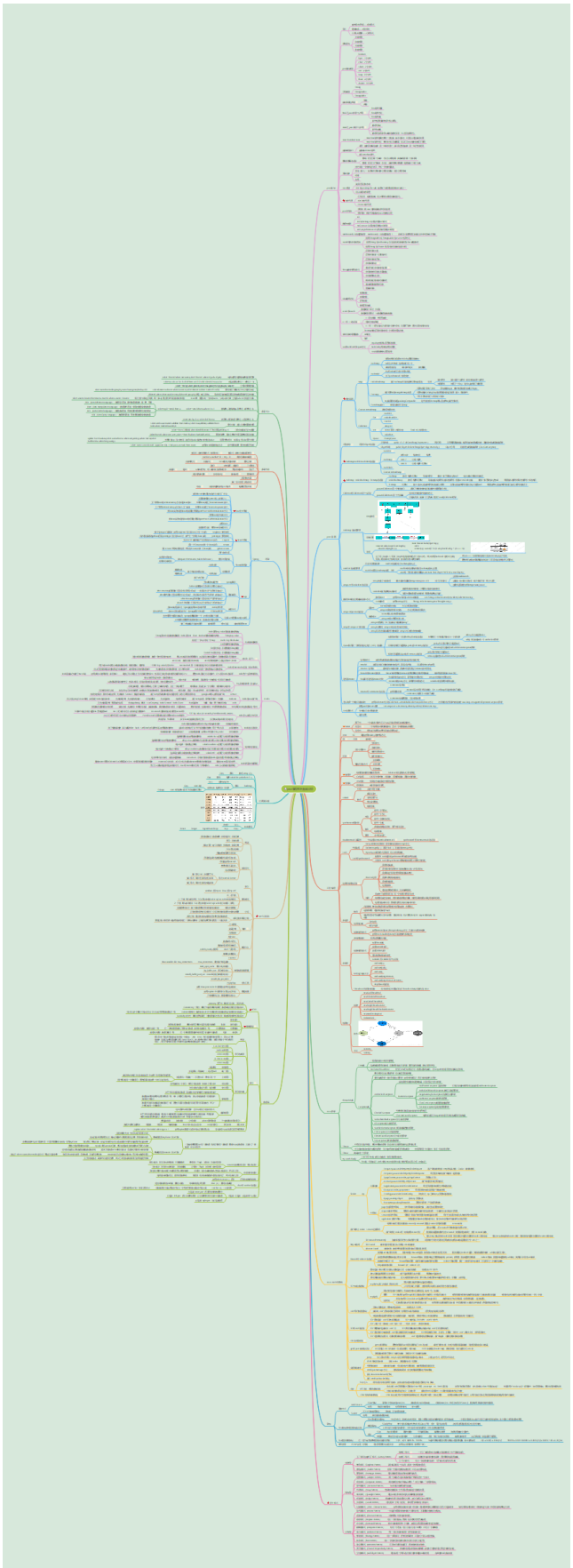
java编程知识全面回顾



7

1. java基础

1.1. 范式

1.1.1. 自然观/世界观——观念范式

1.1.2. 逻辑体系——规则范式

1.1.2.1. 编程范式

1.1.2.1.1. 基于图灵机的命令编程范式

1.1.2.1.2. 基于λ运算的函数编程范式

1.1.2.1.3. 基于一般递归函数的逻辑编程范式

1.1.2.1.4. 人的思路——心智模型

1.1.2.1.4.1. 面向对象编程范式

以人们熟悉的、习惯的现实世界和思维方式为隐喻（metaphor）

以概念/类型或其实例化的对象为思考单元，进行程序组织的编程范式

柏拉图(Plato)原则：类的世界独立存在，对象世界由类创建而来

里氏替换原则（LSP）：子类型(必须)能够替代其父类型

Parnas原则：接口与实现的分离，用户仅需要了解接口

1.1.3. 心理认知因素——心理范式

1.2. 领域驱动

1.2.1. 失血模型

1.2.1.1. 模型仅仅包含数据的定义和getter/setter方法

1.2.1.1.1. 业务逻辑和应用逻辑都放到服务层中。这种类在Java中叫POJO，在.NET中叫POCO

1.2.2. 缺血模型

1.2.2.1. 贫血模型中包含了一些业务逻辑，但不包含依赖持久层的业务逻辑

1.2.3. 充血模型

1.2.3.1. 充血模型中包含了所有的业务逻辑，包括依赖于持久层的业务逻辑

1.2.4. 胀血模型

1.2.4.1. 胀血模型把和业务逻辑不想关的其他应用逻辑（授权、事务等）都放到领域模型中

1.3. java基本类型

1.3.1. boolean

1.3.1.1. 1个bit

1.3.1.1.1. 在内存中只需要1位（bit）即可存储，位是计算机最小的存储单位

1.3.1.2. 1个字节

1.3.1.2.1. 计算机处理数据的最小单位是1个字节

1.3.1.2.1.1. true存储的二进制为：0000 0001

false存储的二进制为：0000 0000

1.3.1.3. 4个字节

1.3.1.3.1. boolean值在编译之后都使用Java虚拟机中的int数据类型来代替

1.3.1.3.1.1. boolean数组会被Java虚拟机编码成byte数组，每个元素boolean元素占8位1个字节

1.3.2. byte： 1个字节

1.3.2.1. 8位带符号的二进制数

1.3.2.1.1. 取值范围是-2^7——2^7-1

1.3.2.1.1.1. [-128, 127]

1.3.3. char： 2个字节

1.3.4. short： 2个字节

1.3.5. int： 4个字节

1.3.5.1. int的取值范围为-2^31——2^31-1

1.3.5.1.1. 负数的补码为：正值二进制取反+1

1.3.6. long： 8个字节

1.3.7. float： 4个字节

1.3.8. double： 8个字节

1.4. 引用类型

1.4.1. String

1.4.1.1. 不可变字符序列、线程安全

final修饰，String类的方法都是返回new String。即对String对象的任何改变都不影响到原对象，对字符串的修改操作都会生成新的对象

1.4.1.2. String s = "aaa";

