Java基础知识：

面向对象、异常、泛型、多线程、接口、内部类、集合、

前端：HTML/HTML5、CSS/CSS3、Javascript、jQuery、RequireJS、AngularJS、Vue

后端：Java、Struts2/Spring MVC、JPA/Mybatis、Spring Boot

安全：Shiro、Spring Security

中间件：Dubbo、ActiveMQ/RabbitMQ、Nginx

数据库：MySQL/Oracle、Redis、MongoDB

搜索引擎：Lucene、Elasticsearch

代码管理：svn、git

构建工具：Maven、Jenkins

规则引擎；

RBAC模型：权限管理；

spring boot :定时任务、批处理、Eureka、Thymeleaf、Jenkins；

Java:垃圾回收机制、类加载机制；

LDAP

数据库：JPA

前端：electron

Mybatis中标签association和collection两个标签能同时使用吗？

**工具：[Notion数字笔记使用教程](http://www.baidu.com/link?url=Jxv7rUU2NY-n6FBJedjl8OUg5Q17bt21sFAqe0VLMKuwmw7cjbb5IhxHkTWr_bcdUhXATJlwxu3W3vz9gWVnzOcUynuyqqu8aUhRcn114NG" \t "_blank)**

Canva；

# 后端

## 面向对象的特征（OO Object–Oriented）

Object–Oriented Programming[面向对象](https://www.baidu.com/s?wd=%E9%9D%A2%E5%90%91%E5%AF%B9%E8%B1%A1&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao" \t "https://zhidao.baidu.com/question/_blank)编程  
Object–Oriented Analysis [面向对象](https://www.baidu.com/s?wd=%E9%9D%A2%E5%90%91%E5%AF%B9%E8%B1%A1&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao" \t "https://zhidao.baidu.com/question/_blank)分析  
Object–Oriented Design [面向对象](https://www.baidu.com/s?wd=%E9%9D%A2%E5%90%91%E5%AF%B9%E8%B1%A1&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao" \t "https://zhidao.baidu.com/question/_blank)设计

抽象。抽象就是忽略一个主题中与当前目标无关的那些方面，以便更充分地注意与当前目标有关的方面。抽象不打算了解全部问题，而是选择其中的一部分，暂时不用部分细节。抽象包括两个方面：过程抽象；数据抽象。

继承。继承是一种连接类的层次模型，并且允许和鼓励类的重用，它提供了一种明确表述共性的方法。对象的一个新类可以从现有的类中派生，这个过程称为类的继承。新类继承了原始类的特性，新类称为原始类的派生类（子类），而原始类称为新类的基类（父类）。派生类可以从它的基类那里继承方法和实例变量，并且派生类可以修改或者增加新的方法使之更适合特殊的需要。

封装。封装是指将客观事物抽象成类，每个类对自身的数据和方法实行保护。类可以把自己的数据和方法只让可信的类或者对象操作，对不可信的进行信息隐藏。

多态。多态是指允许不同类的对象对同一消息做出响应。多态包括参数化多态和包含多态。多态性语言具有灵活、抽象、行为共享、代码共享等优势，很好地解决了应用程序函数同名问题。Java中实现多态主要是方法的重载和方法的覆盖。

## 面向过程（Procedure Oriented）

## 函数式编程（Functional Programming）

### 定义

简单说，"函数式编程"是一种["编程范式"](http://en.wikipedia.org/wiki/Programming_paradigm" \t "https://www.cnblogs.com/feng9exe/p/_blank)（programming paradigm），也就是如何编写程序的方法论。

它属于["结构化编程"](http://en.wikipedia.org/wiki/Structured_programming" \t "https://www.cnblogs.com/feng9exe/p/_blank)的一种，主要思想是把运算过程尽量写成一系列嵌套的函数调用。

### 特点

#### 函数是"第一等公民"

所谓["第一等公民"](http://en.wikipedia.org/wiki/First-class_function" \t "https://www.cnblogs.com/feng9exe/p/_blank)（first class），指的是函数与其他数据类型一样，处于平等地位，可以赋值给其他变量(var result = subtract(multiply(add(1,2), 3), 4);)，也可以作为参数，传入另一个函数，或者作为别的函数的返回值。

举例来说，下面代码中的print变量就是一个函数，可以作为另一个函数的参数。

　　var print = function(i){ console.log(i);};  
  
　　[1,2,3].forEach(print);  //print为函数

#### 只用"表达式"，不用"语句"()

"表达式"（expression）是一个单纯的运算过程，总是有返回值；

"语句"（statement）是执行某种操作(更多的是逻辑语句。)，没有返回值。

函数式编程要求，只使用表达式，不使用语句。也就是说，每一步都是单纯的运算，而且都有返回值。

原因是函数式编程的开发动机，一开始就是为了处理运算（computation），不考虑系统的读写（I/O）。"语句"属于对系统的读写操作，所以就被排斥在外。

当然，实际应用中，不做I/O是不可能的。因此，编程过程中，函数式编程只要求把I/O限制到最小，不要有不必要的读写行为，保持计算过程的单纯性。

#### 没有"副作用"(单一职责，只做一件事，避免耦合关联。)

所谓["副作用"](http://en.wikipedia.org/wiki/Side_effect_(computer_science)" \t "https://www.cnblogs.com/feng9exe/p/_blank)（side effect），指的是函数内部与外部互动（最典型的情况，就是修改全局变量的值），产生运算以外的其他结果。

函数式编程强调没有"副作用"，意味着函数要保持独立，所有功能就是返回一个新的值，没有其他行为，尤其是不得修改外部变量的值。

#### 不修改状态

上一点已经提到，函数式编程只是返回新的值，不修改系统变量。因此，不修改变量，也是它的一个重要特点。

在其他类型的语言中，变量往往用来保存"状态"（state）。不修改变量，意味着状态不能保存在变量中。函数式编程使用参数保存状态，最好的例子就是递归。下面的代码是一个将字符串逆序排列的函数，它演示了不同的参数如何决定了运算所处的"状态"。

　　function reverse(string) {

　　　　if(string.length == 0) {

　　　　　　return string;

　　　　} else {

　　　　　　return reverse(string.substring(1, string.length)) + string.substring(0, 1);

　　　　}

　　}

由于使用了递归，函数式语言的运行速度比较慢，这是它长期不能在业界推广的主要原因。

#### 引用透明

引用透明（Referential transparency），指的是函数的运行不依赖于外部变量或"状态"，只依赖于输入的参数，任何时候只要参数相同，引用函数所得到的返回值总是相同的。

有了前面的第三点和第四点，这点是很显然的。其他类型的语言，函数的返回值往往与系统状态有关，不同的状态之下，返回值是不一样的。这就叫"引用不透明"，很不利于观察和理解程序的行为。

### 意义

#### 代码简洁，开发快速

函数式编程大量使用函数，减少了代码的重复，因此程序比较短，开发速度较快。

Paul Graham在[《黑客与画家》](http://www.ruanyifeng.com/docs/pg/" \t "https://www.cnblogs.com/feng9exe/p/_blank)一书中[写道](http://www.ruanyifeng.com/blog/2010/10/why_lisp_is_superior.html" \t "https://www.cnblogs.com/feng9exe/p/_blank)：同样功能的程序，极端情况下，Lisp代码的长度可能是C代码的二十分之一。

如果程序员每天所写的代码行数度基本相同，这就意味着，"C语言需要一年时间完成开发某个功能，Lisp语言只需要不到三星期。反过来说，如果某个 新功能，Lisp语言完成开发需要三个月，C语言需要写五年。"当然，这样的对比故意夸大了差异，但是"在一个高度竞争的市场中，即使开发速度只相差两三倍，也足以使得你永远处在落后的位置。"

#### 接近自然语言，易于理解

函数式编程的自由度很高，可以写出很接近自然语言的代码。

前文曾经将表达式(1 + 2) \* 3 - 4，写成函数式语言：

　　subtract(multiply(add(1,2), 3), 4)

对它进行变形，不难得到另一种写法：

　　add(1,2).multiply(3).subtract(4)

这基本就是自然语言的表达了。再看下面的代码，大家应该一眼就能明白它的意思吧：

　　merge([1,2],[3,4]).sort().search("2")

因此，函数式编程的代码更容易理解。

#### 更方便的代码管理

函数式编程不依赖、也不会改变外界的状态，只要给定输入参数，返回的结果必定相同。因此，每一个函数都可以被看做独立单元，很有利于进行单元测试（unit testing）和除错（debugging），以及模块化组合。

#### 易于"并发编程"

函数式编程不需要考虑"死锁"（deadlock），因为它不修改变量，所以根本不存在"锁"线程的问题。不必担心一个线程的数据，被另一个线程修改，所以可以很放心地把工作分摊到多个线程，部署"并发编程"（concurrency）。

请看下面的代码：

　　var s1 = Op1();

　　var s2 = Op2();

　　var s3 = concat(s1, s2);

由于s1和s2互不干扰，不会修改变量，谁先执行是无所谓的，所以可以放心地增加线程，把它们分配在两个线程上完成。其他类型的语言就做不到这一 点，因为s1可能会修改系统状态，而s2可能会用到这些状态，所以必须保证s2在s1之后运行，自然也就不能部署到其他线程上了。

多核CPU是将来的潮流，所以函数式编程的这个特性非常重要。

#### 代码的热升级

函数式编程没有副作用，只要保证接口不变，内部实现是外部无关的。所以，可以在运行状态下直接升级代码，不需要重启，也不需要停机。[Erlang](http://en.wikipedia.org/wiki/Erlang_(programming_language)" \t "https://www.cnblogs.com/feng9exe/p/_blank)语言早就证明了这一点，它是瑞典爱立信公司为了管理电话系统而开发的，电话系统的升级当然是不能停机的。

## JDK底层原理

### 数据结构基础

#### 3\*0.1 == 0.3 返回true 还是 false？

本机运行结果参考：3\*0.1 =0.30000000000000004；

false，因为有些浮点数不能完全精确的表示出来。 [浮点数精度](https://www.baidu.com/s?wd=%E6%B5%AE%E7%82%B9%E6%95%B0%E7%B2%BE%E5%BA%A6&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1YYm1-bujw9nWNWm1mdn1Dk0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EPHmznjfkPHf3" \t "https://blog.csdn.net/wuhan_yking/article/details/_blank)默认为6位。

#### “a==b”和”a.equals(b)”有什么区别？

==  比较的是栈内的地址;

equals()  比较的是堆中的内容;

如果 a 和 b 都是对象，则 a==b 是比较两个对象的引用，只有当 a 和 b 指向的是堆中的同一个对象才会返回 true，而 a.equals(b) 是进行逻辑比较，当内容相同时，返回true，所以通常需要重写该方法来提供逻辑一致性的比较。例如，String 类重写 equals() 方法，所以可以用于两个不同对象，但是包含的字母相同的比较。

#### float f=3.4;是否正确?

false。精度不准确,应该用强制类型转换，如下所示：float f=(float)3.4 或float f = 3.4f

在java里面，没小数点的默认是int,有小数点的默认是 double;

3.4是双精度数，将双精度型（double）赋值给浮点型（float）属于下转型（down-casting，也称为窄化）会造成精度损失，因此需要强制类型转换float f =(float)3.4; 或者写成float f =3.4F;。

#### Math.round(11.5) 等于多少? Math.round(-11.5)等于多少?

答：Math.round(11.5)的返回值是12，Math.round(-11.5)的返回值是-11。四舍五入的原理是在参数上加0.5然后进行下取整。

#### “static”关键字是什么意思？Java中是否可以覆盖(override)一个private或者是static的方法？

“static”关键字表明一个成员变量或者是成员方法可以在没有所属的类的实例变量的情况下被访问。

Java中static方法不能被覆盖，因为方法覆盖是基于运行时动态绑定的，而static方法是编译时静态绑定的。static方法跟类的任何实例都不相关，所以概念上不适用。

#### 是否可以在static环境中访问非static变量？

static变量在Java中是属于类的，它在所有的实例中的值是一样的。当类被Java虚拟机载入的时候，会对static变量进行初始化。如果你的代码尝试不用实例来访问非static的变量，编译器会报错，因为这些变量还没有被创建出来，还没有跟任何实例关联上。

#### 自动装箱和自动拆箱的作用？

#### Overload和Override的区别。Overloaded的方法是否可以改变返回值的类型?

方法的重写Overriding和重载Overloading是Java多态性的不同表现。重写Overriding是父类与子类之间多态性的一种表现，重载O verloading是一个类中多态性的一种表现。如果在子类中定义某方法与其父类有相同的名称和参数，我们说该方法被重写(Overriding)。子类的对象使用这个方法时，将调用子类中的定义，对它而言，父类中的定义如同被”屏蔽”了。如果在一个类中定义了多个同名的方法，它们或有不同的参数个数或有不同的参数类型，则称为方法的重载(Overloading)。Overloaded的方法是可以改变返回值的类型。

#### 什么是值传递和引用传递？

对象被值传递，意味着传递了对象的一个副本。因此，就算是改变了对象副本，也不会影响源对象的值。  
对象被引用传递，意味着传递的并不是实际的对象，而是对象的引用。因此，外部对引用对象所做的改变会反映到所有的对象上。

#### 同步方法和同步代码块的区别是什么？

在Java语言中，每一个对象有一把锁。线程可以使用synchronized关键字来获取对象上的锁。synchronized关键字可应用在方法级别(粗粒度锁：这里的锁对象可以是This)或者是代码块级别(细粒度锁：这里的锁对象就是任意对象)。

#### Java中double转BigDecimal的注意事项

|  |
| --- |
| 先上结论：不要直接用double变量作为构造BigDecimal的参数。  线上有这么一段Java代码逻辑：  1，接口传来一个JSON串，里面有个数字：57.3。  2，解析JSON并把这个数字保存在一个float变量。  3，把这个float变量赋值给一个 BigDecimal对象，用的是BigDecimal的double参数的构造：     new BigDecimal(double val)  4，把这个BigDecimal保存到MySQL数据库，字段类型是decimal(15,2)。  这段代码逻辑在线上跑了好久了，数据库保存的值是57.3也没什么问题，但是在今天debug的时候发现，第三步的BigDecimal对象保存的值并不是57.3，而是57.299999237060546875，很明显，出现了精度的问题。  至于数据库最终保存了正确的57.3完全是因为字段类型设置为2位小数，超过2位小数就四舍五入，所以才得到了正确的结果，相当于MySQL给我们把这个精度问题掩盖了。    总觉得这是个坑，所以研究了一下相关的知识。  首先是BigDecimal的double参数构造，在官方JDK文档中对这个构造是这么描述的：  public BigDecimal(double val)  Translates a double into a BigDecimal which is the exact decimal representation of the double's binary floating-point value. The scale of the returned BigDecimal is the smallest value such that (10scale × val) is an integer.  Notes:  The results of this constructor can be somewhat unpredictable. One might assume that writing new BigDecimal(0.1) in Java creates a BigDecimal which is exactly equal to 0.1 (an unscaled value of 1, with a scale of 1), but it is actually equal to 0.1000000000000000055511151231257827021181583404541015625. This is because 0.1 cannot be represented exactly as a double (or, for that matter, as a binary fraction of any finite length). Thus, the value that is being passed in to the constructor is not exactly equal to 0.1, appearances notwithstanding.  The String constructor, on the other hand, is perfectly predictable: writing new BigDecimal("0.1") creates a BigDecimal which is exactly equal to 0.1, as one would expect. Therefore, it is generally recommended that the String constructor be used in preference to this one.  When a double must be used as a source for a BigDecimal, note that this constructor provides an exact conversion; it does not give the same result as converting the double to a String using the Double.toString(double) method and then using the BigDecimal(String) constructor. To get that result, use the static valueOf(double) method.  Parameters:  val - double value to be converted to BigDecimal.  Throws:  NumberFormatException - if val is infinite or NaN.  翻译一下大概是这样的：  1，BigDecimal(double val)构造，用double当参数来构造一个BigDecimal对象。  2，但是这个构造不太靠谱(unpredictable)，你可能以为BigDecimal(0.1)就是妥妥的等于0.1，但是你以为你以为的就是你以为的？还真不是，BigDecimal(0.1)这货实际上等于0.1000000000000000055511151231257827021181583404541015625，因为准确的来说0.1本身不能算是一个double（其实0.1不能代表任何一个定长二进制分数）。  3，BigDecimal(String val)构造是靠谱的，BigDecimal(“0.1”)就是妥妥的等于0.1，推荐大家用这个构造。  4，如果你非得用一个double变量来构造一个BigDecimal，没问题，我们贴心的提供了静态方法valueOf(double)，这个方法跟new Decimal(Double.toString(double))效果是一样的。  说白了就是别直接拿double变量做参数，最好使用String类型做参数或者使用静态方法valueOf(double)，我写了个例子试了一下：           public static void main(String[] args) {                       float a=57.3f;                       BigDecimal decimalA=new BigDecimal(a);                       System.out.println(decimalA);                           double b=57.3;                       BigDecimal decimalB=new BigDecimal(b);                       System.out.println(decimalB);                           double c=57.3;                       BigDecimal decimalC=new BigDecimal(Double.toString(c));                       System.out.println(decimalC);                           double d=57.3;                       BigDecimal decimalD=BigDecimal.valueOf(d);                       System.out.println(decimalD);             }    输出结果：  57.299999237060546875  57.2999999999999971578290569595992565155029296875  57.3  57.3 |

### 常见异常

#### ArithmeticExecption：算数异常类

#### NullPointerException：空指针异常类型

#### ClassCastException：类型强制转换类型

#### NegativeArrayException：数组负下标异常

#### ArrayIndexOutOfBoundsException：数组下标越界异常

#### SecturityException：违背安全原则异常

#### EOFException：文件已结束异常

ObjectInputStream可以自己判断流是否到达末尾，但是它无法告诉我们，我们不能替代他们读取这个标记，不然ObjectInputStream将识别不了下一个内容的实际类型。

对于这种异常的一般解决方法就是，捕获，可以记录日志，也可以不做处理，捕获异常以后，把之前读到的数据进行后续的处理就可以了，因为那就是所以的数据。还有就是如果打算记录日志，不要把它的堆栈信息打印出来，容易给人以错觉。毕竟EOFException实质上只是一个消息而已。

#### FileNotFoundException：文件未找到异常

#### SQLException：操作数据库异常

#### IOException：输入输出异常

#### NoSuchMethodException：方法未找到异常

#### IndexOutOfBoundsExecption：下标越界异常

#### SystemException：系统异常

#### NegativeArraySizeException：创建一个大小为负数的数组错误异常

#### NumberFormatException：数据格式异常

#### SecurityException：安全异常

#### UnsupportedOperationException：不支持的操作异常

#### NetworkOnMainThreadException  ：网络操作在主线程异常

#### IllegalStateException （extends RuntimeException ， 父类：请求状态异常

在不合理或不正确时间内唤醒一方法时出现的异常信息。换句话说，即 Java 环境或 Java 应用不满足请求操作）

#### IllegalArgumentException：非法参数异常

#### IllegalAccessException：非法权限异常

#### InstantiationException 无法实例化异常

没有相关构造器；

#### HttpHostConnectException：网络请求异常

#### ViewRootImpl$CalledFromWrongThreadException：子线程Thread更新UI view 异常

#### SSLExceptionero：证书不匹配的主机名异常

#### InvocationTargetException：反射Method.invoke(obj, args...)方法抛出异常

#### EventBusException：EventBus使用异常

#### ZeroException：参数不能小于0异常

#### [javax.management.InstanceNotFoundException](https://www.cnblogs.com/gradven/p/6323195.html)

## Java集合

### Java集合类框架的基本接口有哪些？

Java集合类提供了一套设计良好的支持对一组对象进行操作的接口和类。Java集合类里面最基本的接口有：

• Collection：代表一组对象，每一个对象都是它的子元素。

• Set：不包含重复元素的Collection。

• List：有顺序的collection，并且可以包含重复元素。

• Map：可以把键(key)映射到值(value)的对象，键不能重复。

### 返回空集合 （实现函数式借口的空集合？）

Collections.emptyList()；

如果返回Lists.newArrayList();或者new ArrayList();这会新建一个对象，而这个对象很可能是没必要的，这样白白浪费性能。

解决方法当然也有，可以用Collections.emptyList();这个方法返回一个空集合，并不会新建对象，而是返回

public static final List EMPTY\_LIST = new EmptyList<>();

这个变量。

当然这也有缺点，如果调用方只是遍历，这没什么不会报错，但是如果要新增，删除里面的元素那就会报错，

那么你可能想为什么，原因就是代码里直接写死了调用时报错，那么为什么要这样写呢？

原因也很简单，如果多个线程对这个集合增删，那么调用方就全乱了，所以采用了直接报错，快速失败的方法

来解决问题。

总结：

返回null，返回new ArrayList<>(),返回EMPTY\_LIST 。

null肯定是不推荐的，那么是新建一个List还是返回空List呢？

这要根据接口的性能要求，如果性能要求高返回EMPTY\_LIST，否则新建一个对象。

## 什么是迭代器(Iterator)？

Iterator接口提供了很多对集合元素进行迭代的方法。每一个集合类都包含了可以返回迭代器实例的迭代方法。迭代器可以在迭代的过程中删除底层集合的元素。

## Iterator和ListIterator的区别是什么？

下面列出了他们的区别：

• Iterator可用来遍历Set和List集合，但是ListIterator只能用来遍历List。

• Iterator对集合只能是前向遍历，ListIterator既可以前向也可以后向。

• ListIterator实现了Iterator接口，并包含其他的功能，比如：增加元素，替换元素，获取前一个和后一个元素的索引，等等。

## Java中的HashMap的工作原理是什么？

Java中的HashMap是以键值对(key-value)的形式存储元素的。HashMap需要一个hash函数，它使用hashCode()和equals()方法来向集合/从集合添加和检索元素。当调用put()方法的时候，HashMap会计算key的hash值，然后把键值对存储在集合中合适的索引上。如果key已经存在了，value会被更新成新值。HashMap的一些重要的特性是它的容量(capacity)，负载因子(load factor)和扩容极限(threshold resizing)。

## HashMap和Hashtable有什么区别？

• HashMap和Hashtable都实现了Map接口，因此很多特性非常相似。但是，他们有以下不同点：

• HashMap允许键和值是null，而Hashtable不允许键或者值是null。

• Hashtable是同步的，而HashMap不是。因此，HashMap更适合于单线程环境，而Hashtable适合于多线程环境。

• HashMap提供了可供应用迭代的键的集合，因此，HashMap是快速失败的。另一方面，Hashtable提供了对键的列举(Enumeration)。

o 一般认为Hashtable是一个遗留的类。

## BigInteger 与 BigDecimal区别，及BigDecimal详解

一般来说，BigInteger用的不是很多，BigDecimal用的稍微多一点，就比如说JDBC中，如果一个字段的数据库类型是Number, 那么getObject().getClass()的结果是java.math.BigDecimal。

除了容量大之外，BigInteger还封装了一些常见的操作，比如+-\*/的基本操作，还有绝对值，相反数，最大公约数，是否是质数等等的运算。

BigDecimal的实现利用到了BigInteger, 所不同的是BigDecimal加入了小数位的概念，比如BigDecimal d = new BigDecimal(new BigInteger(ib)，5);5表示的是5个小数位。BigDecimal可以用来做超大的浮点数的运算，比如+-\*/的运算，其中除法运算是最复杂的，因为商的位数还有除不断的情况下末位小数点的处理都是需要考虑的。

add，substract，multiple，divide和remainder 加、减、乘、除；

java.util包中的Scanner类实现了nextBigInteger()和nextBigDecimal()方法，可以用来读入控制台输入的BigInteger和BigDecimal。

## Java双括号初始化

|  |
| --- |
| Test test = new Test(){  {  num = 2;  s = "aaa";  }  };  test.runTest(); |

