|  |  |
| --- | --- |
| 项目名 | 说明 |
| lvguyuan-common | 公共包 |
| lvguyuan-dao | 持久层 |
| lvguyuan-pojo | 实体类 |
| lvguyuan-sellergoods-interface | 卖家商品借口 |
| lvguyuan-sellergoods-service | 卖家商品实现类 |
| lvguyuan-shop-web | 商城前台 |
| lvguyuan-manager-web | 管理人员后台 |
|  |  |

AngularJS

var app=angular.module('myApp',[]); //定义了一个叫myApp的模块

在body中使用 ng-app=”myApp”

定义控制器

app.controller(‘myController’function($scope){

})

在body中使用 ng-controller=”myController”

$scope的使用贯穿整个angularjs App应用，它域数据模型相管理，同时也是表达式执行上下文，有了￥scope就在

循环执行 ng-repeat=‘x in list’

ng-model=‘’绑定

|  |  |  |
| --- | --- | --- |
| 名称 | 指令 | 说明 |
| 模块名 | ng-app=”” | 指定当前页面的模块名 |
| 初始化 | ng-init=”x=1;t=2”;  对象  ng-init=”person={name=’dd’,age=12}”  数组  ng-init=”points[1,2]” | {{x \* y}}  {{person.name}}  <span ng-bind=”points[2]”>{{poinst[2]}}</span> |
| 指令把元素值（比如输入域的值）绑定到应用程序 | 初始化<body ng-app=’’ ng-init=’firstName=’John’>  <input type=’text’ ng-model=’firstName’>  或{{firstName}} |  |

# javaSE

## 集合

### 常用集合:

collection

list:ArrayList,LinkedList,Vector

set:AbstratSer,HashSet,TreeSet

map:AbstractMap,HashMap,TreeMap

### 集合和数组的区别

长度区别：

数组:固定

集合:可变

内容区别:

数组:可以是基本类型,也可以是引用类型

集合:只能是引用类型

元素内容

数组:只能存储同一种类型

集合:可存储不同类型(其实集合一般存储的也是同一类型),如何存储的是基本类型会自定装箱

### 集合基本方法

boolean add(E e) 在集合末尾添加元素

boolean remove(Object o) 若本类集合有值与o的值相等的元素,则删除该元素,并返回true

void clear() 清除本类集合中所有元素,调用完该方法后本类集合将为空

boolean contains(object o)判断集合中是否包含某元素

boolean isEmpty()判断集合是否为空

int size( ) 返回集合中的元素个数

boolean addAll(Collection c)将一类集合c中的所有元素添加到另一个类集

Object[] toArray() 返回一个包含了本类集合中所有元素的数组,数组类型为:object[]

Iterator iterator() 迭代器,集合专用遍历方式

### List和Set区别

有序性

List:保证an插入顺序排序

Set:存储和取出顺序不一致

唯一性

List:可以重复

Set:元素唯一

获取元素

List:可通过索引直接操作元素

Set：不能根据索引获取元素

HashSet集合不可重复的原因

HashSet安装Hash算法存储集合中的元素,具有很好的存储和查找性能,当向HashSet中添加一些元素是,HashSet会根据该对象的HashCode()方法来查找该对象的HashCode值,然后根据这些HashCode的值来决定该元素的位置

1. 存储顺序和添加的顺序不同
2. HashSet不是同步的,如果多个线程同时访问一个HashSet,假设两个或更多的线程修改了集合中的值,则必须通过代码使线程同步
3. HashSSet运行集合中的元素为null

在HashSet集合中,判断两个元素相同的标准是:两个对象通过equals()方法相等,且HashCode()方法的返回值也相等

LinkedHashSet类

LinkedHashSet是HashSet的子类,它也是根据元素的HashCode值进来绝对元素的位置,但他能够同时使用链表来维护元素的添加次序,使得元素能以插入顺序保存。

TreeSet

TreeSert是SortedSet接口的实现类,TreeSet可以保证了集合元素处于排序状态(所谓排序状态,就是按照一定的规则排序,比如升序排列,降序排列)

什么是Liset集合

List集合是一种元素有序的,可重复的集合,集合中每个元素都有自己的元素索引,List集合允许可重复的元素,可以通过索引来访问指定位置的元素。

java8改进的List接口和ListIterator接口

List作为Collection集合的子接口,可以使用Collection接口的全部方法,另外由于List是有序集合,有添加了根据索引来操纵List集合的一些方法;

ArrayList和Vector实现类

什么是Queue集合

Queue用于模拟队列这种数据结构,队列通常是指”先进先出(FIFO)”的容器,队列的头部保存在队列中存放时间最长的元素,尾部保存存放时间最短的员,新元素插入到队列的尾部,取出元素会返回队列头部的元素,通常,队列不允许随机访问队列中的元素。

Queue接口中定义了如下方法:

void add(Object e) 将制定元素插入到队列的尾部

Object element() 获取队列头部的元素,但是不删除该元素

boolean offer(Object e) 将指定的元素插入此队列的尾部,当使用容量有限的队列时,此方法通常比add(Object e) 有效

object peek() 返回队列头部的元素,但是不删除该元素,如果队列为空,则返回null

object poll() 返回队列头部的元素,并删除元素,如果队列为空,则返回null

object remove() 获取队列头部的元素,并删除该元素

Queue接口有一个PriorityQueue实现类,除此之外,Queue还有一个Deque接口,Deque代表”双端队列”,双端队列可以从两端删除或添加元素,因此Deque可以当做栈来使用,java为Deque提供了ArrayDeque实现类和LinkedList实现类

## 线程

#### 多线程创建

线程的实现方式: Runnable接口,Thread 方法,Callable接口

1. 继承Thread类

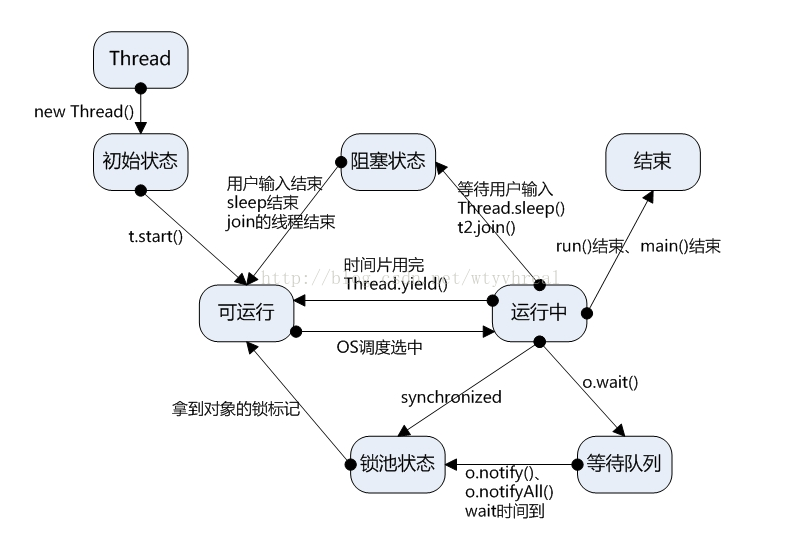
|  |
| --- |
| public class demo01 extends Thread{  *@Override* public void run() {  System.*out*.println("this thread name is:"+Thread.*currentThread*().getName());  }    public static void main(String[] args) {  demo01 d = new demo01();  d.setName("myTest");  d.start();  } } |

1. 实现Runnabe接口

|  |
| --- |
| public class demo02 implements Runnable{   *@Override* public void run() {  System.*out*.println("this thread name is:"+Thread.*currentThread*().getName());  }   public static void main(String[] args) {  System.*out*.println(Thread.*currentThread*().getName());  demo02 d = new demo02();  Thread t = new Thread(d);  t.start();  } } |

#### 线程的生命周期&状态

|  |  |
| --- | --- |
| 状态名称 | 说明 |
| NEW | 初始状态,线程被构建,但是还没有调用start()方法 |
| RUNNABLE | 运行状态,java线程将操作系统中的就绪和运行两种状态笼统地成为”运行中” |
| BLOCKED | 阻塞状态,表示线程阻塞于锁 |
| WAITING | 等待状态,表示线程进入等待状态,进入状态表示当前线程需要等待其他线程做出一些特定动作(通知或中断) |
| TIME\_WAITING | 超时等待状态,该状态不同于WAITING,它是可以在指定的时间自行返回的 |
| TERMINATED | 终止状态,表示当前线程已经执行完毕 |



#### Callable&Future

有返回值的多线程实现

|  |
| --- |
| public static void main(String[] args) throws ExecutionException, InterruptedException {  FutureTask task = new FutureTask(new Callable01());  new Thread(task).start();  System.*out*.println(task.get()); }  static class Callable01<String> implements Callable<String>{   *@Override* public String call() throws Exception {  String name = (String) "你好";  return name;  } } |

java中有几种线程池?

普通线程池

定时线程池

## IO流

### 说明：

### 字节流：

### 字符流：

### 序列化：

概念:

把对象转换为字节序列的过程称为对象的序列化。

把字节序列恢复为对象的过程称为对象的反序列化。

对象的序列化主要有两种途径：

1. 把对象的字节序列永久性地保存在硬盘上,通常存放在一个文件中；
2. 在网络上传送对象的字节序列。

在很多应用中,需要对某些对象进行序列化,让它们离开内存空间,入住物理硬盘,以便长期保存.比如最常见的是Web服务器中的Session对象,当有10万用户并发访问，就有可能出现10万个Session对象,内存可能吃不消,于是Web容器就会吧一些Session先序列化到硬盘中,等要用了,再把保存在硬盘中的对象还原到内存中。

当两个进程在进行远程通信是,彼此可以发送各种类型的数据,无论是何种类型的数据,都会以二进制序列的形式在网络上传送,发送方需要把这个Java对象转换为字节序列,才能在网络传送;接受方则需要把字节序列在恢复为Java对象。

JKD类库中的序列化API

ObjectOutputStream代表对象输出流

writeObject(Object obj)方法可对参数指定的obj对象进行序列化,把得到的字节序列写到一个目标输出流中。

ObjectInputStream代表对象输入流

readObject()方法从一个源输入流中读取字节序列,再把它们反序列化为一个对象,并将其返回。

只有实现了Serializable和Externalizable接口的类对象才能被序列化,Externalizable接口继承自Serializable接口,实现Externalizable接口的类完全自由身来控制序列化的行为,而仅实现Serializable接口的类可以采用默认的序列化方式。

对象序列化包括如下步骤:

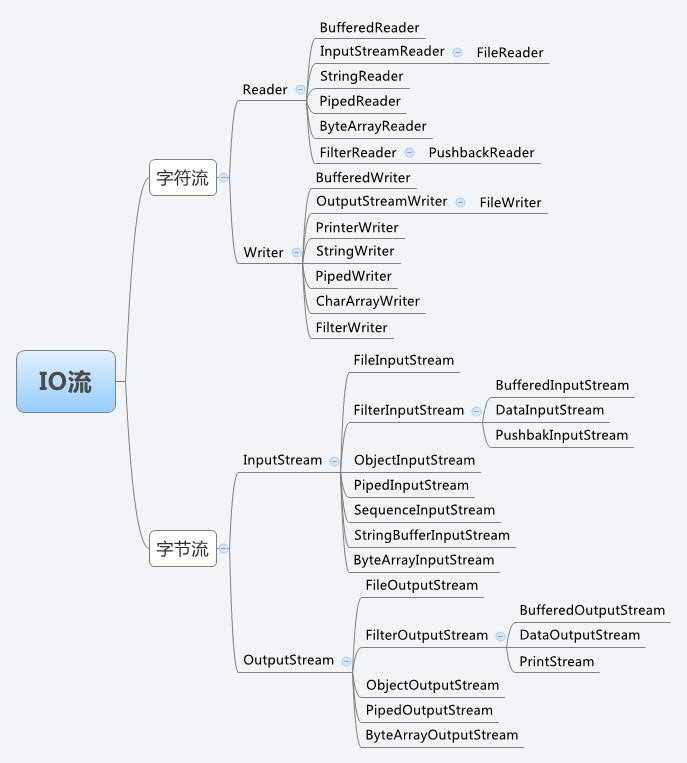
1. 创建一个对象输出流,他可以包装一个其他类型的目标输出流,如文件输出流
2. 通过对象输出流writeObject()方法写对象

对象反序列化步骤如下:

1. 创建一个对象输入流,它可以包装一个其他类型的源输入流,如文件输入流。
2. 通过对象输入流的readObject()方法读取对象

java流操作有关的类或接口

|  |  |
| --- | --- |
| 类 | 说明 |
| File | 文件类 |
| RandomAccessFile | 随机存储文件类 |
| InputStream | 字节输入流 |
| OutputStream | 字节输出流 |
| Reader | 字符输入流 |
| Writer | 字符输出流 |



流的概念和座椅

流是一组有顺序的,有起点和终点的字节集合,是对数据传输的总称或抽象,即数据在两设备间的传输成为流,流的本质是数据传输,根据数据传输特性将流抽象为各种类,方便更直观的进行数据操作。

IO流的分类

根据处理数据类型的不同分为:字符流和字节流

根据数据流向的不同分为:输入流和输出流

字符流和字节流

字符流的由来:因为数据编码的不同,而有了对字符进行高效操作的流对象,本质其实就是基于字节流读取时,去查了指定的码表,字节流和字符流的区别:

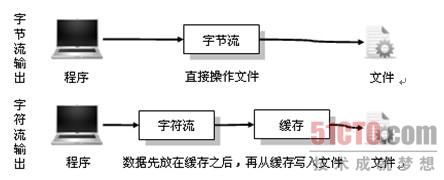
读写单位不同:字节流以字节(8bit)为单位,字符流以字符为单位,根据表映射字符,一次可能读多个字节.

处理对象不同:字节流处理所有类型的数据(如图片、avi等),而字符流只能处理字符类型的数据。

结论:只要是处理纯文本数据,就优先考虑字符流,除此之外都使用字节流。

字节流与字符流的使用非常相似,两者除了操作代码上不同以外,是否还有其他的不同呢？

实际上字节流在操作本身不会用到缓冲区(内存),是文件本身直接操作的,而字符流在操作时使用了缓冲区,通过缓冲区在操作文件



## 网络通讯

## 并发编程

并发：同时拥有两个或者多个线程，如果程序在单核处理器上运行，多个线程将交替地换入或者换出内存，这些线程是同时”存在”的，每个线程都处于执行过程中的某个状态，如果运行在多核处理器上，此时，程序中的每个线程都将分配到一个处理器核上，因此可以同时运行。

高并发:高并发（High Concurrency）是互联网分布式系统架构设计中必须考虑的因素之一，它通常是指，通过设计保证系统能够同时并行处理很多请求

并发：多个线程操作相同的资源，保证线程安全，合理使用资源

高并发：服务器能同时处理很多请求，提高程序性能

CPU多级缓存-缓存一致性（MESI）

用于保证多个CPU cache之间缓存共享数据的一致

java内存模型(java Memory Model,JMM)

java内存模型-同步八种操作

lock(锁定):

作用于主内存的变量,把一个变量标识为一条线程独占状态

unlock(解锁):

作用于主内存的变量,把一个处于锁定状态的变量释放出来,释放后的变量才可以被其他线程锁定

read(读取):

作用于主内存的变量,把一个变量值从主内存传输到线程的工作内存中,以便随后的load动作使用

load(载入):

作用于工作内存的变量,它把read操作从主内存中得到的变量值放入工作内存的变量副本中

use(使用):

作用于工作内存的变量,把工作内存中的一个变量值传递给执行引擎

assign(赋值):

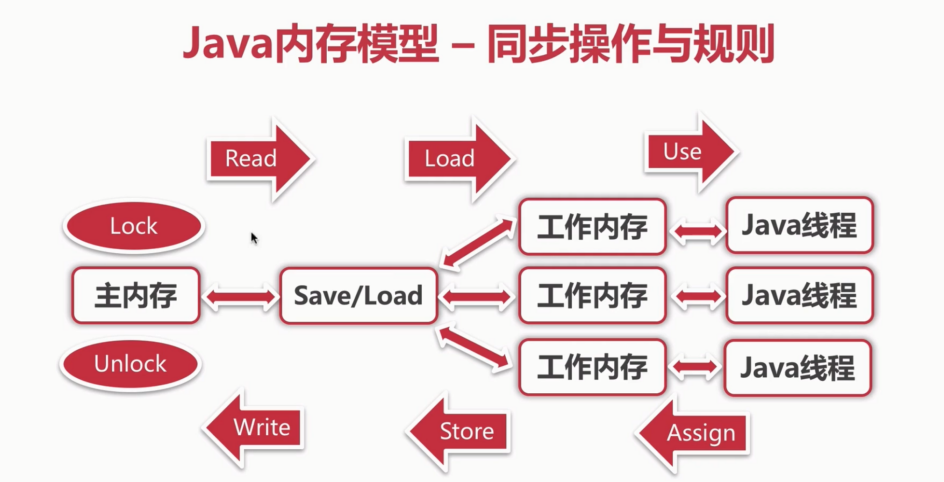
作用于工作内存的变量，它把一个从执行引擎接收到的值赋予给工作内存的变量

store(存储):

作用于工作内存的变量,把工作内存中的一个变量的值传送到主内存中,以便随后的write的操作

writer(写入):

作用于主内存的变量,它把store操作从工作内存中一个变量的值传送到内存的变量中



总结:

CPU多级缓存:缓存一致性，乱序执行优化

java内存模型：JMM规定、抽象结构、同步八种操作及规则

java并发的优势与风险

并发模拟

Postman:Http请求模拟工具

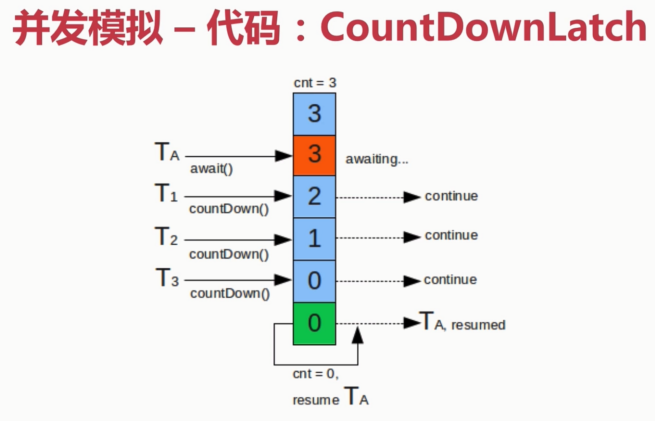
Apache Bench(AB):Apache附带的工具,测试网站性能

JMeter:Apache组织开发的压力测试工具

代码:Semaphore、CountDownLatch等

并发模拟-代码:

CountDownLatch:计数器向下闭锁



Semaphore:信号量

线程安全性:

当多个线程访问某个类时,不管运行时环境采用何种调动方式或者这些进程将如果交替执行，并且在主调代码中不需要任何额外的同步,这类都能表现出正确的行为,那么就称这个类时线程安全的。

原子性:

提供了互斥访问,同一时刻只能有一个线程来对它进行操作

可见性:

一个线程对主内存的修改可以及时的被其他线程观察到

有序性:

一个线程观察其他线程中的指令执行顺序,由于指令重排序的存在,该观察结果一般杂乱无序

原子性-Atomic包(额涛密克)

AtomicXXX:CAS、Unsafe.compareAndSwapInt

AtomicLong、LongAdder

AtomicReference、AtomicRefernceFieldUpdater

AtomicStampReference:CAS的ABA问题

原子性-锁

synchronized:依赖JVM

同步锁

修饰代码块:

大括号括起来的代码,作用于调用的对象

|  |
| --- |
| *//修饰一个代码块* public void test1(){  synchronized (this){  for(int i = 0; i < 10 ;i++){  System.*out*.println(i);  }  } } |

修改方法:

整个方法,作用于调用的对象

|  |
| --- |
| *//修改一个方法* public synchronized void test2(){  synchronized (this){  for(int i = 0; i < 10 ;i++){  System.*out*.println(i);  }  } } |

修饰静态方法:

整个静态方法,作用于所有对象

|  |
| --- |
| *//修改一个静态方法* public static synchronized void test2(){  for(int i = 0; i < 10 ;i++){  System.*out*.println(i);  } } |

修饰类:

括号括起来的部分,作用于所有对象

|  |
| --- |
| *//修饰一个类* public synchronized void test1(){  synchronized (SynchronizedExample2.class){  for(int i = 0; i < 10 ;i++){  System.*out*.println(i);  }  } } |

synchronized:不可中断锁,适合竞争不激烈,可读性好

Lock:可中断锁,多样化同步,竞争激烈时能维持常态

Atomic:竞争激烈时能维持常态,比Lock性能好,只能同步一个值

Lock:依赖特殊的CPU指令,代码实现,ReentrantLock

线程安全性-可见性

可见性：一个线程对主内存的修改可以及时的被其他线程观察到。

导致共享变量在线程之间不可见的原因

1. 线程交叉执行
2. 重排序结合线程交叉执行
3. 共享变量更新后的值没有在工作内存与主存之间及时更新

JMM关于synchronized的两条规定:

1. 线程解锁前,必须把共享变量的最新值刷新到主内存
2. 线程加锁时,将清空工作内存中共享变量的值,从而使用共享变量时需要从主内存中重新读取最新的值(注意:加锁与解锁是同一把锁)

可见性-volatile

通过加入内存屏障和禁止重排序优化来实现

1. 对volatile变量写操作时,会在写操作最后加入一条store屏障指令,将本地内存中的共享变量值刷新到主内存
2. 对volatile变量读操作时,会在读操作前加入一条load屏障指令,从内主内存中读取共享变量





vlatile 比较方式作为变量标记量

volatile boolean inited =false;

//线程1

context = loadContext();

inited = true;

//线程2

while(!inited){

sleep();

}

doSomethingWithConfig(context);

线程安全性-总结

原子性:Atomic包、CAS算法、synchronized、Lock

可见性:synchronized、volatile

有序性:hasppend-before

线程安全:

1. 当多个线程访问某一个类(对象或方式)时,这个类始终都能表现出正确的行为,那么这个类(对象或方法)就是线程安全的。
2. synchronized:可以在任意对象及方法上加锁,而加锁的这段代码名为”互斥区”或”临界区”

线程不安全

|  |
| --- |
| public class MyThread extends Thread{   private int count = 5;   *//synchronized加锁* public void run(){  count--;  System.*out*.println(*currentThread*().getName() +" count = "+count);  }   public static void main(String[] args) {  MyThread myThread = new MyThread();  Thread t1 = new Thread(myThread,"t1");  Thread t2 = new Thread(myThread,"t2");  Thread t3 = new Thread(myThread,"t3");  Thread t4 = new Thread(myThread,"t4");  Thread t5 = new Thread(myThread,"t5");  t1.start();  t2.start();  t3.start();  t4.start();  t5.start();   } } |

总结:当多个线程访问myThread的run方式时,以排队的方式进行处理(这里排队是按照CPU分配的先后顺序而定的),一个线程想要执行synchronized修饰的方法里的代码,首先是尝试获得锁,如果拿到锁,执行synchronized代码体内存;拿不到锁,这个线程就会不断的尝试获得这把锁,直到拿到为止,而且是多个线程同时去竞争这把锁（也就是会有锁竞争的问题）。

多个线程多个锁

在多个线程,每个线程都可以拿到自己指定的锁,分别获得锁之后，执行synchronized方法体的内容。

|  |
| --- |
| public class MultiThread {   private static int num = 0;   */\*\* static \*/* public static synchronized void printNum(String tag){  try {  if(tag.equals("a")){  num = 100;  System.*out*.println("tag a,set num over!");  Thread.*sleep*(1000);  }else {  num = 200;  System.*out*.println("tag b,set num over!");  }  System.*out*.println("tag "+ tag + ", num = "+ num);  }catch(InterruptedException e){  e.printStackTrace();  }   }   *//注意观察run方法输出顺序* public static void main(String[] args) {  final MultiThread m1 = new MultiThread();  final MultiThread m2 = new MultiThread();   Thread t1 = new Thread(new Runnable() {  *@Override* public void run() {  m1.*printNum*("a");  }  });   Thread t2 = new Thread(new Runnable() {  *@Override* public void run() {  m2.*printNum*("b");  }  });   t1.start();  t2.start();   } |

对象锁的同步和异步

同步:synchronized

同步的概念就是共享,我们要牢牢记住”共享”这两个字,如果不是共享资源,就没有必要进行同步。

异步:asynchronized

异步的概念就是独立,互相之间不受到任何制约,就好像我们学习Http的时候,在页面发起的Ajax请求,我们还可以继续浏览或操作页面的内容,二者之间没有任何关系。

同步的目的就是为了线程安全,其实对于线程安全来说,需要满足两个特性:

原子性(同步)

可见性

脏读

对于对象的同步和异步的方法,我们在设计自己的程序的时候,一定要考虑问题的调整体,不然就会出现数据不一致的错误,很经典的错误就是脏读(dirtyread)

|  |
| --- |
| public class Dirtyread {   private String username = "bjsxt";  private String passwrod = "123";   public synchronized void setValue(String username,String passwrod){  this.username = username;  try {  Thread.*sleep*(2000);  } catch (InterruptedException e) {  e.printStackTrace();  }  this.passwrod = passwrod;  System.*out*.println("setValue最终结果:username = "+username+" ,password = "+passwrod);  }   *//synchronized* public void getValue(){  System.*out*.println("getValue方法得到: username = "+this.username+" ,password = "+passwrod);  }   public static void main(String[] args) throws InterruptedException {  final Dirtyread dr = new Dirtyread();  Thread t1 = new Thread(new Runnable() {  *@Override* public void run() {  dr.setValue("z3","456");  }  });  t1.start();  Thread.*sleep*(1000);  dr.getValue();  }   } |

总结：

在我们对一个对象的方法加锁的时候,需要考虑业务的整体性,即为setValue/getValue方法同时加锁synchronized同步关键字,保证业务(service)的原子性,不然会出现业务错误(也从侧面保证业务的一致性)。

synchronized其它概念

关键字synchronized拥有锁重入的功能,也就是在使用synchronized时，当一个线程得到了一个对象的锁后,再次请求对象时时可以再次得到该对象的锁。

|  |
| --- |
| public class SyncDubbo1 {   public synchronized void method1(){  System.*out*.println("method1..");  method2();  }  public synchronized void method2(){  System.*out*.println("method2..");  method3();  }  public synchronized void method3(){  System.*out*.println("method3..");  }   public static void main(String[] aegs){  final SyncDubbo1 sd = new SyncDubbo1();  Thread t1 = new Thread(new Runnable() {  *@Override* public void run() {  sd.method1();  }  });  t1.start();  }  } |

|  |
| --- |
| public class SyncDubbo2 {   static class Main{  public int i = 10;  public synchronized void operationSup(){  try {  i--;  System.*out*.println("Main print i =" +i);  Thread.*sleep*(100);  }catch (InterruptedException e){  e.printStackTrace();  }  }  }   static class Sub extends Main{   public synchronized void operationSub(){  try{  while(i > 0){  i--;  System.*out*.println("Sub print i = "+ i);  Thread.*sleep*(100);  this.operationSup();  }  }catch (InterruptedException e){  e.printStackTrace();  }  }   public static void main(String[] args){  Thread t1 = new Thread(new Runnable() {  *@Override* public void run() {  Sub sub = new Sub();  sub.operationSub();  }  });  t1.start();  }   }  } |

|  |
| --- |
| public class SyncException {  private int i = 0;  public synchronized void opration(){  while(true){  try {  i++;  Thread.*sleep*(200);  System.*out*.println(Thread.*currentThread*().getName()+" ,i = "+i);  if(i == 10){  Integer.*parseInt*("a");  }  } catch (Exception e) {  e.printStackTrace();  System.*out*.println("log info i = "+i);  }  }  }   public static void main(String[] args){  final SyncException se = new SyncException();  Thread t1 = new Thread(new Runnable() {  *@Override* public void run() {  se.opration();  }  });  t1.start();  } } |