1.4.1.2.1. JVM创建了一个变量引用S,在堆中创建了一个对象aaa,将aaa放进常量池中，s指向aaa

1.4.2. StringBuilder

1.4.2.1. 可变字符序列、线程不安全、效率高

不保证线程安全，在方法体内需要进行字符串的修改操作，可以new StringBuilder对象，调用StringBuilder对象的append、replace、delete等方法修改字符串

1.4.3. StringBuffer

1.4.3.1. 可变字符序列、加了synchronized线程安全、效率低

1.5. 自动装箱与拆箱

Java使用自动装箱和拆箱机制，节省了常用数值的内存开销和创建对象的开销，提高了效率，由编译器来完成，编译器会在编译期根据语法决定是否进行装箱和拆箱动作。

1.5.1. 装箱

1.5.1.1. 将基本类型用它们对应的引用类型包装起来

1.5.2. 拆箱

1.5.2.1. 将包装类型转换为基本数据类型

1.6. final 在 java 中有什么作用

final可以修饰：属性，方法，类，局部变量（方法中的变量）
final修饰的属性的初始化可以在编译期，也可以在运行期，初始化后不能被改变。
final修饰的属性跟具体对象有关，在运行期初始化的final属性，不同对象可以有不同的值。
final修饰的属性表明是一个常数（创建后不能被修改）。
final修饰的方法表示该方法在子类中不能被重写，final修饰的类表示该类不能被继承。
对于基本类型数据，final会将值变为一个常数（创建后不能被修改）；但是对于对象句柄（亦可称作引用或者指针），final会将句柄变为一个常数（进行声明时，必须将句柄初始化到一个具体的对象。而且不能再将句柄指向另一个对象。但是，对象的本身是可以修改的。这一限制也适用于数组，数组也属于对象，数组本身也是可以修改的。方法参数中的final句柄，意味着在该方法内部，我们不能改变参数句柄指向的实际东西，也就是说在方法内部不能给形参句柄再另外赋值）。

1.6.1. final修饰变量

1.6.1.1. 当final修饰的是一个基本数据类型数据时，这个数据的值在初始化后将不能被改变;

1.6.1.2. 当final修饰的是一个引用类型数据时，也就是修饰一个对象时，引用在初始化后将永远指向一个内存地址，不可修改. 但是该内存地址中保存的对象信息，是可以进行修改的.

1.6.1.3. 被final修饰的常量在编译阶段会被放入常量池中

1.6.2. final修饰方法

1.6.2.1. 首要作用是锁定方法，不让任何继承类对其进行修改

1.6.3. final修饰类

1.6.3.1. 表明这个类不能被继承

1.6.3.1.1. 所有成员方法都将被隐式修饰为final方法

1.7. static在 java 中有什么作用

static可以修饰：属性，方法，代码段，内部类（静态内部类或嵌套内部类）
static修饰的属性的初始化在编译期（类加载的时候），初始化后能改变。
static修饰的属性所有对象都只有一个值。
static修饰的属性强调它们只有一个。
static修饰的属性、方法、代码段跟该类的具体对象无关，不创建对象也能调用
static修饰的属性、方法等
static和“this、super”势不两立，static跟具体对象无关，而this、super正好跟具体对象有关。
static不可以修饰局部变量。

1.7.1. 修饰成员变量和成员方法(常用)