这种初始化方式实际上是利用了匿名类的特性。外层的括号其实是创建了一个匿名的内部类，而内层的括号是匿名类的构造函数，所以可以直接插入初始化代码。

括号中的代码晚于内部构造函数执行。

有什么利弊

尽管双括号初始化方式让编程的可读性变得简单，但是因为要创建一个匿名的内部类，匿名内部类会持有外部类的引用，有造成内存泄露的风险。

dos窗口查看字节码文件： javap -c D:\Hello.class

## （一）类装载机制

类从被加载到虚拟机内存中开始，到卸载出内存为止，它的整个生命周期包括：加载（Loading）、验证（Verification）、准备(Preparation)、解析(Resolution)、初始化(Initialization)、使用(Using)和卸载(Unloading)7个阶段。其中准备、验证、解析3个部分统称为连接（Linking）。如图所示。



加载、验证、准备、初始化和卸载这5个阶段的顺序是确定的，类的加载过程必须按照这种顺序按部就班地开始，而解析阶段则不一定：它在某些情况下可以在初始化阶段之后再开始，这是为了支持Java语言的运行时绑定（也称为动态绑定或晚期绑定）。以下陈述的内容都已HotSpot为基准。

### 加载 类加载的双亲委派机制？

在加载阶段（可以参考java.lang.ClassLoader的loadClass()方法），虚拟机需要完成以下3件事情：

通过一个类的全限定名来获取定义此类的二进制字节流（并没有指明要从一个Class文件中获取，可以从其他渠道，譬如：网络、动态生成、数据库等）；

将这个字节流所代表的静态存储结构转化为方法区的运行时数据结构；

在内存中生成一个代表这个类的java.lang.Class对象，作为方法区这个类的各种数据的访问入口；

加载阶段和连接阶段（Linking）的部分内容（如一部分字节码文件格式验证动作）是交叉进行的，加载阶段尚未完成，连接阶段可能已经开始，但这些夹在加载阶段之中进行的动作，仍然属于连接阶段的内容，这两个阶段的开始时间仍然保持着固定的先后顺序。

### 验证

验证是连接阶段的第一步，这一阶段的目的是为了确保Class文件的字节流中包含的信息符合当前虚拟机的要求，并且不会危害虚拟机自身的安全。  
验证阶段大致会完成4个阶段的检验动作：

文件格式验证：验证字节流是否符合Class文件格式的规范；例如：是否以魔术0xCAFEBABE开头、主次版本号是否在当前虚拟机的处理范围之内、常量池中的常量是否有不被支持的类型。

元数据验证：对字节码描述的信息进行语义分析（注意：对比javac编译阶段的语义分析），以保证其描述的信息符合Java语言规范的要求；例如：这个类是否有父类，除了java.lang.Object之外。

字节码验证：通过数据流和控制流分析，确定程序语义是合法的、符合逻辑的。

符号引用验证：确保解析动作能正确执行。

验证阶段是非常重要的，但不是必须的，它对程序运行期没有影响，如果所引用的类经过反复验证，那么可以考虑采用-Xverifynone参数来关闭大部分的类验证措施，以缩短虚拟机类加载的时间。

### 准备

准备阶段是正式为类变量分配内存并设置类变量初始值的阶段，这些变量所使用的内存都将在方法区中进行分配。这时候进行内存分配的仅包括类变量（被static修饰的变量），而不包括实例变量，实例变量将会在对象实例化时随着对象一起分配在堆中。其次，这里所说的初始值“通常情况”下是数据类型的零值，假设一个类变量的定义为：

|  |
| --- |
| public static int value=123; |

|  |  |
| --- | --- |
|  |  |

那变量value在准备阶段过后的初始值为0而不是123.因为这时候尚未开始执行任何java方法，而把value赋值为123的putstatic指令是程序被编译后，存放于类构造器()方法之中，所以把value赋值为123的动作将在初始化阶段才会执行。  
至于“特殊情况”是指：public static final int value=123，即当类字段的字段属性是ConstantValue时，会在准备阶段初始化为指定的值，所以标注为final之后，value的值在准备阶段初始化为123而非0.

### 解析

解析阶段是虚拟机将常量池内的符号引用替换为直接引用的过程。解析动作主要针对类或接口、字段、类方法、接口方法、方法类型、方法句柄和调用点限定符7类符号引用进行。

### 初始化

类初始化阶段是类加载过程的最后一步，到了初始化阶段，才真正开始执行类中定义的java程序代码。在准备极端，变量已经付过一次系统要求的初始值，而在初始化阶段，则根据程序猿通过程序制定的主管计划去初始化类变量和其他资源，或者说：初始化阶段是执行类构造器<clinit>()方法的过程.  
<clinit>()方法是由编译器自动收集类中的所有类变量的赋值动作和静态语句块static{}中的语句合并产生的，编译器收集的顺序是由语句在源文件中出现的顺序所决定的，静态语句块只能访问到定义在静态语句块之前的变量，定义在它之后的变量，在前面的静态语句块可以赋值，但是不能访问。如下：

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | | 1  2  3  4  5  6  7  8  9 | public class Test  {      static      {          i=0;          System.out.println(i);//这句编译器会报错：Cannot reference a field before it is defined（非法向前应用）      }      static int i=1;  } | |

<clinit>()方法与实例构造器<init>()方法不同，它不需要显示地调用父类构造器，虚拟机会保证在子类<init>()方法执行之前，父类的<clinit>()方法方法已经执行完毕，回到本文开篇的举例代码中，结果会打印输出：SSClass就是这个道理。  
由于父类的<clinit>()方法先执行，也就意味着父类中定义的静态语句块要优先于子类的变量赋值操作。  
<clinit>()方法对于类或者接口来说并不是必需的，如果一个类中没有静态语句块，也没有对变量的赋值操作，那么编译器可以不为这个类生产<clinit>()方法。  
接口中不能使用静态语句块，但仍然有变量初始化的赋值操作，因此接口与类一样都会生成<clinit>()方法。但接口与类不同的是，执行接口的<clinit>()方法不需要先执行父接口的<clinit>()方法。只有当父接口中定义的变量使用时，父接口才会初始化。另外，接口的实现类在初始化时也一样不会执行接口的<clinit>()方法。  
虚拟机会保证一个类的<clinit>()方法在多线程环境中被正确的加锁、同步，如果多个线程同时去初始化一个类，那么只会有一个线程去执行这个类的<clinit>()方法，其他线程都需要阻塞等待，直到活动线程执行<clinit>()方法完毕。如果在一个类的<clinit>()方法中有好事很长的操作，就可能造成多个线程阻塞，在实际应用中这种阻塞往往是隐藏的。

|  |
| --- |
| package jvm.classload;    public class DealLoopTest  {      static class DeadLoopClass      {          static          {              if(true)              {                  System.out.println(Thread.currentThread()+"init DeadLoopClass");                  while(true)                  {                  }              }          }      }        public static void main(String[] args)      {          Runnable script = new Runnable(){              public void run()              {                  System.out.println(Thread.currentThread()+" start");                  DeadLoopClass dlc = new DeadLoopClass();                  System.out.println(Thread.currentThread()+" run over");              }          };            Thread thread1 = new Thread(script);          Thread thread2 = new Thread(script);          thread1.start();          thread2.start();      }  } |

运行结果：（即一条线程在死循环以模拟长时间操作，另一条线程在阻塞等待）

|  |
| --- |
| Thread[Thread-0,5,main] start  Thread[Thread-1,5,main] start  Thread[Thread-0,5,main]init DeadLoopClass |

需要注意的是，其他线程虽然会被阻塞，但如果执行<clinit>()方法的那条线程退出<clinit>()方法后，其他线程唤醒之后不会再次进入<clinit>()方法。同一个类加载器下，一个类型只会初始化一次。  
将上面代码中的静态块替换如下：

|  |
| --- |
| static          {              System.out.println(Thread.currentThread() + "init DeadLoopClass");              try              {                  TimeUnit.SECONDS.sleep(10);              }              catch (InterruptedException e)              {                  e.printStackTrace();              }          } |

运行结果：

|  |
| --- |
| Thread[Thread-0,5,main] start  Thread[Thread-1,5,main] start  Thread[Thread-1,5,main]init DeadLoopClass (之后sleep 10s)  Thread[Thread-1,5,main] run over  Thread[Thread-0,5,main] run over |

虚拟机规范严格规定了有且只有5中情况（jdk1.7）必须对类进行“初始化”（而加载、验证、准备自然需要在此之前开始）：

遇到new,getstatic,putstatic,invokestatic这失调字节码指令时，如果类没有进行过初始化，则需要先触发其初始化。生成这4条指令的最常见的Java代码场景是：使用new关键字实例化对象的时候、读取或设置一个类的静态字段（被final修饰、已在编译器把结果放入常量池的静态字段除外）的时候，以及调用一个类的静态方法的时候。

使用java.lang.reflect包的方法对类进行反射调用的时候，如果类没有进行过初始化，则需要先触发其初始化。

当初始化一个类的时候，如果发现其父类还没有进行过初始化，则需要先触发其父类的初始化。

当虚拟机启动时，用户需要指定一个要执行的主类（包含main()方法的那个类），虚拟机会先初始化这个主类。

当使用jdk1.7动态语言支持时，如果一个java.lang.invoke.MethodHandle实例最后的解析结果REF\_getstatic,REF\_putstatic,REF\_invokeStatic的方法句柄，并且这个方法句柄所对应的类没有进行初始化，则需要先出触发其初始化。

开篇已经举了一个范例：通过子类引用付了的静态字段，不会导致子类初始化。  
这里再举两个例子。  
1. 通过数组定义来引用类，不会触发此类的初始化：（SuperClass类已在本文开篇定义）

|  |
| --- |
| public class NotInitialization  {      public static void main(String[] args)      {          SuperClass[] sca = new SuperClass[10];      }  } |

运行结果：（无）  
2. 常量在编译阶段会存入调用类的常量池中，本质上并没有直接引用到定义常量的类，因此不会触发定义常量的类的初始化：

|  |
| --- |
| public class ConstClass  {      static      {          System.out.println("ConstClass init!");      }      public static  final String HELLOWORLD = "hello world";  }  public class NotInitialization  {      public static void main(String[] args)      {          System.out.println(ConstClass.HELLOWORLD);      }  } |

## 多线程并发

## IO

## 内部类

## 异常

## 对象克隆以及深拷贝和浅拷贝

### 什么是"克隆"？

在某一时刻已经包含了一些有效值的对象A，可能会需要一个和A完全相同新对象B，并且此后对B任何改动都不会影响到A中的值，也就是说，A与B是两个独立的对象，但B的初始值是由A对象确定的。

在 Java语言中，用简单的赋值语句是不能满足这种需求的。要满足这种需求虽然有很多途径，但实现clone（）方法是其中最简单，也是最高效的手段。   
Java的所有类都默认继承java.lang.Object类，在java.lang.Object类中有一个方法clone()。JDK API的说明文档解释这个方法将返回Object对象的一个拷贝。

要说明的有两点：

一是拷贝对象返回的是一个新对象，而不是一个引用。

二是拷贝对象与用 new操作符返回的新对象的区别就是这个拷贝已经包含了一些原来对象的信息，而不是对象的初始信息。

有三个值得注意的地方，一是希望能实现clone功能的CloneClass类实现了Cloneable接口，这个接口属于java.lang 包，java.lang包已经被缺省的导入类中，所以不需要写成java.lang.Cloneable。另一个值得请注意的是重载了clone()方 法。最后在clone()方法中调用了super.clone()，这也意味着无论clone类的继承结构是什么样的，super.clone()直接或 间接调用了java.lang.Object类的clone()方法。下面再详细的解释一下这几点。

应该说第三点是最重要的，仔细 观察一下Object类的clone()一个native方法，native方法的效率一般来说都是远高于java中的非native方法。这也解释了为 什么要用Object中clone()方法而不是先new一个类，然后把原始对象中的信息赋到新对象中，虽然这也实现了clone功能。对于第二点，也要 观察Object类中的clone()还是一个protected属性的方法。这也意味着如果要应用clone()方法，必须继承Object类，在 Java中所有的类是缺省继承Object类的，也就不用关心这点了。然后重写clone()方法。还有一点要考虑的是为了让其它类能调用这个clone 类的clone()方法，重写之后要把clone()方法的属性设置为public。

那么clone类为什么还要实现 Cloneable接口呢？稍微注意一下，Cloneable接口是不包含任何方法的！其实这个接口仅仅是一个标志，而且这个标志也仅仅是针对 Object类中clone()方法的，如果clone类没有实现Cloneable接口，并调用了Object的clone()方法（也就是调用了 super.Clone()方法），那么Object的clone()方法就会抛出CloneNotSupportedException异常。

### 深拷贝与浅拷贝

浅拷贝是指拷贝对象时仅仅拷贝对象本身（包括对象中的基本变量），而不拷贝对象包含的引用指向的对象。

深拷贝不仅拷贝对象本身，而且拷贝对象包含的引用指向的所有对象。

举例来说更加清楚：对象A1中包含对B1的引用，B1中包含对C1的引用。浅拷贝A1得到A2，A2 中依然包含对B1的引用，B1中依然包含对C1的引用。深拷贝则是对浅拷贝的递归，深拷贝A1得到A2，A2中包含对B2（B1的copy）的引用，B2 中包含对C2（C1的copy）的引用。

## 强引用、弱引用、软引用、虚引用 - Java 7之基础

### 强引用（StrongReference）

      强引用是使用最普遍的引用。如果一个对象具有强引用，那垃圾回收器绝不会回收它。如下：

Object o=new Object();   //  强引用

当内存空间不足，Java虚拟机宁愿抛出OutOfMemoryError错误，使程序异常终止，也不会靠随意回收具有强引用的对象来解决内存不足的问题。如果不使用时，要通过如下方式来弱化引用，如下：

o=null;     // 帮助垃圾收集器回收此对象

 显式地设置o为null，或超出对象的生命周期范围，则gc认为该对象不存在引用，这时就可以回收这个对象。具体什么时候收集这要取决于gc的算法。

举例：

public void test(){

Object o=new Object();

// 省略其他操作

}

在一个方法的内部有一个强引用，这个引用保存在栈中，而真正的引用内容（Object）保存在堆中。当这个方法运行完成后就会退出方法栈，则引用内容的引用不存在，这个Object会被回收。

但是如果这个o是全局的变量时，就需要在不用这个对象时赋值为null，因为强引用不会被垃圾回收。

强引用在实际中有非常重要的用处，举个ArrayList的实现源代码：

|  |
| --- |
| private transient Object[] elementData;  public void clear() {  modCount++;  // Let gc do its work  for (int i = 0; i < size; i++)  elementData[i] = null;  size = 0;  } |

在ArrayList类中定义了一个私有的变量elementData数组，在调用方法清空数组时可以看到为每个数组内容赋值为null。不同于elementData=null，强引用仍然存在，避免在后续调用 add()等方法添加元素时进行重新的内存分配。使用如clear()方法中释放内存的方法对数组中存放的引用类型特别适用，这样就可以及时释放内存。

### 软引用（SoftReference）

如果一个对象只具有软引用，则内存空间足够，垃圾回收器就不会回收它；如果内存空间不足了，就会回收这些对象的内存。只要垃圾回收器没有回收它，该对象就可以被程序使用。软引用可用来实现内存敏感的高速缓存。

String str=new String("abc"); // 强引用

SoftReference<String> softRef=new SoftReference<String>(str); // 软引用

当内存不足时，等价于：

If(JVM.内存不足()) {

str = null; // 转换为软引用

System.gc(); // 垃圾回收器进行回收

}

虚引用在实际中有重要的应用，例如浏览器的后退按钮。按后退时，这个后退时显示的网页内容是重新进行请求还是从缓存中取出呢？这就要看具体的实现策略了。

（1）如果一个网页在浏览结束时就进行内容的回收，则按后退查看前面浏览过的页面时，需要重新构建

（2）如果将浏览过的网页存储到内存中会造成内存的大量浪费，甚至会造成内存溢出

这时候就可以使用软引用

|  |
| --- |
| Browser prev = new Browser(); // 获取页面进行浏览  SoftReference sr = new SoftReference(prev); // 浏览完毕后置为软引用  if(sr.get()!=null){  rev = (Browser) sr.get(); // 还没有被回收器回收，直接获取  }else{  prev = new Browser(); // 由于内存吃紧，所以对软引用的对象回收了  sr = new SoftReference(prev); // 重新构建  } |

这样就很好的解决了实际的问题。

       软引用可以和一个引用队列（ReferenceQueue）联合使用，如果软引用所引用的对象被垃圾回收器回收，Java虚拟机就会把这个软引用加入到与之关联的引用队列中。

### 弱引用（WeakReference）

弱引用与软引用的区别在于：只具有弱引用的对象拥有更短暂的生命周期。在垃圾回收器线程扫描它所管辖的内存区域的过程中，一旦发现了只具有弱引用的对象，不管当前内存空间足够与否，都会回收它的内存。不过，由于垃圾回收器是一个优先级很低的线程，因此不一定会很快发现那些只具有弱引用的对象。

|  |
| --- |
| String str=new String("abc");  WeakReference<String> abcWeakRef = new WeakReference<String>(str);  str=null; |

当垃圾回收器进行扫描回收时等价于：

|  |
| --- |
| str = null;  System.gc(); |

如果这个对象是偶尔的使用，并且希望在使用时随时就能获取到，但又不想影响此对象的垃圾收集，那么你应该用 Weak Reference 来记住此对象。

   下面的代码会让str再次变为一个强引用：

String abc = abcWeakRef.get();

弱引用可以和一个引用队列（ReferenceQueue）联合使用，如果弱引用所引用的对象被垃圾回收，Java虚拟机就会把这个弱引用加入到与之关联的引用队列中。

当你想引用一个对象，但是这个对象有自己的生命周期，你不想介入这个对象的生命周期，这时候你就是用弱引用。

这个引用不会在对象的垃圾回收判断中产生任何附加的影响。

|  |
| --- |
| public class ReferenceTest {    private static ReferenceQueue<VeryBig> rq = new ReferenceQueue<VeryBig>();    public static void checkQueue() {  Reference<? extends VeryBig> ref = null;  while ((ref = rq.poll()) != null) {  if (ref != null) {  System.out.println("In queue: " + ((VeryBigWeakReference) (ref)).id);  }  }  }    public static void main(String args[]) {  int size = 3;  LinkedList<WeakReference<VeryBig>> weakList = new LinkedList<WeakReference<VeryBig>>();  for (int i = 0; i < size; i++) {  weakList.add(new VeryBigWeakReference(new VeryBig("Weak " + i), rq));  System.out.println("Just created weak: " + weakList.getLast());    }    System.gc();  try { // 下面休息几分钟，让上面的垃圾回收线程运行完成  Thread.currentThread().sleep(6000);  } catch (InterruptedException e) {  e.printStackTrace();  }  checkQueue();  }  }    class VeryBig {  public String id;  // 占用空间,让线程进行回收  byte[] b = new byte[2 \* 1024];    public VeryBig(String id) {  this.id = id;  }    protected void finalize() {  System.out.println("Finalizing VeryBig " + id);  }  }    class VeryBigWeakReference extends WeakReference<VeryBig> {  public String id;    public VeryBigWeakReference(VeryBig big, ReferenceQueue<VeryBig> rq) {  super(big, rq);  this.id = big.id;  }    protected void finalize() {  System.out.println("Finalizing VeryBigWeakReference " + id);  }  }  输出结果：  Just created weak: com.javabase.reference.VeryBigWeakReference@1641c0  Just created weak: com.javabase.reference.VeryBigWeakReference@136ab79  Just created weak: com.javabase.reference.VeryBigWeakReference@33c1aa  Finalizing VeryBig Weak 2  Finalizing VeryBig Weak 1  Finalizing VeryBig Weak 0  In queue: Weak 1  In queue: Weak 2  In queue: Weak 0 |

### 虚引用（PhantomReference）

     “虚引用”顾名思义，就是形同虚设，与其他几种引用都不同，虚引用并不会决定对象的生命周期。如果一个对象仅持有虚引用，那么它就和没有任何引用一样，在任何时候都可能被垃圾回收器回收。

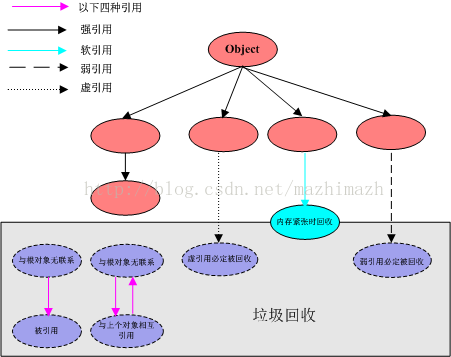
    虚引用主要用来跟踪对象被垃圾回收器回收的活动。虚引用与软引用和弱引用的一个区别在于：虚引用必须和引用队列 （ReferenceQueue）联合使用。当垃圾回收器准备回收一个对象时，如果发现它还有虚引用，就会在回收对象的内存之前，把这个虚引用加入到与之 关联的引用队列中。

### 总结

 Java 4种引用的级别由高到低依次为：

强引用  >  软引用  >  弱引用  >  虚引用

通过图来看一下他们之间在垃圾回收时的区别：



当垃圾回收器回收时，某些对象会被回收，某些不会被回收。垃圾回收器会从根对象Object来标记存活的对象，然后将某些不可达的对象和一些引用的对象进行回收，如果对这方面不是很了解，可以参考如下的文章：

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| ****引用类型**** | | ****被垃圾回收时间**** | | ****用途**** | | ****生存时间**** | |
| 强引用 | 从来不会 | | 对象的一般状态 | | JVM停止运行时终止 | |
| 软引用 | | 在内存不足时 | | 对象缓存 | | 内存不足时终止 | |
| 弱引用 | | 在垃圾回收时 | | 对象缓存 | | gc运行后终止 | |
| 虚引用 | | Unknown | | Unknown | | Unknown | |

## 动态绑定，静态绑定

 程序绑定的概念：绑定指的是一个方法的调用与方法所在的类（方法主题）关联起来。对java来说，绑定分为动态绑定和静态绑定，也叫做前期绑定和后期绑定。

    静态绑定：在程序执行前，方法已经被绑定，此时由编译器或其他链接程序实现。针对java简单的可以理解为程序编译器的绑定；这里特别说明一点，java当中的方法只有final，static，private和构造方法是前期绑定。

    动态绑定（后期绑定）：在运行时根据具体对象的类型进行绑定，若有一种语言实现了后期绑定，同时必须提供一些机制，可在运行期间判断对象类型，并分别调用适当的方法。也就是说编译器此时依然不知道对象的类型，但方法调用机制能自己去调查，找到正确的方法主体。

    动态绑定的过程：虚拟机提取对象的实际类型的方法表，虚拟机搜索方法签名，调用方法。

    在java中，几乎所有的方法都是动态绑定的，在运行时动态绑定方法是属于子类还是基类的。但是也有特殊，针对static和final方法，由于不能被继承，因此在编译时就可以确定他们的值，private声明的方法和成员变量不能被子类继承，所有的private方法都被隐式的final的（由此我们可以知道，将方法声明为final类型意一是为了防止方法被覆盖，二是为了有效的关闭java中的动态绑定）。java中的后期绑定是由JVM实现的。

   java中的向上转型或者说多态是借助于动态绑定实现的。

   比如：Parent p = new Children();其具体编译细节如下：

编译器检查对象的声明类型和方法名。假设我们调用x.f(args)方法，并且x已经被声明为C类的对象那么编译器会列举出C类众所的方法名为f的方法和从C类的超类继承过来的f方法。

接下来编译器检查方法调用中提供的参数类型。如果所有名称为f的方法中有一个参数类型和调用的方法的参数类型最为匹配，那么就调用这个方法，这个过程叫做“重载解析”。

当程序运行并且使用动态绑定调用方法时，虚拟机必须调用同x所指向的对象的实际类型所匹配的方法版本。假设实际类型为D（C的子类），如果D类定义了f，那么该方法被调用否则就在D的超类中搜寻方法f，以此类推。

注意，在处理java的成员变量时，并不是采用动态绑定，而是一般意义上的静态绑定。所以在向上转型的情况下，对象的方法可以找到子类，而对象的属性还是父类。

## 泛型

尖括号 <> 中的 T 被称作是类型参数，用于指代任何类型。实际上这个T你可以任意写，但出于规范的目的，Java还是建议我们用单个大写字母来代表类型参数。常见的如：