总结:

对于web应用程序,异常释放锁的情况,如果不及时处理,很可能对你的应用程序业务逻辑产生严重的错误,比如你现在执行一个队列任务,很多对象都去在等待第一个对象正确执行完毕再去释放锁,但是第一个对象由于异常的出现,导致业务逻辑没有正常执行完毕,就释放了锁,那么可想而知后续的对象执行的都是错误的逻辑。所有这一点一定要引起注意,在编写代码的时候,一定要考虑周全。

volatile关键字

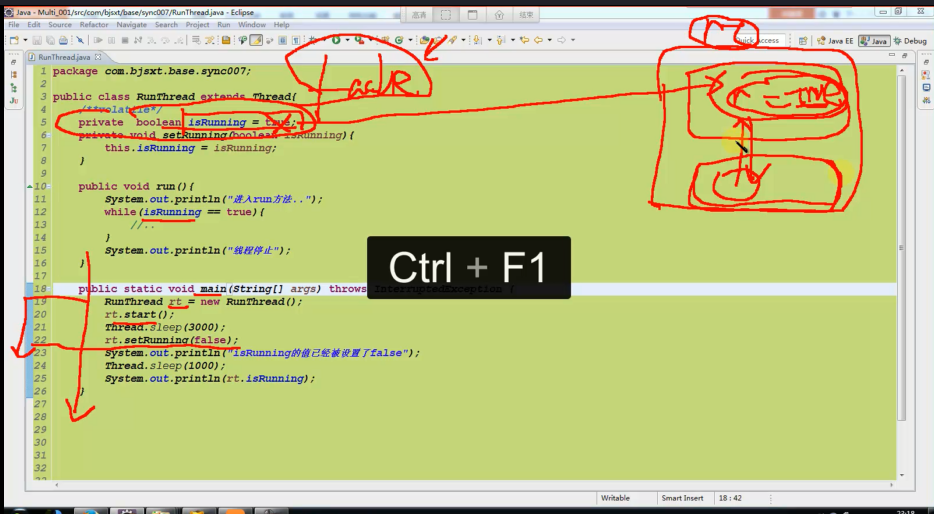
主要作用是使变量在多个线程间可见。

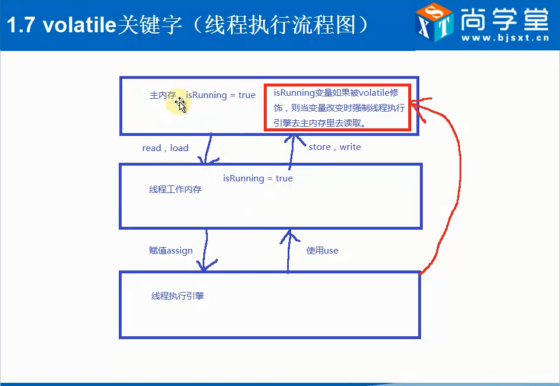
第一种：可以使用锁来解决多线程之间的变量可见性(效率低)

第二种：volatile

|  |
| --- |
| public class RunThread extends Thread{   *//volatile* private boolean isRunning = true;  private void setRunning(boolean isRunning){  this.isRunning = isRunning;  }   public void run(){  System.*out*.println("进入run方法..");  while (isRunning){   }  System.*out*.println("线程停止");  }   public static void main(String[] args) throws InterruptedException {  RunThread rt = new RunThread();  rt.start();  Thread.*sleep*(3000);  rt.setRunning(false);  System.*out*.println("isRunning的值已经被设置了false");  Thread.*sleep*(1000);  System.*out*.println(rt.isRunning);  } |

JDK1.5以后它对每一个线程(Thread)做了一个优化,对每一个线程加了一块独立的运行空间,空间中装载主内存中的引用变量,将主内存内的变量拷贝到了自己的运行空间中,然后线程在运行中去取自己的副本值,从而使线程运行效率提高。





volatile关键字不具备原子性

|  |
| --- |
| public class VolatileNoAtomic extends Thread{   private static volatile int count;  *//private static AtomicInteger count = new AtomicInteger(0);* private static void addCount(){  for(int i = 0;i < 1000;i++){  count++;  *//count.incrementAndGet();* }  System.*out*.println(count);  }   public void run(){  *addCount*();  }   public static void main(String[] args){  VolatileNoAtomic[] arr = new VolatileNoAtomic[10];  for(int i = 0;i < 10;i++){  arr[i] = new VolatileNoAtomic();  }   for(int i = 0;i < 10 ;i++){  arr[i].start();  }   }  } |

原子性使用Atomic\*

volatile关键字虽然拥有多个线程之间的可见性,但是却不具备同步性(也就是原子性),可以算上是一个轻量级的synchronized,性能要比synchronized强很多,不会造成阻塞(在很多开源的架构中,比如netty的底层代码就大量使用了volatile,可见netty性能一定是非常不错的。)这里需要注意:一般volatile用于只针对多个线程可见的变量操作,并不能代替synchronized的同步功能。

如果想保证原子性操作：Atomic、synchronized

总结：volatile关键字只具备可见性,没有原子性。要实现原子性建议使用actomic类的系列对象,支持原子性操作（注意atomic类值保证本身方法原子性,并不保证多次操作的原子性）

保证方法是原子性的操作

|  |
| --- |
| public class AtomicUse {   private static AtomicInteger count = new AtomicInteger();  *//多个addAndGet在方法内是非原子性的,需要加synchronized进行修饰保证4个addAndGet整体原子性  //synchronized* public int multiAdd(){  try {  Thread.*sleep*(100);  } catch (InterruptedException e) {  e.printStackTrace();  }  count.addAndGet(1);  count.addAndGet(2);  count.addAndGet(3);  count.addAndGet(4);  return count.get();  }   public static void main(String[] args){  final AtomicUse au = new AtomicUse();  List<Thread> ts = new ArrayList<Thread>();  for(int i = 0 ;i < 100 ;i++){  ts.add(new Thread(new Runnable() {  *@Override* public void run() {  System.*out*.println(au.multiAdd());  }  }));  }   for(Thread thread:ts){  thread.start();  }   }    } |

线程之间通讯

线程是操作系统中独立的个体,但这些个体如果不经过特殊的处理就不能成为一个整体,线程间的通信就成为整体的必用方式之一。当线程存在通讯指挥,系统间的交互性会更强大,在提高CPU利用率的同时还会使开发人员对线程任务在处理的过程中进行有效的把控与监督。

使用wait/notify方法实现线程间的通讯（注意这两个方法都是object的类的方法,换句话说java为所有的对象都提供了这两个方法）

1. wait和notify必须配合synchronized使用
2. wait方法释放锁,notify方法不释放锁。

|  |
| --- |
| public class ListAdd1 {   private volatile static List list = new ArrayList();   public void add(){  list.add("bjsxt");  }  public int size(){  return list.size();  }   public static void main(String[] args) {  final ListAdd1 list1 = new ListAdd1();   Thread t1 = new Thread(new Runnable() {  *@Override* public void run() {  try {  for(int i = 0;i < 10; i++){  list1.add();  System.*out*.println("当前线程:"+Thread.*currentThread*().getName()+"添加了一个元素..");  Thread.*sleep*(500);  }  } catch (InterruptedException e) {  e.printStackTrace();  }  }  },"t1");   Thread t2 = new Thread(new Runnable() {  *@Override* public void run() {  while(true){  if(list.size() == 5){  System.*out*.println("当前线程收到通知: "+Thread.*currentThread*().getName()+" list size = 5线程停止..");  throw new RuntimeException();  }  }  }  },"t2");   t1.start();  t2.start();   }    } |

|  |
| --- |
| public class ListAdd2 {  private volatile static List list = new ArrayList();   public void add(){  list.add("bjsxt");  }  public int size(){  return list.size();  }   public static void main(String[] args) {  final ListAdd2 list1 = new ListAdd2();   *//1.实例化出来一个lock  //当使用wait 和 notify 的时候，一定要配合着synchronized关键字去使用* final Object lock = new Object();   Thread t1 = new Thread(new Runnable() {  *@Override* public void run() {  try {  synchronized (lock){  for(int i = 0;i < 10; i++){  list1.add();  System.*out*.println("当前线程:"+Thread.*currentThread*().getName()+"添加了一个元素..");  Thread.*sleep*(500);  if(list1.size() == 5){  System.*out*.println("已经发出通知..");  lock.notify();  }  }  }  } catch (InterruptedException e) {  e.printStackTrace();  }  }  },"t1");   Thread t2 = new Thread(new Runnable() {  *@Override* public void run() {  synchronized (lock){  if(list.size() != 5){  try {  lock.wait();  } catch (InterruptedException e) {  e.printStackTrace();  }  }  System.*out*.println("当前线程收到通知: "+Thread.*currentThread*().getName()+" list size = 5线程停止..");  throw new RuntimeException();  }  }  },"t2");  t2.start();  t1.start();  }  } |

|  |
| --- |
| public class ListAdd2 {  private volatile static List list = new ArrayList();   public void add(){  list.add("bjsxt");  }  public int size(){  return list.size();  }   public static void main(String[] args) {  final ListAdd2 list2 = new ListAdd2();   *//1.实例化出来一个lock  //当使用wait 和 notify 的时候，一定要配合着synchronized关键字去使用* final Object lock = new Object();   Thread t1 = new Thread(new Runnable() {  *@Override* public void run() {  try {  synchronized (lock){  for(int i = 0;i < 10; i++){  list2.add();  System.*out*.println("当前线程:"+Thread.*currentThread*().getName()+"添加了一个元素..");  Thread.*sleep*(500);  if(list2.size() == 5){  System.*out*.println("已经发出通知..");  lock.notify();  }  }  }  } catch (InterruptedException e) {  e.printStackTrace();  }  }  },"t1");   Thread t2 = new Thread(new Runnable() {  *@Override* public void run() {  synchronized (lock){  if(list2.size() != 5){  try {  lock.wait();  } catch (InterruptedException e) {  e.printStackTrace();  }  System.*out*.println("----------------");  }  System.*out*.println("当前线程收到通知: "+Thread.*currentThread*().getName()+" list size = 5线程停止.."+list2.size());  throw new RuntimeException();  }  }  },"t2");  t2.start();  t1.start();  }  } |

使用wait/notify模拟Queue

BlockingQueue:首先它是一个队列,并且支持阻塞的机制,阻塞的放入和得到数据,我们要实现LinkedBlockingQueue

|  |
| --- |
| public class MyQueue {   *//1.需要一个乘装元素的集合* private LinkedList<Object> list = new LinkedList<Object>();  *//2.需要一个计数器* private AtomicInteger count = new AtomicInteger(0);   *//3.需要指定上限和下限* private int minSize = 0;  private int mixSize;   *//构造方法指定容器的最大长度* public MyQueue(int size){  this.mixSize = size;  }   *//5.初始化一个对象用户加锁* private final Object lock = new Object();   *//put* public void put(Object obj){  synchronized (lock){  while(this.mixSize == count.get()){  try {  lock.wait();  } catch (InterruptedException e) {  e.printStackTrace();  }  }  *//1.加入元素* list.add(obj);  *//2.计数器累加* count.incrementAndGet();  System.*out*.println("新加入的元素为:"+obj);  *//3.通知另外一个线程(唤醒)* lock.notify();  }  }   *//take* public Object take(){  Object ret = null;  synchronized (lock){  while(count.get() == this.minSize){  try {  lock.wait();  } catch (InterruptedException e) {  e.printStackTrace();  }  }  *//1.做移除元素操作* ret = list.removeFirst();  *//2.计数器递减* count.decrementAndGet();  *//3.唤醒另外一个线程* lock.notify();  }   return ret;  }   public int getSize(){  return count.get();  }   public static void main(String[] args) {  MyQueue mq = new MyQueue(5);  mq.put("n");  mq.put("2");  mq.put("3");  mq.put("4");  mq.put("5");   Thread t1 = new Thread(new Runnable() {  *@Override* public void run() {  mq.put("6");  mq.put("7");  System.*out*.println(mq.getSize());  }  },"t1");  t1.start();   Thread t2 = new Thread(new Runnable() {  *@Override* public void run() {  Object o1 = mq.take();  System.*out*.println(o1);  Object o2 = mq.take();  System.*out*.println(o2);  System.*out*.println(mq.getSize());  }  },"t2");  t2.start();  } } |

ThreadLocal

线程局部变量,是一种多线程间并发访问变量的解决方案,与其synchronized等加锁方式不同,ThreadLocal完全不提供锁,而使用与空间换时间的手段,以每个线程提供变量的独立副本,以保障线程安全。

以性能上说,ThreadLocal不具有绝对的优势,在并发不是很高的时候,加锁的性能会更好,但作为一套与锁完全无关的线程安全解决方案,在高并发量或者竞争激烈的场景,使用ThreadLocal可以在一定程度上减少锁竞争。

|  |
| --- |
| public class ConnThreadLocal {   public static ThreadLocal<String> th = new ThreadLocal<String>();  public void setTh(String value){  th.set(value);  }  public void getTh(){  System.*out*.println(Thread.*currentThread*().getName()+":"+this.th.get());  }   public static void main(String[] args) {  final ConnThreadLocal ct = new ConnThreadLocal();  Thread t1 = new Thread(new Runnable() {  *@Override* public void run() {  ct.setTh("张三");  ct.getTh();  }  },"t1");  Thread t2 = new Thread(new Runnable() {  *@Override* public void run() {  try {  Thread.*sleep*(1000);  ct.getTh();  } catch (InterruptedException e) {  e.printStackTrace();  }   }  },"t2");  t1.start();  t2.start();   } } |

单利和多线程

单利模式,最常见的就是饥饿模式,和懒汉模式,一个直接实例化对象,一个在调用方法时进行实例化对象。在多线程模式中,考虑到性能和线程安全问题,我们一般选择下面两个比较经典的单利模式,在性能提高的同时,又保证线程安全。

dubble check instance

|  |
| --- |
| public class DubbleSingleton {   private static DubbleSingleton ds;   public static DubbleSingleton getDs(){  if(ds == null){  try {  *//模拟初始化对象的准备时间* Thread.*sleep*(3000);  } catch (InterruptedException e) {  e.printStackTrace();  }  synchronized (DubbleSingleton.class){  if(ds == null){  ds = new DubbleSingleton();  }  }  }  return ds;  }   public static void main(String[] args){  Thread t1 = new Thread(new Runnable() {  *@Override* public void run() {  System.*out*.println(DubbleSingleton.*getDs*().hashCode());  }  },"t1");   Thread t2 = new Thread(new Runnable() {  *@Override* public void run() {  System.*out*.println(DubbleSingleton.*getDs*().hashCode());  }  }, "t2");   Thread t3 = new Thread(new Runnable() {  *@Override* public void run() {  System.*out*.println(DubbleSingleton.*getDs*().hashCode());  }  }, "t3");   t1.start();  t2.start();  t3.start();  } } |

static inner class

|  |
| --- |
| public class InnerSingleton {   private static class Singletion{  private static Singletion single = new Singletion();  }   public static Singletion getInstance(){  return Singletion.single;  }  } |

同步类容器

同步类容器都是线程安全的,但在某些场景下可能需要加锁来保护复合操作，复合类操作如:迭代(反复访问元素,遍历完容器中所有的元素)、跳转（根据指定的顺序找到当前元素的下一个元素）、以及条件运算，这些复合操作在多线程并发的修改容器时,可能会表现出意外的行为,最为经典的便是ConcurrentModificationException,原因是当容器迭代的过程中,被并发的修改了内容,这是由于早期迭代器设计的时候并没有考虑并发修改的问题。

同步类容器：如过来的Vector、HashTable。这些容器的同步功能其实都是JDK的Conllections.synchronized\*\*\*等工厂方法去创建的实现的,其底层的机制无非就是用传统的synchronized关键字对每个公用的方法都进行了同步,使得每次只能有一个线程访问容器的状态,这很明显不满足我们今天互联网时代高并发的需求,在保证线程安全的同事,也必须要有足够好的性能。

|  |
| --- |
| */\*\*  \* 多线程使用Vector或者HashTable的示例（简单线程同步问题）  \* @author alienware  \*/* public class Tickets {   public static void main(String[] args) {  *//初始化火车票池并添加火车票:避免线程同步可采用Vector替代ArrayList HashTable替代HashMap* final Vector<String> tickets = new Vector<String>();    *//Map<String, String> map = Collections.synchronizedMap(new HashMap<String, String>());* for(int i = 1; i<= 1000; i++){  tickets.add("火车票"+i);  }    */\*for (Iterator iterator = tickets.iterator(); iterator.hasNext();) {  String string = (String) iterator.next();  tickets.remove(20);  }\*/* for(int i = 1; i <=10; i ++){  new Thread("线程"+i){  public void run(){  while(true){  if(tickets.isEmpty()) break;  System.*out*.println(Thread.*currentThread*().getName() + "---" + tickets.remove(0));  }  }  }.start();  }  } } |

并发类容器

JDK5.0以后提供了多种并发类容器来代替同步类容器从而改善性能，同步类容器的状态都是串行化的,他们虽然实现了线程安全,但是严重降低了并发性,在多线程环境时,严重减低了应用程序的吞吐量。

并发类容器时专门针对并发设计的,使用了ConcurrentHashMap来代替给予散列的传统的HashTable,而且在ConcurrentHashMap中,添加了CopyinWriteArraySet,已经并发的Queue,ConcurrentLinkedQueue和LinkedBlockingQueue,前者是提高性能的队列,后者是以阻塞性能的队列,具体实现Queue还有很多,例如ArrayBlockingQueue、PriorityBlockingQueue、SynchronousQueue等。

ConcurrentMap

接口下有两个重要的实现:

ConcurrentHashMap

ConcurrentSkipListMap（支持并发排序功能,弥补ConcurrentHashMap）

ConcurrentHashMap内部使用段(Segment)来表示这些不同的部分,每个段其实就是一个小的HashTable,它们有自己的锁，这要多个修改操作发生在不同的段上,它们就可以并发进行,把一个整体分成了16个段(Segment),也就是最高支持16个线程的并发修改操作,这也是在多线程场景时减少锁的颗粒度从而降低竞争的一种方案,并且代码中大多共享变量使用volatile关键字声明,目的是第一时间获取修改的内容,性能非常好。

Copy-On-Write容器

Cop-On-Write简称cow,是一种用于程序设计中的优化策略。

JDK里的COW容器有两种:CopyOnWriteArrayList和CopyOnWrlteArraySet

COW容器非常有用,可以在非常多的并发场景中使用到。

什么是CopyOnWrite容器

CopyOnWrite容器即写是复制的容器,通俗的理解是当我们往一个容器添加元素的时候,不直接往当前容器添加,而是先将当前容器进行Copy,复制出一个新的容器,然后新的容器里添加元素,添加完成元素之后,再将原容器的引用指向新的容器,这样做的好处是我们可以对CopyOnWrite容器进行并发的读,而不需要加锁,因为当前容器不会添加任何元素。所有CopyOnWrite容器也是一种读写分离的思想,读写不同的容器。

最好字读多写少的场景下应用

并发Queue

在并发队列上JDK提供了两套实,一个是ConcurrentLinkedQueue为代表的高性能队列,一个是BlockingQueue借口为代表的阻塞队列,无论哪种都继承自Queue

ConcurrentLinkedQueue

是一个适用于高并发场景下的队列,通过无锁的方式,实现了高并发状态下的高性能,通常ConcurrentLinkedQueue性能好于BlockingQueue。它是一个基于链表节点的无界线程安全队列。该队列的元素遵循现进先出原则。头是最先进入的,尾是最近加入,该队列不允许null元素。

ConcurrentLinkedQueue重要方法:

add（）和offer（）都是加入元素的方法（在ConcurrentLinkedQueue中,这两个方法没有任何区别）

poll（）和peck（）都是取头元素节点,区别在于前者会删除元素,后者不会。

BlockingQueue接口(阻塞队列)

ArrayBlockingQueue

基于数组的阻塞队列实现,在ArrayBlockingQueue内部,维护了一个定长数组,以便缓存队列中的数据对象,其内部没有实现读写分离,也就意味着生产和消费不能完全并行,长度是需要定义的,可以指定先进先出或者先进后出,也叫有界队列,在很多场合非常适合使用。

LinkedBlockingQueue

基于链表的阻塞队列,同ArrayBlockingQueue类似,其内部也维持着一个数据缓冲队列(该队列由一个链表构成),LinkedBlockingQueue之所以能够高效的处理并发数据,是因为其内部实现采用分类锁(读写分离两个锁),从而实现生产者和消费者操作的完全并行运行,它是一个无界队列。

PriorityBlockingQueue

基于优先级的阻塞队列(优先级的判断通过构造函数传入的Compator对象来就决定,也就是说传入队列的对象必须实现Comparable接口),在实现PriorityBlockingQueue时,内部控制线程同步的锁采用了公平锁,他也是一个无界的队列。

DelayQueue

带有延迟时间的Queue,其中的元素只有当其指定的延迟时间到了,才能够从队列中获取到该元素。DelayQueue中的元素必须实现Delayed接口,DelayQueue是一个没有大小限制的队列,应用场景很多,比如对缓存超时的数据进行移除、任务超时处理、空闲连接的关闭等等。

SynchronousQueue

一种没有缓存的队列,生产者产生的数据直接会被消费者获取并消费。

AbstractQueuedSynchronizer - JUC- AQS

CountDownLatch（计数器）同步辅助类

使用它可以实现类似计数器的功能,比如有一个任务A，它要等待其他4个任务执行完毕之后才能执行,此时可以利用CountDownLatch来实现功能

|  |
| --- |
| public CountDownLatch(int count) {  };  //参数count为计数值 |

下面三个方式的CountDownlatch类中最重要的方法：

|  |
| --- |
| public void await() throws InterruptedException { };   //调用await()方法的线程会被挂起，它会等待直到count值为0才继续执行  public boolean await(long timeout, TimeUnit unit) throws InterruptedException { };  //和await()类似，只不过等待一定的时间后count值还没变为0的话就会继续执行  public void countDown() { };  //将count值减1 |

|  |
| --- |
| public class Test {       public static void main(String[] args) {           final CountDownLatch latch = new CountDownLatch(2);             new Thread(){               public void run() {                   try {                       System.out.println("子线程"+Thread.currentThread().getName()+"正在执行");                      Thread.sleep(3000);                      System.out.println("子线程"+Thread.currentThread().getName()+"执行完毕");                      latch.countDown();                  } catch (InterruptedException e) {                      e.printStackTrace();                  }               };           }.start();             new Thread(){               public void run() {                   try {                       System.out.println("子线程"+Thread.currentThread().getName()+"正在执行");                       Thread.sleep(3000);                       System.out.println("子线程"+Thread.currentThread().getName()+"执行完毕");                       latch.countDown();                  } catch (InterruptedException e) {                      e.printStackTrace();                  }               };           }.start();             try {               System.out.println("等待2个子线程执行完毕...");              latch.await();              System.out.println("2个子线程已经执行完毕");              System.out.println("继续执行主线程");          } catch (InterruptedException e) {              e.printStackTrace();          }       }  } |

|  |
| --- |
| 线程Thread-0正在执行  线程Thread-1正在执行  等待2个子线程执行完毕...  线程Thread-0执行完毕  线程Thread-1执行完毕  2个子线程已经执行完毕  继续执行主线程 |

Semaphore（信号量）

控制资源被同时访问的数量,Semaphore可以控制同时访问的线程个数,通过Acquire()获取一个许可,如果没有等待,而release()释放一个许可。

|  |
| --- |
| public Semaphore(int permits) {          //参数permits表示许可数目，即同时可以允许多少线程进行访问      sync = new NonfairSync(permits);  }  public Semaphore(int permits, boolean fair) {    //这个多了一个参数fair表示是否是公平的，即等待时间越久的越先获取许可      sync = (fair)? new FairSync(permits) : new NonfairSync(permits);  } |

下面说一下Semaphore类中比较重要的几个方法、首先是acquire()、release()方法:

|  |
| --- |
| public void acquire() throws InterruptedException {  }     //获取一个许可  public void acquire(int permits) throws InterruptedException { }    //获取permits个许可  public void release() { }          //释放一个许可  public void release(int permits) { }    //释放permits个许可 |

CyclicBarrier

ReentrantLock

Condition

FutureTask

### 线程池

为了更好的控制多线程,JDK提供了一套线程框架Executor,帮助开发人员有效的进行线程控制。它们都在java.util.concurrent包中,是JDK并发包的核心,其中有一个比较重要的类:Executors,它扮演者线程工厂的角色,我们通过Exexutors可以创建特定功能的线程池。

Executors创建线程池的方法:

newFixedThreadPool()

方法返回一个固定数量的线程池,该方法的线程数始终不变,当有一个任务提交时,若线程池中空闲,则立即执行,若没有,则会被暂缓在一个任务队列中等待有空闲的线程去执行。

newSingleThreadExecutor()

创建一个线程的线程池,若空闲则执行,若没有空闲线程则暂缓在任务队列中。

newCachedThreadPool()

返回一个可根据实际情况调整线程个数的线程池,不限制最大线程数量,若用空闲的线程则执行任务,若无任务则不创建线程。并且每一个空闲线程会在60秒后自动回收。

newScheduledThreadPool()

该方法返回一个SchededExecutorService对象,但该线程池可以指定线程的数量。

自定义线程池

若Exectors工厂类无法满足我们的需求,可以自己去创建自定义的线程池,其实Executors工厂类里面的创建线程池方法其内部实现均是用了ThreadPoolExecutor这个类,这个类可自定义线程。构造方法如下：

public ThreadPoolExecutor(

int corePoolSize, 当前核心线程相数

int maxinmnmPoolSize,最大线程数

long keepAliveTime,空闲时间

TlmeUnit unit,时间单位

BlockingQueue<Runnable> workQueue,线程排队队列

ThreadFactory threadFactory,

RejectedExecutionHandler handler 拒绝执行的方法,

)

Concurrent.util工具类详细讲解和使用

CyclicBarrier（C扣拜呐）

每一个线程代表一个跑步运动员,当运动员都准备好后,才一起出发,只要有一个人没有准备好,大家都等待

public class UseCyclicBarrier {  
  
  
  
 static class Runner implements Runnable{  
 private CyclicBarrier barrier;  
 private String name;  
  
 public Runner(CyclicBarrier barrier,String name){  
 this.barrier = barrier;  
 this.name = name;  
 }  
  
 *@Override* public void run() {  
 try {  
 Thread.*sleep*(1000\*(new Random().nextInt(5)));  
 System.*out*.println(name + "准备OK");  
 barrier.await();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 } catch (BrokenBarrierException e) {  
 e.printStackTrace();  
 }  
 System.*out*.println(name + "GO!");  
 }  
 }  
  
 public static void main(String[] args) {  
 CyclicBarrier barrier = new CyclicBarrier(3);  
 ExecutorService executor = Executors.*newFixedThreadPool*(3);  
 executor.submit(new Thread(new Runner(barrier,"zhangsan")));  
 executor.submit(new Thread(new Runner(barrier,"lisi")));  
 executor.submit(new Thread(new Runner(barrier,"wangwu")));  
 executor.shutdown();  
 }  
}

