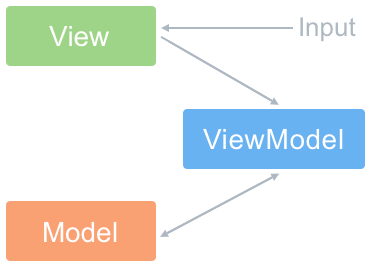
null 3 34 4

3 王武 null

### 17.简单介绍前端MVC/MVVM模式特点及区别

**简介一下MVC**：

如果前端没有框架，只使用原生的html+js，MVC模式可以这样理解。**将html看成view**;**js看成controller**，负责处理用户与应用的交互，响应对view的操作（对事件的监听），调用Model对数据进行操作，完成model与view的同步（根据model的改变，通过选择器对view进行操作）;**将js的ajax当做Mode**l，也就是数据层，通过ajax从服务器获取数据。

**MVVM**:

new Vue({})

这里的html部分相当于View层，可以看到这里的View通过通过模板语法来声明式的将数据渲染进DOM元素，当ViewModel对Model进行更新时，通过数据绑定更新到View。  
  
Vue实例中的data相当于Model层，而ViewModel层的核心是Vue中的双向数据绑定，即Model变化时VIew可以实时更新，View变化也能让Model发生变化。

**区别：**MVVM比MVC精简很多，不仅简化了业务与界面的依赖，还解决了数据频繁更新的问题，不用再用选择器操作DOM元素。因为在MVVM中，View不知道Model的存在，Model和ViewModel也观察不到View，这种低耦合模式提高代码的可重用性。

### 18.对于VUE的理解？为什么使用vue？有什么好处？

是一套构建用户界面的 渐进式框架。与其他重量级框架不同的是，Vue 采用自底向上增量开发的设计。Vue 的核心库只关注视图层，并且非常容易学习，非常容易与其它库或已有项目整合。另一方面，Vue 完全有能力驱动采用[单文件组件](http://link.zhihu.com/?target=http%3A//cn.vuejs.org/v2/guide/single-file-components.html)和 [Vue 生态系统支持的库](http://link.zhihu.com/?target=http%3A//github.com/vuejs/awesome-vue%23libraries--plugins)开发的复杂单页应用。

Vue.js 的目标是通过尽可能简单的 API 实现双向数据绑定。

Vue 的核心库只关注视图层

**好处：**能让基于网页的前端应用程序开发起来这么方便，因为Vue.js有声明式，响应式的数据绑定，与组件化的开发，并且还使用了Virtual DOM这个看名字就觉得高大上的技术

**Virtual DOM则是虚拟DOM的英文:**

js的运行速度已经很快了，然而大量的DOM 操作就会变得很慢，但是前端本身就是要通过JS处理DOM 来更新视图数据的。 这样在更新数据后会重新渲染页面，这样就造成在没有改变数据的地方也重新渲染了DOM节点。这样性能方面就会很受影响。

利用在内存中生成与真实DOM与之对应的数据结构，这个在内存中生成的结构称之为虚拟DOM。

当数据发生变化时，能够智能的计算出重新渲染组件的最小代价并应用到DOM操作上。vue就是利用了这一点。



### 19.vue中this关键字？

答： 1 this代表当前的vm对象，访问属性的时候，可以借助this.属性，相当于就vm.属性访问 2 在html标签中，访问vue的属性的时候，不能加this关键字 3 // 匿名函数，此时this就指当前的function函数，所以通过this.goodsInfo访问的不是vue的data数据，所以会失败 getGoodsInfo(this.skuid).then(function(res){ console.log(res.data) // 此处的赋值 为什么会失败？ this.goodsInfo = res.data }) 使用箭头函数不会影响this关键字的用法

### 20.组件的要点？

**A** **组件名字一定要小写**

**B** **组件的属性** **data**，一定是一个方法，而且方法必须要有返回值

**C** **组件必须要有根标签，而且只能有一个根标签**

**D** **父组件可以给子组件传值，通过：的方式将父组件的值传递进来**

**E** **子组件可以给父子间传值：通过 $emit** 的方式传值

### 21.json和Fastjson的区别

[JSON与fastjson](https://www.cnblogs.com/hh6plus/p/5548050.html)

### 22.JQuery的选择权有哪些？

### 23.java的面向对象？

封装、抽象、继承和多态。

封装：在面向对象语言中，封装特性是由类来体现的，我们将现实生活中的一类实体定义成类，其中包括属性和行为（在Java中就是方法），就好像人类，可以具有name,sex,age等属性，同时也具有eat(),sleep()等行为，我们在行为中实现一定的功能，也可操作属性，这是面向对象的封装特性；

抽象：抽象就是将一类实体的共同特性抽象出来，封装在一个抽象类中，所以抽象在面向对象语言是由抽象类来体现的。比如鸟就是一个抽象实体，因为抽象实体并不是一个真正的对象，它的属性还不能完全描述一个对象，所以在语言中体现为抽象类不能实例化；

继承：继承就像是我们现实生活中的父子关系，儿子可以遗传父亲的一些特性，在面向对象语言中，就是一个类可以继承另一个类的一些特性，从而可以代码重用，其实继承体现的是is-a关系，父类同子类在本质上还是一类实体；

多态：多态就是通过传递给父类对象引用不同的子类对象从而表现出不同的行为

### 24.java的**基本数据类型**?并占字节分别是多少？

四类八种：整数型，字符型，浮点型，布尔型

**整数型**

byte 1字节

short 2字节

int 4字节

long 8字节

**字符型**

char 2字节（C语言中是1字节）可以存储一个汉字

**浮点型**

float 4字节

double 8字节

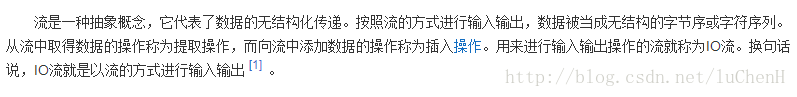
**布尔型**

boolean false/true(理论上占用1bit,1/8字节，实际处理按1byte处理)

