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

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

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

Canva；

# 后端

## （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。

### ****1.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。

### ****2.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。

### ****3.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.

### ****4.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。

### ****5.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

## 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。

### 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。这类内容我将在下篇博客进行阐述。

## 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

#### 1.1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。

#### 1.2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。

### 2.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。

### 3.函数式接口

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。

### 4.默认方法

默认方法就是接口可以有实现方法，而且不需要实现类去实现其方法。

在方法名前面加个default关键字即可实现默认方法。

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。

#### 4.1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!");

    }

}

#### 4.2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();

    }

}

#### 4.3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!!!");

    }

}

### 5.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

#### 5.1什么是 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。

#### 5.2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。

public static void main(String[] args) {

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

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

}

#### 5.3 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);

#### 5.4 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：

|  |
| --- |
| 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());

#### 5.5 filter

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();

#### 5.6 limit

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);

#### 5.7 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);

#### 5.8 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。

#### 5.9 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);

#### 5.10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());

### 6.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。

#### 6.1类声明

以下是一个 java.util.Optional<T> 类的声明：

public final class Optional<T> extends Object

#### 6.2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。

### 7.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。

#### 7.1 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!

#### 7.2 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()

>>

#### 7.3 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>

#### 7.4 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

#### 7.5 JavaScript 中调用 Java

以下实例演示了如何在 JavaScript 中引用 Java 类：

varBigDecimal=Java.type('java.math.BigDecimal');

function calculate(amount, percentage){

   var result =newBigDecimal(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。

#### 8.1 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

#### 8.2 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。

#### 9.1 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。

#### 9.2 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。

#### 9.3 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

## （二）设计模式

### 1. 单例模式

1.1内部类

**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 }

# 数据库

## （一）MySql

### 1．命令行操作

#### 1.1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)

#### 1.2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 \*\*\*\*\*\*

#### 1.3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;

#### 1.4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。

#### 1.5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

### 2. 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）

## （二）Oracle

### 1. count（\*）、count（1）和Count([列])区别？

## （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）

### **1.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]

### **2.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

### **3.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

### **4.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

### **5.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

### 1.JSON

#### 1.1 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。

#### 1.2 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'} |

#### 1.3 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

## （一）Spring

## （二）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)

## （三）MyBatis

## （四）Hibernate

## （五）Spring Boot

### 1. 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

### 2. Spring Security

### 3. 定时任务Quartz和Spring Task

## （六）其他

### 1．安全框架Shiro和Spring Security比较

### 2. S2SH

### 3. Spring、SpringMVC、Mybatis框架，了解底层机制

# 七、计算机网络

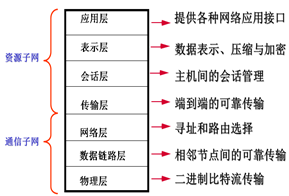
Internet基本协议（如TCP/IP, HTTP等）内容及相关应用

https

## （一）基本概念

### 1.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。

### 2.TCP/IP参考模型

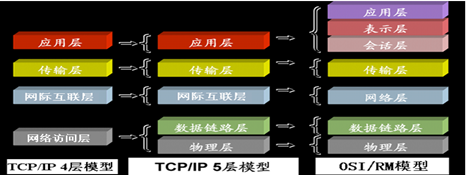
TCP/IP(transmission control /Internet protocol   传输控制协议/网络互联协议）早期的TCP/IP模型是一个四层结构网络接口层网络层传输层应用层后期5层结构将网络接口层分为物理层和数据链路层

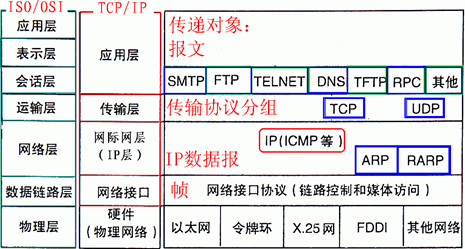
            1.物理层和数据链路层

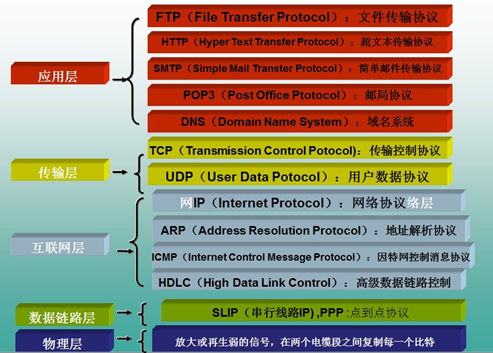
            2.网络层TCP/IP协议定义了网络互联协议（Internet  protocol， IP ）而IP由4个支撑协议组 ARP （地址解析协议）  RARP(逆地址解析协议）  ICMP（网际控制报文协议）   IGMP（网际组管协议）

            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

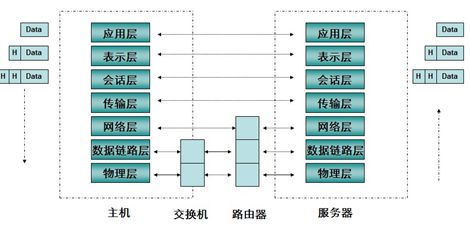
#### 1.数据封装过程

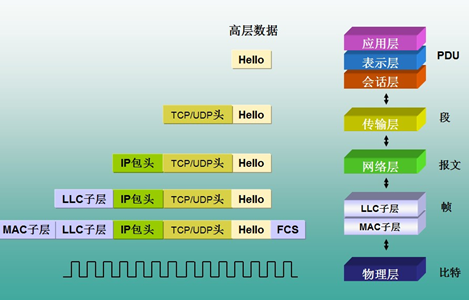
           1.1.应用层在应用层数据被翻译为---二进制编码数据传输层上层数据被分割成小的数据段，并未每个数据段后的数据封装TCP报文头部在TCP头部有一个关键的字段信息----端口号，它用于标识上层的协议或应用程序，确保上层应用数据的正常通信

          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

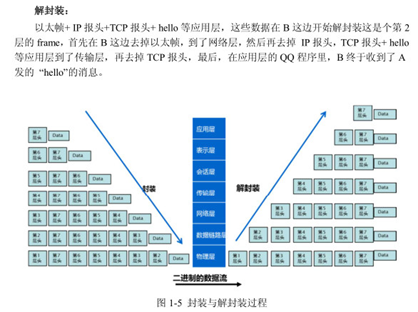




 PDU （Protocol data unit）协议数据单元同层之间的传递数据单位

#### 2.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



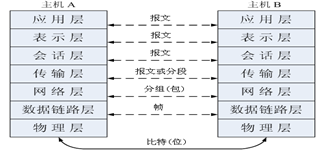


                     常见硬件设备与五层模型的对应关

#### 3.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。

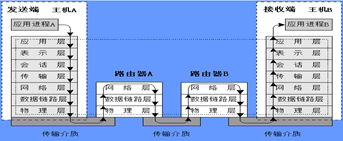
            3.2.在网络体系中，每一层都是独立完成自己的工作，其他层是不干预的，不了解的。



### 4.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

#### 5.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。

# 构建工具

## （一）Maven

# 代码管理

## （一）git

# 中间件

## （一）Nginx

# 其他

## （一）OA

办公自动化（Office Automation，简称OA）；

## （二）CRM

Customer Relationship Management客户关系管理；

## （三）HR

Human Resource 人力资源

## （四）ERP

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

## （五）CRM

Customer Relationship Management 客户关系管理

## （六）IM

即时通讯（Instant Messaging）

## （五）WebService