T 代表一般的任何类。

E 代表 Element 的意思，或者 Exception 异常的意思。

K 代表 Key 的意思

V 代表 Value 的意思，通常与 K 一起配合使用。

S 代表 Subtype。

简单点讲，泛型能节省某些java类型转换（casting）上的操作。泛型的主要好处就是让编译器保存参数的类型信息，执行类型检查，执行类型转换操作，编译器保证了这些类型转换的绝对无误。

    相对于程序员来记住对象类型执行类型转换（这可能会导致程序运行失败），而利用泛型，编译器能够帮助程序员在编译时强制进行大量的类型检查，发现其中的错误。

    在使用java泛型时，自动封转（Autoboxing）和自动拆装（Autounboxing）这两个特征会被自动用，而封装和解封装会带来性能上的损失。

而且可以看到，无论有多少个实例化后的泛型，比如Super<String,String>、Super<Interger,Double>，在虚拟机中，都只存在一个Super类。

在应用泛型类的时候，如果返回值也是泛型，那么返回的将是Object，但是很显然，我们在使用泛型的过程中，并没有要显示的进行类型转换，比如我们不需要这样 String a=(String) super.method("hahaha"); 而是直接 String a= super.method("hahaha"); 就可以了。为什么呢？

则是因为，为了返回实现类型的自动匹配，java编译器会在“类型擦除”的基础上在进行“转换插入”操作。每当使用泛型的返回值时，编译器会制动插入类型转换，所以原始代码是 String a= super.method("hahaha"); ，编译之后，虚拟机看到的却是 String a= (String)super.method("hahaha"); ，类型转换被“插入”了。

## [JAVA 类型转换：隐式转换与强制转换](https://www.cnblogs.com/kouhao/p/5899183.html)

### 隐式转换（  向上转型）

对于基础数据类型 , 可以自动转型 ,比如:   
int a = 10;   
long b = a;   
这里就是把int型转成了long型 , 因为 long范围比int大 , 这样的转型不会有任何影响 , 所以可以自动转型   
又比如 A a = new B();   
其中 A 是B 的父类 , 这也是多态 , 就是父类的对象指向子类的引用 , 你调用a对象 , 实际上是B的类 , 但是B继承A , 拥有A的一切属性和方法 (不谈复写)

### 强制转换（向下转型）

这种需要强转 , 比如   
long a = 10 ;   
int b = (int) a;   
通过 (int)可以强制转型 , 但是这样会丢失精度 , 比如a如果超过了b的范围 , 那么强转成int型, 只会等于int的最大值   
又比如 :   
B b = new A(); 这样是不行的 , 需要有中间途径 , 比如   
A a = new A(); Object obj = a; B b = (A) a;   
而这种强转 , 又有要求 , 必须是同类型 , 也就是说 A 就是B , 不然会报错.   
不同类型之间如果要转型 , 需要通过中间途径 , 比如JSON , 将A转成JSON串 , 然后再转换为B

null值可以强制转换为任何java类类型

|  |
| --- |
| @Data **public class** A {   **private** String **name**;  **private** String **age**; }  @Data **public class** B **extends** A {   **private** String **adress**;   **public void** test() {  System.***out***.println(**"hhahhahaha"**);  } }  A a =**new** B(); ((B) a).test();  向上转型既能调用父类的方法，也能调用子类的方法(在向上转型时，子类单独定义的方法会丢失。)  向下转型：向下转型之前，你必须得先向上转型  当父类对象引用变量引用子类对象时，被引用对象的类型决定了调用谁的成员方法，引用变量类型决定可调用的方法。如果子类没有覆盖该方法，那么会去父类中寻找。 |

## 抽象类和接口

|  |  |  |
| --- | --- | --- |
| 参数 | 抽象类 | 接口 |
| 默认的方法实现 | 它可以有默认的方法实现 | 接口完全是抽象的。它根本不存在方法的实现 |
| 实现 | 子类使用extends关键字来继承抽象类。如果子类不是抽象类的话，它需要提供抽象类中所有声明的方法的实现。 | 子类使用关键字implements来实现接口。它需要提供接口中所有声明的方法的实现 |
| 构造器 | 抽象类可以有构造器 | 接口不能有构造器 |
| 与正常Java类的区别 | 除了你不能实例化抽象类之外，它和普通Java类没有任何区别 | 接口是完全不同的类型 |
| 访问修饰符 | 抽象方法可以有public、protected和default这些修饰符 | 接口方法默认修饰符是public。你不可以使用其它修饰符。 |
| main方法 | 抽象方法可以有main方法并且我们可以运行它 | 接口没有main方法，因此我们不能运行它。 |
| 多继承 | 抽象方法可以继承一个类和实现多个接口 | 接口只可以继承一个或多个其它接口 |
| 速度 | 它比接口速度要快 | 接口是稍微有点慢的，因为它需要时间去寻找在类中实现的方法。 |
| 添加新方法 | 如果你往抽象类中添加新的方法，你可以给它提供默认的实现。因此你不需要改变你现在的代码。 | 如果你往接口中添加方法，那么你必须改变实现该接口的类。 |

抽象类：

类中有抽象方法，继承类必须实现这些抽象方法。（也可以没有抽象方法，而只是实现该类而已，这是纯类型抽象）。

接口：

类中的方法全部为抽象方法，变量为静态变量，接口的方法封装了不同的行为，接口的实现类必须实现这些抽象方法。

继承是“a is A”的关系，接口实现是 "hava a"的关系。

因为继承的子类必须是继承类的子类，接口只是为了添加一个行为功能而已。接口更多的是使用了组合模式，来添加功能。

区别：

抽象类和接口都是超类型，抽象类是对事物的抽象，接口是对行为的抽象。

这样，在需要使用事物抽象时使用抽象类，在需要使用行为抽象时使用接口。

## 静态内部类

将一个类隐藏在另一个类内部，不需要内部类引用外围类对象，可以将内部类声明为static，以便取消产生的引用。

内部类声明在接口中自动成为static和public类；

### Java8中的默认方法和静态方法

Oracle已经开始尝试向接口中引入默认方法和静态方法，以此来减少抽象类和接口之间的差异。现在，我们可以为接口提供默认实现的方法了并且不用强制子类来实现它。

## JDK

### int转换为string？

1.两种方法，一个是再int后面+“”，就可以转为字符串。

另一个，

int i=12345;  
String s="";  
第一种方法：s=i+"";  
第二种方法：s=String.valueOf(i);

第一种方法：s=i+"";   //会产生两个String对象  
第二种方法：s=String.valueOf(i); //直接使用String类的静态方法，只产生一个对象

**String -> int**

s="12345";  
int i;

第一种方法：i=Integer.parseInt(s);//直接使用静态方法，不会产生多余的对象，但会抛出异常  
第二种方法：i=Integer.valueOf(s).intValue();//Integer.valueOf(s) 相当于 new Integer(Integer.parseInt(s))，也会抛异常，但会多产生一个对象

## （一）Java8新特性

### 1．Lambda 表达式

#### 语法

Lambda 允许把函数作为一个方法的参数（函数作为参数传递进方法中）。

以下是lambda表达式的重要特征:

·        可选类型声明：不需要声明参数类型，编译器可以统一识别参数值。

·        可选的参数圆括号：一个参数无需定义圆括号，但多个参数需要定义圆括号。

·        可选的大括号：如果主体包含了一个语句，就不需要使用大括号。

·        可选的返回关键字：如果主体只有一个表达式返回值则编译器会自动返回值，大括号需要指定明表达式返回了一个数值。

 Lambda 表达式免去了使用匿名方法的麻烦，并且给予Java简单但是强大的函数化的编程能力。

#### 变量作用域

lambda 表达式只能引用标记了 final 的外层局部变量，这就是说不能在lambda 内部修改定义在域外的局部变量，否则会编译错误。

### 方法引用

方法引用通过方法的名字来指向一个方法。方法引用使用一对冒号 **::** 。

方法引用可以使语言的构造更紧凑简洁，减少冗余代码。

### 函数式接口

定义：

1.如果一个接口只有一个抽象方法，那么该接口就是一个函数式接口；

2.如果在接口上声明了FunctionalInterface注解，编译期会按照函数式接口的定义来要求该接口。

3.如果某个接口只有一个抽象方法，但并没有给该接口声明FunctionalInterface注解，那么编译期依旧会将该接口看作函数式接口。

Lamada表达式为Java添加了缺失的函数式编程特性，使我们能将函数当作一等公民看待；

在将函数作为一等公民的语言中，Lamada表达式的类型是函数。但在Java中，Lamada表达式是对象，他们必须依附于一类特别的对象类型——函数式接口。

外部迭代；

|  |
| --- |
| @interface FunctionalInterface  Note that instances of functional interfaces can be created with lambda expressions, method references, or constructor references.  list.forEach(System.out::println);//方法引用  // 一、Lamada表达式 map 映射，将一个数据映射成为另一个 list.stream().map(item->item.toUpperCase()).forEach(item-> System.out.println(item)); // 二、方法引用 list.stream().map(String::toUpperCase).forEach(System.out::println);  // 三、匿名函数  list.stream().map(new Function<String, Object>() {  @Override  public Object apply(String s) {  return s.toUpperCase();  } }).forEach(new Consumer() {  @Override  public void accept(Object o) {  System.out.println(o);  } });  Collections.sort(names, new Comparator<String>() {  @Override  public int compare(String o1, String o2) {  return o2.compareTo(o1);  } });  Collections.sort(names,(String o1, String o2)-> {  return o2.compareTo(o1); }); // Lamada表达式复杂传参——语句 Collections.sort(names,(o1,o2)-> {  return o2.compareTo(o1); });  // 表达式  Collections.sort(names,(o1,o2)-> o2.compareTo(o1)); |

Java Lamada表达式是一种匿名函数；它是没有声明的方法，即没有访问修饰符、返回值声明和名字。

Lamada表达式作用：

①传递行为，而不仅仅是传值；

②提升抽象层次；

③API重用性更好；

④更加灵活；

Java Lamada结构：

①一个Lamada表达式可以有零个或多个参数；

②参数的类型可以明确声明，也可以根据上下文来推断。例如：（int a）与（a）效果相同；

③所有参数需包含在圆括号内，参数之间用逗号相隔。例如：（a，b）或（int a，int b）；

④空圆括号代表参数集为空。例如：（）->42；

⑤当只有一个参数，且其类型可推导时，圆括号（）可省略。例如a->return a\*a；

⑥Lamada表达式的主体可包含零条或多条语句；

⑦如果Lamada表达式的主体只有一条语句，花括号{}可省略。匿名函数的返回类型与该主体表达式一致；

⑧如果Lamada表达式的主体包含一条以上语句，则表达式必须包含在花括号{}中（形成代码块）。匿名函数的返回类型与代码块的返回类型一致，若没有返回则为空；

高阶函数：如果一个函数接收一个函数作为参数，或者返回一个函数作为返回值，那么该函数叫做高阶函数；

函数式接口(FunctionalInterface)就是一个有且仅有一个抽象方法，但是可以有多个非抽象方法的接口。

函数式接口可以被隐式转换为lambda表达式。

函数式接口可以现有的函数友好地支持 lambda。

JDK 1.8之前已有的函数式接口:

·        java.lang.Runnable

·        java.util.concurrent.Callable

·        java.security.PrivilegedAction

·        java.util.Comparator

·        java.io.FileFilter

·        java.nio.file.PathMatcher

·        java.lang.reflect.InvocationHandler

·        java.beans.PropertyChangeListener

·        java.awt.event.ActionListener

·        javax.swing.event.ChangeListener

JDK 1.8 新增加的函数接口：

·        java.util.function

java.util.function 它包含了很多类，用来支持 Java的函数式编程，该包中的函数式接口有：

序号 接口 & 描述

1．BiConsumer<T,U>

代表了一个接受两个输入参数的操作，并且不返回任何结果

2．BiFunction<T,U,R>

代表了一个接受两个输入参数的方法，并且返回一个结果

3．BinaryOperator<T>

代表了一个作用于于两个同类型操作符的操作，并且返回了操作符同类型的结果

4．BiPredicate<T,U>

代表了一个两个参数的boolean值方法

5．BooleanSupplier

代表了boolean值结果的提供方

6．Consumer<T>

代表了接受一个输入参数并且无返回的操作

7．DoubleBinaryOperator

代表了作用于两个double值操作符的操作，并且返回了一个double值的结果。

8．DoubleConsumer

代表一个接受double值参数的操作，并且不返回结果。

9．DoubleFunction<R>

代表接受一个double值参数的方法，并且返回结果

10．DoublePredicate

代表一个拥有double值参数的boolean值方法

11．DoubleSupplier

代表一个double值结构的提供方

12．DoubleToIntFunction

接受一个double类型输入，返回一个int类型结果。

13．DoubleToLongFunction

接受一个double类型输入，返回一个long类型结果

14．DoubleUnaryOperator

接受一个参数同为类型double,返回值类型也为double。

15．Function<T,R>

接受一个输入参数，返回一个结果。

16．IntBinaryOperator

接受两个参数同为类型int,返回值类型也为int 。

17．IntConsumer

接受一个int类型的输入参数，无返回值。

18．IntFunction<R>

接受一个int类型输入参数，返回一个结果。

19．IntPredicate

接受一个int输入参数，返回一个布尔值的结果。

20．IntSupplier

无参数，返回一个int类型结果。

21．IntToDoubleFunction

接受一个int类型输入，返回一个double类型结果。

22．IntToLongFunction

接受一个int类型输入，返回一个long类型结果。

23．IntUnaryOperator

接受一个参数同为类型int,返回值类型也为int 。

24．LongBinaryOperator

接受两个参数同为类型long,返回值类型也为long。

25．LongConsumer

接受一个long类型的输入参数，无返回值。

26．LongFunction<R>

接受一个long类型输入参数，返回一个结果。

27．LongPredicate

R接受一个long输入参数，返回一个布尔值类型结果。

28．LongSupplier

无参数，返回一个结果long类型的值。

29．LongToDoubleFunction

接受一个long类型输入，返回一个double类型结果。

30．LongToIntFunction

接受一个long类型输入，返回一个int类型结果。

31．LongUnaryOperator

接受一个参数同为类型long,返回值类型也为long。

32．ObjDoubleConsumer<T>

接受一个object类型和一个double类型的输入参数，无返回值。

33．ObjIntConsumer<T>

接受一个object类型和一个int类型的输入参数，无返回值。

34．ObjLongConsumer<T>

接受一个object类型和一个long类型的输入参数，无返回值。

35．Predicate<T>

接受一个输入参数，返回一个布尔值结果。

36．Supplier<T>

无参数，返回一个结果。

37．ToDoubleBiFunction<T,U>

接受两个输入参数，返回一个double类型结果

38．ToDoubleFunction<T>

接受一个输入参数，返回一个double类型结果

39．ToIntBiFunction<T,U>

接受两个输入参数，返回一个int类型结果。

40．ToIntFunction<T>

接受一个输入参数，返回一个int类型结果。

41．ToLongBiFunction<T,U>

接受两个输入参数，返回一个long类型结果。

42．ToLongFunction<T>

接受一个输入参数，返回一个long类型结果。

43．UnaryOperator<T>

接受一个参数为类型T,返回值类型也为T。

### 接口默认方法

#### JDK8 中为什么有接口默认方法

以前创建了一个接口，并且已经被大量的类实现。

如果需要再扩充这个接口的功能加新的方法，就会导致所有已经实现的子类需要重写这个方法。

如果在接口中使用默认方法就不会有这个问题。

所以从 JDK8 开始新加了接口默认方法，便于接口的扩展。

#### 接口中默认方法的规则

1．默认方法使用 default 关键字，一个接口中可以有多个默认方法。

2．接口中既可以定义抽象方法，又可以定义默认方法，默认方法不是抽象方法。

3．子类实现接口的时候，可以直接调用接口中的默认方法，即继承了接口中的默认方法。

4．接口中同时还可以定义静态方法，静态方法通过接口名调用。

接口中的静态方法不能被实现类继承和子接口继承，但是接口中的非静态的默认方法可以被实现类继承。

首先，之前的接口是个双刃剑，好处是面向抽象而不是面向具体编程，缺陷是，当需要修改接口时候，需要修改全部实现该接口的类，目前的java 8之前的集合框架没有foreach方法，通常能想到的解决办法是在JDK里给相关的接口添加新的方法及实现。然而，对于已经发布的版本，是没法在给接口添加新方法的同时不影响已有的实现。所以引进的默认方法。他们的目的是为了解决接口的修改与现有的实现不兼容的问题。

#### 语法

默认方法语法格式如下：

public interface vehicle {

    default void print() {

        System.out.println("我是一辆车!");

    }

}

#### 多个默认方法

一个接口有默认方法，考虑这样的情况，一个类实现了多个接口，且这些接口有相同的默认方法，以下实例说明了这种情况的解决方法：

public interface vehicle {

    default void print() {

        System.out.println("我是一辆车!");

    }

}

public interface fourWheeler {

    default void print() {

        System.out.println("我是一辆四轮车!");

    }

}

第一个解决方案是创建自己的默认方法，来覆盖重写接口的默认方法：

public class Car implements vehicle, fourWheeler {

    @Override

    public void print() {

        System.out.println("我是一辆四轮汽车!");

    }

}

第二种解决方案可以使用 super 来调用指定接口的默认方法：

public class Car implements vehicle, fourWheeler {

    @Override

    public void print() {

        vehicle.super.print();

    }

}

#### 静态默认方法

Java 8 的另一个特性是接口可以声明（并且可以提供实现）静态方法。例如：

public interface vehicle {

    default void print() {

        System.out.println("我是一辆车!");

    }

    // 静态方法

    static void blowHorn() {

        System.out.println("按喇叭!!!");

    }

}

### Stream

Java 8 API添加了一个新的抽象称为流Stream，可以让你以一种声明的方式处理数据。

Stream使用一种类似用SQL语句从数据库查询数据的直观方式来提供一种对Java集合运算和表达的高阶抽象。

Stream API可以极大提高Java程序员的生产力，让程序员写出高效率、干净、简洁的代码。

这种风格将要处理的元素集合看作一种流，流在管道中传输，并且可以在管道的节点上进行处理，比如筛选，排序，聚合等。

元素流在管道中经过中间操作（intermediate operation）的处理，最后由最终操作(terminal operation)得到前面处理的结果

#### 什么是 Stream

Stream（流）是一个来自数据源的元素队列并支持聚合操作

元素：是特定类型的对象，形成一个队列。Java中的Stream并不会存储元素，而是按需计算。

数据源 ：流的来源。可以是集合，数组，I/O channel，产生器generator等。

聚合操作： 类似SQL语句一样的操作，比如filter, map, reduce, find,match, sorted等。

和以前的Collection操作不同，Stream操作还有两个基础的特征：

Pipelining:：中间操作都会返回流对象本身。这样多个操作可以串联成一个管道，如同流式风格（fluent style）。这样做可以对操作进行优化，比如延迟执行(laziness)和短路( short-circuiting)。

内部迭代：以前对集合遍历都是通过Iterator或者For-Each的方式,显式的在集合外部进行迭代，这叫做外部迭代。Stream提供了内部迭代的方式，通过访问者模式(Visitor)实现。

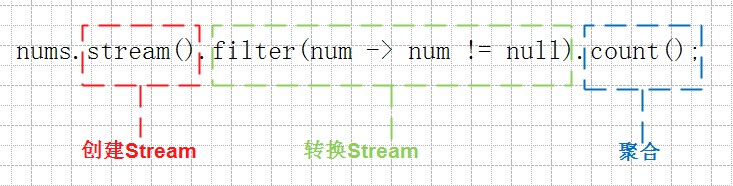
#### 生成流

在Java 8中,集合接口有两个方法来生成流：

stream() −为集合创建串行流。

parallelStream() − 为集合创建并行流。

#### Stream通用语法



使用Stream的基本步骤：

①创建Stream；

②转换Stream，每次转换原有Stream对象不改变，返回一个新的Stream对象（\*\*可以有多次转换\*\*）；

③对Stream进行聚合（Reduce）操作，获取想要的结果；

#### forEach

Stream 提供了新的方法 'forEach' 来迭代流中的每个数据。以下代码片段使用forEach 输出了10个随机数：

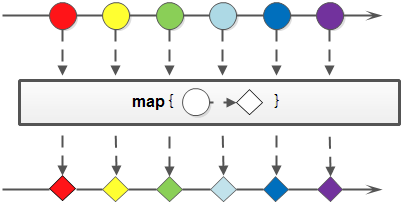
Random random = new Random();

random.ints().limit(10).forEach(System.out::println);

#### map和flatmap

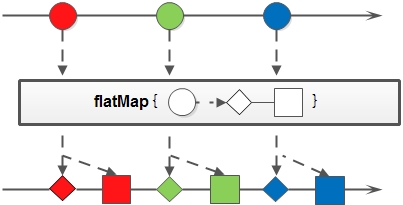
map 方法用于映射每个元素到对应的结果，以下代码片段使用 map 输出了元素对应的平方数：

map方法示意图：



flatMap方法示意图：

和map类似，不同的是其每个元素转换得到的是Stream对象，会把子Stream中的元素压缩到父集合中；



|  |
| --- |
| List<List<String>> lists = Arrays.asList(Arrays.asList("Jordan"),  Arrays.asList("Kobe","James"),Arrays.asList("Durant","Curry")  );  //集合 -> 流：流里的元素也是集合的元素，所以流的元素是List<String>  Stream<List<String>> streamRaw = lists.stream();  //流元素的转换，List<String>，通过Collection的stream方法，变为Stream<String>  //即：List<String> -> Stream<String>，把List<String>进行了一次流化  Stream<Stream<String>> streamMap = streamRaw.map(Collection::stream);  //和上面方法引用效果一样  Stream<Stream<String>> streamMap2 = streamRaw.map(l -> l.stream());  //map方法是单纯的转换，而flatMap方法可以进行流的扁平化，和上面的map做对比  Stream<String> streamFlatmap = streamRaw.flatMap(l -> l.stream()); |

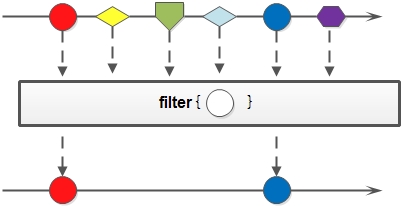
List<Integer> numbers = Arrays.asList(3, 2, 2, 3, 7, 3, 5);

// 获取对应的平方数

List<Integer> squaresList = numbers.stream().map(i -> i \* i).distinct().collect(Collectors.toList());

#### Filter

Filter方法示意图：



filter 方法用于通过设置条件过滤出元素。以下代码片段使用filter 方法过滤出空字符串：

List<String>strings = Arrays.asList("abc", "", "bc", "efg", "abcd","", "jkl");

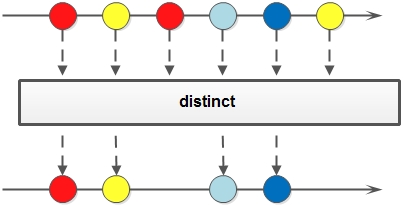
// 获取空字符串的数量

int count = (int) strings.stream().filter(string -> string.isEmpty()).count();

#### distinct

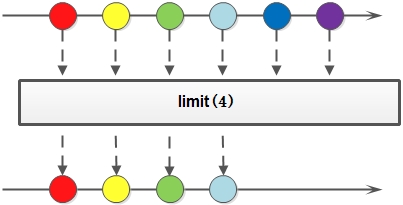
 distinct: 对于Stream中包含的元素进行去重操作（去重逻辑依赖元素的equals方法），新生成的Stream中没有重复的元素；

Distinct方法示意图：



#### Limit

limit方法示意图：



limit 方法用于获取指定数量的流。以下代码片段使用 limit 方法打印出 10 条数据：

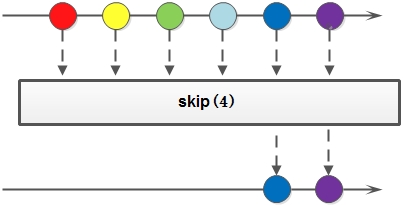
Random random = new Random();

random.ints().limit(10).forEach(System.out::println);