### 25.String 和StringBuffer，StringBuilder的区别

* 运行速度，或者说是执行速度，**在这方面运行速度快慢为：StringBuilder > StringBuffer > String**
* Java中对String对象进行的操作实际上是一个不断创建新的对象并且将旧的对象回收的一个过程，所以执行速度很慢。  
    
  而StringBuilder和StringBuffer的对象是变量，对变量进行操作就是直接对该对象进行更改，而不进行创建和回收的操作，所以速度要比String快很多。
* 线程安全 , **在线程安全上，StringBuilder是线程不安全的，而StringBuffer是线程安全的**
* 如果一个StringBuffer对象在字符串缓冲区被多个线程使用时，StringBuffer中很多方法可以带有synchronized关键字，所以可以保证线程是安全的，但StringBuilder的方法则没有该关键字，所以不能保证线程安全，有可能会出现一些错误的操作。所以如果要进行的操作是多线程的，那么就要使用StringBuffer，但是在单线程的情况下，还是建议使用速度比较快的StringBuilder。
* 总结
* \*\*String：适用于少量的字符串操作的情况\*\*  
  \*\*StringBuilder：适用于单线程下在字符缓冲区进行大量操作的情况\*\*  
  \*\*StringBuffer：适用多线程下在字符缓冲区进行大量操作的情况\*\*

### 26.请解释一下io流是什么？



* 按照流向分为： 输入流和输出流
* 按照数据的表现形式分为 ：字节流（二进制，可以处理一切文件，文本，音频等）和字符流文本文件，只能是纯文本，像txt）

### 27.字节流如何转为字符流？

字节输入流转为字符输入流 通过InputStreamReader实现，该类的构造函数可以传入InputStream对象

字节输出流转为字符输出流 通过OutputStreamWriter实现，该类的构造函数可以传入OutputStream对象

### 28.字节流和字节流的区别？

1）在字节流中输出数据主要是使用OutputStream完成，输入使的是InputStream

2）在字符流中输出主要是使用Writer类完成，输入流主要使用Reader类完成。（这四个都是抽象类）

3） 字符流处理的单元为2个字节的Unicode字符，分别操作字符、字符数组或字符串，而字节流处理单元为1个字节，操作字节和字节数组。

### [**数据结构**](https://zhidao.baidu.com/question/1642368890567281380.html)**：**

### 30.JAVA中的**数据结构有哪些**？List、Set、Map的区别？各有什么特点？List的三个实现类？分别介绍一下？

##### List：

* 可以允许重复的对象。
* **可以插入多个null元素。**
* **是一个有序容器，保持了每个元素的插入顺序，输出的顺序就是插入的顺序。**
* 常用的**实现类有 ArrayList、LinkedList 和 Vector**。ArrayList 最为流行，它提供了使用索引的随意访问，而 **LinkedList 则对于经常需要从 List 中添加或删除元素的场合更为合适**。

##### Set ：

* 不允许重复对象
* 无序容器，你无法保证每个元素的存储顺序，TreeSet通过 Comparator 或者 Comparable 维护了一个排序顺序。

##### **只允许一个 null 元素**

* Set 接口最流行的几个**实现类是 HashSet、LinkedHashSet 以及 TreeSet**。最流行的是基于 HashMap 实现的 HashSet； **TreeSet 还实现了 SortedSet (有序Set)接口**，因此 TreeSet 是一个根据其 compare() 和 compareTo() 的定义进行排序的有序容器。

##### Map：

* 1.**Map不是collection的子接口或者实现类。Map是一个接口**
* 2.Map 的 每个 Entry 都持有两个对象，也就是一个键一个值，Map 可能会持有相同的值对象但键对象必须是唯一的。
* 3.TreeMap 也通过 Comparator 或者 Comparable 维护了一个排序顺序。
* 4.Map 里你可以拥有随意个 null 值但最多只能有一个 null 键。
* 5.Map 接口最流行的几个**实现类是 HashMap、LinkedHashMap、Hashtable 和 TreeMap**。（HashMap、TreeMap最常用）

#### **ArrayList、LinkedList 和 Vector**

**ArrayList**是最常用的List实现类，**内部是通过数组实现的**，它允许对元素进行快速随机访问。数组的缺点是每个元素之间不能有间隔，当数组大小不满足时需要增加存储能力，就要讲已经有数组的数据复制到新的存储空间中。**当从ArrayList的中间位置插入或者删除元素时，需要对数组进行复制、移动、代价比较高。因此，它适合随机查找和遍历，不适合插入和删除。**

**Vector**与ArrayList一样，也是通过**数组实现**的，不同的是**它支持线程的同步**，即**某一时刻只有一个线程能够写Vector**，避免多线程同时写而引起的不一致性，但实现同步需要很高的花费，因此，访问它比访问ArrayList慢。

**LinkedList**是用**链表结构存储数据**的，很**适合数据的动态插入和删除**，随机访问和遍历速度比较慢。另外，他还提供了List接口中没有定义的方法，**专门用于操作表头和表尾元素，可以当作堆栈、队列和双向队列使用**。

### 31.ArrayList内部用什么实现的

数组

### 32.HaspMap安全吗？还了解过什么Map？对比一下？Map 的父类是 Collection吗?Map的key和value可以为null吗？

**不安全**

HashMap底层是一个Entry数组，当发生hash冲突的时候，hashmap是采用链表的方式来解决的，在对应的数组位置存放链表的头结点。对链表而言，新加入的节点会从头结点加入  
  
  
假如A线程和B线程同时对同一个数组位置调用addEntry，两个线程会同时得到现在的头结点，然后A写入新的头结点之后，B也写入新的头结点，那B的写入操作就会覆盖A的写入操作造成A的写入操作丢失  
(删除也是一样的)