1.7.1.1. 方便在没有创建对象的情况下来进行调用

1.7.1.2. static变量/静态变量

1.7.1.2.1. 静态变量被所有的对象所共享，在内存中只有一个副本，它当且仅当在类初次加载时会被初始化。而非静态变量是对象所拥有的，在创建对象的时候被初始化，存在多个副本，各个对象拥有的副本互不影响。

1.7.1.2.2. 所有的静态方法和静态变量都可以通过对象访问

```
public class StaticTest {
□static int value=33;
□public static void main(String[] args) {
□□new StaticTest().print();
□}
□public void print() {
□□int value=3;
□□System.out.println(this.value);
□}
}
```

返回的结果是33
这里面主要考察队this和static的理解。this代表什么？this代表当前对象，那么通过new Main()来调用printValue的话，当前对象就是通过new Main()生成的对象。而static变量是被对象所享有的，因此在printValue中的this.value的值毫无疑问是33。在printValue方法内部的value是局部变量，根本不可能与this关联，所以输出结果是33。在这里永远要记住一点：静态成员变量虽然独立于对象，但是不代表不可以通过对象去访问，所有的静态方法和静态变量都可以通过对象访问（只要访问权限足够）。

1.7.2. 静态代码块

1.7.2.1. 只会在类加载的时候执行一次，以优化程序性能

1.7.2.1.1. 很多时候会将一些只需要进行一次的初始化操作都放在static代码块中进行

```
private static Date startDate,endDate;
    static{
        startDate = Date.valueOf("1946-1-1");
        endDate = Date.valueOf("1964-12-31");
    }
```

1.7.3. 修饰内部类

1.7.3.1. 它的创建是不需要依赖外围类的创建。

1.7.3.2. 它不能使用任何外围类的非static成员变量和方法。

1.7.3.3. Example（静态内部类实现单例模式）

```
public class Singleton {

    声明为 private 避免调用默认构造方法创建对象
    private Singleton() {
    }

    声明为 private 表明静态内部该类只能在该 Singleton 类中被访问
    private static class SingletonHolder {
        private static final Singleton INSTANCE = new Singleton();
    }

    public static Singleton getUniqueInstance() {
        return SingletonHolder.INSTANCE;
    }
}
```

当 Singleton 类加载时，静态内部类 SingletonHolder 没有被加载进内存。只有当调用 getUniqueInstance()方法从而触发 SingletonHolder.INSTANCE 时 SingletonHolder 才会被加载，此时初始化 INSTANCE 实例，并且 JVM 能确保 INSTANCE 只被实例化一次。
这种方式不仅具有延迟初始化的好处，而且由 JVM 提供了对线程安全的支持。

1.7.4. 静态导包(用来导入类中的静态资源，1.5之后的新特性)

1.7.4.1. import static org.springframework.util.Assert.notNull;

1.8. static final/final static

1.8.1. static final修饰的属性表示一旦给值，就不可修改，并且可以通过类名访问

1.8.2. static final修饰方法，表示该方法不能重写，可以在不new对象的情况下调用

1.9. 抽象类和接口

1.9.1. 本质：抽象是对类的抽象，是一种模板设计；接口是行为的抽象，是一种行为的规范

1.9.2. 抽象类 abstract修饰

1.9.2.1. 为子类提供一个公共的类型

1.9.2.1.1. 封装子类中的重复属性和方法

1.9.2.2. 抽象类中不一定包含抽象方法，但是有抽象方法的类必定是抽象类，不可被实例化

1.9.2.3. 构造方法，类方法（用 static 修饰的方法）不能声明为抽象方法

1.9.2.4. 定义抽象方法

抽象方法：由abstract修饰的方法为抽象方法，抽象方法只有方法的定义，没有方法的实现
抽象方法不能被private、static、final或native并列修饰，只能被public、protected修饰 （如果为private则没有意义，因为子类无法继承，子类无法实现）

1.9.2.4.1. abstract修饰的方法

1.9.2.5. 抽象类的子类，除非也是抽象类，否则必须实现该抽象类声明的方法

1.9.2.6. 父类和派生类之间必须存在"is-a"关系，即父类和派生类在概念本质上应该是相同的

1.9.3. 接口 interface修饰

1.9.3.1. 接口的方法默认是public，所有方法在接口中不能有实现，抽象类可以有非抽象的方法

1.9.3.1.1. jdk9中，引入的私有方法(private method)和私有静态方法(private static method)

由于不可被继承，因此私有方法必须定义方法体才有意义。同时也意味着接口的私有方法和私有静态方法不可被实现该接口的类或继承该接口的接口调用或重写。私有方法(private method)和私有静态方法(private static method)的提出都是对jdk8提出的default和public static method的补充。
默认方法和静态方法可以共享接口中的私有方法，因此避免了代码冗余，这也使代码更加清晰。如果私有方法是静态的，那这个方法就属于