#### Skip

返回一个丢弃原Stream的前N个元素后剩下元素组成的新Stream，如果原Stream中包含的元素个数小于N，那么返回空Stream；

skip方法示意图：



#### sorted

sorted 方法用于对流进行排序。以下代码片段使用 sorted 方法对输出的 10 个随机数进行排序：

Random random = new Random();

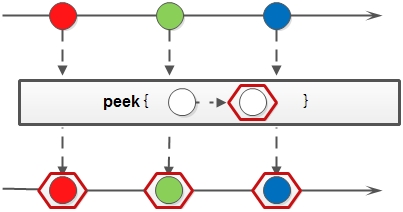
random.ints().limit(10).sorted().forEach(System.out::println);

#### peek

peek: 生成一个包含原Stream的所有元素的新Stream，同时会提供一个消费函数（Consumer实例），新Stream每个元素被消费的时候都会执行给定的消费函数；

peek(System.out::println)

Peek方法示意图：



#### 并行（parallel）程序

parallelStream 是流并行处理程序的代替方法。以下实例我们使用parallelStream 来输出空字符串的数量：

List<String> strings = Arrays.asList("abc", "", "bc", "efg", "abcd", "", "jkl");

// 获取空字符串的数量

int count = (int) strings.parallelStream().filter(string -> string.isEmpty()).count();

我们可以很容易的在顺序运行和并行直接切换。

#### Collectors

Collectors 类实现了很多归约操作，例如将流转换成集合和聚合元素。Collectors可用于返回列表或字符串：

List<String> strings = Arrays.asList("abc", "", "bc", "efg", "abcd", "", "jkl");

List<String> filtered = strings.stream().filter(string -> !string.isEmpty()).collect(Collectors.toList());

System.out.println("筛选列表: " + filtered);

String mergedString = strings.stream().filter(string -> !string.isEmpty()).collect(Collectors.joining(", "));

System.out.println("合并字符串: " + mergedString);

#### 统计

另外，一些产生统计结果的收集器也非常有用。它们主要用于int、double、long等基本类型上，它们可以用来产生类似如下的统计结果。

List<Integer> numbers = Arrays.asList(3, 2, 2, 3, 7, 3, 5);

IntSummaryStatistics stats = numbers.stream().mapToInt((x) -> x).summaryStatistics();

System.out.println("列表中最大的数 : " + stats.getMax());

System.out.println("列表中最小的数 : " + stats.getMin());

System.out.println("所有数之和 : " + stats.getSum());

System.out.println("平均数 : " + stats.getAverage());

#### 汇聚（Reduce）Stream

汇聚操作（也称为折叠）接受一个元素序列为输入，反复使用某个合并操作，把序列中的元素合并成一个汇总的结果。比如查找一个数字列表的总和或者最大值，或者把这些数字累积成一个List对象。Stream接口有一些通用的汇聚操作，比如reduce()和collect()；也有一些特定用途的汇聚操作，比如sum(),max()和count()。注意：sum方法不是所有的Stream对象都有的，只有IntStream、LongStream和DoubleStream是实例才有。

下面会分两部分来介绍汇聚操作：

可变汇聚：把输入的元素们累积到一个可变的容器中，比如Collection或者StringBuilder；

其他汇聚：除去可变汇聚剩下的，一般都不是通过反复修改某个可变对象，而是通过把前一次的汇聚结果当成下一次的入参，反复如此。比如reduce，count，allMatch；

#### 搜索相关

– allMatch：是不是Stream中的所有元素都满足给定的匹配条件；  
– anyMatch：Stream中是否存在任何一个元素满足匹配条件；  
– findFirst: 返回Stream中的第一个元素，如果Stream为空，返回空Optional；  
– noneMatch：是不是Stream中的所有元素都不满足给定的匹配条件；  
– max和min：使用给定的比较器（Operator），返回Stream中的最大|最小值；

### Optional 类

Optional 类是一个可以为null的容器对象。如果值存在则isPresent()方法会返回true，调用get()方法会返回该对象。

Optional 是个容器：它可以保存类型T的值，或者仅仅保存null。Optional提供很多有用的方法，这样我们就不用显式进行空值检测。

Optional 类的引入很好的解决空指针异常。

#### 类声明

以下是一个 java.util.Optional<T> 类的声明：

public final class Optional<T> extends Object

#### 类方法

序号 方法 & 描述

1．static <T> Optional<T> empty()

返回空的 Optional 实例。

2．boolean equals(Object obj)

判断其他对象是否等于 Optional。

3．Optional<T> filter(Predicate<? super <T> predicate)

如果值存在，并且这个值匹配给定的 predicate，返回一个Optional用以描述这个值，否则返回一个空的Option Optional。

4．<U> Optional<U> flatMap(Function<? super T,Optional<U>> mapper)

如果值存在，返回基于Optional包含的映射方法的值，否则返回一个空的Optional

5．T get()

如果在这个Optional中包含这个值，返回值，否则抛出异常：NoSuchElementException

6．int hashCode()

返回存在值的哈希码，如果值不存在返回 0。

7．void ifPresent(Consumer<? super T> consumer)

如果值存在则使用该值调用 consumer , 否则不做任何事情。

8．boolean isPresent()

如果值存在则方法会返回true，否则返回 false。

9．<U>Optional<U> map(Function<? super T,? extends U> mapper)

如果存在该值，提供的映射方法，如果返回非null，返回一个Optional描述结果。

10．static <T> Optional<T> of(T value)

返回一个指定非null值的Optional。

11．static <T> Optional<T> ofNullable(T value)

如果为非空，返回 Optional 描述的指定值，否则返回空的 Optional。

12．T orElse(T other)

如果存在该值，返回值，否则返回 other。

13．T orElseGet(Supplier<? extends T> other)

如果存在该值，返回值，否则触发 other，并返回 other 调用的结果。

14．<X extends Throwable> T orElseThrow(Supplier<? extends X> exceptionSupplier)

如果存在该值，返回包含的值，否则抛出由 Supplier 继承的异常

15．String toString()

返回一个Optional的非空字符串，用来调试

注意： 这些方法是从 java.lang.Object 类继承来的。

### Nashorn JavaScript

Nashorn 一个 javascript 引擎。

从JDK1.8开始，Nashorn取代Rhino(JDK 1.6, JDK1.7)成为Java的嵌入式JavaScript引擎。Nashorn完全支持ECMAScript 5.1规范以及一些扩展。它使用基于JSR292的新语言特性，其中包含在JDK 7中引入的 invokedynamic，将JavaScript编译成Java字节码。

与先前的Rhino实现相比，这带来了2到10倍的性能提升。

#### jjs

jjs是个基于Nashorn引擎的命令行工具。它接受一些JavaScript源代码为参数，并且执行这些源代码。

例如，我们创建一个具有如下内容的sample.js文件：

print('Hello World!');

打开控制台，输入以下命令：

$ jjs sample.js

以上程序输出结果为：

HelloWorld!

#### jjs 交互式编程

打开控制台，输入以下命令：

|  |
| --- |
| $ jjs  jjs>print("Hello, World!")  Hello,World!  jjs> quit()  >> |

#### 传递参数

打开控制台，输入以下命令：

$ jjs -- a b c

jjs>print('字母: '+arguments.join(", "))

字母: a, b, c

jjs>

#### Java 中调用 JavaScript

使用ScriptEngineManager, JavaScript 代码可以在 Java 中执行，实例如下：

Java8Tester.java文件

|  |
| --- |
| public class Java8Tester {      public static void main(String args[]) {          ScriptEngineManager scriptEngineManager = new ScriptEngineManager();          ScriptEngine nashorn = scriptEngineManager.getEngineByName("nashorn");          String name = "Runoob";          Integer result = null;          try {              nashorn.eval("print('" + name + "')");              result = (Integer) nashorn.eval("10 + 2");          } catch (ScriptException e) {              System.out.println("执行脚本错误: " + e.getMessage());          }          System.out.println(result.toString());      }  }  执行以上脚本，输出结果为：  Runoob  12 |

#### JavaScript 中调用 Java

以下实例演示了如何在 JavaScript 中引用 Java 类：

|  |
| --- |
| var BigDecimal=Java.type('java.math.BigDecimal');    function calculate(amount, percentage){       var result =new BigDecimal(amount).multiply(     newBigDecimal(percentage)).divide(newBigDecimal("100"),2,BigDecimal.ROUND\_HALF\_EVEN);       return result.toPlainString();  }    var result = calculate(568000000000000000023,13.9);  print(result); |

我们使用jjs 命令执行以上脚本，输出结果如下：

$ jjs sample.js

78952000000000002017.94

### 8.Java 8 日期时间 API

Java 8通过发布新的Date-Time API (JSR 310)来进一步加强对日期与时间的处理。

在旧版的Java 中，日期时间API 存在诸多问题，其中有：

·        非线程安全 − java.util.Date 是非线程安全的，所有的日期类都是可变的，这是Java日期类最大的问题之一。

·        设计很差 − Java的日期/时间类的定义并不一致，在java.util和java.sql的包中都有日期类，此外用于格式化和解析的类在java.text包中定义。java.util.Date同时包含日期和时间，而java.sql.Date仅包含日期，将其纳入java.sql包并不合理。另外这两个类都有相同的名字，这本身就是一个非常糟糕的设计。

·        时区处理麻烦 − 日期类并不提供国际化，没有时区支持，因此Java引入了java.util.Calendar和java.util.TimeZone类，但他们同样存在上述所有的问题。

Java 8 在 java.time 包下提供了很多新的 API。以下为两个比较重要的 API：

·        Local(本地) − 简化了日期时间的处理，没有时区的问题。

·        Zoned(时区) − 通过制定的时区处理日期时间。

新的java.time包涵盖了所有处理日期，时间，日期/时间，时区，时刻（instants），过程（during）与时钟（clock）的操作。

#### 本地化日期时间 API

LocalDate/LocalTime 和 LocalDateTime 类可以在处理时区不是必须的情况。代码如下：

Java8Tester.java文件

public class Java8Tester {

    public static void main(String args[]) {

        Java8Tester java8tester = new Java8Tester();

        java8tester.testLocalDateTime();

    }

    public void testLocalDateTime() {

        // 获取当前的日期时间

        LocalDateTime currentTime = LocalDateTime.now();

        System.out.println("当前时间: " + currentTime);

        LocalDate date1 = currentTime.toLocalDate();

        System.out.println("date1: " + date1);

        Month month = currentTime.getMonth();

        int day = currentTime.getDayOfMonth();

        int seconds = currentTime.getSecond();

        System.out.println("月: " + month + ", 日: " + day + ", 秒: " + seconds);

        LocalDateTime date2 = currentTime.withDayOfMonth(10).withYear(2012);

        System.out.println("date2: " + date2);

        // 12 december 2014

        LocalDate date3 = LocalDate.of(2014, Month.DECEMBER, 12);

        System.out.println("date3: " + date3);

        // 22 小时 15 分钟

        LocalTime date4 = LocalTime.of(22, 15);

        System.out.println("date4: " + date4);

        // 解析字符串

        LocalTime date5 = LocalTime.parse("20:15:30");

        System.out.println("date5: " + date5);

    }

}

执行以上脚本，输出结果为：

当前时间: 2018-06-08T15:19:16.910

date1:2018-06-08

月: JUNE, 日: 8, 秒: 16

date2:2012-06-10T15:19:16.910

date3:2014-12-12

date4:22:15

date5:20:15:30

#### 使用时区的日期时间API

如果我们需要考虑到时区，就可以使用时区的日期时间API：

Java8Tester.java文件

public class Java8Tester {

    public static void main(String args[]) {

        Java8Tester java8tester = new Java8Tester();

        java8tester.testZonedDateTime();

    }

    public void testZonedDateTime() {

        // 获取当前时间日期

        ZonedDateTime date1 = ZonedDateTime.parse("2015-12-03T10:15:30+05:30[Asia/Shanghai]");

        System.out.println("date1: " + date1);

        ZoneId id = ZoneId.of("Europe/Paris");

        System.out.println("ZoneId: " + id);

        ZoneId currentZone = ZoneId.systemDefault();

        System.out.println("当期时区: " + currentZone);

    }

}

执行以上脚本，输出结果为：

date1:2015-12-03T10:15:30+08:00[Asia/Shanghai]

ZoneId:Europe/Paris

当期时区: Asia/Shanghai

### 9. Base64

在Java8中，Base64编码已经成为Java类库的标准。

Java 8 内置了 Base64 编码的编码器和解码器。

Base64工具类提供了一套静态方法获取下面三种BASE64编解码器：

·        基本：输出被映射到一组字符A-Za-z0-9+/，编码不添加任何行标，输出的解码仅支持A-Za-z0-9+/。

·        URL：输出映射到一组字符A-Za-z0-9+\_，输出是URL和文件。

·        MIME：输出隐射到MIME友好格式。输出每行不超过76字符，并且使用'\r'并跟随'\n'作为分割。编码输出最后没有行分割。

#### 内嵌类

序号 内嵌类 & 描述

1.static class Base64.Decoder

该类实现一个解码器用于，使用 Base64 编码来解码字节数据。

2.static class Base64.Encoder

该类实现一个编码器，使用 Base64 编码来编码字节数据。

#### 方法

序号 方法名 & 描述

1.static Base64.Decoder getDecoder()

返回一个 Base64.Decoder ，解码使用基本型 base64 编码方案。

2.static Base64.Encoder getEncoder()

返回一个 Base64.Encoder ，编码使用基本型 base64 编码方案。

3.static Base64.Decoder getMimeDecoder()

返回一个 Base64.Decoder ，解码使用 MIME 型 base64 编码方案。

4.static Base64.Encoder getMimeEncoder()

返回一个 Base64.Encoder ，编码使用 MIME 型 base64 编码方案。

5.static Base64.Encoder getMimeEncoder(int lineLength, byte[] lineSeparator)

返回一个 Base64.Encoder ，编码使用 MIME 型 base64 编码方案，可以通过参数指定每行的长度及行的分隔符。

6.static Base64.Decoder getUrlDecoder()

返回一个 Base64.Decoder ，解码使用 URL 和文件名安全型 base64 编码方案。

7.static Base64.Encoder getUrlEncoder()

返回一个 Base64.Encoder ，编码使用 URL 和文件名安全型 base64 编码方案。

注意：Base64 类的很多方法从 java.lang.Object 类继承。

#### Base64 实例

以下实例演示了Base64 的使用:

Java8Tester.java文件

public class Java8Tester {

    public static void main(String args[]) {

        try {

            // 使用基本编码

            String base64encodedString = Base64.getEncoder().encodeToString("runoob?java8".getBytes("utf-8"));

            System.out.println("Base64 编码字符串 (基本) :" + base64encodedString);

            // 解码

            byte[] base64decodedBytes = Base64.getDecoder().decode(base64encodedString);

            System.out.println("原始字符串: " + new String(base64decodedBytes, "utf-8"));

            base64encodedString = Base64.getUrlEncoder().encodeToString("TutorialsPoint?java8".getBytes("utf-8"));

            System.out.println("Base64 编码字符串 (URL) :" + base64encodedString);

            StringBuilder stringBuilder = new StringBuilder();

            for (int i = 0; i < 10; ++i) {

                stringBuilder.append(UUID.randomUUID().toString());

            }

            byte[] mimeBytes = stringBuilder.toString().getBytes("utf-8");

            String mimeEncodedString = Base64.getMimeEncoder().encodeToString(mimeBytes);

            System.out.println("Base64 编码字符串 (MIME) :" + mimeEncodedString);

        } catch (UnsupportedEncodingException e) {

            System.out.println("Error :" + e.getMessage());

        }

    }

}

执行以上脚本，输出结果为：

Base64 编码字符串 (基本) :cnVub29iP2phdmE4

原始字符串: runoob?java8

Base64编码字符串(URL):VHV0b3JpYWxzUG9pbnQ\_amF2YTg=

Base64编码字符串(MIME):MjY5OGRlYmEtZDU0ZS00MjY0LWE3NmUtNzFiNTYwY2E4YjM1NmFmMDFlNzQtZDE2NC00MDk3LTlh

ZjItYzNkNGJjNmQwOWE2OWM0NDJiN2YtOGM4Ny00MjhkLWJkMzgtMGVlZjFkZjkyYjJhZDUwYzk0

ZWMtNDE5ZC00MTliLWEyMTAtZGMyMjVkYjZiOTE3ZTkxMjljMTgtNjJiZC00YTFiLTg3MzAtOTA0

YzdjYjgxYjQ0YTUxOWNkMTAtNjgxZi00YjQ0LWFkZGMtMzk1YzRkZjIwMjcyMzA0MTQzN2ItYzBk

My00MmQyLWJiZTUtOGM0MTlmMWIxM2MxYTY4NmNiOGEtNTkxZS00NDk1LThlN2EtM2RjMTZjMWJk

ZWQyZTdhNmZiNDgtNjdiYy00ZmFlLThjNTYtMjcyNDNhMTRhZTkyYjNiNWY2MmEtNTZhYS00ZDhk

LWEwZDYtY2I5ZTUwNzJhNGE1

## 设计模式

    创建型模式：单例模式，抽象工厂模式，建造者模式，工厂模式与原型模式。

    结构型模式：适配器模式，桥接模式，装饰者模式，组合模式，外观模式，享元模式以及代理模式。

    行为型模式：模板方法模式，命令模式，迭代器模式，观察者模式，中介者模式，备忘录模式，解释器模式，状态模式，策略模式，职责链模式和访问者模式。

### 抽象工厂模式（Abstract Factory）

    客户类与工厂类分开，当客户需要某种产品时向对应工厂发出请求。缺点是当产品需要修改时，工厂类也需要做相应的修改。

    意图：

        提供一个创建一系列相关或者相互依赖的对象接口，无需指定它们具体的类。

    适用性：

        一个系统要独立于它的产品创建，组合的时候。

        一个系统要由多个产品系列中的一个来配置的时候。

        强调一系列相关产品对象设计以便进行联合使用时。

        对于一个产品库，只想显示接口而不是具体的实现的时候。

### 工厂方法模式（Factory Method）

    其核心工厂类不再负责所有的产品撞见，而是将具体工作委托给子类，自身成为一个抽象的工厂角色，仅负责给出工厂类必须实现的接口，而不需要关系具体哪种产品被实例化的细节。

    意图：

        定义一个用于创建对象接口，让子类来决定实例化哪一个类，将类的实例化延迟到其子类。

    适用性：

        当一个类不知道它所必须创建的对象的类的时候。

        当一个类希望由其子类来创建对象的时候。

### 建造者模式（Builder）

    将对象的内部表象和对象的生产过程分割开来，从而使得一个建造过程具有不同内部表象的产品对象。建造模式使得产品内部表象可以独立出来变化，客户不需要知道内部细节，可以强制实行一种分步建造过程。

    意图：

        将一个复杂的对象的构建与它的表现分离，使同样的构建形式可以得到不同的表示。

    适用性：

        将复杂对象的算法需要独立于对象的组成部分以及装配方式的时候。

        构造过程允许构造不同的对象表示。

### 适配器模式（Adapter）

    将一个类的接口变为客户端所希望的接口类型，从而使原先因接口不匹配而无法工作的两个类可以一起工作。

    意图：

        将某个接口改装变为所需要类型的接口，也可以是原本不兼容的接口适配成兼容。

    适用性：

        希望使用一个已经存在的类，但是与你所需要的接口有些不吻合时。

        希望创建一个可以复用的类，可以与其他已存在或者未存在的类进行协同工作。

### 桥接模式（Bridge）

    将抽象画和实例化解耦，使得两者可以独立变化。即降低耦合度。也就是说将抽象化和实例化之间的关系使用组合/聚合的关系而非继承关系，两者可以独立的变化。

    意图：

        将抽象部分与它的实现部分分离，使它们可以独立的变化。

    适用性：

        在对于抽象部分和实现部分不希望使用固定的关系。

        类的抽象和类的实现都应该都应该可以可以通过生成子类的方式来加以扩充。

        对一个抽象的实现部分的修改应对客户不产生影响。

        希望在多个对象之间共享（比如使用引用计数），但是客户不知道该点。

### 职责链模式（Chain of Responsibility）

    在职责链模式中，很多对象由每一个对象对其下家的引用而连接起来的一条链。请求在这个链上进行传递直到链上的某一个对象决定处理该请求。对于链上的每一个对象成为处理者，处理者有两个选择：承担处理责任挥着将请求传递给下家，请求可以不被任何对象处理。

    意图：

        使多个对象都有机会处理请求，从而避免请求发送者和接受者之间的耦合关系。将这些对象连成一条链，并且沿着该链传输，直到有对象对他进行处理。

    适用性：

        有对个对象可以处理一个请求，具体是哪个对象处理会在运行时自动确定。

        在不明确接受者情况下，向多个对象提交一个请求。

        可以动态的指定那个对象集。

### 命令模式（Command）

    命令模式将一个请求或者操作封装到一个对象中。命令模式将发出命令和执行命令的责任分割开，委派给不同的对象。命令模式也允许请求方个接受方相互独立，使得请求方不需要知道接收方的接口等。

    意图：

        将一个请求封装为一个对象，从而可以使你应不同的请求对客户进行参数化。对于一些排队的请求可以进行取消操作。

    适用性：

        抽象出待执行的动作以参数化某对象可以使用回调表达这种参数化机制。

        在不同的时刻指定，排列和执行请求。一个 command 对象可以有一个与初始请求无关的生存期。

        支持对于取消，修改操作。

        用构建在原语上的高层操作构造一个系统，一个事务封装了对数据的一组变动，Command模式提供了对事务进行建模的方法，有一个公共的接口，使得可以使用同一种方式来调用所有的事务。也易于添加新事务以扩展系统。

### 组合模式（Composite）

    组合模式间对象组织到树结构中，可以用来描述整体和部分的关系。组合模式就是一个处理对象的树结构的模式。

    意图：

        将对象组合成树形结构以表示“部分 - 整体”的层次关系，使用户对于单个对象和组合对象使用具有一致性。

    适用性：

        对于对象希望表现出 部分 - 整体 的层次结构。

        希望统一使用组合结构中的所有对象。

### 装饰者模式（Decorator）

    装饰者模式以对客户端透明的方式来扩展对象功能，是继承关系的一个替代方案，提供比继承更多的灵活性。动态给对象添加功能，这些功能也能动态撤销。常用于对增加一些基本功能的排列组合而产生的非常大量的功能。

    意图：

        动态的给一个对象添加一些额外的职责。就增加功能来说，装饰者模式比生成子类更为灵活。

    适用性：

        在不影响其他的对象情况下，以动态，透明的方式给单个兑现添加职责。

        当无法使用类似生成子类的方式进行类的拓展的时候。如有大量的独立应用，为支持各种组合那么将出现数量级爆炸的情况，或者类定义被隐藏等不能用于生成子类。

### 外观模式（Facade 门面模式）

    外部与一个子系统的通信必须通过一个统一的门面对象进行。门面模式提供一个更高级的接口使得子系统更容易使用。每一个子系统只有一个门面模式，此门面模式只有一个实例，即单例模式。整个系统中可以有多种门面类。

    意图：

        为子类系统中的一组接口提供一致的界面，外观模式定义一个高级接口以便子类更容易的调用。

    适用性：

        当需要为一个非常复杂的子类系统提供一个简单的接口的时候。

        外观模式提供一个简单的缺省视图，这一视图对大多数用户来说已经OK，而对高级用户可以绕过外观层进行所需要的处理。

### 享元模式（Flyweight）

    享元模式以共享的方式高效的支持大量细粒度对象。共享的关键是区分内蕴状态和外蕴状态。内蕴状态存储在享元内部不会随环境的改变而有所不同。外蕴状态就是随环境的改变而变化的。两者相互独立互不影响。将可以共享的状态和非共享的状态从常规类里面区分出来。客户不可以直接创建共享对象，应当使用一个工厂对象负责创建共享的对象。享元模式可以大幅度的降低内存中对象的数量。