\*\*Map不是collection的子接口或者实现类。Map是一个接口\*\*

键和值可以为空或null，但是不能同时存在两个相同的键，意思就是不能把两个不同的键都设为空或null。   
原因：键是唯一的。\*\*

### 33.HashMap和HashTable的区别？

**HashMap** 非线程安全，但是效率比较高

**HashTable** 是线程安全的，它的每个方法，都是通过**sychronizd**修饰的，问题：效率低下

* HashMap允许将null作为一个entry的key或者value，而Hashtable不允许。
* Hashtable的方法是Synchronize的，而HashMap不是，在多个线程访问Hashtable时，不需要自己为它的方法实现同步，而HashMap 就必须为之提供外同步。

HashMap可以通过下面的语句进行同步： Map m = Collections.synchronizeMap(hashMap);

### 33.数组和链表分别适合用于什么场景？为什么？

### 34.Session是什么

### 35.Cookie和Session的区别？

### 36.什么是jsp？什么是Servlet？jsp和Servlet的区别？

JSP本质上就是一个servlet，因为servlet输出html太麻烦了，所以就有了JSP，JSP就是专门用来书写html的，当然其中也能写java代码。

Servlet本质就是一个继承了HttpServlet的一个类文件,功能是在容器(服务器)根据客户端不同请求给予相应的响应服务.

jsp和servlet的区别和联系：  
  
1.jsp经编译后就变成了Servlet.  
(JSP的本质就是Servlet，JVM只能识别java的类，不能识别JSP的代码,Web容器将JSP的代码编译成JVM能够识别的java类)  
2.jsp更擅长表现于页面显示,servlet更擅长于逻辑控制.  
3.Servlet中没有内置对象，Jsp中的内置对象都是必须通过HttpServletRequest对象，HttpServletResponse对象以及HttpServlet对象得到.  
Jsp是Servlet的一种简化，使用Jsp只需要完成程序员需要输出到客户端的内容，Jsp中的Java脚本如何镶嵌到一个类中，由Jsp容器完成。  
而Servlet则是个完整的Java类，这个类的Service方法用于生成对客户端的响应。  
  
  
联系：   
JSP是Servlet技术的扩展，本质上就是Servlet的简易方式。JSP编译后是“类servlet”。  
Servlet和JSP最主要的不同点在于：  
Servlet的应用逻辑是在Java文件中，并且完全从表示层中的HTML里分离开来。  
而JSP的情况是Java和HTML可以组合成一个扩展名为.jsp的文件。  
JSP侧重于视图，Servlet主要用于控制逻辑  
Servlet更多的是类似于一个Controller，用来做控制。

### 37.jsp的作用域？作用？

page作用域：代表变量只能在当前页面上生效  
  
request：代表变量能在一次请求中生效，一次请求可能包含一个页面，也可能包含多个页面，比如页面A请求转发到页面B  
  
session：代表变量能在一次会话中生效，基本上就是能在web项目下都有效，session的使用也跟cookie有很大的关系。一般来说，只要浏览器不关闭，cookie就会一直生效，cookie生效，session的使用就不会受到影响。  
  
application：代表变量能一个应用下(多个会话)，在服务器下的多个项目之间都能够使用。比如baidu、wenku等共享帐号。  
  
其中操作page域中属性需要借助pageContext对象。

### 38.jsp的九大内置对象？各代表什么？

1 out：页面输出  
  
2 page：当前页面  
  
3 pageContext：当前上下文（编译之后的当前页面）  
  
4 request：请求  
  
5 response：响应  
  
6 session：会话  
  
7 application(ServletContext)：tomcat启动的时候自动创建，tomcat停止的时候，自动销毁  
  
8 config：配置文件（获取在web.xml中配置的Servlet的init-param配置信息）  
  
9 exception:异常

### 39.static的理解？

##### 修饰成员变量----将其变为类的成员，从而实现所有对象对于该成员的共享

##### 修饰成员方法----将其变为类方法，可以直接使用**“类名.方法名”**的方式调用，常用于工具类

##### 静态块-------------将多个类成员放在一起初始化，使得程序更加规整，其中理解对象的初始化过程非常关键

##### 静态导包----------将类的方法直接导入到当前类中，从而直接使用**“方法名”**即可调用类方法，更加方便

### 40.final、finally与finalize的区别

1. final

* 在java中，final可以用来修饰类，方法和变量（成员变量或局部变量）。
  + 当用**final修饰类**的时，表明该类不能被其他类所继承。当我们需要让一个类永远不被继承，此时就可以用final修饰，但要注意：**final类中所有的成员方法都会隐式的定义为final方法**
  + **父类中final方法的访问权限为private**，将导致**子类中不能直接继承该方法**，因此，此时可以在子类中定义相同方法名的函数，此时不会与重写final的矛盾，而是在子类中重新地定义了新方法。
  + **final成员变量表示常量**，只能被赋值一次，赋值后其值不再改变

1. finally
   * finally作为异常处理的一部分，它只能用在try/catch语句中，并且附带一个语句块，表示这段语句最终一定会被执行（不管有没有抛出异常），经常被用在需要释放资源的情况下。（×）（这句话其实存在一定的问题）
2. finalize

### 41.Redis？特点？数据类型？

Redis 是一个高性能的key-value数据库，缓存数据库，内存数据库，读取和修改的效率极高，应用在高并发的系统中非常合适  
 Redis支持数据的持久化，可以将内存中的数据保存在磁盘中，重启的时候可以再次加载进行使用。  
 Redis不仅仅支持简单的key-value类型的数据，同时还提供list，set，zset，hash等数据结构的存储。  
 Redis支持数据的备份，即master-slave模式的数据备份。

* Redis支持数据的持久化，可以将内存中的数据保存在磁盘中，重启的时候可以再次加载进行使用。
* Redis不仅仅支持简单的key-value类型的数据，同时还提供list，set，zset，hash等数据结构的存储。
* Redis支持数据的备份，即master-slave模式的数据备份