CountDownLacth

他经常用于监听某些初始化操作,当初始化执行完毕后,通知主线程继续工作。

public class UseCountDownLach {  
  
 public static void main(String[] args) {  
 //几个线程发送  
 final CountDownLatch countDown = new CountDownLatch(2);  
  
 Thread t1 = new Thread(new Runnable() {  
 *@Override* public void run() {  
 try {  
 System.*out*.println("进入线程t1"+"等待其他线程处理完成...");  
 countDown.await();  
 System.*out*.println("t1线程继续执行");  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 },"t1");  
 Thread t2 = new Thread(new Runnable() {  
 *@Override* public void run() {  
 try {  
 System.*out*.println("t2线程执行初始化操作...");  
 Thread.*sleep*(3000);  
 System.*out*.println("t2线程初始化完毕,通知t1线程继续...");  
 countDown.countDown();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 },"t2");  
 Thread t3 = new Thread(new Runnable() {  
 *@Override* public void run() {  
 try {  
 System.*out*.println("t3线程进行初始化操作...");  
 Thread.*sleep*(4000);  
 System.*out*.println("t3线程初始化完毕,通知t1线程继续...");  
 countDown.countDown();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 }, "t3");  
 t1.start();  
 t2.start();  
 t3.start();  
 }  
}

(重入锁、读写锁使用)锁的高级深化

### 死锁-必要条件

1. 互斥条件
2. 请求和保持条件

多线程的最佳实践

1. 使用本地变量
2. 使用不可变类
3. 最小化锁的使用范围
4. 使用线程池的Executor,
5. 宁可使用同步也不要使用线程的wait和notify方法。
6. 使用blockingQueue实现生产-消费模式
7. 使用并发集合而不是加了锁的同步集合
8. 使用Semaphore创建有界的访问
9. 宁可使用同步代码块,也不要使用同步的方法
10. 避免使用静态变量

### Concurrent.util常用类

#### CountDownLatch

说明：CountDownLatchd类位于java.util.concurrent包下,利用他可以实现类似计数器的功能,比如有一个任务A,它要等待其他4个任务执行完毕之后才执行,此时就可以利用CountDownLatch来实现这个功能。

等待多线程完成的CountDownLatchd

CountDownLatch类只提供了一个构造器:

|  |
| --- |
| public CountDownLatch(int count) { //参数count为计数值  if (count < 0) throw new IllegalArgumentException("count < 0");  this.sync = new Sync(count); } |

然后下面这3个方法是CountDownLatch类中最重要的方法

|  |
| --- |
| public void await() throws InterruptedException { //调用await()方法的线程会被挂起,它会等待直到count值为0才继续执行  sync.acquireSharedInterruptibly(1); }  public boolean await(long timeout, TimeUnit unit)  throws InterruptedException { //和await()类似,只不过等待一定的时间后count值还没有变为0的话就会继续执行  return sync.tryAcquireSharedNanos(1, unit.toNanos(timeout)); }  public void countDown() { //将count值减1  sync.releaseShared(1); } |

下面看一个CountDownlatch的用法

|  |
| --- |
| package com.lvguyuan.demo;  import java.util.concurrent.CountDownLatch; import java.util.concurrent.ExecutorService; import java.util.concurrent.Executors; import java.util.concurrent.TimeUnit;  */\*\*  \* @Auther: tianws  \* @Date: 2018/11/20 17:08  \* @Description: 利用它可以实现类似计数器的功能。比如有一个任务A，它要等待其他4个任务执行完毕之后才能执行，此时就可以利用CountDownLatch来实现这种功能了。  \*/* public class CountDownTest {  static CountDownLatch latch = new CountDownLatch(2);  static ExecutorService pool = Executors.*newFixedThreadPool*(2);   public static void main(String agrs[]){   pool.execute(new Runnable() {  public void run() {  try {  TimeUnit.***MILLISECONDS***.sleep(3);  }catch (InterruptedException e){  e.printStackTrace();  }  System.***out***.println("This is A");  latch.countDown();  }  });    pool.execute(new Runnable() {  public void run() {  try {  TimeUnit.***MILLISECONDS***.sleep(2000);  }catch (InterruptedException e){  e.printStackTrace();  }  System.***out***.println("This is B");  latch.countDown();  }  });   try {  System.***out***.println("等待2个子线程执行完毕...");  latch.await();  System.***out***.println("2个子线程已经执行完毕");  System.***out***.println("继续执行主线程");  }catch (InterruptedException e){   }  System.***out***.println("This is main");  } } |

执行结果

|  |
| --- |
| 等待2个子线程执行完毕...  This is A  This is B  2个子线程已经执行完毕  继续执行主线程  This is main |

#### CyclicBarrier

同步屏蔽CyclicBarrier

说明:字面意思回环栅栏,通过他可以实现让一组现场等待至摸个状态之后在全部同时执行,叫做回环是因为当所有等待线程被释放以后,CyclicBarrier可以被重用,我们暂时把这个状态叫做Barrier,当调用await()方法之后,线程就处于berrier了

CyclicBarrier构造方法

|  |
| --- |
| public CyclicBarrier(int parties, Runnable barrierAction) {  if (parties <= 0) throw new IllegalArgumentException();  this.parties = parties;  this.count = parties;  this.barrierCommand = barrierAction; }  public CyclicBarrier(int parties) {  this(parties, null); } |

参数parties值让多个线程或任务等待至barrier状态;参数barrierAction为这些线程都打到barrier状态是执行的内容。

CyclicBarrier中最重要的方法就是await方法

|  |
| --- |
| public int await() throws InterruptedException, BrokenBarrierException {  try {  return dowait(false, 0L);  } catch (TimeoutException toe) {  throw new Error(toe); *// cannot happen* } }  public int await(long timeout, TimeUnit unit)  throws InterruptedException,  BrokenBarrierException,  TimeoutException {  return dowait(true, unit.toNanos(timeout)); } |

第一个版本比较常用,用来挂起当前线程,直至所有线程都到达barrier状态在同时执行后续任务;

第二个版本是让这些线程等待至一定时间,如果这些线程没有到达barrier状态就直接让到达barrier的线程执行后续任务。

下面举几个列子

#### Semaphore

说明:semaphore翻译成字面意思为信号量,Semaphore可以同时访问的线程个数,通过acquire()获取一个许可,如果没有就等待,而release()释放一个许可。

Semaphore类提供了2个构造器

|  |
| --- |
| public Semaphore(int permits) { //参数permits表示许可数目,即同时可以运行多少线程进行访问  sync = new NonfairSync(permits); }  public Semaphore(int permits, boolean fair) { //这个多了一个参数fair表示是否公平的,即等待时间越久的越优先获取许可  sync = fair ? new FairSync(permits) : new NonfairSync(permits); } |

下面说一下Semaphore类中比较重要的几个方法,首先是acquire(),release()方法

|  |
| --- |
| public void acquire() throws InterruptedException { //获取一个许可  sync.acquireSharedInterruptibly(1); }  public void acquire(int permits) throws InterruptedException { //获取permits个许可  if (permits < 0) throw new IllegalArgumentException();  sync.acquireSharedInterruptibly(permits); }  public void release() { //释放一个许可  sync.releaseShared(1); }  public void release(int permits) { //释放permits个许可  if (permits < 0) throw new IllegalArgumentException();  sync.releaseShared(permits); } |

acquire()用来获取一个许可,若无许可获取,则会一直等待,直到获得许可。

release()用来释放许可,主要,在释放许可之前,必须先获得许可。

这4个方法都会被阻塞,如果想立即得到执行结果,可以使用下面几个方法:

|  |
| --- |
| public boolean tryAcquire() { //尝试获取一个许可,若获取成功,则立即放回true,若获取失败,则立即返回false  return sync.nonfairTryAcquireShared(1) >= 0; }  public boolean tryAcquire(long timeout, TimeUnit unit)  throws InterruptedException { //尝试获取一个许可,若在指定的时间内获取成功,则立即放回true,否则则立即返回false  return sync.tryAcquireSharedNanos(1, unit.toNanos(timeout)); }  public boolean tryAcquire(int permits) { //尝试获取permits个许可,若获取成功,则立即放回true,若获取失败,则立即返回false  if (permits < 0) throw new IllegalArgumentException();  return sync.nonfairTryAcquireShared(permits) >= 0; }  public boolean tryAcquire(int permits, long timeout, TimeUnit unit)  throws InterruptedException { //尝试获取permits个许可,若在指定的时间内获取成功,则立即返回true,否则立即返回false  if (permits < 0) throw new IllegalArgumentException();  return sync.tryAcquireSharedNanos(permits, unit.toNanos(timeout)); } |

另外还可以通过availablePermits()方法得到可用许可数目。

下面通过一个例子来看一下Semaphore的具体使用:

假若一个工厂有5个机器,但是有8个人,一台机器同时只能被一个工人使用,只有使用完了,其他工人才能继续使用,name我们就可以通过Semaphore来实现

|  |
| --- |
| package com.lvguyuan.demo;  import java.util.concurrent.Semaphore;  */\*\*  \* @Auther: tianws  \* @Date: 2018/11/21 13:42  \* @Description:  \*/* public class semaphoreTest {   public static void main(String[] args) {  int N = 8; *//工人数* Semaphore semaphore = new Semaphore(5); *//机器数目* for(int i=0;i<N;i++)  new Worker(i,semaphore).start();  }   static class Worker extends Thread{  private int num;  private Semaphore semaphore;  public Worker(int num,Semaphore semaphore){  this.num = num;  this.semaphore = semaphore;  }   *@Override* public void run() {  try {  semaphore.acquire();  System.***out***.println("工人"+this.num+"占用一个机器在生产...");  Thread.*sleep*(2000);  System.***out***.println("工人"+this.num+"释放出机器");  semaphore.release();  } catch (InterruptedException e) {  e.printStackTrace();  }  }  }  } |

Semaphore型号量非常适合高并发访问

PV(page view)网站的访问量，页面浏览或点击量,用户没刷新一次就会记录一次

UV（unique Visitor）访问网站的一台电脑客户端为一个访客，一般来讲,时间上以00:00-24:00之内相同的ip客户记录一次

QPS(query per second)即每秒查询书,qps很大程度上代表了系统业务上的繁忙程度,每次请求的背后,可能对应着多次磁盘I/O,多次网络请求,多个cpu时间片等,我们通过qps可以非常直观的了解当前系统业务状况，一旦qps超过所设定的预警阈值,可以考虑增加机器全对集群扩容,以免压力过大导致宕机,可以根据前期的压力测试得到估值,在结合后期综合运维情况,估算出阈值。

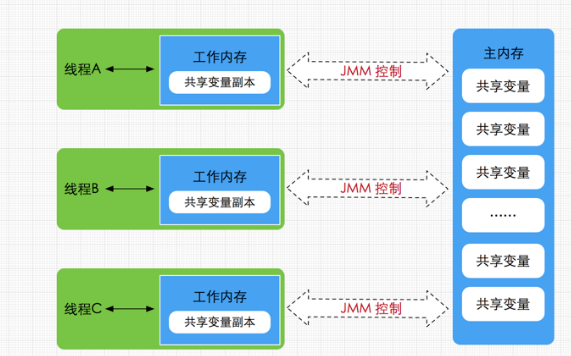
RT(response time)即请求的响应时间,这个指标非常关键,直接说明前端用户的体验,因此任何系统设计师都想降低rt时间。

当然还设计cpu、内存、网络、磁盘等情况，更细节的问题很多，如select、update、ps等数据层面的设计。

### Executor线程池

#### JMM

说明：java内存模型(即java Memory Model 简称JMM)本身是一种抽象的概念,并不是真实存在,它描述的是一组规则或规范,通过这组规范定义了程序中各个变量（包含实例字段,静态字段和构成数组对象的元素）的访问方式,由于JVM运行程序的实体是线程,而每个线程创建时JVM都会为其创建一个工作内存（有些地方称为栈空间）,用于存储线程私有的数据,而java内存模型中规定所有变量都存储在主内存,主内存是共享内存区域,所有线程都可以访问,但线程对变量的操作（读取赋值等）必须在工作内存中进行,首先要将变量从主内存拷贝到自己的工作内存空间,然后对变量进行操作,操作完成后再将变量写回主内存,不能直接操作主内存中的变量,工作内存中存储着主内存中的变量副本拷贝,前面说过,工作内存是每个线程私有的数据区域,因此不同的线程间无法访问对方的工作内存,线程间的通信(传值)必须通过主内存来完成,其简要访问过程如下图：



#### 线程

#### 线程池

### Lock锁

#### Synchronized（同步锁）

#### Volatile（易变的）

#### Condition

#### ReentrantLock（可重入锁）

#### ReentrantReadWriteLock

### atomic

线程安全性

定义：当多个线程访问某一个类时,不管运行时环境采用何种调度方式或者这些进程将如何交替执行,并且在主调代码中不需要任何额外的同步或协同,这类都是表现出正确的行为,那么就称这个类是线程安全的

原子性：提供了互斥访问,同一时刻只能有一个线程来对它进行操作

可见性：一个线程对内存的修改可以及时的被其他线程观察到

有序性：一个线程观察其他线程中的指令执行顺序,由于指令重排序的存在,该观察结果一般杂乱无序

原子性-Atomic包

AtomicXXX ：CAS、Unsafe.compareAndSwapInt

|  |
| --- |
| package com.lvguyuan.atomic;  import java.util.concurrent.CountDownLatch; import java.util.concurrent.ExecutorService; import java.util.concurrent.Executors; import java.util.concurrent.Semaphore;  */\*\*  \* @Auther: tianws  \* @Date: 2018/11/22 10:25  \* @Description:  \*/* public class CountExample1 {  *//请求总数* public static int clientTotal = 5000;   *//同时并发执行的线程数* public static int threadTotal = 200;   public static Integer count = 0;   public static void main(String[] args) throws InterruptedException {  ExecutorService executorService = Executors.*newCachedThreadPool*();  final Semaphore semaphore = new Semaphore(threadTotal);  final CountDownLatch countDownLatch = new CountDownLatch(clientTotal);  for(int i = 0;i < clientTotal ; i++){  executorService.execute(() ->{  try {  semaphore.acquire();  *add*();  semaphore.release();  } catch (InterruptedException e) {  e.printStackTrace();  }  countDownLatch.countDown();  });  }  countDownLatch.await();  executorService.shutdown();  System.***out***.println("count{"+count+"}");  }   public static void add(){  count++;  } } |

|  |
| --- |
| package com.lvguyuan.atomic;  import java.util.concurrent.CountDownLatch; import java.util.concurrent.ExecutorService; import java.util.concurrent.Executors; import java.util.concurrent.Semaphore; import java.util.concurrent.atomic.AtomicInteger;  */\*\*  \* @Auther: tianws  \* @Date: 2018/11/22 10:25  \* @Description:  \*/* public class CountExample2 {  *//请求总数* public static int clientTotal = 5000;   *//同时并发执行的线程数* public static int threadTotal = 200;   public static AtomicInteger count = new AtomicInteger(0);   public static void main(String[] args) throws InterruptedException {  ExecutorService executorService = Executors.*newCachedThreadPool*();  final Semaphore semaphore = new Semaphore(threadTotal);  final CountDownLatch countDownLatch = new CountDownLatch(clientTotal);  for(int i = 0;i < clientTotal ; i++){  executorService.execute(() ->{  try {  semaphore.acquire();  *add*();  semaphore.release();  } catch (InterruptedException e) {  e.printStackTrace();  }  countDownLatch.countDown();  });  }  countDownLatch.await();  executorService.shutdown();  System.***out***.println("count{"+count+"}");  }   public static void add(){  count.incrementAndGet();  } } |

AtomicLog、LongAdder

AtomicReference、AtomicReferenceFieldUpdater

AtomicStampReference

解决CAS的ABA问题

## disruptor

### disruptor框架介绍与Hello World

它能够在一个线程里每秒处理6百万订单

p >队列>c

### disruptor详细说明与使用

### disruptor应用(并发场景实例讲解)

### Spring与线程安全

Spring Bean:singleton 、prototype

## JVM虚拟机

## 网络编程

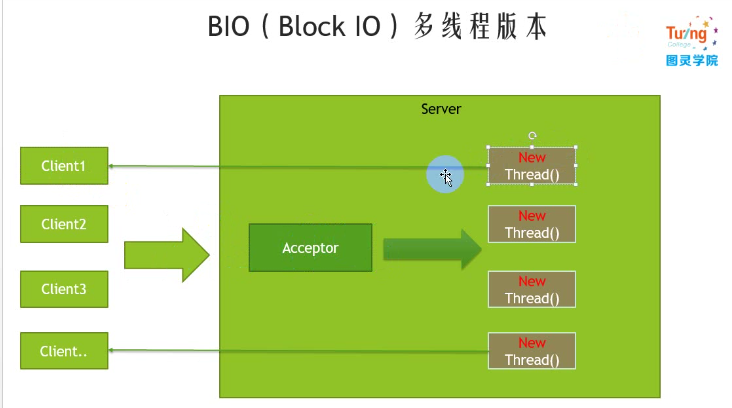
# Netty

BIO的网络连接,传递BIO多线模型

|  |
| --- |
| */\*\*  \* 单线程  \*/* public class TraditionalSocketDemo {   public static void main(String[] args) throws IOException {  ServerSocket serverSocket = new ServerSocket(7777);  System.*out*.println("服务端启动");  while (true){  *//获取socket套接字  //accept()阻塞点* Socket socket = serverSocket.accept();  System.*out*.println("有客户端连接上来了");  *//获取客户端输入流* InputStream inputStream = socket.getInputStream();  byte[] b = new byte[1024];  while (true){  *//循环读取数据  //read()阻塞点* int data = inputStream.read(b);  if(data != -1){  String info = new String(b,0,data,"GBK");  System.*out*.println(info);  }else{  break;  }   }   }     }    } |

多线BIO

|  |
| --- |
| */\*\*  \* 传统BIO多线程版本  \*/* public class TraditionalSocketDemo2 {  public static void main(String[] args) throws IOException {  ServerSocket serverSocket = new ServerSocket(7777);  System.*out*.println("服务端启动...");  while (true){  *//获取socket套接字  //accept()阻塞点* Socket socket = serverSocket.accept();  System.*out*.println("有新客户端连接上来了...");   new Thread(new Runnable() {  *@Override* public void run() {  try {  InputStream is = socket.getInputStream();  byte[] b = new byte[1024];  while (true){  *//循环读取数据  //read() 阻塞点* int data = is.read(b);  if(data != -1){  String info = new String(b,0,data,"GBK");  System.*out*.println(info);  }else{  break;  }  }   } catch (IOException e) {  e.printStackTrace();  }  }  });   }    } } |



多线程为异步IO

|  |
| --- |
| */\*\*  \* 传统BIO 多线程伪异步IO  \*/* public class TraditionalSocketMultiThreadDemo {   public static void main(String[] args) throws IOException {  ExecutorService threadPool = Executors.*newCachedThreadPool*();  ServerSocket serverSocket = new ServerSocket(7777);  System.*out*.println("服务器启动...");  while (true){  *//获取socket套接字* Socket socket = serverSocket.accept();  threadPool.submit(new Runnable() {  *@Override* public void run() {  try {  InputStream is = socket.getInputStream();  byte[] b = new byte[1024];  while (true) {  int data = is.read(b);  String info = new String(b,0,data,"GBK");  System.*out*.println(info);  }   } catch (IOException e) {  e.printStackTrace();  }  }  });   }  } } |

整个BIO最主要的问题

每当一个新的客户端请求接入时,服务端必须创建一个线程来处理这条链路

如果在高性能高并发场景下肯定是没法用的

NIO(Non Block IO)

Non Block io 非阻塞IO

Selector通道的管理器

ServerSocketChannel(ServerSocket)：只关心客户端连接事件

SocketChannel(Socket):关心读事件,写事件,读写事件

SelectionKey:事件集合

Natyy优化了Reactor

Reactor

单线程Reactor

多线程Reactor

主从多线程Reactor

阻塞与非阻塞

线程在访问某一个资源的时候,该资源是否准备就绪的一种处理方式

同步与异步

同步和异步是指访问数据的一种机制

BIO

同步阻塞IO, Block IO,并发处理能力低,耗时久,销耗带宽。

一问一答的形式。

改良后通过线程池,来实现伪异步IO

NIO

同步非阻塞IO，New Io(Non-Blick IO)

AIO

异步非阻塞IO

Netyy的三种线程模型

Reactor线程模型:

单线程模型:所有的IO操作都由同一个NIO线程处理的

多线程模型:由一组NIO线程处理IO操作

主从线程模型:一组线程池接受请求,一组线程池处理IO

实时通信

1. ajax轮训
   * 1. 通过js使用定时每隔一段时间异步进行后的请求。
2. Long pull
   * 1. 原理和ajax差不不多,但是阻塞,一直如服务端没有响应就一直等待
3. webscoket
   * 1. 是h5的一种协议,http 1.0不支持长连接,1.1支持长连接
     2. 持久化的协议

Websoket api

var socket = new WebSocket(“ws://[ip]:[port]”);

生命周期:

onopen() 服务端与客户端建立连接触发

onmessage() 收到消息的时候

onerror() 后端出现异常

onclose() 客户端和服务端连接发生关闭

主动方法：

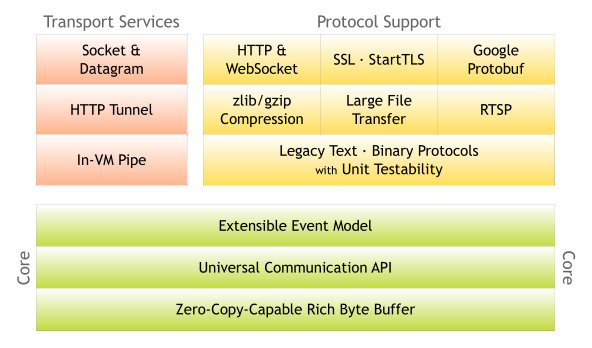
Socket.send()

Socket.close()

## Netty线程模型

说明：

Netty架构



Netty模型组件

bootstrap

Netty服务端及客户端启动类

buffer

缓冲相关,对NIO Buffer做了一些优化、分装

channel

处理客户端以服务端之间的连接通道

chontainer

连接其他容器的代码,例如spring

handler

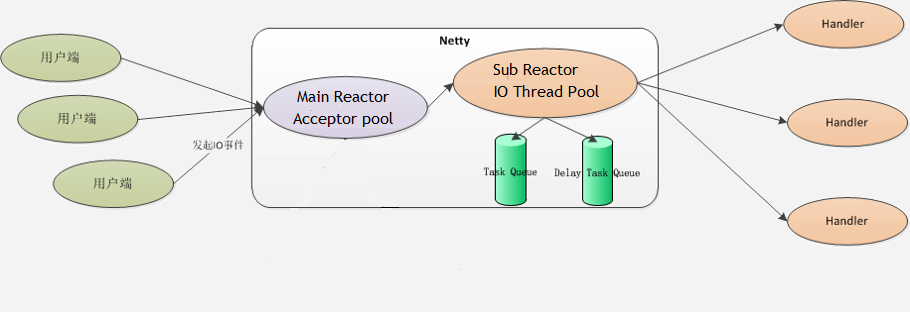
实现协议编码等附加功能

logging日志

util工具类

Netty线程模型

主从多线程Reactor模型



Netty中的对应关系

MainReactor = Boos

SubReactor = Worker

Boos线程

Worker线程

Netty线程模型源码分析

## 高性能反序列化协议protobuf

说明：序列化就是将我们的对象序列化成字节数组,主要用于网络传输。

序列化协议基础

基础类型int在内存中的原始序列化

大端序列

先写高位,在写地位

小端序列

先写地位,在写高位

基于nio的序列化

缺点不能自动扩容

ByteBuffer buffer = ByteBuffer.allocate(8);//序列化

ByteBuffer bb = ByteBuffer.wrap(数组); //反序列化

基于netty的序列化

//创建一个动态扩容

ChannelBuffer buffer = ChannelBuffers.bynamicBuffer();

//byte数组的大小由buffer中写指针的位置决定

//往ChannelBuffer中写数据的时候,这个写指针就会移动写的数据的长度。buffer.writerIndex();

//序列化

buffer.readBytes()

//反序列化

ChannelBuffer wrappedBuffer = ChannelBuffers.wrappedBuffer();

wrappedBuffer.readInt();

对象序列化

java原始对象序列化

性能低

ObjectInpuStream

ObjectOutpuStream

基于protobuf序列化

高性能序列化

Protobut序列化过程源码解析

Netty+Protobuf案例

# MUI H5+ Hbuilder

MNI ：<http://www.dcloud.io>

H5 PLUS(H5+) : <http://www.html5plus.org/doc> 调用原生组件

Hbuilder 开发工具

# 2.javaEE

### ServletConfig

配置Servlet初始化参数

在Servlet的配置文件web.xml中，可以使用一个或多个<init-param>标签为servlet配置一些初始化参数。

|  |
| --- |
| <servlet>  <servlet-name>ServletConfigDemo1</servlet-name>  <servlet-class>gacl.servlet.study.ServletConfigDemo1</servlet-class>  <!--配置ServletConfigDemo1的初始化参数 -->  <init-param>  <param-name>name</param-name>  <param-value>gacl</param-value>  </init-param>  <init-param>  <param-name>password</param-name>  <param-value>123</param-value>  </init-param>  <init-param>  <param-name>charset</param-name>  <param-value>UTF-8</param-value>  </init-param>  </servlet> |

### ServletContext

web容器在启动时，它会为每个web应用创建一个对应的ServleContext对象，它代表当前web应用。

ServletConfig对象中维护了ServletContext对象的引用，开发人员在编写servlet是，可以通过ServletConfig.getServletContext方法获得ServletContext对象。

由于一个web引用中的所有Servle共享一个ServletContext对象，因此Servlet对象之间可以通过ServletContext对象来实现通讯，ServletContext对象通常也被称为context域对象。

ServletContext的应用

获取WEB引用的初始化参数

在web.xml文件中使用<conext-param>标签配置web应用的初始化参数:

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee  http://java.sun.com/xml/ns/javaee/web-app\_3\_0.xsd">  <display-name></display-name>  <!-- 配置WEB应用的初始化参数 -->  <context-param>  <param-name>url</param-name>  <param-value>jdbc:mysql://localhost:3306/test</param-value>  </context-param>  <welcome-file-list>  <welcome-file>index.jsp</welcome-file>  </welcome-file-list>  </web-app> |

获取web应用的初始化参数

|  |
| --- |
| public class ServletContextDemo3 extends HttpServlet {  public void doGet(HttpServletRequest request, HttpServletResponse response)  throws ServletException, IOException {  ServletContext context = this.getServletContext();  //获取整个web站点的初始化参数  String contextInitParam = context.getInitParameter("url");  response.getWriter().print(contextInitParam);  }  public void doPost(HttpServletRequest request, HttpServletResponse response)  throws ServletException, IOException {  doGet(request, response);  }  } |