    意图：

        运用共享技术有效地支持大量细粒度的对象。

    适用性：

        一个程序中使用了大量的对象。

        由于对象数量造成了大量存储开销。

        大多对象的状态可以变为外部的状态

        可以使用较少的共享对象来进行处理。

### 解析器模式（Interpreter）

    给定一个语言后，解析器模式可以定义出其文法的表示，并提供一个解析器。客户端可以使用这个解析器来解析该语言中的语句。解析器模式将描述怎样再有了一个简单的文法后使用模式设计解析这些语句。该语言是指任何解析器对象能够解释的任何组合。在解析器模式中需要定义一个代表文法的命令类的等级结构，每一个命令都有一个对应的解释。

    意图：

        给定一种语言，定义文法表示，并且在定义一个解析器来转化该语言。

    适用性：

        简化文法所需要的逻辑处理，文法简单。

### 迭代器模式（Iterator）

    迭代器模式可以顺序访问一个聚集中的元素而不必暴露内部表象。多个对象聚集在一起形成的整体称之为聚集，聚集对象则是包含一组对象的容器。迭代模式将迭代逻辑封装到一个独立的对象中，与本身的聚集对象分离开来。迭代模式简化了聚集的逻辑思想，将行为独立分割开来方便调整修改替换。所以迭代算法可以独立于聚集对象进行变化。

    意图：

        提供一个方法顺序访问一个聚合对象中的各个元素而又不暴露该对象的内部表示。

    适用性：

        访问聚合对象，而不希望内部暴露。

        支持对于聚合对象的多种遍历。

        为遍历各种对象提供统一的接口。

### 中介者模式（Mediator）

    中介者模式包装了一系列对象相互作用的方式，使得这些对象不必相互有明显的联系。使耦合变得松散。当某些对象发生改变的时候也不会影响其他的对象之间的作用，保证各个对象之间的作用相互独立，这边并不是对象与对象之间，而是作用与作用之间的联系变得独立开。中介者模式将多对多的相互作用转化为一对多的相互作用，将对象的行为和协作抽象画，把对象的一些小行为与其他对象的相互作用分开处理。

    意图：

        使用中介者来封装一系列的对象之间的交互，不显示的表示各个对象的相互引用关系，降低对象之间的耦合度。

    适用性：

        定义良好的一组对象需要很复杂的通信方式。从而产生的相互依赖关系但以理解。

        一个对象直接对其他很多对象进行通信导致了难以复用该对象。

        需要定制一个分布在各个类中的行为又不想生成太多子类。

### 备忘录模式（Memento）

    备忘录对象是一个用来存储另外一个对象内部状态的快照对象，在意在不破坏对象封装的前提下将一个对象的状态捕捉到，存储在外部以便可以随时观察记录，以便将来可以还原到某一个状态。

    意图：

        在不破坏封装性的前提下捕获一个对象的内部状态并将该状态保存在对象之外来标记。

    适用性：

        需要注意必须保持对象的某一时刻状态并且不对对象造成封装性的破坏，不制造多余的暴露接口。

### 观察者模式（Observer）

    观察者模式定义一种一对多的状态，让多个观察者同时监听某一个对象。这个对象主题发生变化的时候会告知所有的观察者对象，使它们自动更新自己。

    意图：

        定义对象间的一种一对多的依赖关系，当一个对象发生改变的时候，所有依赖于该对象的其他对象都将得到通知，并且被自动更新。

    适用性：

        当抽象模型具有两个方面，其中一个方面依赖于另一个方面，将这两者分开封装在独立的对象中使得他们可以独立改变以及复用。

        当一个对象在改变的时候需要同时改变其他对象，并且这个对象集合有可能会改变。并且不能假定这些对象来增加耦合度。

### 原型模式（Prototype）

    通过给出一个原型对象来指明所要创建的对象类型，复用这个原型对象来创建出个更多的同类型对象。原型模式允许动态的增加或者减少产品类，可以不需要任何的等级结构，原型模式适用于任何结构。

    意图：

        用原型实例来指定对象创建的种类，并且通过拷贝这些原型来创建新的对象。

    适用性：

        需要实例化的类在运行时候指定。（对于前端的JSer来说，这是最熟悉不过的了）

### 代理模式（proxy）

    代理模式给某一个对象提供一个代理对象，并由代理对象控制对象源对象的引用。在某些情况下客户不想活着不能够直接引用一个人对象，代理对象在客户和目标对象中起到中介者的作用，代理对象可以仅仅只有一个被代理的接口，这时候代理对象不能创建被代理的对象，被代理的对象必须由系统的其他角色代为创建和传入。

    意图：

        为其他对象提供一种代理来来控制对于某个特定对象的访问或者处理。

    适用性：

        在需要使用比较爱通用和复杂的对象指针来代替简单指针的时候。

### 单例模式（Singleton）

    单例模式确保每一个类只有一个实例，而且自行的实例化并且想整个系统提供该实例的接口。单例模式只有在真正需要单一模式的时候使用。

    结构：

        保证一个类只有一个实例，并提供一个访问它的全局访问点。

    适用性：

        当类只能有一个实例，客户端通过规范的访问点来进行访问。

**第五种（静态内部类）：**

|  |
| --- |
| 1 public class Singleton {  2 private static class SingletonHolder{  3 private static final Singleton INSTSNCE = new Singleton();  4 }  5 private Singleton () {}  6 public static final Singleton getInstance () {  7 return SingletonHolder.INSTANCE;  8 }  9 } |

**第六种（枚举）：**

|  |
| --- |
| 1 public class Singleton {  2 INSTANCE;  3 public void whateverMethod() {  4  5 }  6 } |

### 状态模式（state）

    状态模式允许一个对象在其内部状态改变的时候改变行为。使得看上去和改变了类是一样的。状态模式将所研究的对象的行为包装在不同的对象里，每一个状态都属于一个抽象状态类的一个子类。状态模式需要对每一个系统可能取得的状态创建一个状态类的子类。当系统的状态发生变化的时候，系统便改变所选的子类。

    意图：

        允许一个对象在其内部状态改变的时候改变行为。

    适用性：

        一个对象的行为取决于它的状态，并且必须在运行时刻根据状态来改变行为。

### 策略模式（Strategy）

    策略模式针对一组算法，将每一个算法封装到具有共同接口的独立类中，从而使得它们可以相互替换。策略模式使得算法可以在不影响客户端的情况下发生变化。将行为与环境分开。环境类负责维持和查询行为类，各个算法在具体的策略类中提供。所以算法的增减不会影响环境和客户端。

    意图：

        定义一系列算法进行单独封装，并且是它们可以相互替换，使算法独立于客户变化而变化。

    适用性：

          许多相关的类，仅仅是行为上有所差别，策略模式可以实现在多个行为中选取一个来配置一个类。

### 模板方法模式（Template Method）

    模板方法模式准备一个抽象类，将部分逻辑以及具体方法以构造子类的方式实现，然后声明一些抽象方法来事子类实现剩余的逻辑。不同的子类可以以不同的方式来实现这些抽象方法，从而对剩余逻辑的实现。先定制一个顶级的逻辑框架，而将逻辑的细节刘诶具体的子类去实现。

    意图：

        定义一个算法骨架，将一些步骤延迟到子类中，使子类可以不改变一个算法结构却可以重新定义算法的步骤。

    适用性：

        一次性实现算法的不变部分，将可变部分留在子类中实现。

        一个类定义多种行为，并且这些行为在这个类中的操作是以多个条件语句的形式出现。

### 访问者模式（Visitor）

    访问者模式的目的是封装一些施加于某种数据结构元素之上的操作。一旦操作需要做修改，接受设个操作的数据结构可以保持不变。适用于数据结构相对未定的系统，将数据结构和作用于结构上的操作耦合解脱开使得操作可以相对自由的演化。访问者模式使得增加新的操作变得更为容易。访问者模式将有关的行为集中到一个访问者对象中，而不是分散在一个个环节节点中。当使用访问者模式的时候要将尽可能多的对象浏览逻辑放在访问者类中，而不是子类中。

    意图：

        表示一个作用于某个对象结构中的各元素的操作，使你可以在不改变各个元素的类的情况下定义作用于这些元素的新的操作。

    适用性：

        一个对象结构中包含多个类对象，具有不同的接口，而你希望对这些对象实施一些依赖于具体某个类的操作。

        需要对一个对象结构中的对象进行很多不同的并且不想关的操作，这些操作需要避免污染对象。访问者模式可以将相关的操作集中放入一个类中，被对象结构所共享，从而对象结构中的每个子对象都可以被共享到。

## 面向对象设计七大原则

### 1. 单一职责原则（Single Responsibility Principle）

每一个类应该专注于做一件事情。

### 2. 里氏替换原则（Liskov Substitution Principle）

超类存在的地方，子类是可以替换的。

### 3. 依赖倒置原则（Dependence Inversion Principle）

实现尽量依赖抽象，不依赖具体实现。

### 4. 接口隔离原则（Interface Segregation Principle）

应当为客户端提供尽可能小的单独的接口，而不是提供大的总的接口。

### 5. 迪米特法则（Law Of Demeter）

又叫最少知识原则，一个软件实体应当尽可能少的与其他实体发生相互作用。

### 6. 开闭原则（Open Close Principle）

面向扩展开放，面向修改关闭。

### 7. 组合/聚合复用原则（Composite/Aggregate Reuse Principle CARP）

尽量使用合成/聚合达到复用，尽量少用继承。原则： 一个类中有另一个类的对象。

## J2EE

J2EE平台由一整套服务（Services）、应用程序接口（APIs）和协议构成，它对开发基于Web的多层应用提供了功能支持，下面对J2EE中的13种技术规范进行简单的描述（限于篇幅，这里只能进行简单的描述）：

### JDBC(Java Database Connectivity)

JDBC API为访问不同数据库提供了统一的路径,像[ODBC](https://baike.baidu.com/item/ODBC" \t "_blank)一样,[JDBC](https://baike.baidu.com/item/JDBC" \t "_blank)开发者屏蔽了一些细节问题,另外,JDBC对数据库的访问也具有平台无关性.

### JNDI(Java Naming and Directory Interface)

JNDI API 被用于执行名字和目录服务.它提供了一致的模型来存取和操作企业级的资源DNS和LDAP,本地文件系统,或应用服务器中的对象.

### EJB(Enterprise JavaBean)

J2EE技术之所以赢得广泛重视的原因之一就是EJB.它提供了一个框架来开发和实施分布式商务逻辑,由此很显著的简化了具有可伸缩性和高度复杂的企业级应用程序的开发.EJB规范定义了EJB组件在何时如何与它们的容器进行交互作用.容器负责提供公用的服务,例如目录服务,事务管理,安全性,资源缓冲池以及容错性.但这里值得注意的是,EJB并不是实现J2EE的唯一路径.正是由于J2EE的开放性,使得所有的厂商能够以一种和EJB平行的方式来达到同样的目地.

### RMI(Remote Method Invoke)

远程方法请求,RMI协议调用远程对象上的方法.它使用了序列化的方式在客户端和服务器之间传递数据.RMI是一种被EJB使用的更底层的协议.

### Java IDL/CORBA(通用对象请求代理架构是软件构建的一个标准 )

在Java IDL的支持下,开发人员可以将Java和CORBA集成在一起.他们可以创建Java对象并使之可在CORBA ORB中展开,或者他们还可以创建Java类并和其它ORB一起展开的CORBA对象客户.后一种方法提供了另外一种途径,通过它Java可以被用于将你的新的应用程序和旧的系统集合在一起.

### JSP

JSP页面由[HTML](https://baike.baidu.com/item/HTML" \t "_blank)（[标准通用标记语言](https://baike.baidu.com/item/%E6%A0%87%E5%87%86%E9%80%9A%E7%94%A8%E6%A0%87%E8%AE%B0%E8%AF%AD%E8%A8%80" \t "_blank)下的一个应用）代码和嵌入其中的Java代码组成.服务器在

页面被客户端所请求以后对这些Java代码进行处理,然后将生成的HTML页面返回给客户端浏览器.

### Java Servlet

Servlet 是一种小型的Java程序,它扩展了web服务器的功能.作为一种服务器的应用,当被请求时开始执行,这和CGI Perl脚本很相似.Servlet提供的功能大多和JSP类似,不过实现的方式不同.JSP通常是大多数的HTML代码中嵌入少量的Java代码,而servlet全部由java写成并且生成HTML.

### XML

[XML](https://baike.baidu.com/item/XML" \t "_blank)（[标准通用标记语言](https://baike.baidu.com/item/%E6%A0%87%E5%87%86%E9%80%9A%E7%94%A8%E6%A0%87%E8%AE%B0%E8%AF%AD%E8%A8%80" \t "_blank)的子集）是一种可以用来定其它标记语言的语言.它被用来在不同的商务过程中共享数据.XML的发展和java是相互独立的,但是,它和java具有的相同目标是平台独立性.

### JMS

MS是用于和面向对象消息的中间件相互通信的应用程序接口.它既支持点对点的域,又支持发布/订阅类型的域,并且提供了下列类型的支持:消息传递,事务型消息的传递,一致性消息和具有持久性的订阅者支持.JMS还提供了另一种方式来对新系统和旧后台系统相互集成.

### JTA

[JTA](https://baike.baidu.com/item/JTA" \t "_blank)定义了一种标准API,应用程序由此可以访问各种事务监控.

### JTS

JTS是CORBA OTS事务监控的基本实现.JTS规定了事务管理的实现方法.该事务管理器是在高层支持java Transaction API规范,并且在较低层次实现OMG OTS specification 和Java对象.JTS事务管理器为应用程序服务器,资源管理器,独立的应用以及通信资源管理器提供了事务服务.

### JavaMail

JavaMail是用于存取邮件服务器的API,它提供了一套邮件服务器的抽象类.不仅支持SMTP服务器,也支持IMAP服务器.

### JAF(JavaBeans Activation Framework)

JAVA安全认证框架。提供一些安全控制方面的框架。让开发者通过各种部署和自定义实现自己的个性安全控制策略。

# 数据库

## 注意

#### 新增列位置

MySQL增加列的时候可以指定此列的位置；

需要两个关键字: FIRST 和 AFTER

FIRST 表示增加此列为第一个列；AFTER表示增加在某个列之后。

注意MySQL增加列指定位置时没有BEFORE的用法；

Oracle增加列的时候没有直接方法可以指定此列的位置；

#### 取第一行

对于rownum在oracle的使用的时候，有几点需要注意：

(1) rownum 对于等于某值的查询条件

如果希望找到学生表中第一条学生的信息，可以使用rownum=1作为条件。但是想找到学生表中第二条学生的信息，使用rownum=2结果查不到数据。因为rownum都是从1开始，但是1以上的自然数在rownum做等于判断是时认为都是false条件，所以无法查到rownum = n(n>1的自然数)。

(2) rownum对于大于某值的查询条件

如果想找到从第二行记录以后的记录，当使用rownum>2是查不出记录的，原因是由于rownum是一个总是从1开始的伪列，Oracle 认为rownum> n(n>1的自然数)这种条件依旧不成立，所以查不到记录。

 (3) rownum对于小于某值的查询条件

rownum对于rownum<n(n>1的自然数)的条件认为是成立的，所以可以找到记录。比如 rownum < 3

(4) rownum和排序 使用子查询

Oracle中的rownum的是在取数据的时候产生的序号，所以想对指定排序的数据去指定的rowmun行数据就必须注意了。

## （一）MySql

### 字符集

mysql数据库的默认编码是latin1，show variables like "%character%";

### 命令行操作

#### 连接MYSQL

　　格式： mysql -h主机地址 -u用户名 -p用户密码

　　1、 连接到本机上的MYSQL。

　　首先打开DOS窗口，然后进入目录mysql\bin，再键入命令mysql -u root -p，回车后提示你输密码.；

　　2、连接到远程主机上的MYSQL。假设远程主机的IP为：110.110.110.110，用户名为root,密码为abcd123。则键入以下命 令：

　　mysql -h110.110.110.110 -u root -p 123;(注:u与root之间可以不用加空格，其它也一样)

　　3、 退出MYSQL命令： exit (回车)

#### 修改密码

　　格式：mysqladmin -u用户名 -p旧密码 password 新密码。 例如

　　1、 给root加个密码ab12。首先在DOS下进入目录mysql\bin，然后键入以下命令

　　mysqladmin -u root -password ab12

　　2、 再将root的密码改为djg345。

　　mysqladmin -u root -p ab12 password \*\*\*\*\*\*

#### 常用命令

　　1、显示当前数据库服务器中的数据库列表：

　　mysql> SHOW DATABASES;

　　2、建立数据库：

　　mysql> CREATE DATABASE 库名;

　　mysql> CREATE DATABASE IF NOT EXISTS my\_db default charset utf8 COLLATE utf8\_general\_ci;

　　3、建立数据表：

　　mysql> USE 库名;

　　mysql> CREATE TABLE 表名 (字段名 VARCHAR(20), 字段名 CHAR(1));

　　4、删除数据库：

　　mysql> DROP DATABASE 库名;

　　5、删除数据表：

　　mysql> DROP TABLE 表名;

　　6、将表中记录清空：

　　mysql> DELETE FROM 表名;

　　7、往表中插入记录：

　　mysql> INSERT INTO 表名 VALUES ("hyq","M");

　　8、更新表中数据：

　　mysql-> UPDATE 表名 SET 字段名1='a',字段名2='b' WHERE 字段名3='c';

　　9、用文本方式将数据装入数据表中：

　　mysql> LOAD DATA LOCAL INFILE "D:/mysql.txt" INTO TABLE 表名;

　　10、导入.sql文件命令：

　　mysql> USE 数据库名;

　　mysql> SOURCE d:/mysql.sql;

　　11、命令行修改root密码：

　　mysql> UPDATE mysql.user SET password=PASSWORD('新密码') WHERE User='root';

　　mysql> FLUSH PRIVILEGES;

#### 将文本数据转到数据库中

　　1、文本数据应符合的格式：字段数据之间用tab键隔开，null值用\n来代替.例：

　　3 rose 飞数科技1 1976-10-10

　　4 mike 飞数科技2 1975-12-23

　　假设你把这两组数据存为速度sudu.txt文件，放在c盘根目录下。

　　2、数据传入命令 load data local infile "c:\sudu.txt" into table 表名;

　　注意：你最好将文件复制到\mysql\bin目录下，并且要先用use命令打表所在的库。

#### 备份数据库：(命令在DOS的\mysql\bin目录下执行)

　　1.导出整个数据库

　　导出文件默认是存在mysql\bin目录下

　　mysqldump -u 用户名 -p 数据库名 > 导出的文件名

　　mysqldump -u user\_name -p123456 database\_name > outfile\_name.sql

　　2.导出一个表

　　mysqldump -u 用户名 -p 数据库名 表名> 导出的文件名

　　mysqldump -u user\_name -p database\_name table\_name > outfile\_name.sql

　　3.导出一个数据库结构

　　mysqldump -u user\_name -p -d --add-drop-table database\_name > outfile\_name.sql

　　-d 没有数据 --add-drop-table 在每个create语句之前增加一个drop table

　　4.带语言参数导出

　　mysqldump -uroot -p --default-character-set=latin1 --set-charset=gbk --skip-opt database\_name > outfile\_name.sql

## （二）Oracle

|  |
| --- |
| select \* from v$version;--查看数据库版本  Sqlplus——  Conn /as sysdba;——  select username from dba\_users；——查看用户列表  alter user system identified by 123456;——修改密码  grant create session,resource to <user>;——用户授权 |

今天想要修改一个用户的密码，但是在执行完alter user语句后，提示用户不存在。查看dba\_users视图后，该用户的确是存在的，但是注意到一个细节是用户名是小写的，其他的用户名都是大写。在群内咨询过大神后，原来是在创建用户时，把用户名用双引号包起来了，如果没有用双引号，数据库会自动将用户名改为大写，但是用了双引号后，则保持小写。解决方法为，若创建过程用引号包起来，则在后续的使用过程中，也需要将用户名用引号包起来。在本机测试过程如下：  
创建用户成功后，紧接着修改用户密码，提示用户不存在：

### oracle存储过程

|  |
| --- |
| create or replace procedure test(  param1 varchar2,  param2 out varchar2,  param3 in out varchar2  ) as  v\_name varchar2(20);  begin  v\_name := 'david';  param3 := v\_name;  dbms\_output.put\_line('param3:'||param3);  exception  //抛出异常处理  end;  第一行 是创建语句，如果不存在则创建，如果存在，则去替换,test为存储过程的名称  第二行到第四行 是存储过程的参数，参数名，参数传递方式（IN ,OUT, IN OUT），参数类型  参数注意点   * 参数名不能重复 * 传递方式详解 * IN 表示输入参数，按值传递方式，如果不指定，则默认是IN，在存储过程中 不能被重新赋值，只能使用，相当于java中的final * OUT 表示输出参数,可以理解为按引用传递方式。可以作为存储过程的输出结果 * IN OUT 即可作输入参数，也可作输出参数。即可修改 * 参数的数据类型只需要指明类型名即可，不需要指定宽度 * 参数的宽度由外部调用者决定 * 存储过程可以有参数，也可以没有参数   第五行 变量声明块：紧跟着的as(is)关键字，用于声明变量  第六行 声明变量(存储过程需要用到的变量)，必须指定宽度，  第七行 过程语句块，从begin关键字开始为过程的语句块，实现具体的逻辑  第十一行 异常处理块，关键字为exception，该部分可写可不写  第十二行 结束块，end关键字结尾 |

### count（\*）、count（1）和Count([列])区别？

### 各种 join 用法

(1)INNER JOIN（内连接）

(2)LEFT JOIN（左连接）

(3)RIGHT JOIN（右连接）

(4)OUTER JOIN（外连接）

(5)LEFT JOIN EXCLUDING INNER JOIN（左连接-内连接）

(6)RIGHT JOIN EXCLUDING INNER JOIN（右连接-内连接）

(7)OUTER JOIN EXCLUDING INNER JOIN（外连接-内连接）

### 异常

#### Springboot连接mysql数据库异常：

The server time zone value 'ÖÐ¹ú±ê×¼Ê±¼ä' is unrecognized or represents more than one time zone. You must configure either the server or JDBC driver (via the serverTimezone configuration property) to use a more specifc time zone value if you want to utilize time zone support.

解决方法：

jdbc:mysql://127.0.0.1/my\_blog\_db?characterEncoding=utf-8&&useSSl=false&&autoReconnect=true&&failOverReadOnly=FALSE&&serverTimezone=Asia/Shanghai

## （三）Redis

**redis是键值对的数据库，有5中主要数据类型**：

字符串类型（string），散列类型（hash），列表类型（list），集合类型（set），有序集合类型（zset）

### 字符串类型string

字符串类型是Redis的最基本类型，它可以存储任何形式的字符串。其它的四种类型都是字符串类型的不同形式。

最基本的命令：GET、SET         语法：GET key，SET key value   value如果有空格需要双引号以示区分

整数递增：INCR                      语法：INCR key    默认值为0，所以首先执行命令得到 1 ，不是整型提示错误

增加指定的整数：INCRBY          语法：INCRBY key increment

整数递减：DECR                     语法：DECR key   默认值为0，所以首先执行命令得到 -1，不是整型提示错误

减少指定的整数：DECRBY         语法：DECRBY key increment

增加指定浮点数：INCRBYFLOAT 语法：INCRBYFLOAT key increment  与INCR命令类似，只不过可以递增一个双精度浮点数

向尾部追加值：APPEND             语法：APPEND key value   redis客户端并不是输出追加后的字符串，而是输出字符串总长度

获取字符串长度：STRLEN          语法：STRLEN key  如果键不存在返回0，注意如果有中文时，一个中文长度是3，redis是使用UTF-8编码中文的

获取多个键值：MGET                语法：MGET key [key ...]  例如：MGET key1 key2

设置多个键值：MSET                语法：MSET key value [key value ...]  例如：MSET key1 1 key2 "hello redis"

二进制指定位置值：GETBIT        语法：GETBIT key offset   例如：GETBIT key1 2 ，key1为hello 返回 1，返回的值只有0或1，

　　　　　　　　　　　　　　　　　　 当key不存在或超出实际长度时为0

设置二进制位置值：SETBIT       语法：SETBIT key offset value ，返回该位置的旧值

二进制是1的个数：BITCOUNT    语法：BITCOUNT key [start end] ，start 、end为开始和结束字节

位运算：BITOP                       语法：BITOP operation destkey key [key ...]  ，operation支持AND、OR、XOR、NOT

偏移：BITPOS                        语法：BITPOS key bit [start] [end]

### 散列类型hash

设置单个：HSET                      语法：HSET key field value，不存在时返回1，存在时返回0，没有更新和插入之分

设置多个：HMSET                    语法：HMSET key field value [field value ...]