Redis支持五种数据类型：string（字符串），hash（哈希），list（列表），set（集合）及zset(sorted set：有序集合)。

### 42.Redis和Session的区别

答：redis是需要启动服务器的，数据会持久化，比如登录的验证码等很多信息都适合放入redis

session只是一次会话，当浏览器关闭之后，消息就丢失了，session比较适合存储当前登录用户信息

### 43.Redis最适合那些场景？

1. 会话缓存（Session Cache）:用Redis缓存会话比其他存储（如memcached）的优势在于：redis提供持久化。当维护一个不是严格要求一致性的缓存时，如果用户的购物车信息全部丢失，大部分人都会不高兴。
2. 全页缓存（FPC）:除基本的会话token之外，Redis还提供很简便的FPC平台。
3. 队列：Redis在内存存储引擎领域的一大优点是提供list和set操作，这使得Redis能作为一个很好的消息队列平台来使用。Redis作为队列使用的操作，就类似于本地程序语言对list的push/pop操作。
4. 排行榜/计数器：Redis在内存中对数据进行递增递减的操作实现的非常好。集合（Set）和有序集合（Sorted Set）也使得我们在执行这些操作的时候变得非常简单，Redis只是正好提供了这两种数据结构。
5. 订阅/发布

### 44.Redis的优缺点？Redis的持久化？

优点：

1 读写性能优异

2 支持数据持久化，支持AOF和RDB两种持久化方式

3 支持主从复制，主机会自动将数据同步到从机，可以进行读写分离。

4 [数据结构](http://lib.csdn.net/base/datastructure)丰富：除了支持string类型的value外还支持string、hash、set、sortedset、list等数据结构。

缺点：

1 [Redis](http://lib.csdn.net/base/redis)不具备自动容错和恢复功能，主机从机的宕机都会导致前端部分读写请求失败，需要等待机器重启或者手动切换前端的IP才能恢复。

2 主机宕机，宕机前有部分数据未能及时同步到从机，切换IP后还会引入数据不一致的问题，降低了系统的可用性。

3 [redis](http://lib.csdn.net/base/redis)的主从复制采用全量复制，复制过程中主机会fork出一个子进程对内存做一份快照，并将子进程的内存快照保存为文件发送给从机，这一过程需要确保主机有足够多的空余内存。若快照文件较大，对集群的服务能力会产生较大的影响，而且复制过程是在从机新加入集群或者从机和主机网络断开重连时都会进行，也就是网络波动都会造成主机和从机间的一次全量的数据复制，这对实际的系统运营造成了不小的麻烦。

4 Redis较难支持在线扩容，在集群容量达到上限时在线扩容会变得很复杂。为避免这一问题，运维人员在系统上线时必须确保有足够的空间，这对资源造成了很大的浪费。

### 45.什么是MQ?

答：MQ message Queue 消息队列，他有很多实现，kafka、activeMQ、rabbitMQ，我们学习的是avtiveMQ

它是JMS的实现  
  
ActiveMQ中提供了两种消息模型，一个是Queue队列，一个是Topic广播模型  
  
这两个模型的特点：  
  
1 Queue队列一个生产者对应多个消费者，但是消息只能被一次，Topic广播模型，一个消息可以被消费多次  
  
2 Queue队列消息一定要等到成功消费，才会退出队列；而广播模型无所谓是否成功消费  
  
3 Queue队列在时间上对消费者没有要求，而广播模型一定要先有消费者，而后再有生产者生产消息，才可被消费，在时间上有要求

#### 45.1MQ中的广播队列为什么先有消费者后才有生产者?

没有消费者的话，生产者的消息接收不到

#### 45.2如果我使用queue队列生产一个消息,有一百个消费者去进行消费,那么谁消费了?

不确定，随机的，不能去定是谁消费了

### 46.ActiveMQ如果发送消息失败怎么办

### 47.elasticsearch的好处? 为什么不用mysql的查询而使用elasticsearch去查询?

### 48.谈谈你对Restful的理解以及在项目中的使用？

## Shiro？

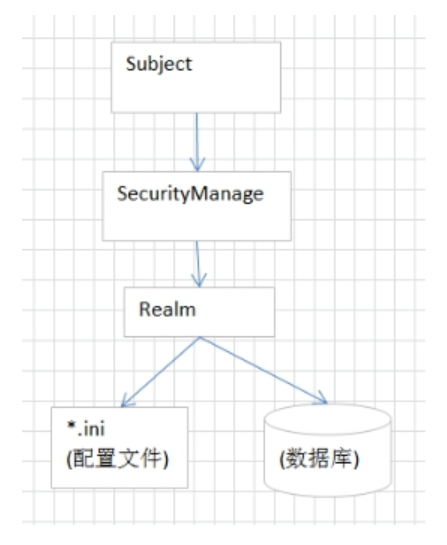
Apache Shiro是一个强大且易用的Java安全框架,执行身份验证、授权、密码和会话管理。使用Shiro的易于理解的API,您可以快速、轻松地获得任何应用程序,从最小的移动应用程序到最大的网络和企业应用程序。