### Servlet

### 过滤器（Filter）

开发步骤

Filter开发分为两个步：

1. 编写java类实现Filter接口，并实现其doFiter方法。
2. 在web.xml文件中使用<filter>和<filter-mapping>元素对编写的filter类进行注册，并设置它所能拦截的资源。

|  |
| --- |
| public class MyFilter implements Filter {  *@Override* public void init(FilterConfig filterConfig) throws ServletException {  System.*out*.println("----过滤器初始化----");  }   *@Override* public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) throws IOException, ServletException {  *//对request和response进行一些预处理* request.setCharacterEncoding("UTF-8");  response.setCharacterEncoding("UTF-8");  response.setContentType("text/html;charset=UTF-8");   System.*out*.println("FilterDemo01执行前！！！");  chain.doFilter(request, response); *//让目标资源执行，放行* System.*out*.println("FilterDemo01执行后！！！");  }   *@Override* public void destroy() {  System.*out*.println("----过滤器销毁----");  } } |
| *<!--配置过滤器-->* <filter>  <filter-name>FilterDemo01</filter-name>  <filter-class>com.demo.filter.MyFilter</filter-class> </filter> *<!--映射过滤器-->* <filter-mapping>  <filter-name>FilterDemo01</filter-name>  *<!--“/\*”表示拦截所有的请求 -->* <url-pattern>/\*</url-pattern> </filter-mapping> |

init方法中使用FilterConfig读取配置

|  |
| --- |
| <filter>  <filter-name>FilterDemo01</filter-name>  <filter-class>com.demo.filter.MyFilter</filter-class>  <init-param>  <param-name>usernam</param-name>  <param-value>tianweishuo</param-value>  </init-param>  <init-param>  <param-name>password</param-name>  <param-value>123456</param-value>  </init-param> </filter> |
| *@Override* public void init(FilterConfig filterConfig) throws ServletException {  *//得到过滤器的名字* String filterName = filterConfig.getFilterName();  System.*out*.println("----过滤器名称:"+filterName+"----");  *//得到在web.xml文件中配置的初始化参数* String name = filterConfig.getInitParameter("name");  String password = filterConfig.getInitParameter("password");  System.*out*.println(name);  System.*out*.println(password);  *//返回过滤器的所有初始化参数的名字的枚举集合。* Enumeration<String> initParameterNames = filterConfig.getInitParameterNames();  while (initParameterNames.hasMoreElements()){  String paramName = (String) initParameterNames.nextElement();  System.*out*.println(paramName);  } |

### 监听器

基本概念：

javaweb中的监听器是servlet规范中的一种特殊类，它用于监听web应用程序中的ServletContext，HttpSession和ServletRequest等域对象的创建与销毁事件，已经监听这些域对象中的属性发生修改的事件。

Servlet监听器的分类

在servlet规范中定义了多种类型的监听器，它们用于监听的事件源分别为

ServletContext，HttpSession，ServletRequest这三个域对象，Servlet规范这三个对象的操作，又把多种类型的监听器划分为三种类型：

1. 监听对象自身的创建和销毁的事件监听器。
2. 监听域对象中农的属性的增加和删除的事件监听器。
3. 监听绑定到HttpSession域中的某个对象的状态的事件监听器。

监听ServletContext域对象的创建和销毁

ServletContextListener接口用于监听ServletContext对象的创建和销毁事件。实现了ServletContextListener接口的类都可以对ServletContext对象的创建和销毁进行监听。

当ServletContext对象创建时，激发contextInitialized(ServletContextEcent sce)方法

当ServletContext对象销毁时，激发contextDestroyed(ServletContextEvent sce)方法

|  |
| --- |
| public class MyServletContextListener implements ServletContextListener {  *@Override* public void contextInitialized(ServletContextEvent sce) {  System.*out*.println("ServletContext对象创建");  }   *@Override* public void contextDestroyed(ServletContextEvent sce) {  System.*out*.println("ServletContext对象销毁");  } } |
| *<!-- 注册针对ServletContext对象进行监听的监听器 -->* <listener>  <description>ServletContextListener监听器</description>  *<!--实现了ServletContextListener接口的监听器类 -->* <listener-class>com.demo.listener.MyServletContextListener</listener-class> </listener> |

监听HttpSession域对象的创建和销毁

httpSessionListener接口用于监听HttpSession对象的创建和销毁

创建一个Session是，激发sessionCreate（httpSessionEvent se）方法

销毁一个Session是，激发sessionDestroye(HttpSessionEvent se) 方法

|  |
| --- |
| public class MyHttpSeesionListener implements HttpSessionListener {  *@Override* public void sessionCreated(HttpSessionEvent se) {  System.*out*.println("HttpSession对象创建");  }    */\*\* HttpSession的销毁时机需要在web.xml中进行配置，如下：  <session-config>  <session-timeout>1</session-timeout>  </session-config>  这样配置就表示session在1分钟之后就被销毁  \*/  @Override* public void sessionDestroyed(HttpSessionEvent se) {  System.*out*.println("HttpSession对象创建");  } } |
| *<!--注册针对HttpSession对象进行监听的监听器-->* <listener>  <description>HttpSessionListener监听器</description>  <listener-class>com.demo.listener.MyHttpSeesionListener</listener-class> </listener> *<!-- 配置HttpSession对象的销毁时机 -->* <session-config>  *<!--配置HttpSession对象的1分钟之后销毁 -->* <session-timeout>1</session-timeout> </session-config> |

监听ServletRequest域对象的创建和销毁

ServletRequestListener接口用于监听ServletRequest对象的创建和销毁

Request对象创建时，监听器的requestInitelalized(ServletRequestEvent sre)方法将会被调用

Request对象销毁时，监听器的requestDestroyed(ServletEvent sre)方法被调用

ServlletRequest域对象创建和销毁时机：

创建:用户每一次访问都会创建request对象

销毁:当访问结束，request对象就会销毁

|  |
| --- |
| public class MyServletRequestListener implements ServletRequestListener {  *@Override* public void requestDestroyed(ServletRequestEvent sre) {  System.*out*.println(sre.getServletRequest() + "销毁了！！");  }   *@Override* public void requestInitialized(ServletRequestEvent sre) {  System.*out*.println(sre.getServletRequest() + "创建了！！");  } } |
| *<!--注册针对ServletRequest对象进行监听的监听器-->* <listener>  <description>ServletRequestListener监听器</description>  <listener-class>com.demo.listener.MyServletRequestListener</listener-class> </listener> |

监听域对象中属性的变更的监听器

ServletContextAttributeListener

HttpSeesionAttributeListener

ServletRequestAttributeListener

这三个接口中都定义了三个方法来处理被监听对象中的属性的增加，删除和替换的事件，同一个事件在这三个借口中对应的方法名完全相同，只是接受的参数类型不同。

# 3.spring Scheduled定时器

## 一:注解方式

@Scheduled(cron = "0/5 \* \* \* \* ?")

# 4.spring Security(权限框架)

# 5.Apache Shiro

Apache Shiro说明:

Primary Concerns:

Authentication

Authorization

Session Management

Cryptography

Supporting Features:

Web Support

Caching

Concurrency

Testing

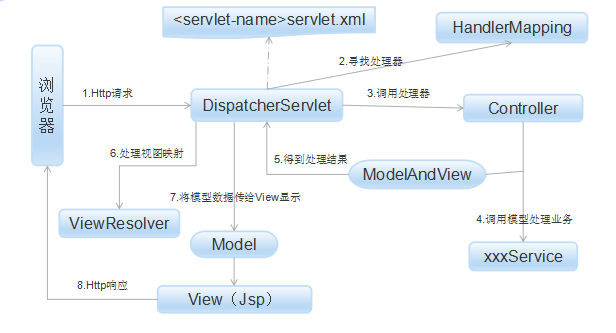
Run As

RememberMe

ApplicationCode -> Subject ->Shiro SecurityManager -> Realm

Apache Shiro身份认证

# springmvc



1. http请求:客户端请求提交到DispatcherServlet
2. 寻找处理器:由DispatcherServlet控制器查询一个或多个HandlerMapping，找到处理请求的Controller。
3. 调用处理器:DispatcherServlet将请求提交到Controller。
4. 5.调用业务处理和返回结果:Controller调用业务逻辑处理后，返回ModelAndView。

6.7.处理视图映射并返回模型:DispatcherSerlvet查询一个或多个ViewResoler视图解析器，找到ModelAndView指定的视图。

8.Http响应：视图负责将结果显示到客户端。

### DispatcherServlet

DispatcherServlet接口:

Spring提供的前端控制器，所有的请求都经过他来统一分发，在DispatcherServlet将请求分发给Spring Controller之前，需要借助于Spring提供的HandlerMapping定位到具体的Controler。

HandlerMapping接口:

能够完成客户请求到Controller映射。

Controller接口:

需要为并发用户处理上述请求，因此实现Controller接口时，必须保证线程安全并且可重用。

ViewResolver接口:

Spring提供的视图解析器(ViewResolver)在Web应用中查找View对象，从而将相应结果渲染给客户。

DipatcherServlet是整个Spring MVC的核心，它负责接收HTTP请求组织协调Spring MVC的各个组成部分。

主要工作有一下三项：

1. 截获符合特点格式的URL请求。
2. 初始化DispatcherServlet上下文对应WebApplicationContext，并将其业务层、持久化层的WebApplicationContext建立关联。
3. 初始化Spring MVC的各个组成组件，并装配到DispatcherServlet中。

|  |
| --- |
| <servlet>  <servlet-name>springServlet</servlet-name>  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>  <init-param>  <param-name>contextConfigLocation</param-name>  <param-value>classpath:spring/servlet-context.xml</param-value>  </init-param>  <load-on-startup>1</load-on-startup> </servlet> <servlet-mapping>  <servlet-name>springServlet</servlet-name>  <url-pattern>/\*</url-pattern> </servlet-mapping> |

Spirngmvc-context.xml文件配置，这一部分主要开启注解功能，配置视图解析器

|  |
| --- |
| *<?*xml version="1.0" encoding="UTF-8"*?>* <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:p="http://www.springframework.org/schema/p"  xmlns:context="http://www.springframework.org/schema/context"  xmlns:mvc="http://www.springframework.org/schema/mvc" xmlns:aop="http://www.springframework.org/schema/aop"  xmlns:tx="http://www.springframework.org/schema/tx"  xsi:schemaLocation="http://www.springframework.org/schema/beans   http://www.springframework.org/schema/beans/spring-beans-4.2.xsd   http://www.springframework.org/schema/context   http://www.springframework.org/schema/context/spring-context-4.2.xsd   http://www.springframework.org/schema/aop   http://www.springframework.org/schema/aop/spring-aop-4.2.xsd   http://www.springframework.org/schema/mvc   http://www.springframework.org/schema/mvc/spring-mvc-4.2.xsd">  *<!--扫描Controller-->* <context:component-scan base-package="com.vacomall.controller" />  *<!-- 这个类用于Spring MVC视图解析 -->* <bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver">  <property name="prefix" value="/WEB-INF/pages/" />  <property name="suffix" value=".jsp" /> </bean>  *<!-- 这两个类用来启动基于Spring MVC的注解功能，将控制器与方法映射加入到容器中 -->* <bean class="org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping" /> <bean class="org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter" />  </beans> |

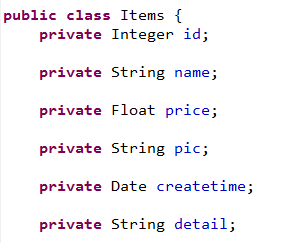
### 参数绑定

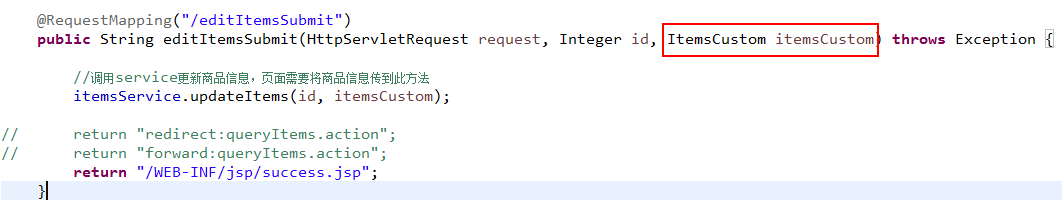
springMVC有支持的默认类型的绑定，也就是说，直接在controller方法形参上定义默认数据的对象，就可以使用这些对象。

1. HttpServletRequest对象
2. HttpServletResponse对象
3. HttpSession对象
4. Model/ModelMap对象

### POJO类型的绑定

创建实体类





### 自定义转换类

|  |
| --- |
| *//需要实现Converter接口，这里是将String类型转换成Date类型* public class CustomDateConverter implements Converter<String,Date> {  *@Override* public Date convert(String source) {  *//实现 将日期串转成日期类型(格式是yyyy-MM-dd HH:mm:ss)* SimpleDateFormat simpleDateFormat = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");  try {  *//直接返回结果* return simpleDateFormat.parse(source);  }catch (Exception e){  e.printStackTrace();  }  *//如果参数绑定失败返回null* return null;  } } |

|  |
| --- |
| <mvc:annotation-driven conversion-service="conversionService"/>  *<!--自定义参数绑定-->* <bean id="conversionService" class="org.springframework.format.support.FormattingConversionServiceFactoryBean">  *<!--转换器-->* <property name="converters">  <list>  *<!--日期类型转换-->* <bean class="com.vacomall.common.CustomDateConverter"></bean>  </list>  </property> </bean> |

### 包装POJO类

包装类



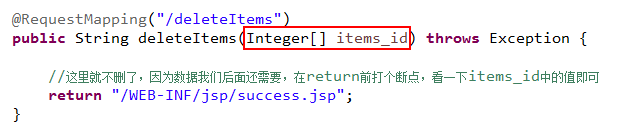
前台指定

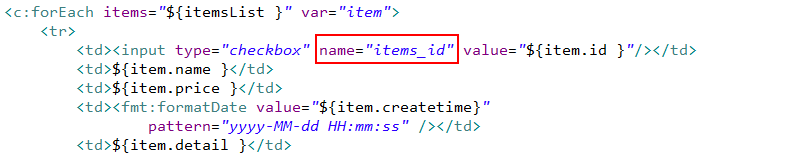
IMG_256

controller

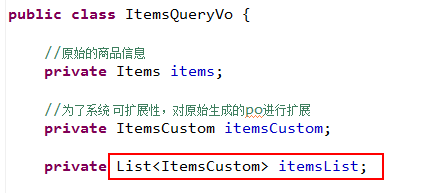


### 集合类绑定

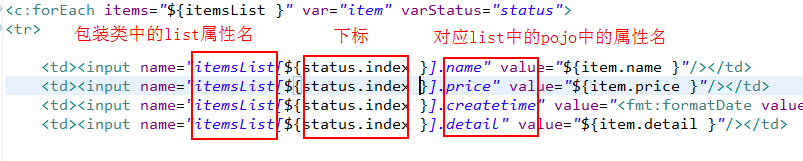




### List绑定







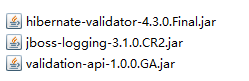
### Map绑定

|  |
| --- |
| Public class QueryVo {  private Map<String, Student> itemInfo = new HashMap<String, Student>(); //get/set方法..  } |

|  |
| --- |
| <tr> <td>学生信息：</td>  <td>  姓名：<input type="text" name="itemInfo['name']"/>  年龄：<input type="text" name="itemInfo['price']"/> .. .. ..  </td>  </tr> |

### 数据校验

springMVC中我们使用hibernate的校验框架validation（注：和hibernate没有任何关系），使用这个框架的话需要导入jar包

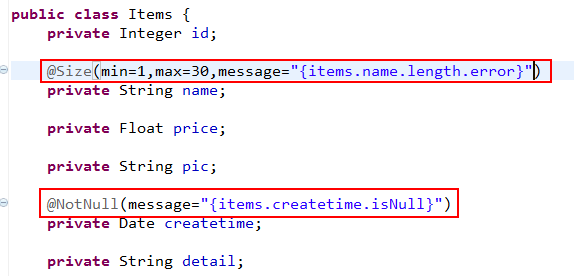


配置校验器

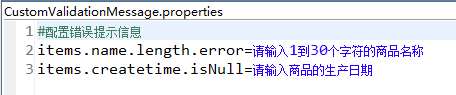
|  |
| --- |
| *<!-- 配置校验器 -->* <bean id="validator" class="org.springframework.validation.beanvalidation.LocalValidatorFactoryBean">  *<!-- 校验器，使用hibernate校验器 -->* <property name="providerClass" value="org.hibernate.validator.HibernateValidator"/>  *<!-- 指定校验使用的资源文件，在文件中配置校验错误信息，如果不指定则默认使用classpath下面的ValidationMessages.properties文件 -->* <property name="validationMessageSource" ref="messageSource"/> </bean> *<!-- 校验错误信息配置文件 -->* <bean id="messageSource" class="org.springframework.context.support.ReloadableResourceBundleMessageSource">  *<!-- 资源文件名 -->* <property name="basenames">  <list>  <value>classpath:CustomValidationMessage</value>  </list>  </property>  *<!-- 资源文件编码格式 -->* <property name="fileEncodings" value="utf-8"/>  *<!-- 对资源文件内容缓存时间，单位秒 -->* <property name="cacheSeconds" value="120"/> </bean> |

|  |
| --- |
| <mvc:annotation-driven conversion-service="conversionService" validator="validator"/> |

| 注解 | 运行时检查 |
| --- | --- |
| @AssertFalse | 被注解的元素必须为false |
| @AssertTrue | 被注解的元素必须为true |
| @DecimalMax(value) | 被注解的元素必须为一个数字，其值必须小于等于指定的最小值 |
| @DecimalMin(Value) | 被注解的元素必须为一个数字，其值必须大于等于指定的最小值 |
| @Digits(integer=, fraction=) | 被注解的元素必须为一个数字，其值必须在可接受的范围内 |
| @Future | 被注解的元素必须是日期，检查给定的日期是否比现在晚 |
| @Max(value) | 被注解的元素必须为一个数字，其值必须小于等于指定的最小值 |
| @Min(value) | 被注解的元素必须为一个数字，其值必须大于等于指定的最小值 |
| @NotNull | 被注解的元素必须不为null |
| @Null | 被注解的元素必须为null |
| @Past(java.util.Date/Calendar) | 被注解的元素必须过去的日期，检查标注对象中的值表示的日期比当前早 |
| @Pattern(regex=, flag=) | 被注解的元素必须符合正则表达式，检查该字符串是否能够在match指定的情况下被regex定义的正则表达式匹配 |
| @Size(min=, max=) | 被注解的元素必须在制定的范围(数据类型:String, Collection, Map and arrays) |
| @Valid | 递归的对关联对象进行校验, 如果关联对象是个集合或者数组, 那么对其中的元素进行递归校验,如果是一个map,则对其中的值部分进行校验 |
| @CreditCardNumber | 对信用卡号进行一个大致的验证 |
| @Email | 被注释的元素必须是电子邮箱地址 |
| @Length(min=, max=) | 被注解的对象必须是字符串的大小必须在制定的范围内 |
| @NotBlank | 被注解的对象必须为字符串，不能为空，检查时会将空格忽略 |
| @NotEmpty | 被注释的对象必须为空(数据:String,Collection,Map,arrays) |
| @Range(min=, max=) | 被注释的元素必须在合适的范围内 (数据：BigDecimal, BigInteger, String, byte, short, int, long and 原始类型的包装类 ) |
| @URL(protocol=, host=, port=, regexp=, flags=) | 被注解的对象必须是字符串，检查是否是一个有效的URL，如果提供了protocol，host等，则该URL还需满足提供的条件 |



我们看到注解中可以指定message，那么这个message中的内容是错误消息配置文件中对应的key，取出来的就是对应的错误消息，所以针对这两个错误消息，我们写一下配置文件：



捕获校验错误信息

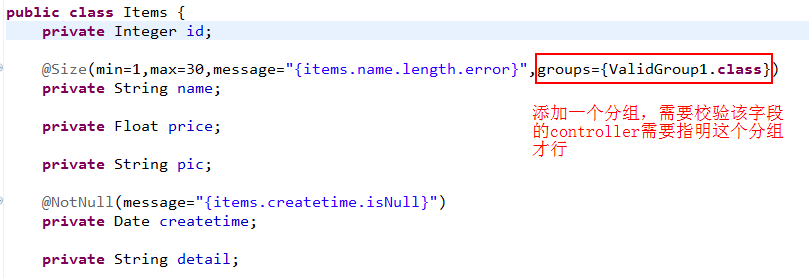


分组检验

1. 首先定义一个检验分组

|  |
| --- |
| public interface ValidGroup1 {  //接口中不需要定义任何方法，仅仅是对不同的校验规则进行分组 //此分组只校验商品名称的长度  } |

1. 然后我们在刚刚的pojo中添加这个分组



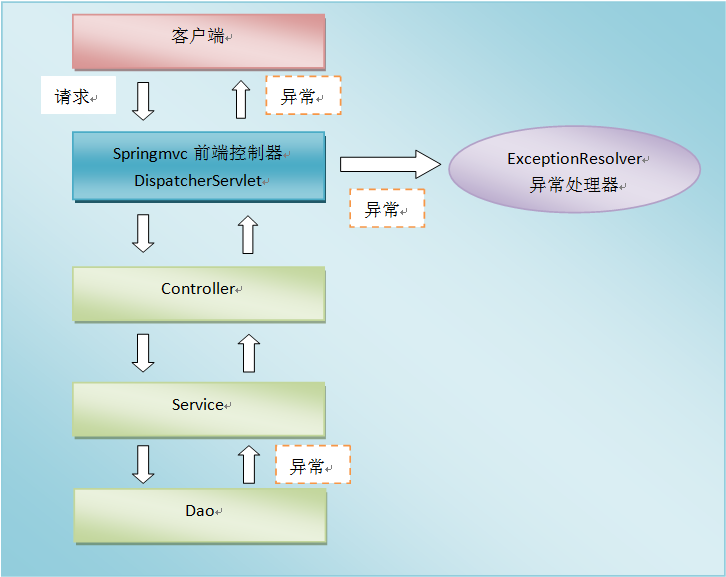
1. 再看一下想要检验这个name字段的controller中使如何配置的



这样该controller就不会去检验生产日期的字段了，通过这种方式可以解决不同的controller检验不同的字段问题。

### 统一异常处理

springMVC中，异常处理的思路



SpringMVC自带的简单异常处理器

SimpleMappingExceptionResolver改处理器实现了HandlerExceptionResolver接口，全局异常处理器都需要实现该接口，我们使用这个自带的异常处理器

1. springmvc.xml文件中配置该处理器:

|  |
| --- |
| *<!-- springmvc提供的简单异常处理器 -->* <bean class="org.springframework.web.servlet.handler.SimpleMappingExceptionResolver">  *<!-- 定义默认的异常处理页面 -->* <property name="defaultErrorView" value="/WEB-INF/jsp/error.jsp"/>  *<!-- 定义异常处理页面用来获取异常信息的变量名，也可不定义，默认名为exception -->* <property name="exceptionAttribute" value="ex"/>  *<!-- 定义需要特殊处理的异常，这是重要点 -->* <property name="exceptionMappings">  <props>  <prop key="com.vacomall.common.CustomException">/WEB-INF/jsp/custom\_error.jsp</prop>  </props>  *<!-- 还可以定义其他的自定义异常 -->* </property> </bean> |

|  |
| --- |
| public class CustomException extends Exception {   *//异常信息* public String message;   public CustomException(String message) {  super(message);  this.message = message;  }   public String getMessage() {  return message;  }   public void setMessage(String message) {  this.message = message;  }   } |



### 自定义全局异常处理器

全局异常处理器处理思路:

1. 解析出异常类型
2. 如果该异常类型是系统自定义的异常，直接取出异常信息，在错误页面展示
3. 如果异常类型不是系统自定义的异常，构造一个自定义的异常类型（信息为：”未知错误”）

springMVC提供一个HandlerExceptionResolver接口，自定义全局异常处理器必须要实现这个接口，如下：

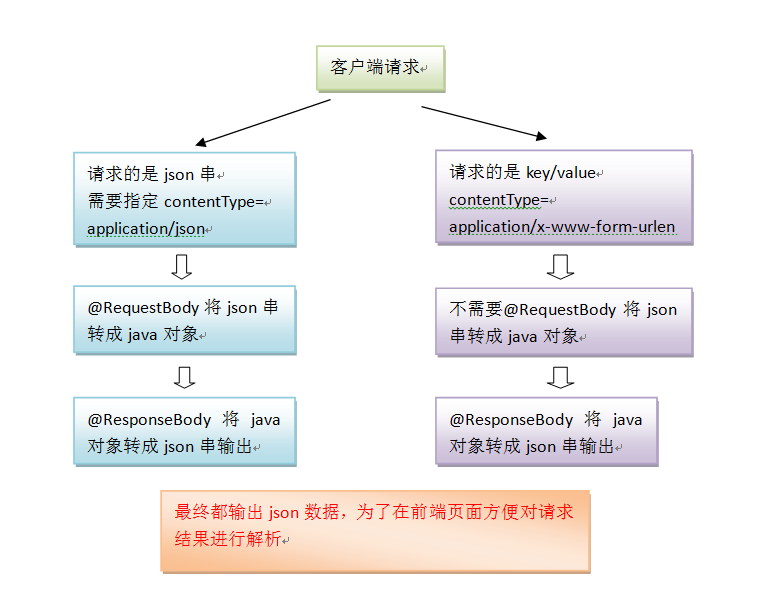
|  |
| --- |
| public class CustomExceptionResolver implements HandlerExceptionResolver {   *@Override* public ModelAndView resolveException(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex) {   ex.printStackTrace();  CustomException customException = null;   *//如果抛出的是系统自定义的异常则直接转换* if(ex instanceof CustomException) {  customException = (CustomException) ex;  } else {  *//如果抛出的不是系统自定义的异常则重新构造一个未知错误异常  //这里我就也有CustomException省事了，实际中应该要再定义一个新的异常* customException = new CustomException("系统未知错误");  }   return null;  } } |

|  |
| --- |
| *<!-- 自定义的全局异常处理器 只要实现HandlerExceptionResolver接口就是全局异常处理器-->* <bean class="com.vacomall.common.CustomExceptionResolver"/> |

### SpringMCV与前台的JSON数据交互

json数据格式在接口调用中、html页面中比较长用，json格式比较简单，解析也比较方便，所有使用很普遍，在springMVC中，也支持对json数据的解析和转换。

两种交互形式



加载json的jar包

IMG_256

配置json转换器

配置json转换器有两种方式，如果是配置了注解适配器org.springframework.web.servlet.mvc.annotation.AnnotationMehodHandlerAdapter的话，需要在该适配器中配置json转换器

|  |
| --- |
| *<!-- 用于将对象转换为 JSON -->* <bean id="stringConverter" class="org.springframework.http.converter.StringHttpMessageConverter">  <property name="supportedMediaTypes">  <list>  <value>text/plain;charset=UTF-8</value>  </list>  </property> </bean> <bean id="jsonConverter" class="org.springframework.http.converter.json.MappingJackson2HttpMessageConverter"/>  <bean class="org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter">  <property name="messageConverters">  <list>  <ref bean="stringConverter" />  <ref bean="jsonConverter" />  </list>  </property> </bean> |

但是如果使用

<mvc:annotation-driven/>

注解驱动的话就不用以上配置了，默认已经配好了，建议使用这种，比较方便

json交互的测试

### ApplicationContextAware

### ApplicationListener

在使用过程中可以监听某一事件的发生,可以做出相应的处理,这个方式不常用,但是在特殊情况下还是有用的。

在一些业务场景中,当容器初始化完成之后,需要处理一些操作,比如一些数据的加载,初始化缓存,特定任务的注册等等,这个时候我们可以使用Spring提供的ApplcationListener来操作。