读取单个：HGET                      语法：HGET key field，不存在是返回nil

读取多个：HMGET                    语法：HMGET key field [field ...]

读取全部：HGETALL                 语法：HGETALL key，返回时字段和字段值的列表

判断字段是否存在：HEXISTS      语法：HEXISTS key field，存在返回1 ，不存在返回0

字段不存在时赋值：HSETNX       语法：HSETNX key field value，与hset命令不同，hsetnx是键不存在时设置值

增加数字：HINCRBY                 语法：HINCRBY key field increment ，返回增加后的数，不是整数时会提示错误

删除字段：HDEL                      语法：HDEL key field [field ...] ，返回被删除字段的个数

只获取字段名：HKEYS               语法：HKEYS key ，返回键的所有字段名

只获取字段值：HVALS              语法：HVALS key  ，返回键的所有字段值

字段数量：HLEN                      语法：HLEN key ，返回字段总数

### 列表类型（list）

内部使用双向链表实现，所以获取越接近两端的元素速度越快，但通过索引访问时会比较慢

添加左边元素：LPUSH               语法：LPUSH key value [value ...]  ，返回添加后的列表元素的总个数

添加右边元素：RPUSH              语法：RPUSH key value [value ...]  ，返回添加后的列表元素的总个数

移除左边第一个元素：LPOP        语法：LPOP key  ，返回被移除的元素值

移除右边第一个元素：RPOP        语法：RPOP key ，返回被移除的元素值

列表元素个数：LLEN                语法：LLEN key， 不存在时返回0，redis是直接读取现成的值，并不是统计个数

获取列表片段：LRANGE           语法：LRANGE key start stop，如果start比stop靠后时返回空列表，0 -1 返回整个列表

                                                    正数时：start 开始索引值，stop结束索引值（索引从0开始）

                                                    负数时：例如 lrange num -2 -1，-2表示最右边第二个，-1表示最右边第一个，

删除指定值：LREM                  语法：LREM key count value，返回被删除的个数

                                                   count>0，从左边开始删除前count个值为value的元素

                                                   count<0，从右边开始删除前|count|个值为value的元素

                                                   count=0，删除所有值为value的元素

索引元素值：LINDEX               语法：LINDEX key index ，返回索引的元素值，-1表示从最右边的第一位

设置元素值：LSET                  语法：LSET key index value

保留列表片段：LTRIM              语法：LTRIM key start stop，start、top 参考lrange命令

一个列表转移另一个列表：RPOPLPUSH      语法：RPOPLPUSH source desctination ，从source列表转移到desctination列表，

                                                                 该命令分两步看，首先source列表RPOP右移除，再desctination列表LPUSH

### 集合类型（set）

集合类型值具有唯一性，常用操作是向集合添加、删除、判断某个值是否存在，集合内部是使用值为空的散列表实现的。

添加元素：SADD                    语法：SADD key member [member ...] ，向一个集合添加一个或多个元素，因为集合的唯一性，所以添加相同值时会被忽略。

　　　　　　　　　　　　　　　         返回成功添加元素的数量。

删除元素：SREM                    语法：SREM key member [member ...] 删除集合中一个或多个元素，返回成功删除的个数。

获取全部元素：SMEMBERS      语法：SMEMBERS key ，返回集合全部元素

值是否存在：SISMEMBER        语法：SISMEMBER key member ，如果存在返回1，不存在返回0

差运算：SDIFF                      语法：SDIFF key [key ...] ，例如：集合A和集合B，差集表示A-B，在A里有的元素B里没有，返回差集合；多个集合(A-B)-C

交运算：SINTER             　　 语法：SINTER key [key ...]，返回交集集合，每个集合都有的元素

并运算：SUNION　　　　　　  语法：SUNION key [key ...]，返回并集集合，所有集合的元素

集合元素个数：SCARD           语法：SCARD key ，返回集合元素个数

集合运算后存储结果                语法：SDIFFSTROE destination key [key ...] ，差运算并存储到destination新集合中

　　　　　　　　　　 　　　　　　　 SINTERSTROE destination key [key ...]，交运算并存储到destination新集合中

                                                  SUNIONSTROE destination key [key ...]，并运算并存储到destination新集合中

随机获取元素：SRANDMEMGER 语法：SRANDMEMBER key [count]，根据count不同有不同结果，count大于元素总数时返回全部元素

　　　　　　　　　　　　　　　　　　count>0 ，返回集合中count不重复的元素

　　　　　　　　　　　　　　　　　　count<0，返回集合中count的绝对值个元素，但元素可能会重复

弹出元素：SPOP                     语法：SPOP key [count] ，因为集合是无序的，所以spop会随机弹出一个元素

### 有序集合类型

添加集合元素：ZADD              语法：ZADD key [NX|XX] [CH] [INCR] score member [score member ...]，不存在添加，存在更新。

获取元素分数：ZSCORE          语法：ZSCORE key member ，返回元素成员的score 分数

元素小到大：ZRANGE             语法：ZRANGE key start top [WITHSCORES] ，参考LRANGE ，加上withscores 返回带元素，即元素，分数

                                                  当分数一样时，按元素排序

元素大到小：ZREVRANGE       语法：ZREVRANGE key start [WITHSCORES] ，与zrange区别在于zrevrange是从大到小排序

指定分数范围元素：ZRANGEBYSCORE   语法：ZRANGEBYSCORE key min max [WITHSCORE] [LIMIT offest count]

 　　　　　　　　　　　　　　　返回从小到大的在min和max之间的元素，( 符号表示不包含，例如：80-100，(80 100，

　　　　　　　　　　　　　　    withscore返回带分数

　　　　　　　　　　　　　　    limit offest count 向左偏移offest个元素，并获取前count个元素

指定分数范围元素：ZREVRANGESCORE   语法：ZREVRANGEBYSCORE key max  min [WITHSCORE] [LIMIT offest count]

　　　　　　　　　　　　　　　 与zrangebyscore类似，只不过该命令是从大到小排序的。

增加分数：ZINCRBY                语法：ZINCRBY key increment member ，注意是增加分数，返回增加后的分数；如果成员不存在，则添加一个为0的成员。

# 前端

## （一）JavaScript

### JSON

#### JSON概念

JSON(JavaScript Object Notation, JS 对象简谱)：

(1)是一种轻量级的数据交换格式。

(2)基于 ECMAScript (欧洲计算机协会制定的js规范)的一个子集

(3)采用完全独立于编程语言的文本格式来存储和表示数据。

(4)易于人阅读和编写，同时也易于机器解析和生成，并有效地提升网络传输效率。

#### JSON 和 JS 对象互转

要实现从对象转换为 JSON 字符串，使用 JSON.stringify() 方法：

|  |  |
| --- | --- |
| 1 | var json = JSON.stringify({a: 'Hello', b: 'World'}); //结果是 '{"a": "Hello", "b": "World"}' |

要实现从 JSON 转换为对象，使用 JSON.parse() 方法：

|  |  |
| --- | --- |
| 1 | var obj = JSON.parse('{"a": "Hello", "b": "World"}'); //结果是 {a: 'Hello', b: 'World'} |

#### JSON支持的数据类型

一、4种基本类型  
字符串(Strings)     如:"a"

数字(numbers)      如:1，1.02

布尔型(booleans)   true  false  
Null值(null)        　null  
二、两种结构类型  
对象        {"name":"zhangsan","age":18}    键为String ，值为Json的任意数据类型之一。对象的键值对是无序的。

数组        [1,2,3]                顺序排列的零个或多个Json数据类型

## （二）AJAX

## （三）jQuery

## （四）CSS

## （五）HTML

# WEB服务器

## （一）Tomcat

# WEB应用

## （一）Cookie

Cookie，有时也用其复数形式 Cookies，指某些网站为了辨别用户身份、进行 session 跟踪而储存在用户本地终端上的数据（通常经过加密）。最新规范是 RFC6265。（可以叫做浏览器缓存）

Cookie 在计算机中是个存储在浏览器目录中的文本文件，当浏览器运行时，存储在 RAM 中发挥作用 （此种 Cookies 称作 Session Cookies），一旦用户从该网站或服务器退出，Cookie 可存储在用户本地的硬盘上 （此种 Cookies 称作 Persistent Cookies）。

通常情况下，当用户结束浏览器会话时，系统将终止所有的 Cookie。当 Web 服务器创建了Cookies 后，只要在其有效期内，当用户访问同一个 Web 服务器时，浏览器首先要检查本地的Cookies，并将其原样发送给 Web 服务器。这种状态信息称作“Persistent Client State HTTP Cookie” ，简称为 Cookies 。

在同一个页面中设置 Cookie，实际上是按从后往前的顺序进行的。如果要先删除一个 Cookie，再写入一个 Cookie，则必须先写写入语句，再写删除语句，否则会出现错误。

Cookie 必须在 HTML 文件的内容输出之前设置；不同的浏览器 (Netscape Navigator、Internet Explorer) 对 Cookie 的处理不一致，使用时一定要考虑；客户端用户如果设置禁止 Cookie，则 Cookie 不能建立。 并且在客户端，一个浏览器能创建的 Cookie 数量最多为 300 个，并且每个不能超过 4KB，每个 Web 站点能设置的 Cookie 总数不能超过 20 个。

Cookie在生成时就会被指定一个Expire值，这就是Cookie的生存周期，在这个周期内Cookie有效，超出周期Cookie就会被清除。有些页面将Cookie的生存周期设置为“0”或负值，这样在关闭浏览器时，就马上清除Cookie，不会记录用户信息，更加安全。

# 框架

## Spring

使用构造器注入的方法，可以明确成员变量的加载顺序。

## Spring Boot

Maven 整合了所有的 Jar 包，Spring Boot 整合了所有的框架。

pom.xml文件中默认有两个模块：

spring-boot-starter：核心模块，包括自动配置支持、日志和YAML；

spring-boot-starter-test：测试模块，包括JUnit、Hamcrest、Mockito。

|  |
| --- |
| 该模块在完整的打包环境下运行的时候会被禁用。如果你使用java -jar启动应用或者用一个特定的classloader启动，它会认为这是一个“生产环境”。  <dependencies>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-devtools</artifactId>  <optional>true</optional>  </dependency>  </dependencies>  <build>  <plugins>  <plugin>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-maven-plugin</artifactId>  <configuration>  <fork>true</fork>  </configuration>  </plugin>  </plugins>  </build> |

## Spring Cloud

 spring cloud子项目包括：

  Spring Cloud Config：配置管理开发工具包，可以让你把配置放到远程服务器，目前支持本地存储、Git以及Subversion。

  Spring Cloud Bus：事件、消息总线，用于在集群（例如，配置变化事件）中传播状态变化，可与Spring Cloud Config联合实现热部署。

  Spring Cloud Netflix：针对多种Netflix组件提供的开发工具包，其中包括Eureka、Hystrix、Zuul、Archaius等。

  Netflix Eureka：云端负载均衡，一个基于 REST 的服务，用于定位服务，以实现云端的负载均衡和中间层服务器的故障转移。

  Netflix Hystrix：容错管理工具，旨在通过控制服务和第三方库的节点,从而对延迟和故障提供更强大的容错能力。

  Netflix Zuul：边缘服务工具，是提供动态路由，监控，弹性，安全等的边缘服务。

  Netflix Archaius：配置管理API，包含一系列配置管理API，提供动态类型化属性、线程安全配置操作、轮询框架、回调机制等功能。

  Spring Cloud for Cloud Foundry：通过Oauth2协议绑定服务到CloudFoundry，CloudFoundry是VMware推出的开源PaaS云平台。

  Spring Cloud Sleuth：日志收集工具包，封装了Dapper,Zipkin和HTrace操作。

  Spring Cloud Data Flow：大数据操作工具，通过命令行方式操作数据流。

  Spring Cloud Security：安全工具包，为你的应用程序添加安全控制，主要是指OAuth2。

  Spring Cloud Consul：封装了Consul操作，consul是一个服务发现与配置工具，与Docker容器可以无缝集成。

  Spring Cloud Zookeeper：操作Zookeeper的工具包，用于使用zookeeper方式的服务注册和发现。

  Spring Cloud Stream：数据流操作开发包，封装了与Redis,Rabbit、Kafka等发送接收消息。

  Spring Cloud CLI：基于 Spring Boot CLI，可以让你以命令行方式快速建立云组件。

#### Spring Cloud Netflix

## Spring Security

### 优缺点

优点：

①提供了一套安全框架，而且这个框架是可以用的；

②提供了很多用户认证的功能，实现相关借口即可，节约大量开发工作；

③基于spring，易于集成到spring项目中，且封装了许多方法；

缺点：

①配置文件多，角色被“编码”到配置文件和源文件中，RBAC不明显；

②对于系统中用户、角色、权限之间的关系，没有可操作的界面；

③大数据量情况下，几乎不可用；

### 定时任务Quartz和Spring Task

## SpringMVC

### 接收Json字符串

在SpringMVC后台控制层获取参数的方式主要有两种，

一种是request.getParameter("name")，

另一种是用注解

@RequestParam直接获取。

@RequestBody+实体类；

@RequestBody+ Map<String, String> map；

@RequestBody+ List<Accomodation> list；

@PathVariable绑定uri模板变量值：通过@PathVariable可以绑定占位符参数到方法参数中，例如：  
https://images2018.cnblogs.com/blog/1106621/201808/1106621-20180829110012185-1551808708.png

@CookieValue绑定Cookie数据值

public String test(@CookieValue(value="SESSIONID", defaultValue="") String sessionId)

自动将SESSIONID的值绑定到字符串sessionId上，如果Cookie中没有sessionId时则默认为空

传入数据类型也可以是javax.servlet.http.Cookie类型

public String test2(@CookieValue(value="SESSIONID", defaultValue="") Cookie sessionId)

**@RequestHeader绑定请求头数据**

@RequestMapping(value="/header")

public String test(

@RequestHeader("User-Agent") String userAgent,

@RequestHeader(value="Accept") String[] accepts)

### 注解@PostMapping

1.  Content-Type

  MediaType，即是Internet Media Type，互联网媒体类型；也叫做MIME类型，在Http协议消息头中，使用Content-Type来表示具体请求中的媒体类型信息。

|  |
| --- |
| 类型格式：type/subtype(;parameter)? type  主类型，任意的字符串，如text，如果是\*号代表所有；  subtype 子类型，任意的字符串，如html，如果是\*号代表所有；  parameter 可选，一些参数，如Accept请求头的q参数， Content-Type的 charset参数。 |

常见的媒体格式类型如下：

    text/html ： HTML格式

    text/plain ：纯文本格式

    text/xml ：  XML格式

    image/gif ：gif图片格式

    image/jpeg ：jpg图片格式

    image/png：png图片格式

   以application开头的媒体格式类型：

   application/xhtml+xml ：XHTML格式

   application/xml     ： XML数据格式

   application/atom+xml  ：Atom XML聚合格式

   application/json    ： JSON数据格式

   application/pdf       ：pdf格式

   application/msword  ： Word文档格式

   application/octet-stream ： 二进制流数据（如常见的文件下载）

   application/x-www-form-urlencoded ： <form encType=””>中默认的encType，form表单数据被编码为key/value格式发送到服务器（表单默认的提交数据的格式）

   另外一种常见的媒体格式是上传文件之时使用的：

    multipart/form-data ： 需要在表单中进行文件上传时，就需要使用该格式

     以上就是我们在日常的开发中，经常会用到的若干content-type的内容格式。

2.   Spring MVC中关于关于Content-Type类型信息的使用

    首先我们来看看RequestMapping中的Class定义：

|  |
| --- |
| @Target({ElementType.METHOD, ElementType.TYPE})  @Retention(RetentionPolicy.RUNTIME)  @Documented  @Mapping  public @interface RequestMapping {        String[] value() default {};        RequestMethod[] method() default {};        String[] params() default {};        String[] headers() default {};        String[] consumes() default {};        String[] produces() default {};  } |

value:  指定请求的实际地址， 比如 /action/info之类。  
method：  指定请求的method类型， GET、POST、PUT、DELETE等  
consumes： 指定处理请求的提交内容类型（Content-Type），例如application/json, text/html;  
produces:    指定返回的内容类型，仅当request请求头中的(Accept)类型中包含该指定类型才返回  
params： 指定request中必须包含某些参数值是，才让该方法处理  
headers： 指定request中必须包含某些指定的header值，才能让该方法处理请求

其中，consumes， produces使用content-typ信息进行过滤信息；headers中可以使用content-type进行过滤和判断。

|  |
| --- |
| @RequestMapping(value = "/test", method = RequestMethod.GET, headers="Referer=http://www.ifeng.com/")  public void testHeaders(@PathVariable String ownerId, @PathVariable String petId) {    // implementation omitted  } |

 这里的Headers里面可以匹配所有Header里面可以出现的信息，不局限在Referer信息。

|  |
| --- |
| @RequestMapping(value = "/response/ContentType", headers = "Accept=application/json")  public void response2(HttpServletResponse response) throws IOException {      //表示响应的内容区数据的媒体类型为json格式，且编码为utf-8(客户端应该以utf-8解码)      response.setContentType("application/json;charset=utf-8");      //写出响应体内容      String jsonData = "{\"username\":\"zhang\", \"password\":\"123\"}";      response.getWriter().write(jsonData);  } |

服务器根据请求头“Accept=application/json”生产json数据。

当你有如下Accept头，将遵守如下规则进行应用：  
①Accept：text/html,application/xml,application/json  
      将按照如下顺序进行produces的匹配 ①text/html ②application/xml ③application/json  
②Accept：application/xml;q=0.5,application/json;q=0.9,text/html  
      将按照如下顺序进行produces的匹配 ①text/html ②application/json ③application/xml  
      参数为媒体类型的质量因子，越大则优先权越高(从0到1)  
③Accept：\*/\*,text/\*,text/html  
      将按照如下顺序进行produces的匹配 ①text/html ②text/\* ③\*/\*

即匹配规则为：最明确的优先匹配。

### Requests部分

|  |  |  |
| --- | --- | --- |
| **Header** | **解释** | **示例** |
| Accept | 指定客户端能够接收的内容类型 | Accept: text/plain, text/html |
| Accept-Charset | 浏览器可以接受的字符编码集。 | Accept-Charset: iso-8859-5 |
| Accept-Encoding | 指定浏览器可以支持的web服务器返回内容压缩编码类型。 | Accept-Encoding: compress, gzip |
| Accept-Language | 浏览器可接受的语言 | Accept-Language: en,zh |
| Accept-Ranges | 可以请求网页实体的一个或者多个子范围字段 | Accept-Ranges: bytes |
| Authorization | HTTP授权的授权证书 | Authorization: Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ== |
| Cache-Control | 指定请求和响应遵循的缓存机制 | Cache-Control: no-cache |
| Connection | 表示是否需要持久连接。（HTTP 1.1默认进行持久连接） | Connection: close |
| Cookie | HTTP请求发送时，会把保存在该请求域名下的所有cookie值一起发送给web服务器。 | Cookie: $Version=1; Skin=new; |
| Content-Length | 请求的内容长度 | Content-Length: 348 |
| Content-Type | 请求的与实体对应的MIME信息 | Content-Type: application/x-www-form-urlencoded |
| Date | 请求发送的日期和时间 | Date: Tue, 15 Nov 2010 08:12:31 GMT |
| Expect | 请求的特定的服务器行为 | Expect: 100-continue |
| From | 发出请求的用户的Email | From: user@email.com |
| Host | 指定请求的服务器的域名和端口号 | Host: www.zcmhi.com |
| If-Match | 只有请求内容与实体相匹配才有效 | If-Match: “737060cd8c284d8af7ad3082f209582d” |
| If-Modified-Since | 如果请求的部分在指定时间之后被修改则请求成功，未被修改则返回304代码 | If-Modified-Since: Sat, 29 Oct 2010 19:43:31 GMT |
| If-None-Match | 如果内容未改变返回304代码，参数为服务器先前发送的Etag，与服务器回应的Etag比较判断是否改变 | If-None-Match: “737060cd8c284d8af7ad3082f209582d” |
| If-Range | 如果实体未改变，服务器发送客户端丢失的部分，否则发送整个实体。参数也为Etag | If-Range: “737060cd8c284d8af7ad3082f209582d” |
| If-Unmodified-Since | 只在实体在指定时间之后未被修改才请求成功 | If-Unmodified-Since: Sat, 29 Oct 2010 19:43:31 GMT |
| Max-Forwards | 限制信息通过代理和网关传送的时间 | Max-Forwards: 10 |
| Pragma | 用来包含实现特定的指令 | Pragma: no-cache |
| Proxy-Authorization | 连接到代理的授权证书 | Proxy-Authorization: Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ== |
| Range | 只请求实体的一部分，指定范围 | Range: bytes=500-999 |
| Referer | 先前网页的地址，当前请求网页紧随其后,即来路 | Referer: http://www.zcmhi.com/archives/71.html |
| TE | 客户端愿意接受的传输编码，并通知服务器接受接受尾加头信息 | TE: trailers,deflate;q=0.5 |
| Upgrade | 向服务器指定某种传输协议以便服务器进行转换（如果支持） | Upgrade: HTTP/2.0, SHTTP/1.3, IRC/6.9, RTA/x11 |
| User-Agent | User-Agent的内容包含发出请求的用户信息 | User-Agent: Mozilla/5.0 (Linux; X11) |
| Via | 通知中间网关或代理服务器地址，通信协议 | Via: 1.0 fred, 1.1 nowhere.com (Apache/1.1) |
| Warning | 关于消息实体的警告信息 | Warn: 199 Miscellaneous warning |

### Responses 部分

|  |  |  |
| --- | --- | --- |
| **Header** | **解释** | **示例** |
| Accept-Ranges | 表明服务器是否支持指定范围请求及哪种类型的分段请求 | Accept-Ranges: bytes |
| Age | 从原始服务器到代理缓存形成的估算时间（以秒计，非负） | Age: 12 |
| Allow | 对某网络资源的有效的请求行为，不允许则返回405 | Allow: GET, HEAD |
| Cache-Control | 告诉所有的缓存机制是否可以缓存及哪种类型 | Cache-Control: no-cache |
| Content-Encoding | web服务器支持的返回内容压缩编码类型。 | Content-Encoding: gzip |
| Content-Language | 响应体的语言 | Content-Language: en,zh |
| Content-Length | 响应体的长度 | Content-Length: 348 |
| Content-Location | 请求资源可替代的备用的另一地址 | Content-Location: /index.htm |
| Content-MD5 | 返回资源的MD5校验值 | Content-MD5: Q2hlY2sgSW50ZWdyaXR5IQ== |
| Content-Range | 在整个返回体中本部分的字节位置 | Content-Range: bytes 21010-47021/47022 |
| Content-Type | 返回内容的MIME类型 | Content-Type: text/html; charset=utf-8 |
| Date | 原始服务器消息发出的时间 | Date: Tue, 15 Nov 2010 08:12:31 GMT |
| ETag | 请求变量的实体标签的当前值 | ETag: “737060cd8c284d8af7ad3082f209582d” |
| Expires | 响应过期的日期和时间 | Expires: Thu, 01 Dec 2010 16:00:00 GMT |
| Last-Modified | 请求资源的最后修改时间 | Last-Modified: Tue, 15 Nov 2010 12:45:26 GMT |
| Location | 用来重定向接收方到非请求URL的位置来完成请求或标识新的资源 | Location: http://www.zcmhi.com/archives/94.html |
| Pragma | 包括实现特定的指令，它可应用到响应链上的任何接收方 | Pragma: no-cache |
| Proxy-Authenticate | 它指出认证方案和可应用到代理的该URL上的参数 | Proxy-Authenticate: Basic |
| refresh | 应用于重定向或一个新的资源被创造，在5秒之后重定向（由网景提出，被大部分浏览器支持） | Refresh: 5; url=  http://www.zcmhi.com/archives/94.html |
| Retry-After | 如果实体暂时不可取，通知客户端在指定时间之后再次尝试 | Retry-After: 120 |
| Server | web服务器软件名称 | Server: Apache/1.3.27 (Unix) (Red-Hat/Linux) |
| Set-Cookie | 设置Http Cookie | Set-Cookie: UserID=JohnDoe; Max-Age=3600; Version=1 |
| Trailer | 指出头域在分块传输编码的尾部存在 | Trailer: Max-Forwards |
| Transfer-Encoding | 文件传输编码 | Transfer-Encoding:chunked |
| Vary | 告诉下游代理是使用缓存响应还是从原始服务器请求 | Vary: \* |
| Via | 告知代理客户端响应是通过哪里发送的 | Via: 1.0 fred, 1.1 nowhere.com (Apache/1.1) |
| Warning | 警告实体可能存在的问题 | Warning: 199 Miscellaneous warning |
| WWW-Authenticate | 表明客户端请求实体应该使用的授权方案 | WWW-Authenticate: Basic |

 params的示例

|  |
| --- |
| @RequestMapping(value = "/test/{userId}", method = RequestMethod.GET, params="myParam=myValue")  public void findUser(@PathVariable String userId) {    // implementation omitted  } |