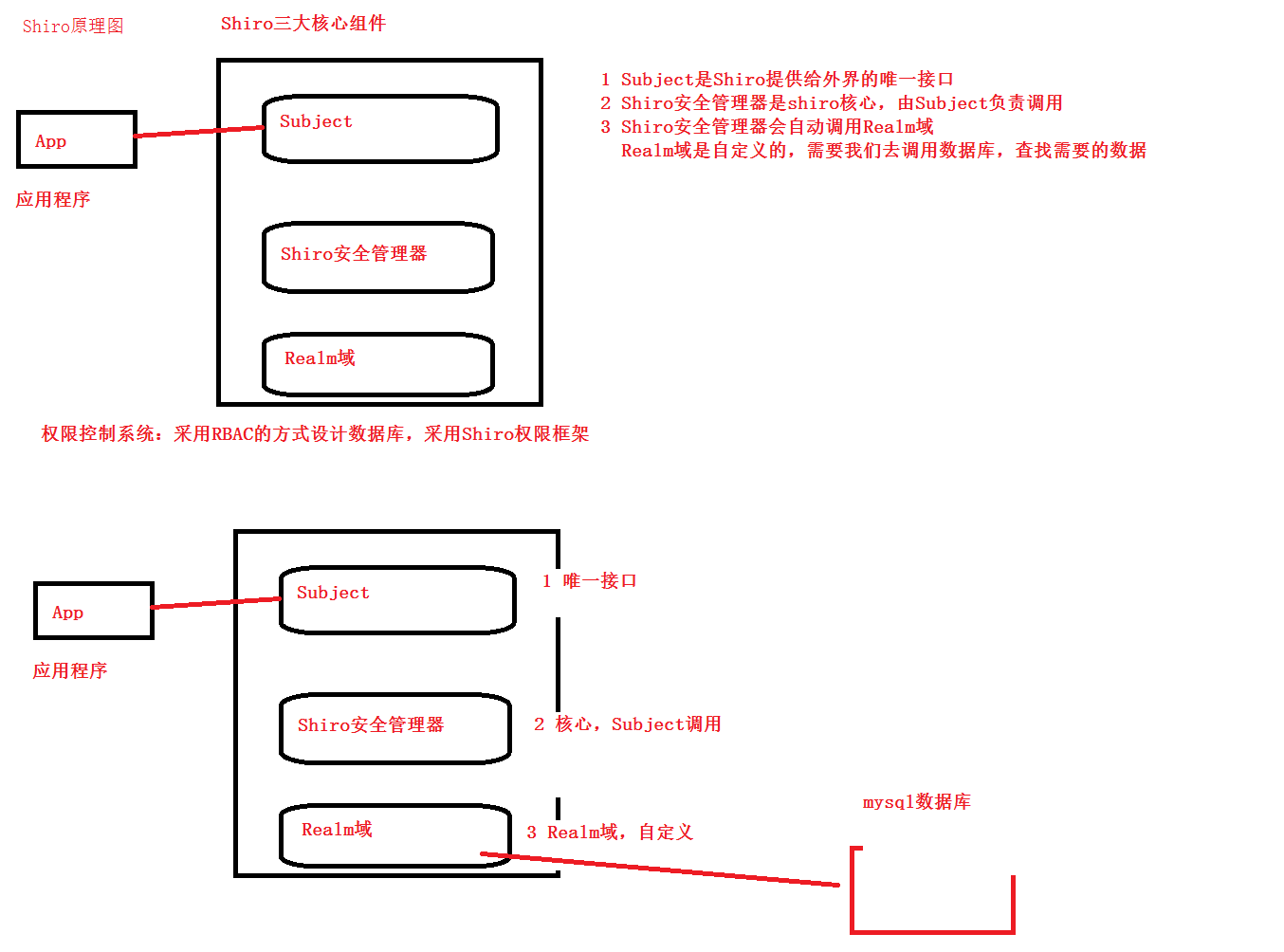
官网：http://shiro.apache.org/



### 1.1**三个核心组件**

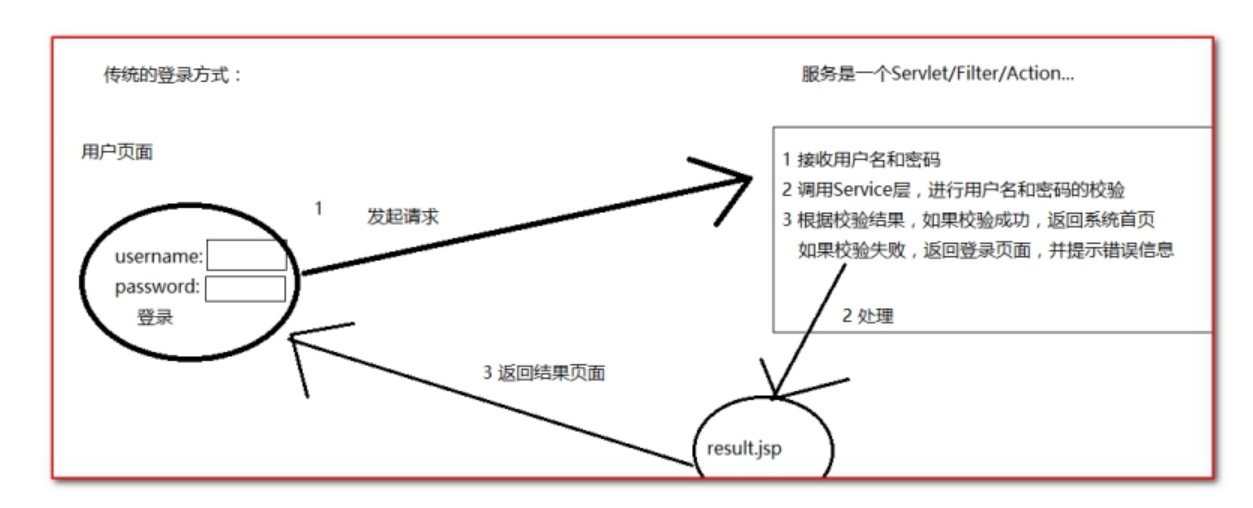
三个核心组件：Subject, SecurityManager 和 Realms.