:

### 文件上传

第一步：

引入核心jar文件

springmvc实现文件上传，需要添加两个jar包，一个是文件上传的jar包，一个是其所依赖的IO包，这两个jar包，均在Spring支持库的org.apache.commons中

IMG_256

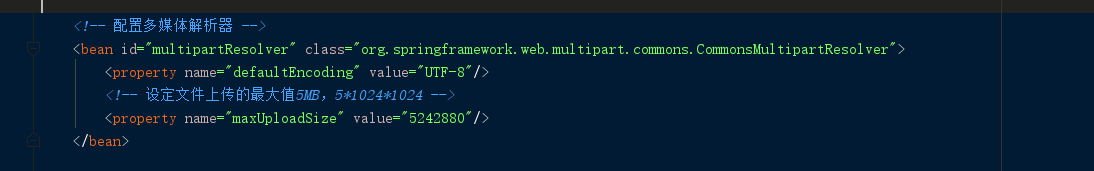
第二步：

编写控制器方法



注：必须创建MultipartFile实例，要不出现500错误





|  |
| --- |
| <body>     <form action="${pageContext.request.contextPath }/first.do" method="post" enctype="multipart/form-data">     <h2>文件上传</h2>                  文件:<input type="file" name="uploadFile"/><br/><br/>        <input type="submit" value="上传"/>     </form>   </body> |

二：多文件上传



三：文件下载



# 7.spring

## aop

什么是AOP以及实现方式

AOP:Aspect Oriented Programming的缩写,意为:面向切面编程,通过预编译方式和运行期动态代理实现程序功能的统一维护的一种技术

主要的功能是:日志记录,性能统计,安全控制,事务处理,异常处理等等。

预编译

-Aspectj

运行期动态代理（JDK动态代理、CGLib动态代理）

-SpringAop 、JbossAOP

AOP基本概念

|  |  |
| --- | --- |
| 名称 | 说明 |
| 切面（Aspect） | 一个关注点的模块化,这个关注点可能会横切多个对象 |
| 连接点（Joinpoint） | 程序执行过程中的某个特定的点 |
| 通知（Advice） | 在切面的某个特定的连接点执行的动作 |
| 切入点（Pointcut） | 匹配连接点的断言,在AOP中通知和一个切入点表达式关联 |
| 引入（Introduction） | 在不修改类代码的前提下,为类添加新的方法和属性 |
| 目标对象（Target Object） | 被一个或者多个切面所通知的对象 |
| AOP代理（AOP Proxy） | AOP框架创建的对象,用来实现切面契约（aspect contract）（包括通知方法执行等功能） |
| 织入（Weaving） | 把切面连接到其它的应用程序类型或者对象上,并创建一个被通知的对象,分为:编译时织入、类加载是织入、执行时织入 |

Advice的类型

|  |  |
| --- | --- |
| 名称 | 说明 |
| 前置通知（Before advice） | 在某连接点（join point）之前执行的通知,但不能阻止连接点前的执行（除非它抛出一个异常） |
| 返回后通知（After returning advice） | 在某连接点（join point）正常完成后执行的通知 |
| 抛出异常后通知（After throwing advice） | 在方法抛出异常退出时执行的通知 |
| 后通知（After（finally）advice） | 当某连接点退出的时候执行的通知（不论是正常返回还是异常退出） |
| 环绕通知（Around Advice） | 包围一个连接点（join point）的通知 |

有接口和无接口的Spirng AOP实现区别

1. Spring AOP默认使用标准的JavaSE动态代理作为AOP代理,这使得任何接口（或者接口集）都可以被代理
2. Spring AOP中也可以使用CGLIB代理（如果一个业务对象并没有实现一个接口）

Schema-based AOP（基于配置的切面）

Spring所有的切面和通知器都必须放在一个<aop:config>内（可以配置多个<aop:config>元素）,每个<aop:config>可以包含pointcut,advisor和aspect元素

（它们必须按照这个顺序进行声明）

<aop:config>风格的配置大量使用了Spring的自动代理机制

声明方式:

|  |
| --- |
| <aop:config>  <aop:aspect>  <aop:config>  <aop:aspect id=”myAspect” ref=”aBean”> //ref就是切面类  ...  </aop:aspect>  </aop:config>  <bean id=”aBean” class=”...”>  ..  </bean> |

|  |
| --- |
| <bean id=”moocAspect” class=”..MoocAspect”> //切面类  <bean id=”aspectBiz” class=”..AspectBiz”> //要增强的业务类  <aop:config>  <aop:aspect id=”moocAspectAOP” ref=”moocAspect”>  </aop:aspect>  </aop:config> |

pointcut（execution用于匹配方法执行的连接点）

|  |
| --- |
| 任意公共方法的执行：  execution(public \* \*(..))  任何一个以“set”开始的方法的执行：  execution(\* set\*(..)) AccountService  接口的任意方法的执行：  execution(\* com.xyz.service.AccountService.\*(..))  定义在service包里的任意方法的执行：  execution(\* com.xyz.service.\*.\*(..))  定义在service包和所有子包里的任意类的任意方法的执行：  execution(\* com.xyz.service..\*.\*(..))  定义在pointcutexp包和所有子包里的JoinPointObjP2类的任意方法的执行：execution(\* com.test.spring.aop.pointcutexp..JoinPointObjP2.\*(..))")  \*\*\*> 最靠近(..)的为方法名,靠近.\*(..))的为类名或者接口名,如上例的JoinPointObjP2.\*(..))  pointcutexp包里的任意类  within(com.test.spring.aop.pointcutexp.\*)  pointcutexp包和所有子包里的任意类.  within(com.test.spring.aop.pointcutexp..\*)  实现了MyInterface接口的所有类,如果MyInterface不是接口,限定MyInterface单个类.  this(com.test.spring.aop.pointcutexp.MyInterface)  \*\*\*>当一个实现了接口的类被AOP的时候,用getBean方法必须cast为接口类型,不能为该类的类型. |

|  |
| --- |
| <aop:config>  <aop:aspect id=”” ref=””>  <aop:pointcut id=”businessService” expression=”..”>  <aop:pointcutid>  </aop:config> |

Spring AOP API

AspectJ

## ioc

# 8.swagger2接口管理工具

使用swagger2构建restfule接口

首页/swagger

说明:

可以生成文档形式的api并提供给不同的团队

使用自测,也便于领导查阅任务量

无需过多冗余的word文档

配置maven

|  |
| --- |
| *<!-- swagger2 配置 -->* <dependency>  <groupId>io.springfox</groupId>  <artifactId>springfox-swagger2</artifactId>  <version>2.4.0</version> </dependency> <dependency>  <groupId>io.springfox</groupId>  <artifactId>springfox-swagger-ui</artifactId>  <version>2.4.0</version> </dependency> |

# 9.mybatis

# 10.SSM整合

|  |
| --- |
|  |

# 11.springboot

## SpringBoot和SpringMVC的关系

两者没有必然的关系，是springmvc的升级版本。

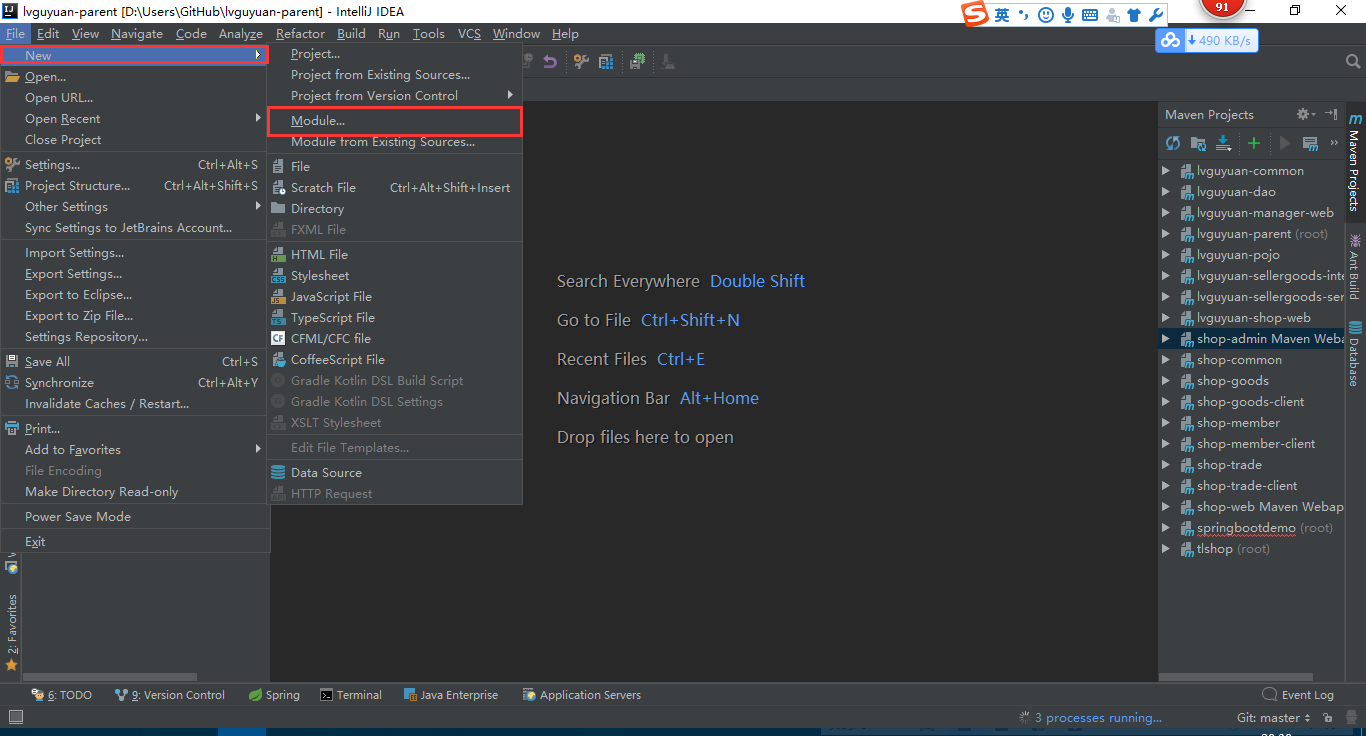
## springBoot的特点

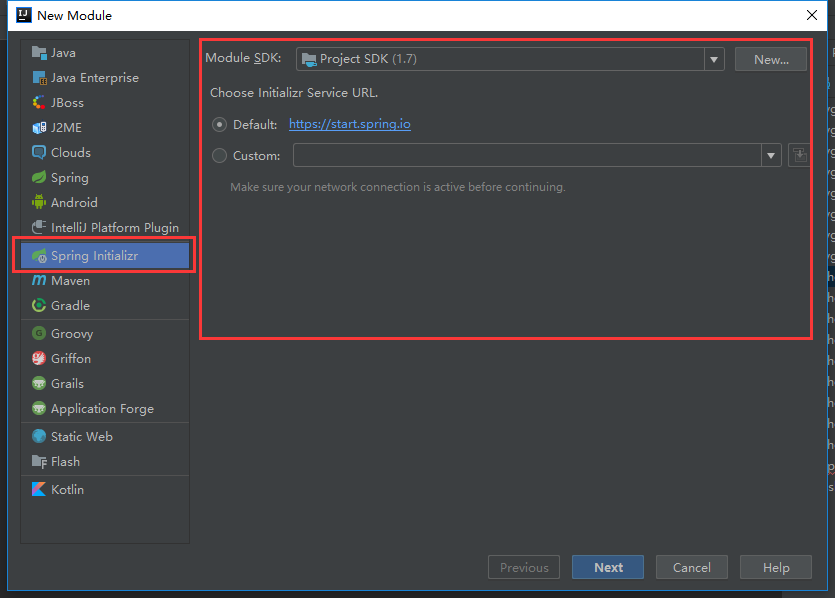
1. 化繁为简，简化配置
2. 备受关注，是下一代框架。
3. 入门级微服务框架。

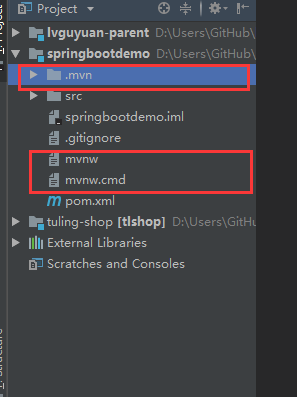
微服务 > SpringCloud > SpringBoot

课程目录

## 第一个SpringBoot程序





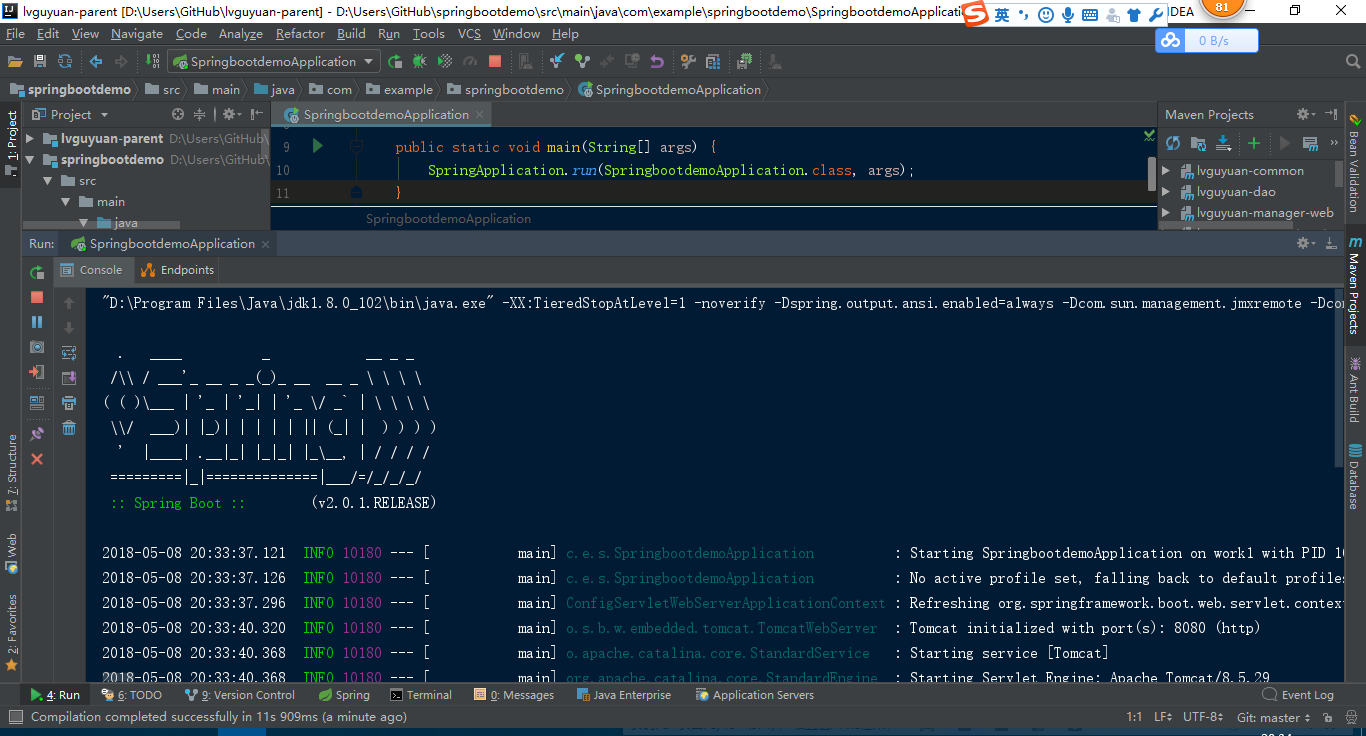


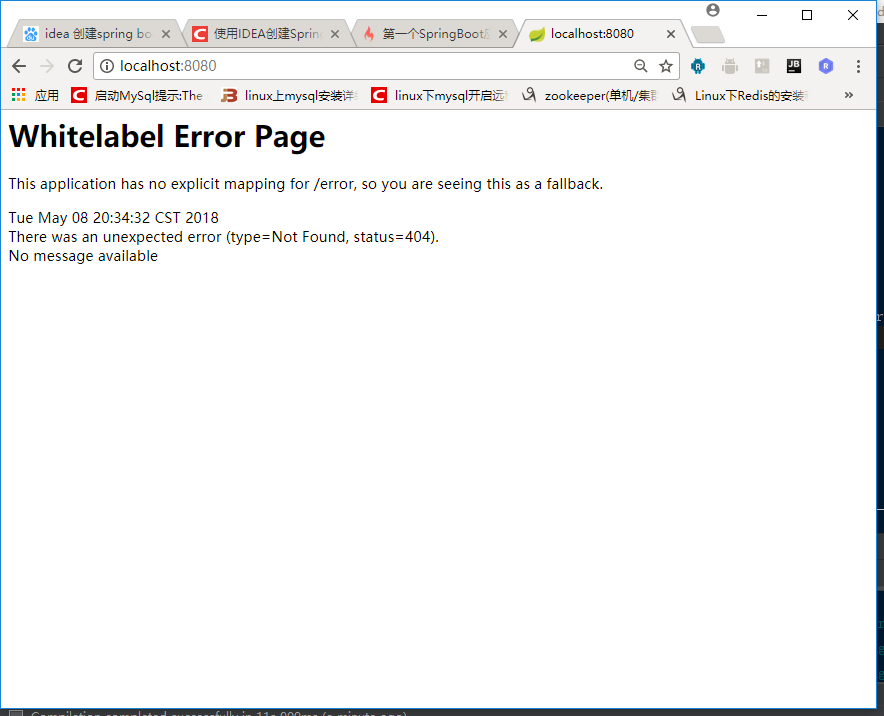
删除勾选



springboot启动入口：使用注解

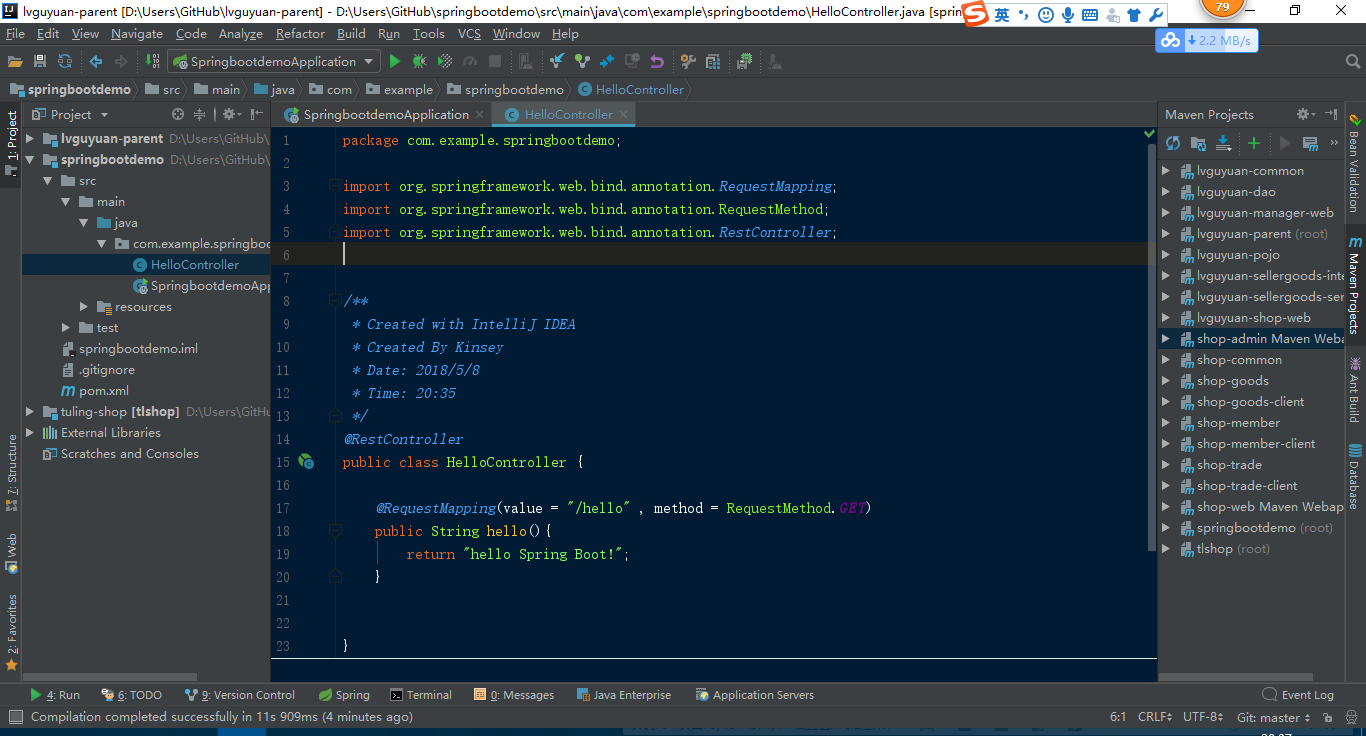
@SpringBootApplication





启动成功

创建一个Controller

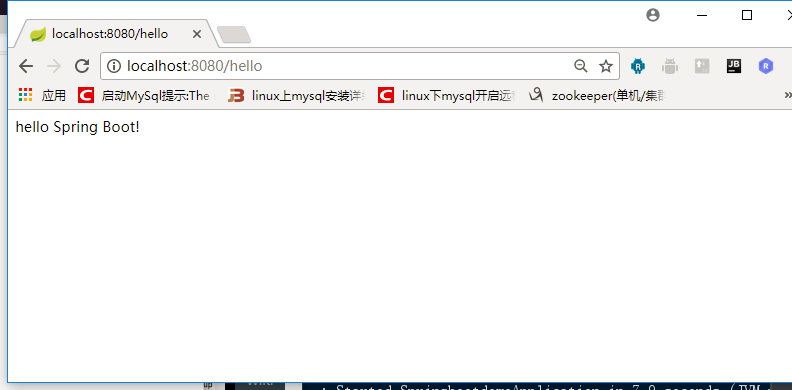


@SpringBootApplication开启了Spring的组件扫描和springboot的自动配置功能，相当于将以下三个注解组合在了一起

（1）@Configuration：表名该类使用基于Java的配置,将此类作为配置类

（2）@ComponentScan：启用注解扫描

（3）@EnableAutoConfiguration：开启springboot的自动配置功能



第二种启动方式:

进入项目的目录下

运行:mvn spring-boot:run 命令来启动

第三种启动方式:

1. 先编译项目：mvn install
2. 进入taget目录下
3. 输入 java -jar 项目包-0.0.1-SNAPSHOT.jar

### 自定义属性配置

属性配置：

|  |
| --- |
| spring.datasource.url : jdbc:mysql://127.0.0.1:3306/  spring.datasource.username : root  spring.datasource.password : 123456  spring.datasource.driver-class-name : com.mysql.jdbc |

项目中resources下的配置application.properties，这个文件是springboot的配置文件

server.port:8080 配置端口号

server.context-path=/springboot 配置项目路径

|  |
| --- |
| server.port=8080 server.servlet.context-path=/springboot |

也可以使用application.yml的格式配置文件

|  |
| --- |
| server:  port: 8080  servlet:  context-path: /springboot |

注解配置文件

配置文件: age:18

@Value(“${age}”)

private Integer age;

使用实体类读取配置文件

使用类注解：

@ConfigurationProperties(prefix = “springboot”)

prefix 配置文件中前缀为什么的

如:springboot.age

通过注解bin的方式进行加载

@Component

使用时使用@Autowired进行注入

springboot多环境配置文件

默认配置环境：application-yml

|  |
| --- |
| spring:  profiles:  active:dev //表示使用dev环境配置 |

开发环境：application-dev.yml

生产环境：applicaiotn-prod.vml

启动方式中进行环境切换

java -jar target/项目名.jar --spring.profiles.active=prod

### Controller的使用

|  |  |  |
| --- | --- | --- |
| @Controller | 处理http请求 |  |
| @RestController | Spring4之后新加的注解，原来返回json需要@ResponseBody配合@Controller |  |
| @RequestMapping | 配置url映射 |  |
| @PathVariable | 获取url中的数据  locahost:8080/sat/100 | @ReuestMapping(“/sat/{id}”)  public String sat(@PathVariabl(“id”) Integer id){ } |
| @RequestParam | 获取请求参数的值  localhost:8080/sat?id=100 | @RequestMapping(“/sat”)  public Sting sat (@ReqeuestParam(“id”) Integer id){} |
| @GetMapping | 组合注解 |  |

### spring-data-jpa

JAP(java Persistence API)定义了一系列对象持久化的标准，目前实现这一规范的产品有Hibernate、Toplink等。

### 事务管理

## springboot进阶之web进阶

web进阶：使用@valid表单验证,使用AOP处理请求，统一异常处理，单元测试

### 表单验证

@RequestMapping()

public string save(@valid 封装类){}

配置JSR303定义的校验类型

|  |
| --- |
| 空检查  @Null 验证对象是否为null  @NotNull 验证对象是否不为null, 无法查检长度为0的字符串  @NotBlank 检查约束字符串是不是Null还有被Trim的长度是否大于0,只对字符串,且会去掉前后空格.  @NotEmpty 检查约束元素是否为NULL或者是EMPTY.    Booelan检查  @AssertTrue 验证 Boolean 对象是否为 true  @AssertFalse 验证 Boolean 对象是否为 false    长度检查  @Size(min=, max=) 验证对象（Array,Collection,Map,String）长度是否在给定的范围之内  @Length(min=, max=) Validates that the annotated string is between min and max included.    日期检查  @Past 验证 Date 和 Calendar 对象是否在当前时间之前  @Future 验证 Date 和 Calendar 对象是否在当前时间之后  @Pattern 验证 String 对象是否符合正则表达式的规则    数值检查，建议使用在Stirng,Integer类型，不建议使用在int类型上，因为表单值为“”时无法转换为int，但可以转换为Stirng为"",Integer为null  @Min 验证 Number 和 String 对象是否大等于指定的值  @Max 验证 Number 和 String 对象是否小等于指定的值  @DecimalMax 被标注的值必须不大于约束中指定的最大值. 这个约束的参数是一个通过BigDecimal定义的最大值的字符串表示.小数存在精度  @DecimalMin 被标注的值必须不小于约束中指定的最小值. 这个约束的参数是一个通过BigDecimal定义的最小值的字符串表示.小数存在精度  @Digits 验证 Number 和 String 的构成是否合法  @Digits(integer=,fraction=) 验证字符串是否是符合指定格式的数字，interger指定整数精度，fraction指定小数精度。    @Range(min=, max=) Checks whether the annotated value lies between (inclusive) the specified minimum and maximum.  @Range(min=10000,max=50000,message="range.bean.wage")  private BigDecimal wage;    @Valid 递归的对关联对象进行校验, 如果关联对象是个集合或者数组,那么对其中的元素进行递归校验,如果是一个map,则对其中的值部分进行校验.(是否进行递归验证)  @CreditCardNumber信用卡验证  @Email 验证是否是邮件地址，如果为null,不进行验证，算通过验证。  @ScriptAssert(lang= ,script=, alias=)  @URL(protocol=,host=, port=,regexp=, flags=) |