仅处理请求中包含了名为“myParam”，值为“myValue”的请求，起到了一个过滤的作用。

3.3 consumes/produces

|  |
| --- |
| @Controller  @RequestMapping(value = "/users", method = RequestMethod.POST, consumes="application/json", produces="application/json")  @ResponseBody  public List<User> addUser(@RequestBody User userl) {      // implementation omitted      return List<User> users;  } |

3.3 consumes/produces

|  |
| --- |
| @Controller  @RequestMapping(value = "/users", method = RequestMethod.POST, consumes="application/json", produces="application/json")  @ResponseBody  public List<User> addUser(@RequestBody User userl) {      // implementation omitted      return List<User> users;  } |

方法仅处理request Content-Type为“application/json”类型的请求. produces标识==>处理request请求中Accept头中包含了"application/json"的请求，同时暗示了返回的内容类型为application/json;

4. 总结

  在本文中，首先介绍了Content-Type主要支持的格式内容，然后基于@RequestMapping标注的内容介绍了主要的使用方法，其中,headers, consumes,produces,都是使用Content-Type中使用的各种媒体格式内容，可以基于这个格式内容来进行访问的控制和过滤。

## MyBatis

### 常用标签

#### ExecutorType

SIMPLE 就是普通的执行器；

REUSE 执行器会重用预处理语句 （prepared statements）；

BATCH 执行器将重用语句并执行批量更新。

#### @MapperScan和@Mapper

在启动类中添加对 mapper 包扫描@MapperScan

@SpringBootApplication@MapperScan("com.neo.mapper")public class MybatisAnnotationApplication {

public static void main(String[] args) {

SpringApplication.run(MybatisAnnotationApplication.class, args);

}}

或者直接在 Mapper 类上面添加注解@Mapper，建议使用上面那种，不然每个 mapper 加个注解也挺麻烦的

#### mybatis collection column 传常量

|  |
| --- |
| 项目中在使用association和collection实现一对一和一对多关系时需要对关系中结果集进行筛选，如果使用懒加载模式，即联合使用select标签时，主sql和关系映射里的sql是分开的，查询参数传递成为问题。  想要在mybatis 的collection关联查询中，添加一个常量：classifyId=1作为参数，原先使用的添加方式为：  <collection property="imageList" column="{aaaId=aaa\_id,classifyId='1'}"  javaType="ArrayList"  select="com.fsti.information.dao.ImageManageMapper.queryGoodsImage">  </collection>  会报找不到行：”1 “的错误。  需要将关联的语句改为：  <resultMap id="GoodsVO" type="com.fsti.aaa.bean.vo.aaaVO" >  <collection property="imageList" column="{aaaId=aaa\_id,classifyId=classifyId}"  javaType="ArrayList"  select="com.fsti.information.dao.ImageManageMapper.queryGoodsImage">  </collection>  <collection property="goodsTags" column="{goodsId = goods\_id}"  javaType="ArrayList"  select="com.fsti.goods.dao.GoodsTagsReleMapper.queryGoodsTags">  </collection>  </resultMap>  基础查询的语句改为：  <select id="queryAaaVO" resultMap="aaaVO" parameterType="java.util.Map" >  select  <include refid="Base\_Column\_List" />,  1 as classifyId  from aaa  where  aaa\_id=#{aaaId,jdbcType=BIGINT}  </select>  也就是在查询时添加一句 1 as classifyId  然后，将其作为变量在column中引用即可：classifyId=classifyId |

### mybatis-spring——事务

一个使用 MyBatis-Spring 的其中一个主要原因是它允许 MyBatis 参与到 Spring 的事务管理中。而不是给 MyBatis 创建一个新的专用事务管理器，MyBatis-Spring 借助了 Spring 中的 DataSourceTransactionManager 来实现事务管理。

一旦配置好了 Spring 的事务管理器，你就可以在 Spring 中按你平时的方式来配置事务。并且支持 @Transactional 注解和 AOP 风格的配置。在事务处理期间，一个单独的 SqlSession 对象将会被创建和使用。当事务完成时，这个 session 会以合适的方式提交或回滚。

事务配置好了以后，MyBatis-Spring 将会透明地管理事务。这样在你的 DAO 类中就不需要额外的代码了。

无论 JDBC 连接是否设置为自动提交，调用 SqlSession 数据方法或在 Spring 事务之外调用任何在映射器中方法，事务都将会自动被提交。

## Hibernate

## Caffeine

[Caffeine](https://link.jianshu.com/?t=https://github.com/ben-manes/caffeine" \t "https://blog.csdn.net/xiaolyuh123/article/details/_blank) — 一个高性能的 Java 缓存库。

缓存和 Map 之间的一个根本区别在于缓存可以回收存储的 item。

回收策略为在指定时间删除哪些对象。此策略直接影响缓存的命中率 — 缓存库的一个重要特征。

Caffeine 因使用 Window TinyLfu 回收策略，提供了一个近乎最佳的命中率。

### 填充策略（Population）

Caffeine 为我们提供了三种填充策略：手动、同步和异步

#### 手动加载（Manual）

|  |
| --- |
| Cache<String, Object> manualCache = Caffeine.newBuilder()  .expireAfterWrite(10, TimeUnit.MINUTES)  .maximumSize(10\_000)  .build();    String key = "name1";  // 根据key查询一个缓存，如果没有返回NULL  graph = manualCache.getIfPresent(key);  // 根据Key查询一个缓存，如果没有调用createExpensiveGraph方法，并将返回值保存到缓存。  // 如果该方法返回Null则manualCache.get返回null，如果该方法抛出异常则manualCache.get抛出异常  graph = manualCache.get(key, k -> createExpensiveGraph(k));  // 将一个值放入缓存，如果以前有值就覆盖以前的值  manualCache.put(key, graph);  // 删除一个缓存  manualCache.invalidate(key);    ConcurrentMap<String, Object> map = manualCache.asMap();  cache.invalidate(key); |

Cache接口允许显式的去控制缓存的检索，更新和删除。

我们可以通过cache.getIfPresent(key) 方法来获取一个key的值，通过cache.put(key, value)方法显示的将数控放入缓存，但是这样子会覆盖缓原来key的数据。更加建议使用cache.get(key，k - > value) 的方式，get 方法将一个参数为 key 的 Function (createExpensiveGraph) 作为参数传入。如果缓存中不存在该键，则调用这个 Function 函数，并将返回值作为该缓存的值插入缓存中。get 方法是以阻塞方式执行调用，即使多个线程同时请求该值也只会调用一次Function方法。这样可以避免与其他线程的写入竞争，这也是为什么使用 get 优于 getIfPresent 的原因。

注意：如果调用该方法返回NULL（如上面的 createExpensiveGraph 方法），则cache.get返回null，如果调用该方法抛出异常，则get方法也会抛出异常。

可以使用Cache.asMap() 方法获取ConcurrentMap进而对缓存进行一些更改。

#### 同步加载（Loading）

|  |
| --- |
| LoadingCache<String, Object> loadingCache = Caffeine.newBuilder()  .maximumSize(10\_000)  .expireAfterWrite(10, TimeUnit.MINUTES)  .build(key -> createExpensiveGraph(key));    String key = "name1";  // 采用同步方式去获取一个缓存和上面的手动方式是一个原理。在build Cache的时候会提供一个createExpensiveGraph函数。  // 查询并在缺失的情况下使用同步的方式来构建一个缓存  Object graph = loadingCache.get(key);    // 获取组key的值返回一个Map  List<String> keys = new ArrayList<>();  keys.add(key);  Map<String, Object> graphs = loadingCache.getAll(keys); |

LoadingCache是使用CacheLoader来构建的缓存的值。

批量查找可以使用getAll方法。默认情况下，getAll将会对缓存中没有值的key分别调用CacheLoader.load方法来构建缓存的值。我们可以重写CacheLoader.loadAll方法来提高getAll的效率。

注意：您可以编写一个CacheLoader.loadAll来实现为特别请求的key加载值。例如，如果计算某个组中的任何键的值将为该组中的所有键提供值，则loadAll可能会同时加载该组的其余部分。

#### 异步加载（Asynchronously Loading）

|  |
| --- |
| AsyncLoadingCache<String, Object> asyncLoadingCache = Caffeine.newBuilder()  .maximumSize(10\_000)  .expireAfterWrite(10, TimeUnit.MINUTES)  // Either: Build with a synchronous computation that is wrapped as asynchronous  .buildAsync(key -> createExpensiveGraph(key));  // Or: Build with a asynchronous computation that returns a future  // .buildAsync((key, executor) -> createExpensiveGraphAsync(key, executor));    String key = "name1";    // 查询并在缺失的情况下使用异步的方式来构建缓存  CompletableFuture<Object> graph = asyncLoadingCache.get(key);  // 查询一组缓存并在缺失的情况下使用异步的方式来构建缓存  List<String> keys = new ArrayList<>();  keys.add(key);  CompletableFuture<Map<String, Object>> graphs = asyncLoadingCache.getAll(keys);  // 异步转同步  loadingCache = asyncLoadingCache.synchronous(); |

AsyncLoadingCache是继承自LoadingCache类的，异步加载使用Executor去调用方法并返回一个CompletableFuture。异步加载缓存使用了响应式编程模型。

如果要以同步方式调用时，应提供CacheLoader。要以异步表示时，应该提供一个AsyncCacheLoader，并返回一个CompletableFuture。

synchronous()这个方法返回了一个LoadingCacheView视图，LoadingCacheView也继承自LoadingCache。调用该方法后就相当于你将一个异步加载的缓存AsyncLoadingCache转换成了一个同步加载的缓存LoadingCache。

默认使用ForkJoinPool.commonPool()来执行异步线程，但是我们可以通过Caffeine.executor(Executor) 方法来替换线程池。

### 驱逐策略（eviction）

Caffeine提供三类驱逐策略：基于大小（size-based），基于时间（time-based）和基于引用（reference-based）。

#### 基于大小（size-based）

基于大小驱逐，有两种方式：一种是基于缓存大小，一种是基于权重。

|  |
| --- |
| // Evict based on the number of entries in the cache  // 根据缓存的计数进行驱逐  LoadingCache<Key, Graph> graphs = Caffeine.newBuilder()  .maximumSize(10\_000)  .build(key -> createExpensiveGraph(key));    // Evict based on the number of vertices in the cache  // 根据缓存的权重来进行驱逐（权重只是用于确定缓存大小，不会用于决定该缓存是否被驱逐）  LoadingCache<Key, Graph> graphs = Caffeine.newBuilder()  .maximumWeight(10\_000)  .weigher((Key key, Graph graph) -> graph.vertices().size())  .build(key -> createExpensiveGraph(key)); |

我们可以使用Caffeine.maximumSize(long)方法来指定缓存的最大容量。当缓存超出这个容量的时候，会使用Window TinyLfu策略来删除缓存。

我们也可以使用权重的策略来进行驱逐，可以使用Caffeine.weigher(Weigher) 函数来指定权重，使用Caffeine.maximumWeight(long) 函数来指定缓存最大权重值。

maximumWeight与maximumSize不可以同时使用。

#### 基于时间（Time-based）

|  |
| --- |
| // Evict based on a fixed expiration policy  // 基于固定的到期策略进行退出  LoadingCache<Key, Graph> graphs = Caffeine.newBuilder()  .expireAfterAccess(5, TimeUnit.MINUTES)  .build(key -> createExpensiveGraph(key));  LoadingCache<Key, Graph> graphs = Caffeine.newBuilder()  .expireAfterWrite(10, TimeUnit.MINUTES)  .build(key -> createExpensiveGraph(key));    // Evict based on a varying expiration policy  // 基于不同的到期策略进行退出  LoadingCache<Key, Graph> graphs = Caffeine.newBuilder()  .expireAfter(new Expiry<Key, Graph>() {  @Override  public long expireAfterCreate(Key key, Graph graph, long currentTime) {  // Use wall clock time, rather than nanotime, if from an external resource  long seconds = graph.creationDate().plusHours(5)  .minus(System.currentTimeMillis(), MILLIS)  .toEpochSecond();  return TimeUnit.SECONDS.toNanos(seconds);  }    @Override  public long expireAfterUpdate(Key key, Graph graph,  long currentTime, long currentDuration) {  return currentDuration;  }    @Override  public long expireAfterRead(Key key, Graph graph,  long currentTime, long currentDuration) {  return currentDuration;  }  })  .build(key -> createExpensiveGraph(key)); |

Caffeine提供了三种定时驱逐策略：

expireAfterAccess(long, TimeUnit):在最后一次访问或者写入后开始计时，在指定的时间后过期。假如一直有请求访问该key，那么这个缓存将一直不会过期。

expireAfterWrite(long, TimeUnit): 在最后一次写入缓存后开始计时，在指定的时间后过期。

expireAfter(Expiry): 自定义策略，过期时间由Expiry实现独自计算。

缓存的删除策略使用的是惰性删除和定时删除。这两个删除策略的时间复杂度都是O(1)。

测试定时驱逐不需要等到时间结束。我们可以使用Ticker接口和Caffeine.ticker(Ticker)方法在缓存生成器中指定时间源，而不必等待系统时钟。如：

|  |
| --- |
| FakeTicker ticker = new FakeTicker(); // Guava's testlib  Cache<Key, Graph> cache = Caffeine.newBuilder()  .expireAfterWrite(10, TimeUnit.MINUTES)  .executor(Runnable::run)  .ticker(ticker::read)  .maximumSize(10)  .build();    cache.put(key, graph);  ticker.advance(30, TimeUnit.MINUTES)  assertThat(cache.getIfPresent(key), is(nullValue()); |

#### 基于引用（reference-based）

见 强引用、弱引用、软引用、虚引用 - Java 7之基础

|  |
| --- |
| // Evict when neither the key nor value are strongly reachable  // 当key和value都没有引用时驱逐缓存  LoadingCache<Key, Graph> graphs = Caffeine.newBuilder()  .weakKeys()  .weakValues()  .build(key -> createExpensiveGraph(key));    // Evict when the garbage collector needs to free memory  // 当垃圾收集器需要释放内存时驱逐  LoadingCache<Key, Graph> graphs = Caffeine.newBuilder()  .softValues()  .build(key -> createExpensiveGraph(key)); |

我们可以将缓存的驱逐配置成基于垃圾回收器。为此，我们可以将key 和 value 配置为弱引用或只将值配置成软引用。

注意：AsyncLoadingCache不支持弱引用和软引用。

Caffeine.weakKeys() 使用弱引用存储key。如果没有其他地方对该key有强引用，那么该缓存就会被垃圾回收器回收。由于垃圾回收器只依赖于身份(identity)相等，因此这会导致整个缓存使用身份 (==) 相等来比较 key，而不是使用 equals()。

Caffeine.weakValues() 使用弱引用存储value。如果没有其他地方对该value有强引用，那么该缓存就会被垃圾回收器回收。由于垃圾回收器只依赖于身份(identity)相等，因此这会导致整个缓存使用身份 (==) 相等来比较 key，而不是使用 equals()。

Caffeine.softValues() 使用软引用存储value。当内存满了过后，软引用的对象以将使用最近最少使用(least-recently-used ) 的方式进行垃圾回收。由于使用软引用是需要等到内存满了才进行回收，所以我们通常建议给缓存配置一个使用内存的最大值。 softValues() 将使用身份相等(identity) (==) 而不是equals() 来比较值。

注意：Caffeine.weakValues()和Caffeine.softValues()不可以一起使用。

### 移除监听器（Removal）

概念：

驱逐（eviction）：由于满足了某种驱逐策略，后台自动进行的删除操作

无效（invalidation）：表示由调用方手动删除缓存

移除（removal）：监听驱逐或无效操作的监听器

#### 手动删除缓存：

在任何时候，您都可能明确地使缓存无效，而不用等待缓存被驱逐。

|  |
| --- |
| // individual key  cache.invalidate(key)  // bulk keys  cache.invalidateAll(keys)  // all keys  cache.invalidateAll() |

#### Removal 监听器：

|  |
| --- |
| Cache<Key, Graph> graphs = Caffeine.newBuilder()  .removalListener((Key key, Graph graph, RemovalCause cause) ->  System.out.printf("Key %s was removed (%s)%n", key, cause))  .build(); |

您可以通过Caffeine.removalListener(RemovalListener) 为缓存指定一个删除侦听器，以便在删除数据时执行某些操作。 RemovalListener可以获取到key、value和RemovalCause（删除的原因）。

删除侦听器的里面的操作是使用Executor来异步执行的。默认执行程序是ForkJoinPool.commonPool()，可以通过Caffeine.executor(Executor)覆盖。当操作必须与删除同步执行时，请改为使用CacheWrite，CacheWrite将在下面说明。

注意：由RemovalListener抛出的任何异常都会被记录（使用Logger）并不会抛出。

### 刷新（Refresh）

|  |
| --- |
| LoadingCache<Key, Graph> graphs = Caffeine.newBuilder()  .maximumSize(10\_000)  // 指定在创建缓存或者最近一次更新缓存后经过固定的时间间隔，刷新缓存  .refreshAfterWrite(1, TimeUnit.MINUTES)  .build(key -> createExpensiveGraph(key)); |

刷新和驱逐是不一样的。刷新的是通过LoadingCache.refresh(key)方法来指定，并通过调用CacheLoader.reload方法来执行，刷新key会异步地为这个key加载新的value，并返回旧的值（如果有的话）。驱逐会阻塞查询操作直到驱逐作完成才会进行其他操作。

与expireAfterWrite不同的是，refreshAfterWrite将在查询数据的时候判断该数据是不是符合查询条件，如果符合条件该缓存就会去执行刷新操作。例如，您可以在同一个缓存中同时指定refreshAfterWrite和expireAfterWrite，只有当数据具备刷新条件的时候才会去刷新数据，不会盲目去执行刷新操作。如果数据在刷新后就一直没有被再次查询，那么该数据也会过期。

刷新操作是使用Executor异步执行的。默认执行程序是ForkJoinPool.commonPool()，可以通过Caffeine.executor(Executor)覆盖。

如果刷新时引发异常，则使用log记录日志，并不会抛出。

### Writer

|  |
| --- |
| LoadingCache<Key, Graph> graphs = Caffeine.newBuilder()  .writer(new CacheWriter<Key, Graph>() {  @Override public void write(Key key, Graph graph) {  // write to storage or secondary cache  }  @Override public void delete(Key key, Graph graph, RemovalCause cause) {  // delete from storage or secondary cache  }  })  .build(key -> createExpensiveGraph(key)); |

CacheWriter允许缓存充当一个底层资源的代理，当与CacheLoader结合使用时，所有对缓存的读写操作都可以通过Writer进行传递。Writer可以把操作缓存和操作外部资源扩展成一个同步的原子性操作。并且在缓存写入完成之前，它将会阻塞后续的更新缓存操作，但是读取（get）将直接返回原有的值。如果写入程序失败，那么原有的key和value的映射将保持不变，如果出现异常将直接抛给调用者。

CacheWriter可以同步的监听到缓存的创建、变更和删除操作。加载（例如，LoadingCache.get）、重新加载（例如，LoadingCache.refresh）和计算（例如Map.computeIfPresent）的操作不被CacheWriter监听到。

注意：CacheWriter不能与weakKeys或AsyncLoadingCache结合使用。

#### 可能的用例（Possible Use-Cases）

CacheWriter是复杂工作流的扩展点，需要外部资源来观察给定Key的变化顺序。这些用法Caffeine是支持的，但不是本地内置。

#### 写模式（Write Modes）

CacheWriter可以用来实现一个直接写（write-through ）或回写（write-back ）缓存的操作。

write-through式缓存中，写操作是一个同步的过程，只有写成功了才会去更新缓存。这避免了同时去更新资源和缓存的条件竞争。

write-back式缓存中，对外部资源的操作是在缓存更新后异步执行的。这样可以提高写入的吞吐量，避免数据不一致的风险，比如如果写入失败，则在缓存中保留无效的状态。这种方法可能有助于延迟写操作，直到指定的时间，限制写速率或批写操作。

通过对write-back进行扩展，我们可以实现以下特性：

批处理和合并操作

延迟操作并到一个特定的时间执行

如果超过阈值大小，则在定期刷新之前执行批处理

如果操作尚未刷新，则从写入后缓冲器（write-behind）加载

根据外部资源的特点，处理重审，速率限制和并发

可以参考一个简单的例子，使用RxJava实现。

#### 分层（Layering）

CacheWriter可能用来集成多个缓存进而实现多级缓存。

多级缓存的加载和写入可以使用系统外部高速缓存。这允许缓存使用一个小并且快速的缓存去调用一个大的并且速度相对慢一点的缓存。典型的off-heap、file-based和remote 缓存。

受害者缓存（Victim Cache）是一个多级缓存的变体，其中被删除的数据被写入二级缓存。这个delete(K, V, RemovalCause) 方法允许检查为什么该数据被删除，并作出相应的操作。

#### 同步监听器（Synchronous Listeners）

同步监听器会接收一个key在缓存中的进行了那些操作的通知。监听器可以阻止缓存操作，也可以将事件排队以异步的方式执行。这种类型的监听器最常用于复制或构建分布式缓存。

### 统计（Statistics）

|  |
| --- |
| Cache<Key, Graph> graphs = Caffeine.newBuilder()  .maximumSize(10\_000)  .recordStats()  .build(); |

使用Caffeine.recordStats()，您可以打开统计信息收集。Cache.stats() 方法返回提供统计信息的CacheStats，如：

hitRate()：返回命中与请求的比率

hitCount(): 返回命中缓存的总数

evictionCount()：缓存逐出的数量

averageLoadPenalty()：加载新值所花费的平均时间

### Cleanup

缓存的删除策略使用的是惰性删除和定时删除，但是我也可以自己调用cache.cleanUp()方法手动触发一次回收操作。cache.cleanUp()是一个同步方法。

### 策略（Policy）

在创建缓存的时候，缓存的策略就指定好了。但是我们可以在运行时可以获得和修改该策略。这些策略可以通过一些选项来获得，以此来确定缓存是否支持该功能。

#### Size-based

|  |
| --- |
| cache.policy().eviction().ifPresent(eviction -> {  eviction.setMaximum(2 \* eviction.getMaximum());  }); |

如果缓存配置的时基于权重来驱逐，那么我们可以使用weightedSize() 来获取当前权重。这与获取缓存中的记录数的Cache.estimatedSize() 方法有所不同。

缓存的最大值(maximum)或最大权重(weight)可以通过getMaximum()方法来读取，并使用setMaximum(long)进行调整。当缓存量达到新的阀值的时候缓存才会去驱逐缓存。