* Subject，主体，即“当前操作用户”。Subject代表了当前用户的安全操作。（需要被认证的对象）
* SecurityManager：它是Shiro框架的核心，管理所有用户的安全操作，并提供安全管理的各种服务。
* Realm： Realm充当了Shiro与应用安全数据间的“桥梁”或者“连接器”。也就是说，当对用户执行认证（登录）和授权（访问控制）验证时，Shiro会从应用配置的Realm中查找用户及其权限信息。
* 

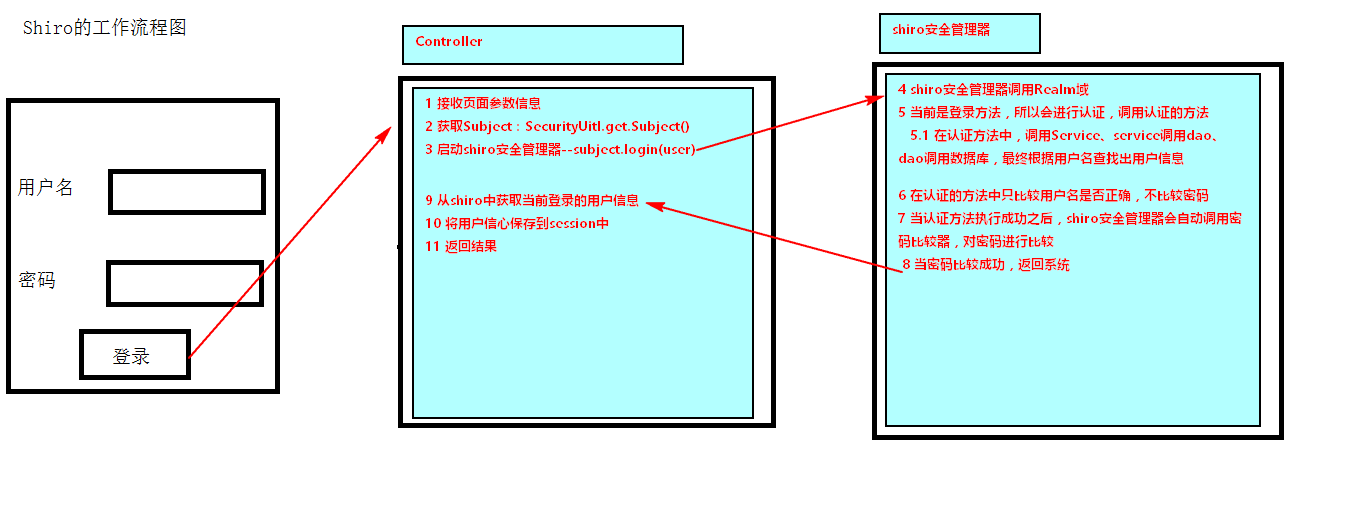


### 1.2传统的登录和shiro登录的比较

#### 传统：



#### shiro：



传统的登录与shiro的区别在于：

传统的是Controller直接调用Service处理，现在用了shiro后Controller调用Shiro安全框架去处理，也就是说将认证授权抽取出来，有一个框架专门为你做认证做授权