AOP统一处理请求日志

AOP是一种编程范式

与语言无关，是一种程序设计思想

面向切面（AOP）Aspect Oriented Programming

面向对象（OOP）Object Oriented Programming

面向过程到面向对象

换个角度看世界，换个姿态处理问题

将通用逻辑从业务逻辑中分离出来

@aspect使用类注解定义切面类

@Pointcut(表达式)统一切点

@Before 前置通知

@AfterReturning后置通知

@After最终通知

@AfterThrowing例外通知

@Around循环通知

execution表达式:execution(\* com.sample.service.impl..\*.\*(..))

|  |  |
| --- | --- |
| 符号 | 含义 |
| execution() | 表达式的主体; |
| 第一个“\*”符号 | 表示返回值得数据类型； |
| com.sample.service.impl | AOP所切的服务的包名，即，我们的业务部分 |
| 包后面的”..” | 表示当前包及子包 |
| 第二个”\*” | 表示类名，\*即所有的类。此处可以自定义，下文有举例 |
| .\*(..) | 表示任何方法名，括号表示参数，两个点表示任何参数类型 |

RequestContextHolder持有上下文的Request容器

|  |
| --- |
|  |

获取返回内容

@AfterReturning(returning=”object”,pointcut=”log()”)

public void doAfterReturning(Object object){

logger.info(“response={}”,object);

}

### 统一异常处理

为什么要统一异常处理？

1. 响应格式统一

创建Result响应类

|  |
| --- |
| //http请求返回的最外层对象  public class Result<T>{  //错误编码  private Integer code;  //提示信息  private String msg;  //具体的内容  private T data;  } |

代码优化

创建ResultUtil

|  |
| --- |
| public class ResultUtil{    //成功  public static Result success(Object object){  Result result = new Result();  result.setCode(0);  result.setMsg(“成功”);  result.setData(object);  return result;  }  //成功对象为null  public static Result success(){  return success(null);  }  //失败  public static Result error(Integer code,String msg){  Result result = new Result();  result.setCode(code);  result.setMsg(msg);  return result;  }  } |

统一异常处理中

|  |
| --- |
| public void getAge(Integer id){  Girl girl = girlRepository.findOne(id);  Integer age = girl.getAge();  if(age < 10){  throw new Exception(“还在上小学吧”);  }  } |

通过handle进行捕获异常

|  |
| --- |
| @ControllerAdvice  public class ExceptionHandle{    @ExceptionHandler(value = Exception.class)  @ResponseBody  public Result handle(Exception e){  if(e instanceof GirlException ){  GirlException girlException = (GirlException)e;  return ResultUtil.error(girlException.getCode,gir;Exception.getMessage());  }else{  return ResultUtil.error(100,”请联系管理员”);  }  }  } |

自定义Exception

|  |
| --- |
| public class GirlException extends RuntiomExeception{  private Integer code;    public GirlException(Integer code,String message){  super(message);  this.code = code;  }  } |

code统一管理

ResultEnum

|  |
| --- |
| public enum ResultEnum{  UNKONW\_ERROR(-1,”位置错误”);  success(0,”成功”);  private Integer code;  private String getMsg;  } |

### 单元测试

# 12.shiro配置

第一步配置xml

|  |
| --- |
| <filter>  <filter-name>shiroFilter</filter-name>  <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>  <init-param>  <param-name>targetFilterLifecycle</param-name>  <param-value>true</param-value>  </init-param> </filter> <filter-mapping>  <filter-name>shiroFilter</filter-name>  <url-pattern>/\*</url-pattern> </filter-mapping> |

# 13.项目

### ThreadLocal<>

多线程保存线程的方式，保存当前线程的数据

### 页面优化技术

#### 页面缓存+URL缓存+对象缓存

@RequestMapping(value=”/to\_list”,produces=”text/html”)

可以直接返回html

1. 取缓存

使用Thymeleaf渲染

将html页面存储到redis中。

1. 手动渲染模板
2. 结果输出

#### 页面静态化,前后端分离

#### 静态资源优化

#### CND优化

超卖解决：

1. 数据库添加唯一索引:防止用户重复购买
2. sql加库存数量判断:防止库存变成负数

### 秒杀接口优化MQ

RabbitMQ-4种交换机模式

Direct模式 交换机Exchange

### InitializingBean

接口中实现afterPropertiestSet方法即可执行响应任务代码

### valuator

实体类验证工具

|  |
| --- |
| <dependency>  <groupId>javax.validation</groupId>  <artifactId>validation-api</artifactId>  <version>1.1.0.Final</version> </dependency> <dependency>  <groupId>org.hibernate</groupId>  <artifactId>hibernate-validator</artifactId>  <version>5.2.4.Final</version> </dependency>  *@Getter @Setter* public class AclModuleParam {  *@NotBlank*(message = "权限模块名称不可以为空")  *@Length*(min = 2,max = 20,message = "权限模块内容2至20")  private String name;   *@NotNull*(message = "权限模块展示顺序不可以为空")  private Integer seq;   *@NotNull*(message = "权限状态不可以为空")  *@Min*(value = 0,message = "权限模块状态不合法")  *@Max*(value = 1,message = "权限模块状态不合法")  private Integer status; } |
| */\*\*  \* 表单校验工具类  \* Created with IntelliJ IDEA  \* Created By 田伟硕  \* Date: 2018/5/24  \* Time: 15:54  \*/* public class BeanValidator {  *//validator工厂* private static ValidatorFactory validatorFactory = Validation.*buildDefaultValidatorFactory*();   private static <T> Map<String,String> validate(T t,Class... groups){  Validator validator = validatorFactory.getValidator();  Set validateResult = validator.validate(t,groups);  if(validateResult.isEmpty()){  return Collections.*emptyMap*();  }else{  LinkedHashMap errors = Maps.*newLinkedHashMap*();  Iterator iterator = validateResult.iterator();  while (iterator.hasNext()){  ConstraintViolation violation = (ConstraintViolation)iterator.next();  errors.put(violation.getPropertyPath().toString(),violation.getMessage());  }  return errors;  }  }   public static Map<String,String> validateList(Collection<?> collection){  Preconditions.*checkNotNull*(collection);  Iterator iterator = collection.iterator();  Map errors = null;  do{  if(!iterator.hasNext()){  return Collections.*emptyMap*();  }  Object object = iterator.next();  errors = *validate*(object,new Class[0]);  }while (iterator.hasNext());  return errors;  }   public static Map<String,String> validateObject(Object first,Object... objects) {  if(objects != null && objects.length > 0){  return *validateList*(Lists.*asList*(first,objects));  }else{  return *validate*(first,new Class[0]);  }  }   *//检查参数* public static void ceck(Object param)throws ParamException{  Map<String, String> map = BeanValidator.*validateObject*(param);  if(MapUtils.*isNotEmpty*(map)){  throw new ParamException(map.toString());  }  }  } |

### lombok

提供了注解方式的getset方法

|  |
| --- |
| *@Getter 生成get @Setter 生成set*  *@Builder 使用builer构建 @NoArgsConstructor 空参构造方法 @AllArgsConstructor 实参构造方法*  *@ToString*  <dependency>  <groupId>org.projectlombok</groupId>  <artifactId>lombok</artifactId>  <version>1.16.12</version> </dependency> |

### 日志系统

SLF4J+Logback

日志配置

区分info和error日志

每天产生一个日志文件

1. application.yml

|  |
| --- |
| *<?*xml version="1.0" encoding="UTF-8" *?>* <configuration>   *<!--配置控制台-->* <appender name="consoleLog" class="ch.qos.logback.core.ConsoleAppender">  <layout class="ch.qos.logback.classic.PatternLayout">  <pattern>*<!--打印格式-->* %d - %msg%n  </pattern>  </layout>  </appender>   *<!--配置文件输入-->* <appender name="fileInfoLog" class="ch.qos.logback.core.rolling.RollingFileAppender">  <filter class="ch.qos.logback.classic.filter.LevelFilter">  <level>EROOR</level>  <onMathc>DENY</onMathc>  <onMismatch>ACCEPT</onMismatch>  </filter>  <encoder>  <pattern>  %msg%n  </pattern>  </encoder>  *<!--滚动策略-->* <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">  *<!--路径-->* <fileNamePattern>  D:\info.%d.log  </fileNamePattern>  </rollingPolicy>  </appender>   <appender name="fileErrorLog" class="ch.qos.logback.core.rolling.RollingFileAppender">  <filter class="ch.qos.logback.classic.filter.ThresholdFilter">  <level>EROOR</level>  </filter>  <encoder>  <pattern>  %msg%n  </pattern>  </encoder>  *<!--滚动策略-->* <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">  *<!--路径-->* <fileNamePattern>  D:\error.%d.log  </fileNamePattern>  </rollingPolicy>  </appender>   <root level="info">  <appender-ref ref="consoleLog"/>  <appender-ref ref="fileInfoLog"/>  <appender-ref ref="fileErrorLog"/>  </root>   </configuration> |

# 14.serialize序列化

# 15.Redis

## 1.NoSQL的概述

什么是NoSql: NoSQL = Not Only SQL

非关系型的数据库

为什么需要NoSQL

High performance -高并发读写

Huge Storage -海量数据的高效率存储和访问

High Scalability && High Availability-高可扩展性和高可用性

NoSQL数据库的四大分类

1. 键值（Key-Value）存储
2. 列存储
3. 文档数据库
4. 图形数据库

NoSQL的特点

1. 易扩展
2. 大数据量,高性能
3. 灵活的数据模型
4. 高可用

Redis的概述

高性能键值对数据库,支持的键值数据类型

1. 字符串类型
2. 列表类型
3. 有序集合类型
4. 散列类型
5. 集合类型

Redis的应用场景

1. 缓存（数据的查询，商品的信息内容）
2. 任务队列
3. 网站访问的统计
4. 数据过期处理
5. 应用排行榜
6. 分布式集群架构中的session分离

Redis的安装和使用

Jedis的入门

普通方式

Jedis jedis = new Jedis(pi,端口);

jedis.set(名称,值);

连接池方式

//获取连接池配置对象;

JedisPoolConfig config = new JedisPooleConfig();

//设置最大的连接数

config.setMaxTotal(30);

//设置最大的空闲连接数

config.setMaxIdle(10);

//获得连接池

JedisPool jedisPoole = new JedisPool(config,ip,端口);

//获得核心对象

Jedis jedis = null;

try{

//通过连接池设置连接

jedis = jedisPool.getResource();

}catch(Exceotion e){

e.printStackTrace();

}finally{

}

Reids的数据类型

Redis的数据结构

value最大存储的长度是512兆

五种数据类型

1. 字符串（String）
   1. 二进制安全的,存入和获取的数据相同
   2. value最大可以容纳的数据长度是512M
   3. 常用命令
      1. 赋值

set key val（set company imooc）

* + 1. 删除

del key （del company）

* + 1. 扩展

incrby key value（将指定的value增几）

decrby key value（将指定的value减几）

append key value（字符串拼接追加）

* + 1. 取值

get key （get company）

getset key val （getset company imooc 先获取在修改）

* + 1. 数据增减

incr key （将指定key的value增1,如果值不在则初始值为0,如果响应的值不能转换为整形如hello,则操作失败返回错误信息）

decr key (递减默认为0 减1)

1. 字符串列表（list）
   1. ArrayList使用数组方式,LinkedList使用双向链表方式,双向链表中增加数据,双向链表中删除数据
   2. 存储list常用命令:
      1. 两端添加
         1. lpush mylist a b c（左侧添加lpush key values）
         2. rpush mylist c d x（从右插入）
      2. 查看列表
         1. lrange mylist 0 5（查看列表lrange key值 开始位置 结束位置，0 -1查看所有）
      3. 两端弹出
         1. lpop key （左边弹出移除）
         2. rpop key （右边弹出移除）
      4. 获取列表元素个数
         1. llen mylist （获取列表的元素数量）
      5. 扩展命令
         1. lpushx mylist x（头部值插入）
         2. rpushx mylist y（尾部值插入）
         3. lrem mylist 1 （lrem mylist 删除数量 删除的值，删除几个值为及的值，lrem mylist -2 1 负数代表从后面删除2个1，数量为0是代表删除所有）
         4. lset mylist 3（修改指定角标的值）
         5. linsert mylist befare b 11（在哪个值之前插入的值）
         6. linset mylist after b 22
         7. rpoplpush mylist5 mylist6（将第一个列表中的最右边元素弹出,亚入第二个列表中的左边）
2. 有序字符串集合（sorted set）
   1. Sorted-Set和Set的区别:Sorted-Set有个分数
   2. Sorted-Set中的成员在集合中的位置是有序的
   3. 主要的应用场景是在游戏的排名,微博热点话题等
   4. 存储Sorted-Set常用命令：
      1. 添加元素
         1. zadd mysort 70 zs 80 ls 90 Ww (返回存储元素个数)
         2. zincrby mysort 3 ls （指定成员增加分数）
      2. 获得元素
         1. zscore mysort zs （获得成员对应的分数）
         2. zcard mysort（获得成员的数量）
         3. zrangebyscore mysort 0 100（返回某个分数的成员，并从低到高排序）
         4. zrangebyscore mysort 0 100 withscores 显示分数
         5. zrangebyscore mysort 0 100 withscores limit 0 2
         6. zcount mysort 80 90 （某个分数之间的成员数量）
      3. 删除元素
         1. zrem mysort tom Ww （删除指定的成员）
         2. zremrangebyrank mysort 0 4（按照排名的范围删除）
         3. zremrangebyscore mysort 80 100（按照分数范围删除）
      4. 范围查询
         1. zrange mysort 0 -1（按照下标查找返回成员）
         2. zrange mysort 0 -1 withscores（返回元素和分数，默认由小到大）
         3. zrevrange mysort 0 -1 withscores（由大到小的排名）
      5. 扩展命令
3. 哈希（hash）
   1. String key和String value的map容器
   2. 每个hash可以存储4294967295个键值对
   3. 存储hash常用命令
      1. 赋值

hset key 属性 值 (hasht myhash username jack)

hmset key value value..（设置多个属性值）

hkeys key （获取所有的key）

* + 1. 删除

hdel key 属性 ..(hdel myhash username age)

del key （del myhash2）

* + 1. 自学命令

hexists myhash username （判断某个属性是否存在，存在返回1）

hlen myshas（获取集合中的属性数量）

* + 1. 取值

hget key value (hget myhash username)

hmget key value ..

hgetall key （获取全部key的属性）

hkeys key （获取所有的key，hkeys myhash）

hvals key （获取所有key的属性值）

* + 1. 增加数字

hincrby mhash age 5

1. 字符串集合（set）
   1. set列无序不可重复集合,和list类不同的是,Set集合不允许出现重复的元素
   2. set可包含的最大元素数量是4294967295
   3. 存储set常用命令
      1. 添加/删除元素
         1. sadd myset 1 2 3 （sadd key value..）
         2. srem myset 1 2 （删除）
      2. 获得集合中的元素
         1. smembers myset （查看所有元素）
         2. sismember myset a （查看元素是否存在，如果存返回1，不存在返回0）
      3. 集合中的差集匀速
         1. sdiff mya1 mya2
      4. 集合中的交集运算
         1. sinter myb1 myb2
      5. 集合中的并集运算
         1. sunion mya3 myb3
      6. 扩展命令
         1. smembers myset（获得set集合成员）
         2. scard myset （获取set数量）
         3. srandmember myset（随机然后成员）
         4. sdiffstore my1 mya1 myb1（两个集合相差成员存储到另外集合中）
         5. sinterstore my1 my2 myb1（求两个集合的交集，存储起来）
         6. sunionstore my3 mya3 myb3（并集）

key定义的注意点:

1. 不要太长（避免消耗内存，便于索引）
2. 不要果断
3. 统一命名规范

Keys的通用操作

1. keys \*（获得所有的keys）
2. keys my?（以my开发的所有keys）
3. del my1 my2 my3（删除某个指定的key）
4. exists my1（查看某个k是否存在1代表存在,0不存在）
5. rename compary newcompary（重命名key）
6. expire newcompary 1000（设置过期时间单位秒）
7. ttl newcompary（查看k所剩的超时时间，没有设置返回-1）
8. type newcompary（获得指定k的类型）

Redis的特性

1. 多数据库
   * 1. 0-15个数据库
     2. select 1 切换连接数据库，默认是0号数据库
     3. move key 1 将某个数据移动到几号数据库
2. 事务支持
   * 1. multi | exec | discard 事务命令，串行化存续执行
     2. multi 开启事务，exec 提交 ，discard 回滚
     3. 1.mult
     4. 2.incr num
     5. 3.incr num
     6. 4.exec 或 discard

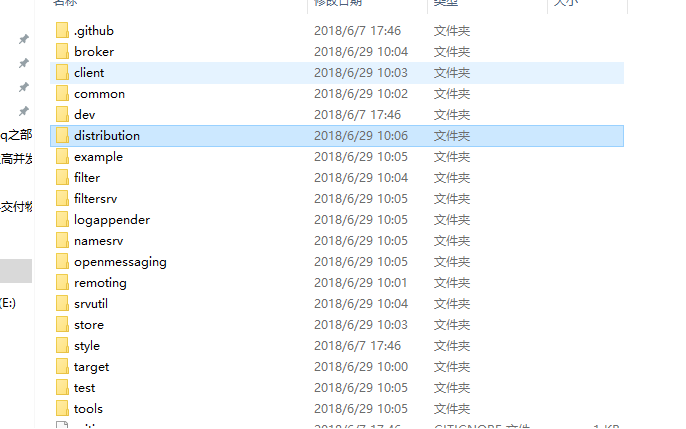
Redis的持久化

1. 两种持久化方式：RDB 方式 AOF方式
2. RDB持久化（默认支持不需要配置）
   1. 在指定的时间内将数据库的快照存储到磁盘中
   2. 优势
      1. 整个redis数据库只包含一个文件,对备份比较不错
   3. 劣势
   4. 配置
      1. 修改redis.conf配置文件（save 900 1 表示最少9百秒有一个k发生变化才会发生快照）
3. AOF持久化（以日志方式存储redis的每一次操作）
   1. 优势
   2. 劣势
   3. 配置
4. 无持久化
5. 同时使用RDB和AOF

# 16.RocketMq

项目部署:

下载项目:https://github.com/apache/rocketmq



项目编译: mvn -Prelease-all -DskipTests clean install -U

项目模块说明

|  |  |
| --- | --- |
| 名称 | 作用 |
| broker | broker模块：c和p端消息存储逻辑 |
| client | 客户端api：produce、consumer端 接受与发送api |
| common | 公共组件：常量、基类、数据结构 |
| tools | 运维tools：命令行工具模块 |
| store | 存储模块：消息、索引、commitlog存储 |
| namesrv | 服务管理模块：服务注册topic等信息存储 |
| remoting | 远程通讯模块：netty+fastjson |
| logappender | 日志适配模块 |
| example | Demo列子 |
| filtersrv | 消息过滤器模块 |
| srvutil | 辅助模块 |
| filter | 过滤模块：消息过滤模块 |
| distribution | 部署、运维相关zip包中的代码 |
| openmessaging | 兼容openmessaging分布式消息模块 |

# GIT使用

## 为什么学习git

牛逼的互联网,牛逼的大神们都在用

完整的版本控制功能,解决多人写作的问题

提高开发效率

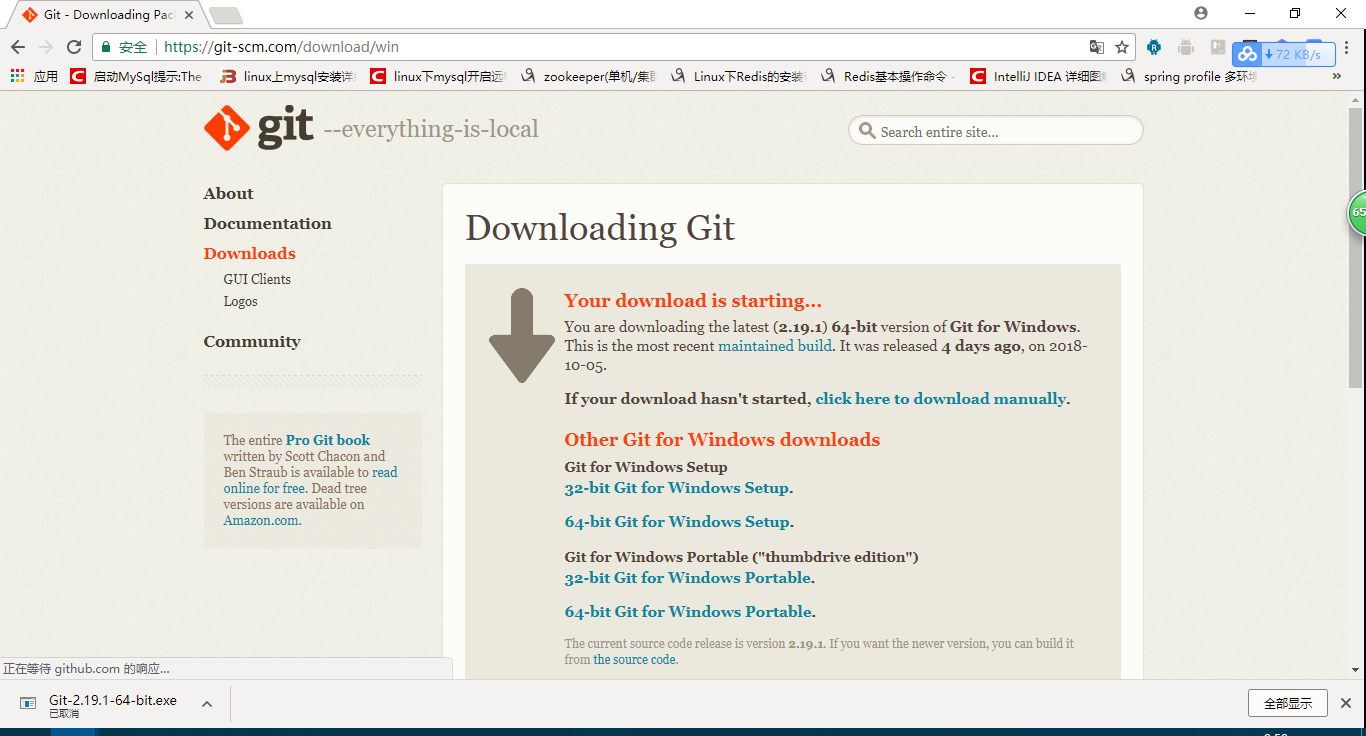
如果你是程序猿,请务必学Git,并习惯吧自己的代码同步到GitHub上

Git不等于GitHub大家注意了!Git和GitHub就是球和球场的关系

## Git安装

下载window git

地址：<https://git-scm.com/download/win>

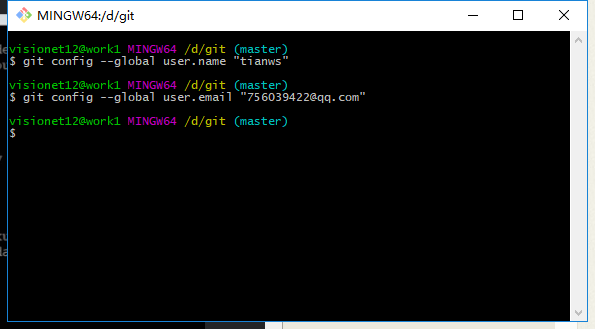


配置用户信息

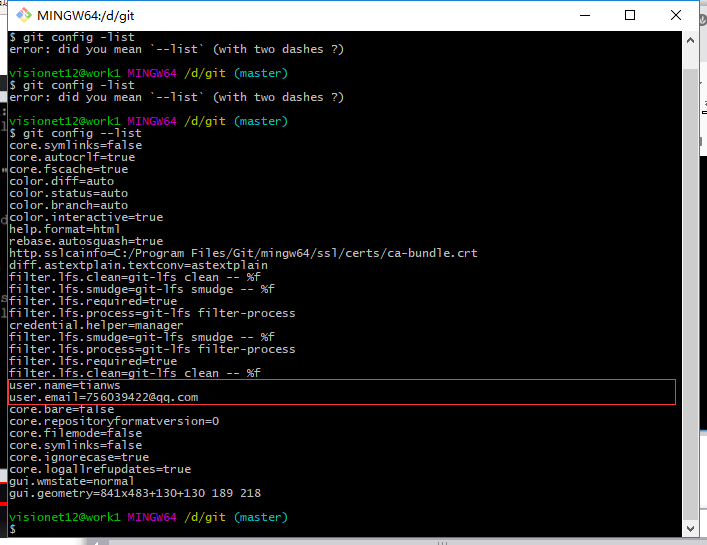
-git config --global user.name “tianweishuo”

-git config --global user.email “756039422@qq.com”

global 表示所有的仓库使用统一配置



查看配置信息确认是否配置成功 git config -list



客户端操作SourceTree

下载都中:https://www.sourcetreeapp.com/

## Git仓库创建及工作量

### 创建仓库

-pwd 显示当前路径

初始化版本库

-git init

添加文件到版本库

-git add 添加新增文件 ./符号表示所有文件

-git commit -m “” 提交到本地仓库m表示备注

查看版本库状态

-git status

### 工作流

暂存区到工作区代码回滚

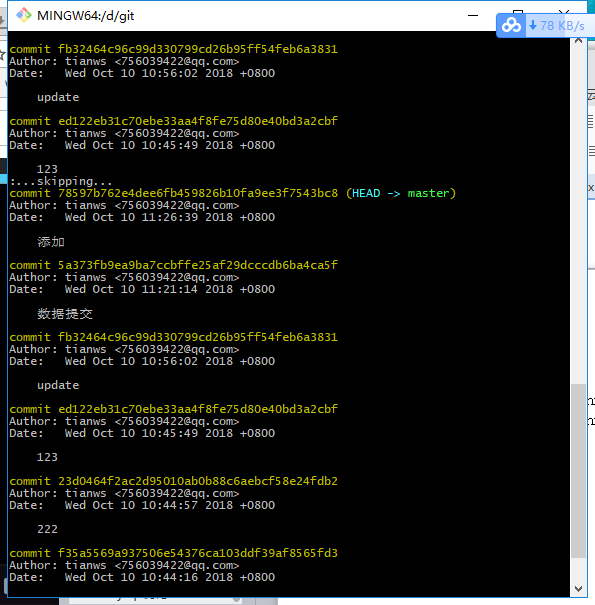
暂存区代码回滚

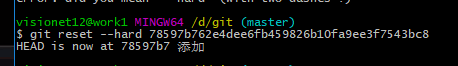
git reset HEAD wenjianming.text

git checkout -- wenjianming.text

本地仓库代码回滚

git log 查看提交日志





代码中追加内容

echo "new code" >> demo.txt

## Git主要功能

远程仓库

创建SSH key

ssh-keygen -t rsa -C “756039422@qq.com”

测试是否已经通了

ssh -T [git@github.com](mailto:git@github.com)

gitee.com

添加远程仓库

git remote add origin [git@gitee.com:TianWeiShuo/demo.git](mailto:git@gitee.com:TianWeiShuo/demo.git)

git pull origin master --allow-unrelated-histories

git push -u origin master

克隆仓库

git clone [git@gitee.com:TianWeiShuo/demo.git](mailto:git@gitee.com:TianWeiShuo/demo.git)

## 分支管理

## 标签管理

查看所有的标签 git tag

创建标签 git tag name

指定提交信息 git tage -a name -m “comment”