如果有需用我们可以通过hottest(int) 和 coldest(int)方法来获取最有可能命中的数据和最有可能驱逐的数据快照。

#### Time-based

|  |
| --- |
| cache.policy().expireAfterAccess().ifPresent(expiration -> ...);  cache.policy().expireAfterWrite().ifPresent(expiration -> ...);  cache.policy().expireVariably().ifPresent(expiration -> ...);  cache.policy().refreshAfterWrite().ifPresent(expiration -> ...); |

ageOf(key，TimeUnit) 提供了从expireAfterAccess，expireAfterWrite或refreshAfterWrite策略的角度来看条目已经空闲的时间。最大持续时间可以从getExpiresAfter(TimeUnit)读取，并使用setExpiresAfter(long，TimeUnit)进行调整。

如果有需用我们可以通过hottest(int) 和 coldest(int)方法来获取最有可能命中的数据和最有可能驱逐的数据快照。

### 测试（Testing）

|  |
| --- |
| FakeTicker ticker = new FakeTicker(); // Guava's testlib  Cache<Key, Graph> cache = Caffeine.newBuilder()  .expireAfterWrite(10, TimeUnit.MINUTES)  .executor(Runnable::run)  .ticker(ticker::read)  .maximumSize(10)  .build();    cache.put(key, graph);  ticker.advance(30, TimeUnit.MINUTES)  assertThat(cache.getIfPresent(key), is(nullValue()); |

测试的时候我们可以使用Caffeine..ticker(ticker)来指定一个时间源，并不需要等到key过期。

FakeTicker这个是guawa test包里面的Ticker，主要用于测试。依赖：

|  |
| --- |
| <dependency>  <groupId>com.google.guava</groupId>  <artifactId>guava-testlib</artifactId>  <version>23.5-jre</version>  </dependency> |

### 常见问题（Faq）

#### 固定数据（Pinning Entries）

固定数据是不能通过驱逐策略去将数据删除的。当数据是一个有状态的资源时（如锁），那么这条数据是非常有用的，你有在客端使用完这个条数据的时候才能删除该数据。在这种情况下如果驱逐策略将这个条数据删掉的话，将导致资源泄露。

通过使用权重将该数据的权重设置成0，并且这个条数据不计入maximum size里面。 当缓存达到maximum size 了以后，驱逐策略也会跳过该条数据，不会进行删除操作。我们还必须自定义一个标准来判断这个数据是否属于固定数据。

通过使用Long.MAX\_VALUE（大约300年）的值作为key的有效时间，这样可以将一条数据从过期中排除。自定义到期必须定义，这可以评估条目是否固定。

将数据写入缓存时我们要指定该数据的权重和到期时间。这可以通过使用cache.asMap()获取缓存列表后，再来实现引脚和解除绑定。

#### 递归调用（Recursive Computations）

在原子操作内执行的加载，计算或回调可能不会写入缓存。 ConcurrentHashMap不允许这些递归写操作，因为这将有可能导致活锁（Java 8）或IllegalStateException（Java 9）。

解决方法是异步执行这些操作，例如使用AsyncLoadingCache。在异步这种情况下映射已经建立，value是一个CompletableFuture，并且这些操作是在缓存的原子范围之外执行的。但是，如果发生无序的依赖链，这也有可能导致死锁。

## 其他

### Apache Shiro

### 安全框架Shiro和Spring Security比较

Spring Security架构：

安全拦截器：

认证管理器、访问抉择管理器、运行身份管理器。

http Basic 认证 BASE64；

http Digest 认证 MD5；

http X509 证书公钥，序列号CA签发；

LDAP

[LDAP](http://www.openldap.org/doc/admin24/intro.html" \t "_blank)是轻量目录访问协议(Lightweight Directory Access Protocol)的缩写，LDAP是从X.500目录访问协议的基础上发展过来的，目前的版本是v3.0。与LDAP一样提供类似的目录服务软件还有[ApacheDS](http://directory.apache.org/apacheds/" \t "_blank)、[Active Directory](https://technet.microsoft.com/en-us/library/dd448614.aspx" \t "_blank)、[Red Hat Directory Service](http://www.redhat.com/en/technologies/cloud-computing/directory-server" \t "_blank) 。

Form；

Spring Security；

### S2SH

### Spring、SpringMVC、Mybatis框架，了解底层机制

### 日志

在spring Cloud Netflix栈中，各个微服务都是以HTTP接口的形式暴露自身服务的，因此在调用远程服务时就必须使用HTTP客户端。我们可以使用JDK原生的URLConnection、Apache的Http Client、Netty的异步HTTP Client, Spring的RestTemplate。但是，用起来最方便、最优雅的还是要属Feign了。

@Slf4j以后，默认的Slf4j对象就是log，所以使用时候可以直接log.info()、log.error()……，你需要引入:

|  |
| --- |
| <dependency>  <groupId>org.projectlombok</groupId>  <artifactId>lombok</artifactId>  <version>1.16.12</version>  <scope>provided</scope>  </dependency> |

如果你使用@Slf4j注解后发现没有log对象，说明你的idea工具中没有安装lombok插件；

### JAXB框架转报文 XML

JAXB（Java Architecture for XML Binding) 是一个业界的标准，是一项可以根据XML Schema产生Java类的技术。该过程中，JAXB也提供了将XML实例文档反向生成Java对象树的方法，并能将Java对象树的内容重新写到XML实例文档。从另一方面来讲，JAXB提供了快速而简便的方法将XML模式绑定到Java表示，从而使得Java开发者在Java应用程序中能方便地结合XML数据和处理函数。

@XmlRootElement——

类级别的注解，将类映射为xml全局元素，也就是根元素。就像spring配置文件中的beans。上面的例子中我将该注解用在了person类上，生成了<person>根元素。常与@XmlType，@XmlAccessorType，@XmlAccessorOrder连用。

@XmlAccessorType(XmlAccessType.FIELD)——

# 计算机网络

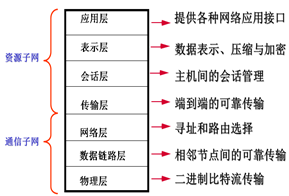
Internet基本协议（如TCP/IP, HTTP等）内容及相关应用

https

## （一）基本概念

### OSI 参考模型

OSI（open system interconnected，OS）参考模型是一个开放式体系结构，它规定将网络分为七层，从下往上依次是物理层，数据链路层，网络层，传输层，会话层，表示层和应用层。



        1.物理层（physical layer）的主要功能:是完成相邻节点之间原始比特流的传输物理层的设计主要涉及物理接口的机械电气功能和过程特性，以及物理层接口连接的传输介质等问题。

       2.数据链路层（data link layer）:负责将上层数据封装成固定格式的帧

       3.网络层(network layer）的主要功能:是实现数据从源端到目的端的传输

       4.传输层（transport layer）的主要功能:是实现网络中不同主机上用户进程之间的数据通信。

传输层要决定对会话层用户（最终的网络用户）提供什么样的服务。因此我们把1-3层的协议称为点对点的协议，而把4-7层的协议叫做端对端协议

       5.会话层（session layer）:允许不同机器上的用户之间建立会话关系。管理会话控制令牌管理（token management）数据中插入同步点

       6.表示层（presentation  layer ）:表示层以下各层只关心从源端机到目标机可靠地传输比特，而表示层关心的是所传输的语法和语义。用一种大家一致选定的标准方法对数据进行编码。同时，也能提供压缩解压、加密解密。

       7.应用层（application layer）:应用层能与应用程序界面沟通，以达到展示给用户的目的。在此常见的协议有:HTTP，HTTPS，FTP，TELNET，SSH，SMTP，POP3等。

### TCP/IP参考模型

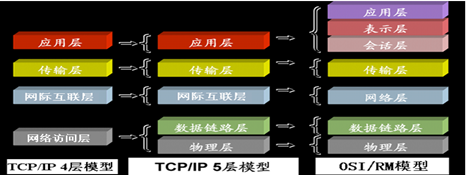
TCP/IP(transmission control /Internet protocol   传输控制协议/网络互联协议）早期的TCP/IP模型是一个四层结构网络接口层网络层传输层应用层后期5层结构将网络接口层分为物理层和数据链路层

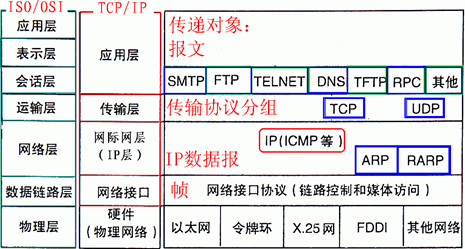
            1.物理层和数据链路层

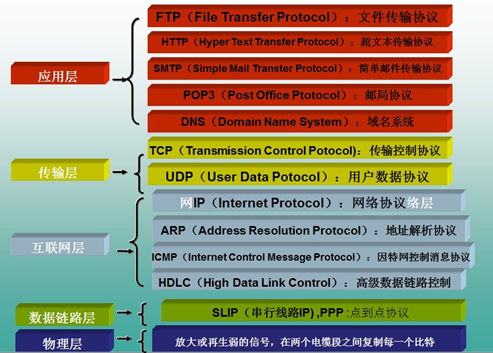
            2.网络层TCP/IP协议定义了网络互联协议（Internet  protocol， IP ）而IP由4个支撑协议组 ARP （地址解析协议）  RARP(逆地址解析协议）  ICMP（网际控制报文协议）   IGMP（网际组管协议）

            3.传输层TCP(传输控制协议）和UDP（用户数据报协议）   TCP 协议传输更加稳定可靠，UDP协议传输效率更高

            4.应用层







### 3.数据封装和解封过程

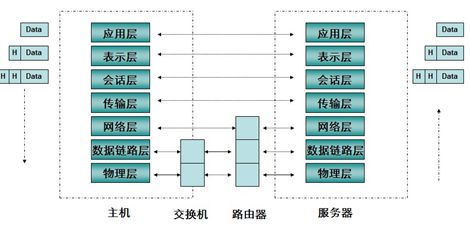
#### 数据封装过程

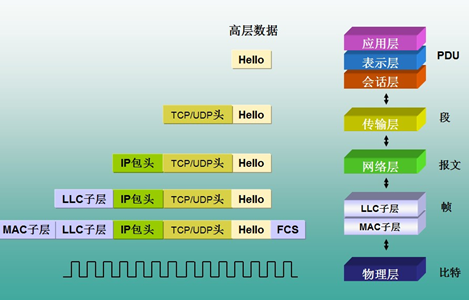
           1.1.应用层在应用层数据被翻译为---二进制编码数据传输层上层数据被分割成小的数据段，并未每个数据段后的数据封装TCP报文头部在TCP头部有一个关键的字段信息----端口号，它用于标识上层的协议或应用程序，确保上层应用数据的正常通信

          1.2.网络层上层数据被封装上新的报文头部------IP头部IP头部有一个关键字段信息---IP地址，由一组32位2进制组成，用于标识网络的逻辑地址

          1.3.数据链路层上层数据被封装一个MAC头部，其内部有一个关键字段信息-------MAC地址

          1.4.物理层封装的信息都是2进制组成的，在物理层，将这些2进制数字组成的比特流转换成电信号在网络中传输

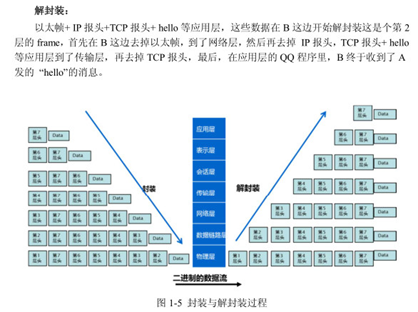




 PDU （Protocol data unit）协议数据单元同层之间的传递数据单位

#### 数据解封过程

物理层电信号转换成2进制，并将数据送至数据链路层查看mac地址，若吻合就将mac头部拆掉，并将数据传入上一层，不吻合数据将被丢弃网络层核实IP地址应用层将2进制数据解码还原



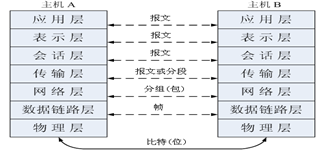


                     常见硬件设备与五层模型的对应关

#### 网络中对等通信原理

            3.1.对等层（peer layers）通信原理：1.只有双方是对等层次的会话才可能使用相同的类型的协议，彼此才能听的懂，才能有共同语言。

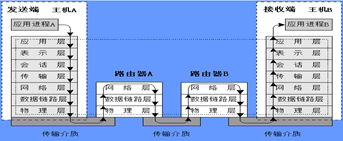
            3.2.在网络体系中，每一层都是独立完成自己的工作，其他层是不干预的，不了解的。



### 网络中数据通信原理

            在网络连接和数据传输流程方面，发送端是自上而下（从高到低），接收端是从下而上（从低到高）进行的，也就是数据通信原理相同。

            通信会话方面，双方都必须是逻辑上的对等层次，也就是对等通信原理相同。



                         网络通信原理

#### 扩展内容

        OSI/RM对各个层次的划分遵循下列原则

       5.1.同一层中的各个网络节点都有相同的层次结构，具有同样的功能

       5.2.同一节点内相邻层之间通过接口（可以是逻辑接口）进行通信

       5.3.七层结构中的每一层使用下一层提供的服务，并向其上层提供服务

       5.4.不同节点的同等层按照协议实现对等通信

      5.5.网络设备（不包括计算机主机）间自身通信仅需要低三层，用来构建数据通信的网络平台。网络平台构建好后，用户应用数据就可以利用这个平台进行各种网络应用通信，但所有的网络应用通信都需要经过网络体系结构中的所有层次，其中最上面4层用来为用户的网络应用通信提供各种服务支持，构建数据通信平台。

## （二）长连接和短连接

HTTP分为长连接和短连接，本质上是说的TCP连接。TCP连接是一个双向的通道，可以保持一段时间不关闭的，因此TCP连接才有真正的长连接和短连接这一说。

HTTP协议是应用层的协议，而TCP才是真正的传输层协议

服务器和客户端都设置Connection为keep-alive就是长连接。

HTTP1.1默认是长连接，现在用的基本上都是HTTP1.1协议，基本上Connection都是keep-alive。而且HTTP协议文档上也提到了，HTTP1.1默认是长连接，也就是默认Connection的值就是keep-alive）

普通的Web应用使用长连接原因：多次HTTP请求（这些请求包括请求网页内容，CSS文件，JS文件，图片等等），其实使用的都是一个TCP连接，可以节省很多消耗的。

长连接并不是永久连接的。如果一段时间内（具体的时间长短，是可以在header当中进行设置的，也就是所谓的超时时间），这个连接没有HTTP请求发出的话，那么这个长连接就会被断掉。

### 长短轮询和长短连接的区别

第一个区别是决定的方式，一个TCP连接是否为长连接，是通过设置HTTP的Connection Header来决定的，而且是需要两边都设置才有效。而一种轮询方式是否为长轮询，是根据服务端的处理方式来决定的，与客户端没有关系。

第二个区别就是实现的方式，连接的长短是通过协议来规定和实现的。而轮询的长短，是服务器通过编程的方式手动挂起请求来实现的。

## RPC好和RESTful

RPC主要是基于TCP/IP协议的，而HTTP服务主要是基于HTTP协议的，我们都知道HTTP协议是在传输层协议TCP之上的，所以效率来看的话，RPC当然是要更胜一筹啦！下面来具体说一说RPC服务和HTTP服务。

REST，即Representational State Transfer的缩写。翻译过来是表现层状态转换。  
如果一个架构符合REST原则，就称它为RESTful架构。

接口调用通常包含两个部分，序列化和通信协议。常见的序列化协议包括json、xml、hession、protobuf、thrift、text、bytes等；通信比较流行的是http、soap、websockect，RPC通常基于TCP实现，常用框架例如netty。

RESTful通常采用http+JSON实现。

JSON-RPC是指通信协议采用二进制方式，而不是http，序列化采用JSON的形式。

我认为，微服务的盛行，开放平台的盛行，的确让RESTful过于“流行”了。你可以看看，无论是Google、Amazon、netflix（据说很可能转向grpc），还是阿里，实际上内部都是采用性能更高的RPC方式。而对外开放的才是RESTful。

如果你的应用非常简单，无论用哪种都无所谓了，基本都能满足要求。

RPC的思想是把本地函数映射到API，也就是说一个API对应的是一个function，我本地有一个getAllUsers，远程也能通过某种约定的协议来调用这个getAllUsers。至于这个协议是Socket、是HTTP还是别的什么并不重要；

RPC中的主体都是动作，是个动词，表示我要做什么。

而REST则不然，它的URL主体是资源，是个名词。而且也仅支持HTTP协议，规定了使用HTTP Method表达本次要做的动作，类型一般也不超过那四五种。这些动作表达了对资源仅有的几种转化方式。

# 构建工具

## Maven

### mvn compile与mvn install、mvn deploy、mvn package的区别

mvn compile，编译类文件

mvn install，包含mvn compile，mvn package，然后上传到本地仓库

mvn deploy,包含mvn install,然后，上传到私服

# 代码管理

## Git

### Git设置用户名和邮箱

git config --global user.name "yangxueping1993@163.com" 设置用户名

git config --global user.email "yangxueping1993@163.com" 设置邮箱

### 生成SSH指令与github建立凭证关系

ssh-keygen -t rsa -C [yangxueping1993@163.com](mailto:yangxueping1993@163.com) cat ~/.ssh/id\_rsa.pub

ssh git@github.com

### 基本操作

git init 建立git仓库

git clone https://github.com/yangxueping1993/timerTask.git 从远程仓库拉代码

git branch -a 查看所有分支

git branch -r 查看远程分支

git branch 查看本地分支

git branch test 新建分支

git checkout test 切换本地分支

git push origin test 将新建本地分支推送远端

git checkout -b iss53 = git branch iss53 新建分支

git checkout iss53 切换分支

### 添加注释

git commit -m 'your message' 添加注释

接在命令行添加message信息，当然也可以打开编辑模式添加。

步骤如下：

（1）在插入编辑模式下编辑文件。

（2）按下 “ESC” 键，退出编辑模式，切换到命令模式。

（3）在命令模式下键入"ZZ"或者":wq"保存修改并且退出 。

（4）如果只想保存文件，则键入":w"，回车后底行会提示写入操作结果，并保持停留在命令模式。

git默认的文本编辑器是nano，执行下面的命令将git的文本编辑器改为我们熟悉的vim

git config --global core.editor vim

# 中间件

## Nginx

# 其他

## OA

办公自动化（Office Automation，简称OA）；

## CRM

Customer Relationship Management客户关系管理；

## HR

Human Resource 人力资源

## ERP

企业资源计划即 ERP (Enterprise Resource Planning)

## CRM

Customer Relationship Management 客户关系管理

## IM

即时通讯（Instant Messaging）

## WebService

# 保险业务

## 重疾险VS医疗险



重疾险大多数是确诊即赔，医疗险是住院后根据发票报销，这是赔付标准上的区别，也是很好理解的。但是在赔付金额上的差别就显得比较微妙了。重疾险很清楚，要赔就是赔合同约定的金额。

而医疗险则不是。这里尤其值得注意的一个条款是“合理且必要”的医疗费用。

重疾险的保险金理赔，保险公司只要把钱给了你，剩下的事就不管了。这笔钱不管你是拿去治病了，还是花在其他用途了，都可自由支配。

而医疗险的保险金理赔，必须是治疗费用报销，自费花了多少，保险公司按相关比例进行报销，最高报销金额不会超过治疗总开销。



## 意外险VS寿险

# Java编程思想

**第1章 对象导论**

1.1 抽象过程

1.2 每个对象都有一个接口

1.3 每个对象都提供服务

1.4 被隐藏的具体实现

1.5 复用具体实现

1.6 继承

1.6.1 “是一个”（is-a）与“像是一个”（is-like-a）关系

1.7 伴随多态的可互换对象

1.8 单根继承结构

1.9 容器

1.9.1 参数化类型（范型）

1.10 对象的创建和生命期

1.11 异常处理：处理错误

1.12 并发编程

1.13 Java与Internet

1.13.1 Web是什么

1.13.2客户端编程

1.13.3 服务器端编程

1.22 总结

**第2章 一切都是对象**

2.1 用引用操纵对象

2.2 必须由你创建所有对象

2.2.1 存储到什么地方

2.2.2 特例：基本类型

2.2.3 Java中的数组

2.3 永远不需要销毁对象

2.3.1 作用域

2.3.2 对象的作用域

2.4 创建新的数据类型：类

2.4.1 域和方法

2.4.2 基本成员默认值

2.5 方法、参数和返回值

2.5.1 参数列表

2.6 构建一个Java程序

2.6.1 名字可见性

2.6.2 运用其他构件

2.6.3 static 关键字

2.7 你的第一个Java程序

编译和运行

2.8 注释和嵌入式文档

2.8.1 注释文档

2.8.2 语法

2.8.3 嵌入式HTML

2.8.4 一些标签示例

2.8.5 文档示例

2.9 编码风格

2.10 总结

2.11 练习

**第3章 操作符**

3.1 更简单的打印语句

3.2 使用Java操作符

3.3 优先级

3.4 赋值

3.4.1 方法调用中的别名问题

3.5 算术操作符

3.5.1 一元加、减操作符

3.6 自动递增和递减

3.7 关系操作符

3.7.1 测试对象的等价性

3.8 逻辑操作符

3.8.1 短路

3.9 直接常量

3.9.1 指数记数法

3.10 按位操作符

3.11 移位操作符

3.12 三元操作符 if-else

3.13 字符串操作符 + 和 +=

3.14 使用操作符时常犯的错误

3.15 类型转换操作符

3.15.1 截尾和舍入

3.15.2提升

3.16 Java没有“sizeof”

3.17 操作符小结

3.18 总结

**第4章 控制执行流程**

4.1 true和false

4.2 if-else

4.3 迭代

4.3.1 do-while

4.3.2 for

4.3.3 逗号操作符

4.4 Foreach语法

4.5 return

4.6 break和 continue

4.7 臭名昭著的“goto”

4.8 switch

4.9 总结

**第5章 初始化与清理**

5.1 用构造器确保初始化

5.2 方法重载

5.2.1 区分重载方法

5.2.2 涉及基本类型的重载

5.2.3 以返回值区分重载方法

5.3 缺省构造器

5.4 this关键字

5.4.1 在构造器中调用构造器

5.4.2 static的含义

5.5 清理：终结处理和垃圾回收

5.5.1 finalize()的用途何在

5.5.2 你必须实施清理

5.5.3 终结条件

5.5.4 垃圾回收器如何工作

5.6 成员初始化

5.6.1 指定初始化

5.7 构造器初始化

5.7.1 初始化顺序

5.7.2. 静态数据的初始化

5.7.3. 显式的静态初始化

5.7.4. 非静态实例初始化

5.8 数组初始化

5.8.1 可变参数列表

5.9 枚举类型

5.10 总结

**第6章 访问权限控制**

第7章 复用类

第8章 多态

第9章 接口

第10章 内部类

第11章 持有对象

第12章 通过异常处理错误

第13章 字符串

第14章 类型信息

第15章 泛型

第16章 数组

第17章 容器深入研究

第18章 Java I/O系统

第19章 枚举类型

第20章 注解

第21章 并发

第22章 图形化用户界面

# 《JAVA核心技术(卷1):基础知识(原书第8版)》

# 《JAVA核心技术卷2:高级特征》

1. 流与文件

1.1 流

1.1.1 读写字节

1.1.2 完整的流家族

1.1.3 组合流过滤器

1.2 文本输入与输出

1.2.1 如何写出文本输出

1.2.2 如何读入文本输入

1.2.3 以文本格式存储对象

1.2.4 字符集

1.3 读写二进制数据

1.4 ZIP文档

1.5 对象流与序列化

1.5.1 理解对象序列化的文件格式

1.5.2 修改默认的序列化机制

1.5.3 序列化单例和类型安全的枚举

1.5.4 版本管理

1.5.5 为克隆使用序列化

1.6 文件管理

1.7 新I/O

1.7.1 内存映射文件

1.7.2 缓冲区数据结构

1.7.3 文件加锁机制

1.8 正则表达式 ……

第2章 XML

第3章 网络

第4章 数据库编程

第5章 国际化

第6章 高级Swing

第7章 高级AWT

第8章 JavaBean构件

第9章 安全

第10章 分布式对象

第11章 脚本、编译与注解处理

第12章 本地方法