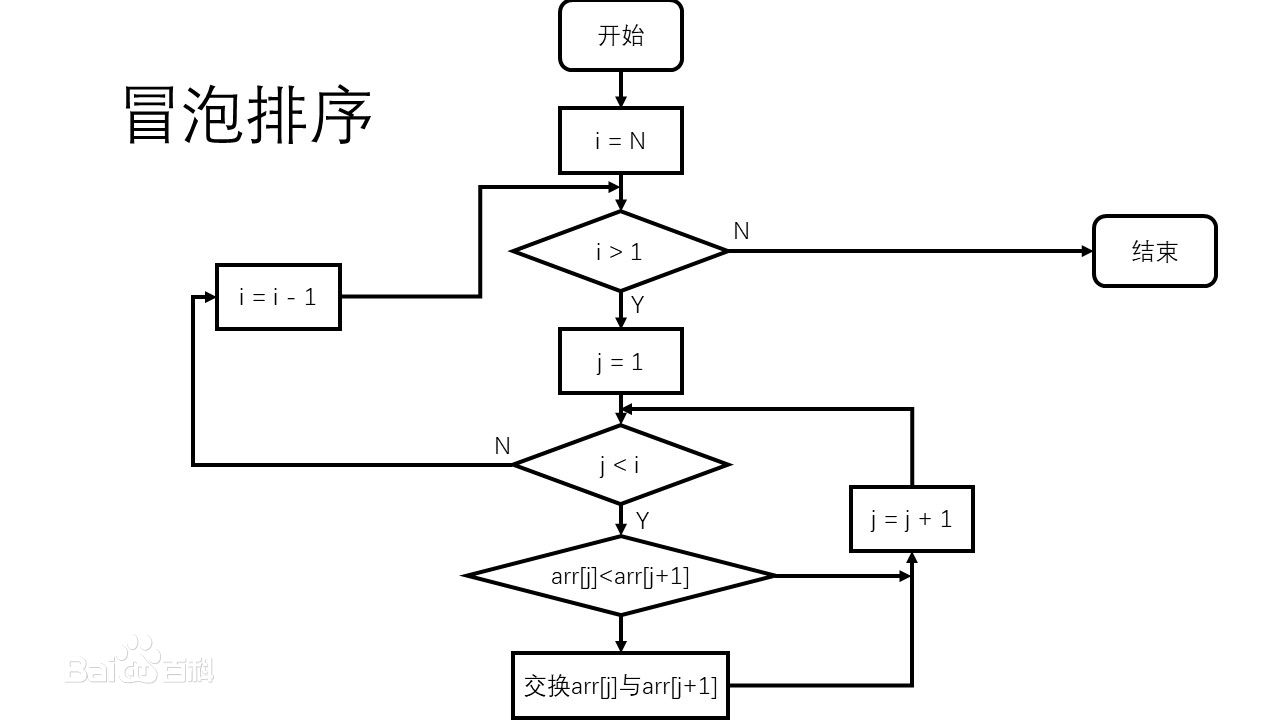
### 49.设计模式（至少了解三种并解释）

### 50.盒子模型

### 51.手写冒泡，并总结数学公式

它重复地走访过要排序的元素列，依次比较两个相邻的元素，如果他们的顺序（如从大到小、首字母从A到Z）错误就把他们交换过来，直到元素排序完成

* /\*  
    
  - 冒泡排序  
   \*/  
   public class BubbleSort {  
   　　public static void main(String[] args) {  
   　　　　int[] arr={6,3,8,2,9,1};  
   　　　　System.out.println("排序前数组为：");  
   　　　　for(int num:arr){  
   　　　　　　System.out.print(num+" ");  
   　　　　}  
   　　　　for(int i=0;i<arr.length-1;i++){//外层循环控制排序趟数  
   　　　　　　for(int j=0;j<arr.length-1-i;j++){//内层循环控制每一趟排序多少次  
   　　　　　　　　if(arr[j]>arr[j+1]){  
   　　　　　　　　　　int temp=arr[j];  
   　　　　　　　　　　arr[j]=arr[j+1];  
   　　　　　　　　　　arr[j+1]=temp;  
   　　　　　　　　}  
   　　　　　　}  
   　　　　}   
   　　　　System.out.println();  
   　　　　System.out.println("排序后的数组为：");  
   　　　　for(int num:arr){  
   　　　　　　System.out.print(num+" ");  
   　　　　}   
   　　}  
   }



### 52.讲一下你理解的递归

### 53.单例是什么？特点？Calendar是单例吗？

单例模式是Java中常用的设计模式之一。单例模式确保某个类只有一个实例，而且自行实例化并向整个系统提供这个实  
  
例。选择单例模式就是为了避免不一致状态，避免出现一个端口同时被两个请求同时调用。常用的单利有：懒汉式、  
  
饿汉式、线程安全式、双重检测式、登记式。

1、单例类只能有一个实例。  
  
2、单例类必须自己创建自己的唯一实例。  
  
3、单例类必须自行向整个系统提供这个实例。  
  
单例模式保证了全局对象的唯一性，比如系统启动读取配置文件就需要单例保证配置的一致性。

答：不是  
Calendar c1 = Calendar.getInstance();  
 Calendar c2 = Calendar.getInstance();  
 c1.add(Calendar.DAY\_OF\_YEAR,+1);  
 System.out.println(c1.getTime().toLocaleString());  
 System.out.println(c2.getTime().toLocaleString());

### 手写单例模式？

饿汉式  
class Singleton {  
 private static Singleton instance=new Singleton();  
 private Singleton(){}  
 static Singleton getInstance() {  
 return instance;  
 }  
}

懒汉式  
class Singleton {  
 private static Singleton instance=null;  
 private Singleton(){}  
 static Singleton getInstance() {  
 if(instance==null)  
 instance=new Singleton();  
 return instance;  
 }  
}

### 54.说一下事务管理,有哪些?你是如何进行事务管理的?

### 55.为什么学it？

# 单例模式

## 1.模式简介

## 2.模式特点

## 3.模式优缺点

优点  
  
1、实例控制  
  
单例模式会阻止其他对象实例化其自己的单例对象的副本，从而确保所有对象都访问唯一实例  
  
2、灵活性  
  
因为类控制了实例化过程，所以类可以灵活更改实例化过程。  
  
缺点  
  
1、开销  
  
虽然数量很少，但如果每次对象请求引用时都要检查是否存在类的实例，将仍然需要一些开销。  
  
可以通过使用静态初始化解决此问题。  
  
2、可能的开发混淆  
  
使用单例对象(尤其在类库中定义的对象) 时，开发人员必须记住自己不能使用new关键字实例化对象。  
  
因为可能无法访问库源代码，因此应用程序开发人员可能会意外发现自己无法直接实例化此类。  
  
3、对象生存期  
  
不能解决删除单个对象的问题。 在提供内存管理的语言中，只有单例类能够导致实例被取消分配，因为  
  
它包含对该实例的私有引用。在某些语言中，其他类可以删除对象实例，但这样会导致单例类中出现悬浮引用

## 4.适用场合

1、需要频繁的进行创建和销毁的对象  
  
2、创建对象时耗时过多或耗费资源过多，但又经常用到的对象  
  
3、工具类对象  
  
4、频繁访问数据库或文件的对象

## 5.实例

**1、饿汉式**

public class Singleton {  
 //在自己内部定义自己的一个实例，只供内部调用  
 private final static Singleton instance = new Singleton();  
   
 private Singleton(){  
 }  
 //这里提供了一个供外部访问本class的静态方法，可以直接访问  
 public static Singleton getInstance(){  
 return instance;  
 }  
}  
  
优点：这种写法比较简单，就是在类装载的时候就完成实例化。避免了线程同步问题。  
  
缺点：在类装载的时候就完成了实例化，如果从始至终从未使用过这个实例，则会造成内存的浪费。

**2、懒汉式**

public class Singleton {  
   
 private Singleton() {  
 }   
 //单例对象  
 private static Singleton instance = null;   
 //静态工厂方法  
 public static Singleton getInstance() {  
 if (instance == null) {  
 instance = new Singleton();  
 }  
 return instance;  
 }  
}  
  
解释这样写的理由  
  
1、要想让一个类只能构建一个对象，自然不能让他随便去做new操作，因此构造方法是私有的  
  
2、instance是singleton类的静态成员，也是我们的单例对象。它的初始值可以写成null，也可以  
  
写成new Singletion()。如果单例初始值是null，还未构建，则构建单例对象并返回，这个  
  
写法就属于单例模式当中的懒汉模式。如果单例对象一开始就被new Singleton()主动创建，则  
  
不再需要非空判断操作，这种写法就属于上面介绍的饿汉模式。  
  
3、getInstance是获取单例对象的方法。

**但是这样的懒汉式不是线程安全的单例！**

假设Singleton类刚刚被初始化，instance对象还是空，这个时候两个线程同时访问getInstance方法

因为instance是空，所以两个线程同时通过了条件判断，开始执行new操作：

这样一来，显然instance被构建了两次。

**3.线程安全式**

解决上面实现方式的线程不安全问题做个线程同步就可以了，但是这就会造成程序性能下降。

public class Singleton {  
 //私有构造方法  
 private Singleton() {  
 }  
 //单例对象  
 private static Singleton instance = null;   
 //静态工厂方法  
 /\*public static synchronized Singleton getInstance() {  
 if (singleton == null) {  
 singleton = new Singleton();  
 }  
 return singleton;  
 }\*/  
 public static Singleton getInstance() {  
 if (instance == null) {  
 synchronized (Singleton.class) { //同步锁  
 instance = new Singleton();  
 }  
 }  
 return instance;  
 }  
}  
  