删除标签 git tag -d name

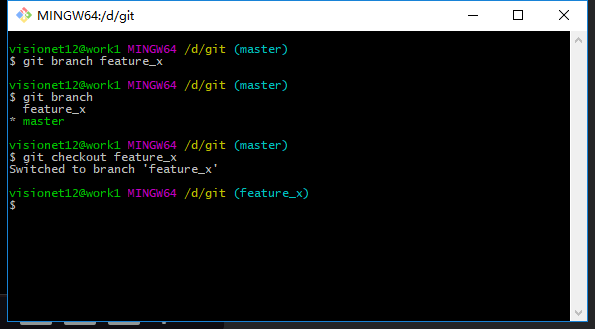
标签发布 git push origin name

## 分支管理

创建分支 git branch name

查看所有分支 git branch

切换分支 git checkout name



分支合并

1. 切换到主分支
2. 分支合并 git merge name
3. 分支删除 git branch -d name

# javaScprit

## 基础

输出语句：

alert（“这是第一个程序”）;//弹出框

document.write(“内容”);//向内body中输入一个内容

console.log();//向控制台中输出内容

定义变量

var a;

数据类型

String 字符串

Number 数值

Boolean 布尔值

Null 控制

Undefined 未定义

Object 对象

转移符

\“ 表示“ \‘ 表示‘ \n 表示换行 \t 制表符 \\ 表示\

检测变量类型

typeof

检测字符串，会返回String

检测数值，会返回number

var a = 100；

console.log(typeof a);

Number() 转换数字

parseInt() 可以将字符中有效的整数内容取出来转换

parseFloat()转换小数

parseInt(值,进制)

Boolean()数字转boolean处理0和Nan其余都是true

null和undefined都会转换为false

对象也会转换为true

对象的基本使用

var obj = new Object();

//向obj中添加一个name对象

obj.name = “孙悟空”;

//向obj中添加一个gender属性

obj.gender = ‘男’

console.log(obj.name);//读取

如果要使用特殊的属性名，不能采用.的方式来操作

语法：对象[“属性名”] = 属性值

使用[]这种形式去操作属性，更加的灵活

在[]中可以直接传递变量，这样变量值是多少就读取哪个属性

obj[“123”] = 789

console.log(obj[“123”]) //读取

in运算符

通过该运算符可以检查一个对象中是否含有指定的属性

如果有则返回true，没有则返会回false

console.log(“test” in obj);

对象字面值

//创建一个对象

var obj = new Object();

//使用对象字面量来创建一个对象

var obj = {};

使用对象字面量，可以在创建对象时，直接指定对象中的属性

语法:{属性名:属性值,属性名:属性值...}

对象字面量的属性可以加引号也可以不加，建议不加，如果要使用一些特殊的名字则必须加。

var obj = {

name:”猪八戒”

age：28

};

console.log（obj）;

函数

//可以将要封装的代码一字符串的形式传递给构造函数

var fun = new Function(“console.log(‘nihao’)”);

//语法

function 函数名(形参1,形参2...){ }

//使用 函数表达式 来创建函数

var 函数名 = function(形参...){ }

立即执行函数

函数定义玩，立即被调用

立即执行函数只会执行一次

(function(){

alert(“立即执行函数”);

})();

(function(a,b){

console.log(a+b);

})(1,2);

对象的属性值可以是任何的函数类型，也可以是函数

var obj = new Object();

obj.name = ‘wukong’

obj.sayName = function(){

console.log(obj.name);

};

for in语句

语法 for(var n in obj){

console.log(“属性名:”+n);

console.log(“属性值:”+obj[n]);

}

for...in 语句 对象中有几个属性，循环体就会执行几次

原型对象

原型prototype

数组（Array）

var arr = new Array();

arr[0] = 10;

arr[1] = 22;

arr[2] = 33;

//读取

console.log(arr[0]);

//读取数组长度

arr.length

//向数组的最后一个位置添加元素

push(“1”，“2”)

//向数组的末尾添加一个或多个元素，并返回数组的长度

pop()

//该方法可以删除数组的最后一个元素，并将被删除的元素返回

unshift(“牛魔王”)

//向数组开通添加一个或多个元素，返回新的数组长度

shift()

//可以删除数组的第一个元素，并将删除的元素作为返回值

forEach（function（value,index,obj）{

//value 值

//index 所有

//obj 当前对象

}）

slice()

//可以用来从数组提取指定元素

参数：

1.截取开始的位置的索引,包含开始

2.截取结束的位置的索引，不包含结束

第二个参数可以忽略不写

splice( , )

//可以用于删除数组中的指定元素

参数：

第一个:开始位置索引

第二个：删除数量

concat（）可以连接两个或多个数组，并将新的数组返回

-该方法不会对原数组产生影响

join（）

该方法可以将数组转换为一个字符串

该方法不会对原数组产生影响

arr.reverse()

该方法用来返回转数组

arr.sort()

可以用来对数组中的元素进行排序

call（）和apply()

这两个方法都是函数对象的方法，需要通过函数对象来调用

当前函数调用call()和apply()都会调用函数执行

在调用call（）和apply（）可以将一个对象指定为第一个参数

DOM对象

document.getElementById();

//修改按钮的文字

btn.inerHTML

事件

获取对象然后进行事件

//点击事件

btn.onclick = function(){};

//为windo绑定noload事件，

该事件对应的相应函数将会在页面加载完成后执行

wind.onload = function(){};

获取元素节点：

通过document对象调用

getElementById()

通过id属性获取一个元素节点对象

getElementsByTagName()

通过标签名获取一组元素节点对象

getElementsByName()

通过name属性获取一组元素节点对象

获取元素节点的子节点

getElementsByTagName（）

-方法，返回当前节点的指定标签后代节点

childNodes

-属性，表示当前节点的所有子节点

firstChild

-属性，表示当前节点的第一个子节点

lastChild

-属性，表示当前节点的最后一个子节点

获取父节点和兄弟节点

parentNode

-

previousSibling

nextSibling

## Jquery

## vue

## java秒杀系统

## tree-grid(基于bootstrap扩展)

# 19.vue.js

## vue.js模板语法

使用表达式{{ }}

|  |
| --- |
| <div id="app">  <p>{{message}}</p> </div>  <script>  new Vue({  el:'#app',  data:{  message:'hellp world!'  }  }) </script> |

v-once:表示只执行一次插值,后面如果数据有改变,插值将不会更新。

|  |
| --- |
| <div id="app">  <p v-once>{{message}}</p> </div>  <script>  var app = new Vue({  el:'#app',  data:{  message:'hellp world!'  }  })   app.message = "asd"; </script> |

输入html代码,可以使用v-html指令

|  |
| --- |
| <div id="app">  <h2 v-html="message"></h2> </div>  <script>  new Vue({  el:'#app',  data:{  message:'<span style="color: red">这是我的标题</span>'  }  }) </script> |

注意:在页面上动态的渲染html是一件很危险的事情,因为他很容易导致XSS攻击(跨站脚本攻击)。

【v-bind】

要给html元素绑定属性,不能直接使用文本插值,vue有特定的指令进行属性的绑定。

|  |
| --- |
| <div id="app">  <h2 v-bind:class="value">在线实例</h2> </div>  <script>  new Vue({  el:'#app',  data:{  value:"context"  }  }) </script> |

多属性绑定

|  |
| --- |
| <div id="app">  <h2 v-bind:id="vid" :class="vclass" :title="vtitle">vue 多属性绑定</h2> </div>  <script>  new Vue({  el:'#app',  data:{  vid:"myid",  vclass: "myclass",  vtitle:"vue 多属性绑定"  }  }) </script> |

javaScript表达式：在输出时把大写的MYAPP转换成了小写的myapp

|  |
| --- |
| <div id="app">  <h2>{{name.toLowerCase()}}</h2> </div>  <script>  new Vue({  el:'#app',  data:{  name:"MYAPP"  }  }) </script> |

过滤器

在两个大括号中 {{name | myfilters}}

在v-bind指令中 <div v-bind:id=”rawId | formatId”> </div>

定义了一个myfilters的过滤器,功能是将插值进行翻转输出。

|  |
| --- |
| <div id="app">  <h2>{{name | myfilter}}</h2> </div>  <script>  new Vue({  el:'#app',  data:{  name:"myapp"  },  filters:{  myfilter(value){  return value.split('').reverse().join();  }  }  }) </script> |

多个过滤器可以串联使用

|  |
| --- |
| <div id="app">  <h2>{{name | touppercase | myfilter }}</h2> </div>  <script>  new Vue({  el:'#app',  data:{  name:"myapp"  },  filters:{  myfilter(value){  return value.split('').reverse().join();  },  touppercase(value){  return value.toUpperCase();  }  }  }) </script> |

缩写

对于【v-on】和【v-bind】这两个常用的指令,vue提供了简写

|  |
| --- |
| <!--完整语法--> <a v-bind:href="url"></a>  <!--缩写--> <a :href="url"></a> |

|  |
| --- |
| <!--完整语法--> <a v-on:click="doSomething"></a> <!--缩写--> <a @click="doSomething"></a> |

修饰符

<form v-on:submit.prevent=”onSubmit”> </fomr>

.prevent是修饰符,告诉v-on 指令对于触发的事件调用event.preventDefault();

## vue.js计算属性（computed）

可以把复杂的表达式写在computed

|  |
| --- |
| <div id="app">  a={{a}},b={{b}} </div>  <script>  new Vue({  el:'#app',  data:{  a:1  },  computed:{  b:function(){  return this.a + 1;  }  }   }) </script> |

计算属性 缓存

当数据源未发生变动是,获取的值将一直是缓存值

我们对b进行了重新赋值,但是没有改变a的值,因为b是依赖于a的,所有最终b的结果还是2.

|  |
| --- |
| <script>  var vm = new Vue({  el:'#app',  data:{  a:1  },  computed:{  b:function(){  return this.a + 1;  }  }  })  vm.b=8; </script> |

计算属性Getter和Setter方法

vue计算属性有两个方法,分别是getter和setter,当没有指明方法是,默认使用getter

在getmessage重新赋值是,vue会执行set方法,从而改变message的值,如果不给set方法,message值为“my first lesson”

|  |
| --- |
| <div id="app">  {{message}} </div>  <script>  var vm = new Vue({  el:'#app',  data:{  title:"my first lesson"  },  computed:{  message:{  //getter  get:function(){  return this.title;  },  //setter  set:function(newValue){  this.title = newValue;  }  }  }  })   vm.message = "你好"; </script> |

## vue.js观察值（watch）

watch属性用于监视vue实例上的数据变动,并相应的改变其他变量的值

curVal表示当前数据,oldVal表示前一步数据(或默认数据)

|  |
| --- |
| <div id="app">  <input type="text" v-model="firstName">  <input type="text" v-model="lastName">  <p>我的名字是:{{fullName}}</p> </div> <script>  new Vue({  el:"#app",  data:{  firstName:"田",  lastName:"伟硕",  fullName:"田 伟硕"  },  watch:{  firstName:function(curVal,oldVal){  this.fullName = curVal + ' '+this.lastName;  },  lastName:function(curVal,oldVal){  this.fullName = this.firstName +' '+curVal;  }  }  }) </script> |

|  |
| --- |
| <div id="app">  <input type="text" v-model="number">  <p>result:{{result}}</p> </div> <script>  new Vue({  el:"#app",  data:{  number:1,  result:1  },  watch:{  number:function(curVal,oldVal){  this.result = curVal\*1+oldVal\*1  }  }  }) </script> |

|  |
| --- |
| <div id="app">  <input type="text" v-model="fruits.count">  <input type="text" v-model="fruits.name">  <p>fruits:{{message}}</p> </div> <script>  new Vue({  el:"#app",  data:{  fruits:{  name:"香蕉",  count:5  },  message:'5条香蕉'  },  watch:{  fruits:{  handler(obj){  this.message = obj.count + '条'+obj.name  },  deep:true  }  }  }) </script> |

## vue.js样式绑定

html中,样式的绑定就是对元素的class和style属性进行操作,前者可以是一个字符串列表,后者允许你直接在html文件中写css代码

vue.js对样式的操作相比HTML增强了不少,除了可以使用字符串外,还允许使用对象数组。

在vue.js中,使用【v-bind】指令进行样式绑定

|  |
| --- |
| <!DOCTYPE html> <html lang="en"> <head>  <meta charset="UTF-8">  <title>样式绑定</title>  <script src="vue.min.js"></script>  <style>  .current{text-decoration:underline}  </style> </head> <body>  <di id="app">  <span class="side" v-bind:class="{current:isCurrent}">当前导航</span> </di>  <script>  new Vue({  el:'#app',  data:{  isCurrent: true  }  }) </script>   </body> </html> |

绑定多个class

|  |
| --- |
| <!DOCTYPE html> <html lang="en"> <head>  <meta charset="UTF-8">  <title>样式绑定</title>  <script src="vue.min.js"></script>  <style>  .current{text-decoration:underline}  .is\_nav{background-color: #2FA4E7}  .color{color:gray}  </style> </head> <body>  <di id="app">  <span class="side" v-bind:class="objClass">当前导航</span> </di>  <script>  new Vue({  el:'#app',  data:{  objClass: {  'current':true,  'is\_nav':false,  'color':true  }  }  }) </script>   </body> </html> |

计算属性

特点是从缓存里面获取值,对于一些依赖被绑定的状态值变化来更新数据变量,我们可以吧它放到计算属性中

|  |
| --- |
| <!DOCTYPE html> <html lang="en"> <head>  <meta charset="UTF-8">  <title>样式绑定</title>  <script src="vue.min.js"></script>  <style>  .color{color:gray}  </style> </head> <body>  <di id="app">  <span class="side" v-bind:class="comClass">当前导航</span> </di>  <script>  new Vue({  el:'#app',  data:{  isColor:true  },  computed:{  comClass:function(){  return{  'color':this.isColor  }  }  }  }) </script> </body> </html> |

数组

表达式放回数组,可以使用中括号[]进行绑定

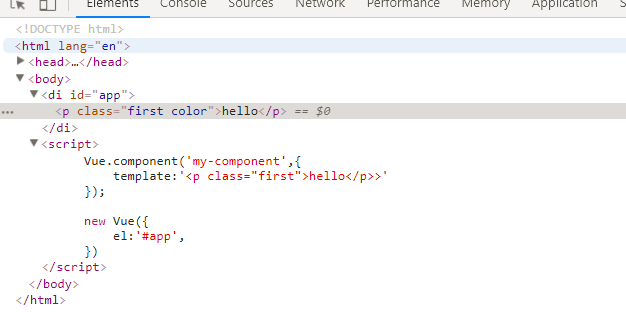
|  |
| --- |
| <style>  .current{text-decoration:underline}  .color{color:gray}  </style> </head> <body>  <di id="app">  <span class="side" v-bind:class="[currentClass,colorClass]">当前导航</span> </di>  <script>  new Vue({  el:'#app',  data:{  currentClass:'current',  colorClass:'color',  }  }) </script> |

数组里也可以使用对象来更新class列表

|  |
| --- |
| <style>  .current{text-decoration:underline}  .color{color:gray}  </style> </head> <body>  <di id="app">  <span class="side" v-bind:class="[{current:isCurrent},colorClass]">当前导航</span> </di>  <script>  new Vue({  el:'#app',  data:{  isCurrent:true,  colorClass:'color',  }  }) </script> |

组件（component）

|  |
| --- |
| <di id="app">  <my-component class="color"></my-component> </di>  <script>  Vue.component('my-component',{  template:'<p class="first">hello</p>>'  });   new Vue({  el:'#app',  }) </script> |



自定义组件上可以使用【v-bind】指令

<my-component v-bind:class="{first: true, color: true}"></my-component>

绑定内联样式(style)

|  |
| --- |
| <di id="app">  <h2 v-bind:style="styleObj">内联样式</h2> </di>  <script>  new Vue({  el:'#app',  data:{  styleObj:{  color:'red',  fontSize:'14px'  }  }  }) </script> |

<h2 v-bind:style="{color: 'red', fontSize: '14px'}">明灯小站</h2>

计算属性

|  |
| --- |
| <di id="app">  <h2 v-bind:style="comObj">计算属性</h2> </di>  <script>  new Vue({  el:'#app',  data:{  color:'red',  fontSize:'14px'  },  computed:{  comObj:function(){  return{  color:this.color,  fontSize:this.fontSize  }  }  }  }) </script> |

数组

第一种方法是直接在元素中操作

<h2 v-bind:style="[{color:'red'}, {fontSize: '14px'}]">明灯小站</h2>

第二种是使用data属性进行赋值

|  |
| --- |
| <di id="app">  <h2 :style="arrStyle">数组</h2> </di>  <script>  new Vue({  el:'#app',  data:{  arrStyle:[  {color:'red'},  {fontSize:'14px'}  ]  }  }) </script> |

第三种方法是以对象数组的形式绑定

|  |
| --- |
| <di id="app">  <h2 :style="[colorObj,fontObj]">明灯小站</h2> </di>  <script>  new Vue({  el:'#app',  data:{  colorObj:{color:'red'},  fontObj:{fontSize:'14px'}  }  }) </script> |

## vue.js条件判断

在视图中,控制元素的显示与否是常用的需求,vue.js有多个指令来实现条件判断

v-if,v-else和v-else-if

当v-if后面的值为真时,将显示iphone,当为假时,继续判断v-else-if的值,如果真,则显示对应内容,如果为假,则继续执行,直到执行到v-else语句

注意:v-if可以单独使用,v-else和v-else-if前必须有v-if指令

|  |
| --- |
| <div id="app">  <p v-if="type == 'iphone'">iPhone</p>  <p v-else-if="type == 'xiaomi'">小米</p>  <p v-else-if="type == huawei">华为</p>  <p v-else>没有相关数据</p> </div> <script>  new Vue({  el:'#app',  data:{  type:'xiaomi'  }  }) </script> |

如果需要对一组元素进行判断,我们需要用<template>元素

在<template>中使用v-else和v-else-if的方法跟普通元素的一样

|  |
| --- |
| <div id="app">  <!--将showme的值改为false在试试-->  <template v-if="showne">  <p>我是谁？</p>  <p>我在那？</p>  <p>我要干什么？</p>  </template>  </div> <script>  new Vue({  el:'#app',  data:{  showne:true  }  }) </script> |

v-show

控制元素的显示域否,我们可以使用v-show指令

|  |
| --- |
| <div id="app">  <p v-show="showme">看我显示不显示</p> </div> <script>  new Vue({  el:'#app',  data:{  showme:false  }  }) </script> |

下面表格列出的是v-show和v-if这两个指令的主要不同点

|  |  |  |
| --- | --- | --- |
|  | v-if | v-show |
| 渲染时间 | 第一次为真时 | 刚开始就渲染 |
| 切换形式 | 动态生成，局部编译/卸载 | 控制display属性 |
| 生成消耗 | 较小（只生成为真的部分） | 较大（生成全部） |
| 切换消耗 | 较大（切换时需要局部编译） | 较小（因为生成时已经渲染完成） |

## vue.js列表渲染

列表的渲染使用v-for指令

变量对象

|  |
| --- |
| <div id="app">  <ul>  <li v-for="value in object">{{value}}</li>  </ul> </div> <script>  new Vue({  el:'#app',  data:{  object:{  first:"a",  second:"b",  third:"c"  }  }  }) </script> |

跟js类似,遍历是你也可以获取对象的索引和键值

<ul>

<li v-for="(value, key, index) in object">{{value}}</li>

</ul>

<div id="app">  
 <ul>  
 <li v-for="(value, key, index) in object">这是第{{index+1}}个字母:{{value}},键为{{key}}</li>  
 </ul>  
</div>  
<script>  
 new Vue({  
 el:'#app',  
 data:{  
 object: {  
 first: "A",  
 second: "B",  
 third: "C"  
 }  
 }  
 })  
</script>

遍历数组

在变量数组时,你也可以使用of代替in,结果是一样的。

<div id="app">  
 <ul>  
 <li v-for="(item, index) in items">这是第{{index+1}}个字母:{{item.letter}}</li>  
 </ul>  
</div>  
<script>  
 new Vue({  
 el:'#app',  
 data:{  
 items:[  
 {letter:'a'},  
 {letter:'b'},  
 {letter:'c'}  
 ]  
 }  
 })  
</script>

遍历<template>

遍历中如果要赋值给一组元素,可在<template>标签中使用v-for指令

<body>  
<div id="app">  
 <template v-for="item in items">  
 <div>  
 <h2>{{item.title}}</h2>  
 <p>{{item.content}}</p>  
 </div>  
 </template>  
</div>  
<script>  
 new Vue({  
 el:'#app',  
 data:{  
 items:[  
 {title:"Vue.js",content:"学习vue,我们需要..."},  
 {title:"HTML",content:"HTML 是前端只是的入门基础"}  
 ]  
 }  
 })  
</script>  
</body>

## vue.js事件处理

v-on指令用于监听dom事件,对应的处理方法在methods属性中执行

<!--完整语法-->  
<button v-on:click="showme"></button>  
  
<!--缩写-->  
<button @:click="showme"></button>

方法处理器

实例1

|  |
| --- |
| <div id="app">  <button v-on:click="showme">{{message}}</button> </div> <script>  new Vue({  el:'#app',  data:{  message:"点击查看"  },  methods:{  showme:function(e){  alert("执行showme方法");  }  }  }) </script> |

内联语句处理器

内联语句允许你传递数值到方法中

|  |
| --- |
| <div id="app">  <a href="http:://www.mingtern.com" v-on:click="say('hello',$event)">{{message}}</a> </div> <script>  new Vue({  el:'#app',  data:{  message:'点击看看'  },  methods:{  say:function(value,e){  e.preventDefault();  alert(value);  }  }  }) </script> |

## vue.js v-text指令

标签中使用v-text ，插入文本

## vue.js v-html指令

插入html

## vue.js v-show指令

## vue.js v-if指令

## vue.js v-else 指令

## vue.js v-else-if指令

## vue.js v-for 指令

## vue.js v-on指令

## vue.js v-bind指令

<!--绑定一个属性-->  
<h2 v-bind:属性名="属性值"></h2>  
<!--缩写-->  
<h2 :属性名="属性值"></h2>

多属性绑定

<di id="app">  
 <h2 v-bind:id="vid" :class="vclass" :title="vtitle">vue 多属性绑定</h2>  
</di>  
  
<script>  
 new Vue({  
 el: "#app",  
 data: {  
 vid:"myid",  
 vclass: "myclass",  
 vtitle:"vue 多属性绑定"  
 }  
 })  
</script>

绑定了style属性

<di id="app">  
 <h2 v-bind:style="[styleA,styleB]">{{message}}</h2>  
</di>  
  
<script>  
 new Vue({  
 el: "#app",  
 data: {  
 message:'多样式绑定',  
 styleA:{  
 fontSize:'30px',  
 color:'blue'  
 },  
 styleB:{  
 textDecoration:'underline'  
 }  
 }  
 })  
</script>

## vue.js v-model指令

v-model指令可实现数据的双向绑定,主要用于监听用户的输入事件,使vue实例中的data数据与渲染的dom元素的数据保持一致。

v-model指令只能用于<input> <textarea> <select> 等表单元素上

文本输入

<div id="app">  
 <p>单行文本:</p>  
 <input type="text" v-model="message1" placeholder="单行文本">  
 <p>当前输入:{{message1}}</p>  
  
 <p>多行文本:</p>  
 <textarea v-model="message2" placeholder="多行文本"></textarea>  
 <p>当前输入:{{message2}}</p>  
</div>  
  
<script>  
 new Vue({  
 el:'#app',  
 data:{  
 message1:'',  
 message2:''  
 }  
 })  
</script>

复选框

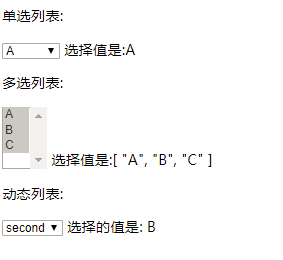
<div id="app">  
 <p>单个复选框:</p>  
 <input type="checkbox" id="checkbox" v-model="checked">  
 <label for="checkbox">{{checked}}</label>  
  
  
 <p>多个复选框</p>  
 <input type="checkbox" id="iphone" value="iphoen" v-model="checkedNames">  
 <label for="iphone">iPhone</label>  
 <input type="checkbox" id="xiaomi" value="xiaomi" v-model="checkedNames">  
 <label for="xiaomi">小米</label>  
 <input type="checkbox" id="huawei" value="huawei" v-model="checkedNames">  
 <label for="huawei">华为</label>  
 <span>选择的值：{{checkedNames}}</span>  
  
</div>  
  
<script>  
 new Vue({  
 el:'#app',  
 data:{  
 checked:false,  
 checkedNames:[]  
 }  
 })  
</script>

单选按钮

<div id="app">  
 <input type="radio" id="migtern" value="mingtern" v-model="choosed">  
 <label for="migtern">migtern</label>  
 <br>  
  
 <input type="radio" id="baidu" value="baidu" v-model="choosed">  
 <label for="baidu">baidu</label>  
 <br>  
 <span>选择为：{{choosed}}</span>  
  
</div>  
  
<script>  
 new Vue({  
 el:'#app',  
 data:{  
 choosed:'mingtern'  
 }  
 })  
</script>

select列表

<div id="app">  
 <p>单选列表:</p>  
 <select v-model="single">  
 <option disabled value="">请选择</option>  
 <option>A</option>  
 <option>B</option>  
 <option>C</option>  
 </select>  
 <span>选择值是:{{single}}</span>  
  
 <p>多选列表:</p>  
 <select v-model="multiple" style="width:50px" multiple>  
 <option>A</option>  
 <option>B</option>  
 <option>C</option>  
 </select>  
 <span>选择值是:{{multiple}}</span>  
  
 <p>动态列表:</p>  
 <select v-model="dynamic">  
 <option disabled value="">请选择</option>  
 <option v-for="item in items" v-bind:value="item.value">  
 {{ item.text }}  
 </option>  
 </select>  
 <span>选择的值是: {{ dynamic }}</span>  
  
  
</div>  
  
  
<script>  
  
 new Vue({  
 el:'#app',  
 data:{  
 single:'',  
 multiple:'',  
 dynamic: '',  
 items: [  
 {text: 'first', value: 'A' },  
 {text: 'second', value: 'B' },  
 {text: 'third', value: 'C' }  
 ]  
 }  
 })  
</script>

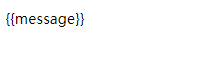


v-model支持的修饰符

## vue.js v-pre指令

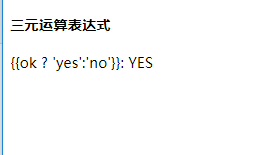
vue实例化时会遍历页面所有节点,如果想跳过某些节点,可以在该节点中使用v-pre指令,编译器会自动忽略该节点和其字子节点。

<div id="app">  
 <p v-pre>{{message}}</p>  
</div>  
<script>  
 new Vue({  
 el:'#app',  
 data:{  
 message:'hello world!'  
 }  
 })  
</script>



实例2

<div id="app">  
 <h4>三元运算表达式</h4>  
 <span v-pre>{{ok ? 'yes':'no'}}:</span>  
 <span>{{ok ? 'YES':'NO'}}</span>  
</div>  
<script>  
 new Vue({  
 el:'#app',  
 data:{  
 ok:true  
 }  
 })  
</script>



## vue.js v-cloak指令

在vuejs开发的项目中,有些页面在加载不完全时,有可能出现一些vue.js变量名,为了避免这种情况,我们可以使用v-cloak指令,该指令会在关联元素的所有vuejs变量编译完成后才正常显示。