被注释掉的是同步方法，下面的是同步代码块，之所以不用同步方法是因为同步方法效率太低了，每个线程在想  
  
获得类的实例的时候，执行getInstance（）方法都要进行同步。而其实这个方法只执行一次实例化代码就够  
  
了，  
  
为了防止new Singleton被执行多次，因此在new操作之前加上Synchronized 同步锁，锁住整个类  
  
后面的想获得该类实例，直接return就行了。所以摒弃同步方法，改为同步产生实例化的代码块。但是这种同  
  
步并不能起到线程同步的作用。跟第二种实现方式遇到的情形一样，假如一个线程进入了判断语句块，还  
  
未来得及往下执行，另一个线程也通过了这个判断语句，这时便会产生多个实例。

**4.双重检测（双重锁）**

public class Singleton {  
 //私有构造方法  
 private Singleton() {}  
 //单例对象  
 private static Singleton instance = null;   
 //静态工厂方法  
 public static Singleton getInstance() {  
 if (instance == null) { //双重检测机制  
 synchronized (Singleton.class){ //同步锁  
 if (instance == null) { //双重检测机制  
 instance = new Singleton();  
 }  
 }  
 }  
 return instance;  
 }  
}  
  
再次解释几个地方：  
  
1、为了防止new Singleton被执行多次，因此在new操作之前加上Synchronized 同步锁，锁住整个类  
  
（注意，这里不能使用对象锁）  
  
2、进入Synchronized 临界区以后，还要再做一次判空。因为当两个线程同时访问的时候，线程A构建完  
  
对象，线程B也已经通过了最初的判空验证，不做第二次判空的话，线程B还是会再次构建instance对象。

使用双重检查进一步做了优化，可以避免整个方法被锁，只对需要锁的代码部分加锁，可以提高执行效率

像这样两次判空的机制就叫做双重检测机制。但是！

**这段代码仍然不是绝对的线程安全!（看不懂可不看）**

假设这样的场景，当两个线程一先一后访问getInstance方法的时候，当A线程正在构建对象，B线程刚进入方法。这种情况表面看似没什么问题，要么Instance还没被线程A构建，线程B执行 if（instance == null）的时候得到true；要么Instance已经被线程A构建完成，线程B执行 if（instance == null）的时候得到false。

其实这里还涉及到了JVM编译器的指令重排。

(我就知道你肯定对指令重排不太了解)

比如Java中简单的一句 instance = new Singleton,会被编译器编译成如下JVM指令：

memory =allocate(); //1：分配对象的内存空间   
  
ctorInstance(memory); //2：初始化对象   
  
instance =memory; //3：设置instance指向刚分配的内存地址

但是这些指令顺序并非一成不变，有可能会经过JVM和CPU的优化，指令重排成下面的顺序：

memory =allocate(); //1：分配对象的内存空间   
  
instance =memory; //3：设置instance指向刚分配的内存地址   
  
ctorInstance(memory); //2：初始化对象

当线程A执行完1,3的时候，instance对象还未完成初始化，但已经不再指向null。此时如果线程B抢占到CPU资源，执行 if（instance == null）的结果会是false，从而返回一个没有初始化完成的instance对象。

为了避免这种情况 我们需要在instance对象前面增加一个修饰符volatile。

public class Singleton {  
 //私有构造方法  
 private Singleton() {}  
 //单例对象  
 private volatile static Singleton instance = null;   
 //静态工厂方法  
 public static Singleton getInstance() {  
 if (instance == null) { //双重检测机制  
 synchronized (Singleton.class){ //同步锁  
 if (instance == null) { //双重检测机制  
 instance = new Singleton();  
 }  
 }  
 }  
 return instance;  
 }  
}  
  
用最简单的方式理解，volatile修饰符阻止了变量访问前后的指令重排，保证了指令执行顺序！

经过volatile的修饰，当线程A执行instance = new Singleton的时候始终保证是下面的顺序：

memory =allocate(); //1：分配对象的内存空间   
  
ctorInstance(memory); //2：初始化对象   
  
instance =memory; //3：设置instance指向刚分配的内存地址

这样的话线程B看来，instance对象的引用要么指向null，要么指向一个初始化完毕的Instance，而不会出现某个中间态，保证了安全。

**5.登记式/静态内部类**

这种方式跟饿汉式方式采用的机制类似，但又有不同。两者都是采用了类装载的机制来保证初始化实例时只有一个线程。不同的地方在饿汉式方式是只要Singleton类被装载就会实例化，没有Lazy-Loading的作用，而静态内部类方式在Singleton类被装载时并不会立即实例化，而是在需要实例化时，调用getInstance方法，才会装载SingletonInstance类，从而完成Singleton的实例化。

public class Singleton {  
  
 private Singleton() {}  
  
 private static class SingletonInstance {  
 private static final Singleton instance = new Singleton();  
 }  
  
 public static Singleton getInstance() {  
 return SingletonInstance.instance;  
 }  
}  
  
注意几个点：  
  
1、从外部无法访问静态内部类SingletonInstance，只有当调用Singleton.getInstance方法的时候  
  
才能得到单例对象instance。  
  
2、instance对象初始化的时机并不是在单例类Singleton被加载的时候，而是在调用getInstance方法，  
  
使得静态内部类SingletonInstance被加载的时候。因此这种实现方式是利用classloader（类加载器）  
  
的加载机制来实现懒加载，并保证构建单例的线程安全。