<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <title>Title</title>  
</head>  
<style>  
 [v-cloak] {  
 display: none;  
 }  
</style>  
<body>  
  
<div id="app" v-cloak>  
 <h2>{{title}}</h2>  
 <p>{{content}}</p>  
 <script src="vue.min.js"></script>  
</div>  
  
<script>  
 new Vue({  
 el: '#app',  
 data: {  
 title: 'mytitle',  
 content: 'mycontent'  
 }  
 })  
  
</script>  
  
</body>  
</html>

上述代码添加了 v-cloak 指令，但是并不生效，还需要在 css 文件中添加如下代码：

[v-cloak] {

display: none;

}

这样才会最终生效。

上述代码中，当 id 为 app 的元素内的所有 vuejs 变量编译完成后，该元素才会正常显示。

## vue.js v-once指令

使用v-once指令,表示只对关联元素或组件进行一次渲染,随后会被当做静态内容跳过,不会再次渲染。

<div id="app">  
 <h2 v-once>{{title}}</h2>  
 <p>{{content}}</p>  
</div>  
<script>  
 var app = new Vue({  
 el:'#app',  
 data:{  
 title:"mytitle",  
 context:'mycontext'  
 }  
 })  
 app.title = "标题不能改变";  
 app.content = "内容可以改变";  
</script>

## vue.js组件

组件是vue.js非常强大的一个功能,它可以扩展HTML元素,还可以封装可重用代码。

组件系统的知识点较多,我们将结合实例一一讲解。

注册一个组件的语法如下:

Vue.component(tagName,options);

调用一个组件的语法如下:

<tagName></tagName>

tagName表示组件的名字,调用的时候直接使用即可,options表示配置选项,可在里面配置相关信息

全局组件:可在所有实例中使用

<div id="app">  
 <mingtern></mingtern>  
</div>  
  
<script>  
 //注册  
 Vue.component("mingtern",{  
 template:'<h2>自定义组件</h2>'  
 });  
 //创建根实例  
 new Vue({  
 el:'#app'  
 })  
  
</script>

局部注册

只能当前实例中生效,使用关键字components注册

<div id="app">  
 <mingtern></mingtern>  
</div>  
  
<script>  
 var Child = {  
 template:"<h2>局部注册</h2>"  
 }  
 //创建实例  
 new Vue({  
 el:'#app',  
 components:{  
 'mingtern':Child  
 }  
 })  
</script>

## vue组件细节

is标签解决，is标签显示row组件，组件小bug使用

<body>  
 <div id="root">  
 <table>  
 <tbody>  
 <tr is="row"></tr>  
 </tbody>  
 </table>  
 </div>  
<script>

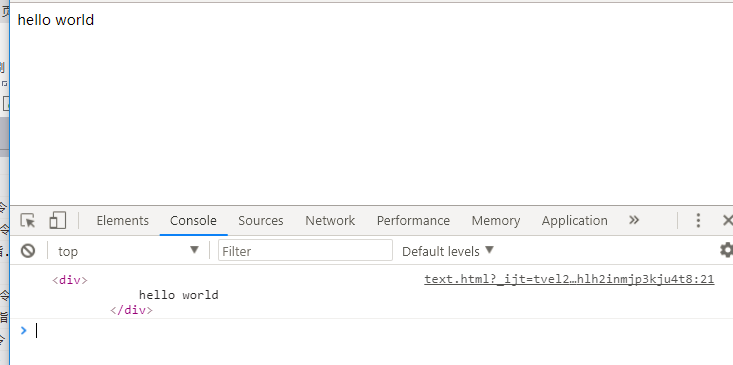
Vue.component('row',{  
 template:'<tr><td>this is row</td></tr>'  
 });  
   
 var vm = new Vue({  
 el : '#root'  
 });  
  
</script>

子组件定义data必须是一个function并且定义return 返回

<body>  
  
 <div id="root">  
 <table>  
 <tbody>  
 <tr is="row"></tr>  
 </tbody>  
 </table>  
 </div>  
<script>  
  
 Vue.component('row',{  
 data:function(){  
 return{  
 content : 'this is content'  
 }  
 },  
 template:'<tr><td>{{content}}</td></tr>'  
 });  
  
 var vm = new Vue({  
 el : '#root'  
 });  
  
</script>

vue中使用ref操作Dom对象

<body>  
  
 <div id="root">  
 <div ref="hello"  
 @click="handleClick">  
 hello world  
 </div>  
 </div>  
<script>  
 var vm = new Vue({  
 el : '#root',  
 methods:{  
 handleClick:function(){  
 console.log(this.$refs.hello);  
 }  
 }  
 });



## 子组件向父组件传值 this.$emit(‘’)

<body>  
 <div id="root">  
 <counter ref="one" @change="handleChange"></counter>  
 <counter ref="two" @change="handleChange"></counter>  
 <div>{{total}}</div>  
 </div>  
<script>  
 Vue.component('counter',{  
 template:'<div @click="handleClick">{{number}}</div>',  
 data:function(){  
 return{  
 number:0  
 }  
 },  
 methods:{  
 handleClick:function(){  
 this.number++;  
 this.$emit('change');  
 }  
 }  
 });  
  
 var vm = new Vue({  
 el : '#root',  
 data:{  
 total:0  
 },  
 methods:{  
 handleChange:function(){  
 this.total = this.$refs.one.number + this.$refs.two.number;  
 }  
 }  
 });  
</script>

## 父子组件的数据传递

props用来接受父组件传递的值

<body>  
  
 <div id="root">  
 <counter :count="3" @inc="handleIncrease"></counter>  
 <counter :count="2" @inc="handleIncrease"></counter>  
 <div>{{total}}</div>  
 </div>  
  
<script>  
 var counter = {  
 props:['count'],  
 data:function(){  
 return{  
 number:this.count  
 }  
 },  
 template:'<div @click="handleClick">{{number}}</div>',  
 methods:{  
 handleClick:function(){  
 this.number++;  
 this.$emit('inc',this.number);  
 }  
 }  
 }  
  
 var vm = new Vue({  
 el:'#root',  
 data:{  
 total:5  
 },  
 components:{  
 counter:counter  
 },  
 methods:{  
 handleIncrease:function(number){  
 this.total = number;  
 }  
 }  
 })  
</script>

## 组件参数校验与非Props特性

props使用对象方式接受值，定义对象中的数据类型

Vue.component("child",{  
 props:{  
 content:{  
 type:String,//数据类  
 required:true,//是否必传  
 default:'default value',//默认值  
 validator:function(value){ //自定义校验器  
 return (value.length > 5)  
 }  
 }  
 },  
 template:"<div>{{content}}</div>"  
});

<body>  
 <div id="root">  
 <child :content="123"></child>  
 </div>  
<script>  
 Vue.component("child",{  
 props:{  
 content:[Number,String]  
 },  
 template:"<div>{{content}}</div>"  
 });  
  
 var vm = new Vue({  
 el:"#root"  
 });  
  
</script>

给组件绑定原生事件

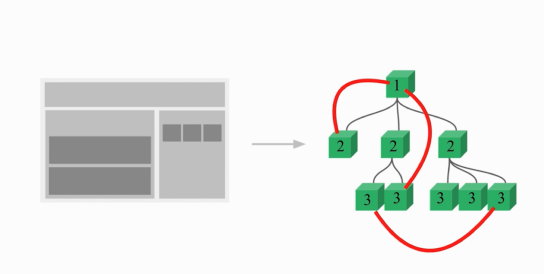
@click.native = “”

## 非父子组件间的传值

父组件通过props向子组件传值,子组件通过触发向父组件传值

非父子组件直接传值,通过插件进行完成。

1. 借助vuex解决问题，使用比较父子
2. 发布订阅模式解决，总线机制



Bus/总线/发布订阅模式/观察者模式

<div id="root">  
 <child content="Dell"></child>  
 <child content="Lee"></child>  
 </div>  
  
<script>  
  
 Vue.prototype.bus = new Vue();  
  
 Vue.component('child',{  
 data:function(){  
 return{  
 selfContext:this.content  
 }  
 },  
 props:{  
 content:String  
 },  
 template:'<div @click="handleClick">{{selfContext}}</div>',  
 methods:{  
 handleClick:function(){  
 this.bus.$emit('change',this.selfContext)  
 }  
 },  
 mounted:function(){//生命周期钩子  
 var this\_ = this;  
 this.bus.$on('change',function(msg){  
 this\_.selfContext = msg;  
 })  
 }  
 });  
  
 var vm = new Vue({  
 el:'#root'  
 });  
</script>

## 在vue中使用插槽（slot）

<div id="root">  
 <body-content>  
 <div class="header" slot="header">header</div>  
 <div class="footer" slot="footer">footer</div>  
 </body-content>  
 </div>  
  
<script>  
 Vue.component('body-content',{  
 template:'<div><slot name="header"></slot><div class="content">content</div><slot name="footer"></slot></div>'  
 });  
 var vm = new Vue({  
 el:'#root'  
 });  
</script>

## vue作用于插槽

<div id="root">  
 <child>  
 <template slot-scope="props">  
 <li>{{props.item}}</li>  
 </template>  
 </child>  
 </div>  
   
<script>  
 Vue.component('child',{  
 data:function(){  
 return{  
 list:[1,2,3,4]  
 }  
 },  
 template:'<div>\n' +  
 ' <slot v-for="item of list" :item="item"></slot>\n' +  
 ' </div>'  
 });  
  
 var vm = new Vue({  
 el:'#root'  
 });  
</script>

## 动态组件与v-once指令

<componen></componen>

## Vue中css动画原理

transition标签 vue动画执行流程

显示流程



隐藏流程



<style>  
 .fade-enter,  
 .fade-leave-to{  
 opacity: 0;  
 }  
 .fade-enter-active,  
 .fade-leave-active{  
 transition: opacity 3s;  
 }  
 </style>  
</head>  
<body>  
  
 <div id="root">  
 <transition name="fade">  
 <div v-if="show">hello world</div>  
 </transition>  
  
 <button @click="handleClick">切换</button>  
 </div>  
  
<script>  
 var vm = new Vue({  
 el:'#root',  
 data:{  
 show:true  
 },  
 methods:{  
 handleClick:function(){  
 this.show = !this.show;  
 }  
 }  
 })  
</script>

## 在Vue中使用Animate.css库

## 单文件组件与Vue中的路由

main.js是项目的入口文件

路由就是根据网址的不同,返回不同的内容给用户

## html中meta标签配置

<meta name="viewport" content="width=device-width,initial-scale=1.0,minimun-scale=1.0,maximum-scale=1.0,er-scalable=no">

引入reset.css样式表,统一移动端样式,对基础样式的修饰

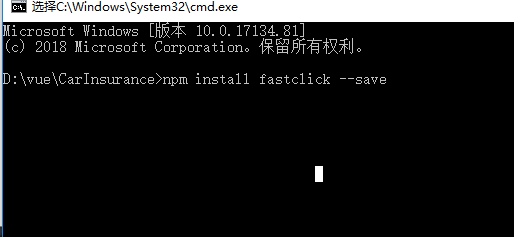
@charset "utf-8";html{background-color:#fff;color:#000;font-size:12px}  
body,ul,ol,dl,dd,h1,h2,h3,h4,h5,h6,figure,form,fieldset,legend,input,textarea,button,p,blockquote,th,td,pre,xmp{margin:0;padding:0}  
body,input,textarea,button,select,pre,xmp,tt,code,kbd,samp{line-height:1.5;font-family:tahoma,arial,"Hiragino Sans GB",simsun,sans-serif}  
h1,h2,h3,h4,h5,h6,small,big,input,textarea,button,select{font-size:100%}  
h1,h2,h3,h4,h5,h6{font-family:tahoma,arial,"Hiragino Sans GB","微软雅黑",simsun,sans-serif}  
h1,h2,h3,h4,h5,h6,b,strong{font-weight:normal}  
address,cite,dfn,em,i,optgroup,var{font-style:normal}  
table{border-collapse:collapse;border-spacing:0;text-align:left}  
caption,th{text-align:inherit}  
ul,ol,menu{list-style:none}  
fieldset,img{border:0}  
img,object,input,textarea,button,select{vertical-align:middle}  
article,aside,footer,header,section,nav,figure,figcaption,hgroup,details,menu{display:block}  
audio,canvas,video{display:inline-block;\*display:inline;\*zoom:1}  
blockquote:before,blockquote:after,q:before,q:after{content:"\0020"}  
textarea{overflow:auto;resize:vertical}  
input,textarea,button,select,a{outline:0 none;border: none;}  
button::-moz-focus-inner,input::-moz-focus-inner{padding:0;border:0}  
mark{background-color:transparent}  
a,ins,s,u,del{text-decoration:none}  
sup,sub{vertical-align:baseline}  
html {overflow-x: hidden;height: 100%;font-size: 50px;-webkit-tap-highlight-color: transparent;}  
body {font-family: Arial, "Microsoft Yahei", "Helvetica Neue", Helvetica, sans-serif;color: #333;font-size: .28em;line-height: 1;-webkit-text-size-adjust: none;}  
hr {height: .02rem;margin: .1rem 0;border: medium none;border-top: .02rem solid #cacaca;}  
a {color: #25a4bb;text-decoration: none;}

移动端1像素边框问题,引入border.css

@charset "utf-8";  
.border,  
.border-top,  
.border-right,  
.border-bottom,  
.border-left,  
.border-topbottom,  
.border-rightleft,  
.border-topleft,  
.border-rightbottom,  
.border-topright,  
.border-bottomleft {  
 position: relative;  
}  
.border::before,  
.border-top::before,  
.border-right::before,  
.border-bottom::before,  
.border-left::before,  
.border-topbottom::before,  
.border-topbottom::after,  
.border-rightleft::before,  
.border-rightleft::after,  
.border-topleft::before,  
.border-topleft::after,  
.border-rightbottom::before,  
.border-rightbottom::after,  
.border-topright::before,  
.border-topright::after,  
.border-bottomleft::before,  
.border-bottomleft::after {  
 content: "\0020";  
 overflow: hidden;  
 position: absolute;  
}  
/\* border  
 \* 因，边框是由伪元素区域遮盖在父级  
 \* 故，子级若有交互，需要对子级设置  
 \* 定位 及 z轴  
 \*/  
.border::before {  
 box-sizing: border-box;  
 top: 0;  
 left: 0;  
 height: 100%;  
 width: 100%;  
 border: 1px solid #eaeaea;  
 transform-origin: 0 0;  
}  
.border-top::before,  
.border-bottom::before,  
.border-topbottom::before,  
.border-topbottom::after,  
.border-topleft::before,  
.border-rightbottom::after,  
.border-topright::before,  
.border-bottomleft::before {  
 left: 0;  
 width: 100%;  
 height: 1px;  
}  
.border-right::before,  
.border-left::before,  
.border-rightleft::before,  
.border-rightleft::after,  
.border-topleft::after,  
.border-rightbottom::before,  
.border-topright::after,  
.border-bottomleft::after {  
 top: 0;  
 width: 1px;  
 height: 100%;  
}  
.border-top::before,  
.border-topbottom::before,  
.border-topleft::before,  
.border-topright::before {  
 border-top: 1px solid #eaeaea;  
 transform-origin: 0 0;  
}  
.border-right::before,  
.border-rightbottom::before,  
.border-rightleft::before,  
.border-topright::after {  
 border-right: 1px solid #eaeaea;  
 transform-origin: 100% 0;  
}  
.border-bottom::before,  
.border-topbottom::after,  
.border-rightbottom::after,  
.border-bottomleft::before {  
 border-bottom: 1px solid #eaeaea;  
 transform-origin: 0 100%;  
}  
.border-left::before,  
.border-topleft::after,  
.border-rightleft::after,  
.border-bottomleft::after {  
 border-left: 1px solid #eaeaea;  
 transform-origin: 0 0;  
}  
.border-top::before,  
.border-topbottom::before,  
.border-topleft::before,  
.border-topright::before {  
 top: 0;  
}  
.border-right::before,  
.border-rightleft::after,  
.border-rightbottom::before,  
.border-topright::after {  
 right: 0;  
}  
.border-bottom::before,  
.border-topbottom::after,  
.border-rightbottom::after,  
.border-bottomleft::after {  
 bottom: 0;  
}  
.border-left::before,  
.border-rightleft::before,  
.border-topleft::after,  
.border-bottomleft::before {  
 left: 0;  
}  
@media (max--moz-device-pixel-ratio: 1.49), (-webkit-max-device-pixel-ratio: 1.49), (max-device-pixel-ratio: 1.49), (max-resolution: 143dpi), (max-resolution: 1.49dppx) {  
 /\* 默认值，无需重置 \*/  
}  
@media (min--moz-device-pixel-ratio: 1.5) and (max--moz-device-pixel-ratio: 2.49), (-webkit-min-device-pixel-ratio: 1.5) and (-webkit-max-device-pixel-ratio: 2.49), (min-device-pixel-ratio: 1.5) and (max-device-pixel-ratio: 2.49), (min-resolution: 144dpi) and (max-resolution: 239dpi), (min-resolution: 1.5dppx) and (max-resolution: 2.49dppx) {  
 .border::before {  
 width: 200%;  
 height: 200%;  
 transform: scale(.5);  
 }  
 .border-top::before,  
 .border-bottom::before,  
 .border-topbottom::before,  
 .border-topbottom::after,  
 .border-topleft::before,  
 .border-rightbottom::after,  
 .border-topright::before,  
 .border-bottomleft::before {  
 transform: scaleY(.5);  
 }  
 .border-right::before,  
 .border-left::before,  
 .border-rightleft::before,  
 .border-rightleft::after,  
 .border-topleft::after,  
 .border-rightbottom::before,  
 .border-topright::after,  
 .border-bottomleft::after {  
 transform: scaleX(.5);  
 }  
}  
@media (min--moz-device-pixel-ratio: 2.5), (-webkit-min-device-pixel-ratio: 2.5), (min-device-pixel-ratio: 2.5), (min-resolution: 240dpi), (min-resolution: 2.5dppx) {  
 .border::before {  
 width: 300%;  
 height: 300%;  
 transform: scale(.33333);  
 }  
 .border-top::before,  
 .border-bottom::before,  
 .border-topbottom::before,  
 .border-topbottom::after,  
 .border-topleft::before,  
 .border-rightbottom::after,  
 .border-topright::before,  
 .border-bottomleft::before {  
 transform: scaleY(.33333);  
 }  
 .border-right::before,  
 .border-left::before,  
 .border-rightleft::before,  
 .border-rightleft::after,  
 .border-topleft::after,  
 .border-rightbottom::before,  
 .border-topright::after,  
 .border-bottomleft::after {  
 transform: scaleX(.33333);  
 }  
}

移动端项目300毫秒点击延迟问题,在某些浏览器上使用click时间会有延迟

引入fastclick库解决问题 --save表示线上测试都使用fastclick



npm run start 重启服务

引入



iconfont技术,登录官方网站需要注册

npm install stylus --save

npm install stylus-loader --save

# React Native

# MUI框架

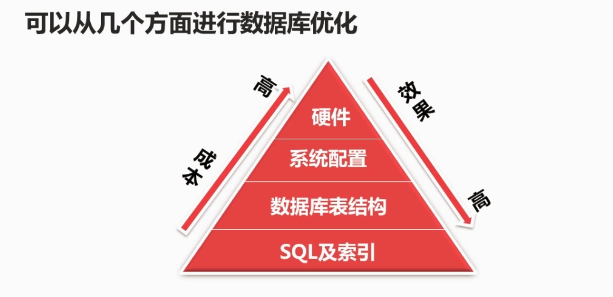
环境安装：

1. 安装node.js环境
2. 通过nodejs安装后在控制条中使用 npm install -g react-native-cli 命令安装

# mysql优化

Mysql数据库优化

1. 数据库优化的目的：
   1. 避免出现页面访问错误
      1. 由于数据库连接timeout产生页面5xx错误。
      2. 由于慢查询造成页面无法加锁
      3. 由于阻塞造成数据无法提交
2. 可以从几个方面进行数据库优化



SQL及索引优化

如何发现有问题的SQL?

使用Mysql慢查询日志对有效率问题的SQL进行监控

show variables like 'slow\_query\_log'

--查看是否开启慢查询日志

set global shlow\_query\_log\_file='/home/mysql/sql\_log/mysql-slow.log'

--指定慢查询日志所存储的位置

set global log\_queries\_not\_using\_indexes = on; off

--是否将没有使用索引的sql记录在慢查询日志中

set global log\_query\_time=1;

--将超过多少秒的查询记录在慢查询日志中

慢查日志的存储格式

1.执行SQL的时间

Time:140606 12:30:17

2.执行SQL的主机信息

User@Host:root[root] @ localhost []

3.SQL执行信息

Query\_time:0.000031 Lock\_time:0.000000 Rows\_sent:0 Rows\_examined:0

SQL执行时间

SET timestamp = 1402029017

SQL的内容

select CONCAT(‘storage engine:’,@@storage\_engine) as INFO;

Mysql慢查询日志分析工具之mysqldumpslow

1. mysqldumpslow
2. pt-query-digest
   1. 输出到文件
      1. pt-query-digest slow-log > slow\_log.report

输出到数据库表

pt-query-digest slow.log -review \

h=127.0.0.1,D=test,p=root,P=3306,u=root,t=query\_review \

--create-reviewtable \

--review-history t=hostname\_slow

通过explain查询和分析SQL的执行计划（a死破len）

explain返回各列的含义

table:显示这一行的数据是关于那张表的

type:这是重要的列,显示连接使用了哪种类型,从最好到最差的连接类型为const/eq\_reg/ref/range/index/all

possible\_keys:显示可能应用在这张表中的索引,如果为空,没有可能的所有。

key:实际使用的索引,如果为null,则没有使用索引

key\_len:使用的索引长度,在不损失精确性的情况下,长度越短越好

ref:显示索引的那一列使用了,如果可能的话,是一个常数

rows:MYSQL认为必须检查的用来返回请求数据的行数

extra列需要注意的返回值

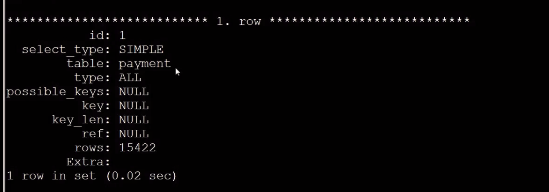
Using filesort:看到这个的时候,查询需要优化了，Mysql需要进行额外的步骤来发现如何反对返回的行排序,它根据连接类型以及存储排序键值和匹配条件的全部行的行指针来排序全部行

Using temporary 看到这个的时候，查询需要优化了,这里mysql需要创建一个临时表来存储结果,这通常发生在对不同的列集进行order by上,而不是group by上

Count() 和Max()的优化

查询之后支付时间-优化max()函数

select max(payment\_date) from payment;



优化步骤：

create index idx\_paydate on payment(payment\_date);

在payment\_data上建立索引

Count()优化

在一条sql中同时查出2006年和2007年电影的数量-优化count函数

错误的方式:

select count(release\_year=’2006’ or release\_year=’2007’) from film;

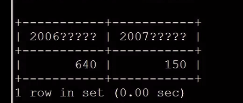
无法分开计算2006和2007年的电影数量

select count(\*) from film where release\_year = ‘2006’ and release\_year = ‘2007’;

release\_year不可能同事为2006和2007,因此上有逻辑错误

正确的方式:

select count(release\_year=’2006’ or null) as ‘2006年电影数量’ ，count(release\_year = ‘2007’ or null) as ‘2007年电影数量’ from filem



注意:count(\*) 和 count(id) 的数量不同count(\*)包含空值

子查询语句的优化

通常情况下,需要把子查询优化为join查询,但在优化是需要注意关联键是否有一对多的关系,要注意重复数据

（查询sandra出验的所有电影）

explain select title,release\_year,LENGTH from film

where film\_id in(select film\_id from film\_actor where actor\_id in(select actor\_id from actor where first\_name = ‘sandra’))

优化group by查询

limit 查询的优化

limit常用于分页查询,时常会伴随order by从语句使用,因此大多时候会使用Filesorts这样会造成大量的IO问题

select film\_id , description from sakila.film order by title limit 50,5

优化步骤1:使用索引的列或主键进行order by操作

select film\_id,description from sakila.film order by film\_id limit 50,5;

优化步骤2:记录上一次返回的主键,在下一次查询时使用主键过滤（缺点:主键必须顺序排序）-这样一般都在自增长字段中如没有顺序自增长可以创建一列专做自增长使用

select film\_id,description from sakila.film where film\_id > 55 and film\_id <= 60 order by film\_id limit 1,5;

SQL及索引优化

如何选择合适的列建立索引?

1. 在where从句,group by 从句,order by从句,on 从句中出现的列
2. 索引字段越小越好
3. 离散度大的列放到联合索引的前面
   1. select \* from payment where staff\_id = 2 and customer\_id = 584
   2. 是index(sftaff\_id,customer\_id)好,还是index(customer\_id,staff\_id)好?

使用count方式进行数量统计,统计数量越高的离散度越高

如果使用联合索引

由于customer\_id的离散度更大,所以应该使用index(customer\_id,staff\_id)

索引的维护及优化 -- 重复及冗余索引

重复索引是指相同的列以相同的顺序建立的同类型的索引,如下表中primary key和ID列上的索引就是重复索引

create table test(

id int not null primay key,

name varchar(10) not null,

titile varchar(50) not null,

unique(id)

)engine = innodb;

数据结构优化

选择合适的数据类型

数据类型的选择,重点在于合适二字,如何确定选择的数据类型是否合适?

1. 使用可以存下你的数据的最小的数据类型。
2. 使用简单的数据类型。int要比varchar类型在mysql处理上简单。
3. 尽可能的使用not null 定义字段。
4. 尽量少用text类型,非用不可是最好考虑分表

选择合适的数据类型

使用int来存储日期时间,利用from\_unixtime(),unix\_timestamp() 两个函数来进行转换

create table test(

id int auto\_increment not null,

timestr int,

primary key(id)

)

insert into test(timestr) values(unix\_timestamp(‘2014-05-01 12:33:22’));

select from\_unixtime(timestr) from test;

使用bigint来存储ip地址,利用inet\_aton(),inet\_ntoa()两个函数来转换

create table sessions(id int auto\_increment not null ,ipaddress bigint,primary key(id))

insert into sessions(ipaddress) values(inet\_auto(‘192.168.0.1’))

select inet\_ntoa(ipaddress) from sessions;

表的范式化和反范式化

# node.js

# mvan

jar包打到maven仓库指令

|  |
| --- |
| mvn install:install-file -Dfile=D:\EasyCaptcha-1.0.0.jar -DgroupId=com.wf.captcha -DartifactId=EasyCaptcha -Dversion=0.0.1-SNAPSHOT -Dpackaging=jar |

# 微信小程序

微信布局:

display:flex; 在css样式中定义为flex布局

flex-direction: 中可以使用

row 默认从做到有排序

row-reverse 从右到做排序

column 从上到下排序

column-reverse 从下到上排序

flex-wrap:

nowrap默认不换行

wrap 换行

wrap-reverse 逆向换行

justify-content

flex-start 默认左对齐

flex-end 默认右对齐

center 居中对齐

space-around 在成员元素周围包含空白

space-between 在成员元素之间留空白

flex布局

flex容器成员的属性

order：通过数字对flex容器内部的成员设置显示顺序

flex:配置每个成员元素所占行级的显示比例