这个接口的。并且没有静态的私有方法只能被在接口中的实例调用。

1.9.3.2. 接口的实现，必须通过子类，子类使用关键字implements，而且接口可以多实现

1.9.3.3. 一个类实现接口的话要实现接口的所有方法，而抽象类不一定

1.9.3.4. 接口不能用new实例化，但可以声明，但是必须引用一个实现该接口的对象

1.9.3.5. 接口中的成员变量只能是public static final类型的

1.9.3.6. 实现者仅仅是实现了接口定义的行为契约而已，"like-a"的关系

1.10. 重载和重写的区别

1.10.1. 重载：发生在同一个类中，方法名必须相同，参数类型不同、个数不同

1.10.2. 重写：发生在父子类中，方法名、参数列表必须相同，返回值小于等于父类

1.10.2.1. 如果父类方法访问修饰符为private则子类中就不是重写

1.11. 克隆对象

1.11.1. 对已有的一个对象进行拷贝，形成一个对象的副本

1.11.2. 目的（意义）：被克隆和克隆对象之间完全复制、相互之间不影响

1.11.3. 场景

1.11.3.1. 方法需要 return 引用类型，但又不希望自己持有引用类型的对象被修改

1.11.3.2. 方法间参数的传递

1.11.4. 实现

1.11.4.1. 浅克隆

1.11.4.1.1. 克隆原对象属性值

1.11.4.1.1.1. 被克隆类上实现Cloneable接口，并重写clone方法

1.11.4.1.2. 浅克隆只是克隆原对象中的引用类型指向，并非克隆了原对象中的全部数据

1.11.4.2. 深克隆

1.11.4.2.1. 深克隆则是克隆了原对象的所有

1.11.4.2.2. 被克隆类上实现Cloneable接口，并使用递归或多层复制重写Clone（）方法

1.11.4.3. 利用序列化完成深克隆

1.12. Java 反射

1.12.1. 就是在运行状态中

1.12.1.1. 获取任意类的名称、package信息、所有属性、方法、注解、类型、类加载器等

1.12.1.2. 获取任意对象的属性，并且能改变对象的属性

1.12.1.3. 调用任意对象的方法

1.12.1.4. 判断任意一个对象所属的类

1.12.1.5. 实例化任意一个类的对象

1.12.2. JDK 中 java.lang.Class 类，就是为了实现反射提供的核心类之一

1.12.3. 可以实现动态装配

1.12.3.1. 降低代码的耦合度

1.12.3.1.1. 动态代理

1.13. 动态代理

1.13.1. 在运行时，创建目标类，可以调用和扩展目标类的方法。

1.13.2. JDK 动态代理

1.13.2.1. 实现 InvocationHandler 接口

1.13.2.2. 重写 invoke 方法，添加业务逻辑

1.13.2.3. 持有目标类对象

1.13.2.4. 提供静态方法获取代理

1.13.3. CGLib 动态代理

1.13.3.1. 使用了ASM（字节码操作框架）来操作字节码生成新的类

1.14. java 序列化

1.14.1. 序列化：将 Java 对象转换成字节流的过程

1.14.2. 反序列化：将字节流转换成 Java 对象的过程

1.15. 数据传输流

1.15.1. IO

1.15.1.1. 流式

1.15.1.1.1. 字节流

1.15.1.1.1.1. InputStream

1.15.1.1.1.2. OutputStream

1.15.1.1.2. 字符流

1.15.1.1.2.1. Writer

1.15.1.1.2.2. Reader

1.15.1.2. 非流式

1.15.1.2.1. file

1.15.1.3. 其他

1.15.1.3.1. SerializablePermission

1.15.1.3.2. FileSystem

1.15.2. BIO (Blocking I/O): 同步阻塞I/O模式

BIO (Blocking I/O): 同步阻塞I/O模式，数据的读取写入必须阻塞在一个线程内等待其完成。在活动连接数不是特别高（小于单机1000）的情况下，这种模型是比较不错的，可以让每一个连接专注于自己的 I/O 并且编程模型简单，也不用过多考虑系统的过载、限流等问题。线程池本身就是一个天然的漏斗，可以缓冲一些系统处理不了的连接或请求。但是，当面对十万甚至百万级连接的时候，传统的 BIO 模型是无能为力的。因此，我们需要一种更高效的 I/O 处理模型来应对更高的并发量。

1.15.3. NIO (New I/O): 同步非阻塞的I/O模型

NIO是一种同步非阻塞的I/O模型，在Java 1.4 中引入了NIO框架，对应 java.nio 包，提供了 Channel , Selector，Buffer等抽象。NIO中的N可以理解为Non-blocking，不单纯是New。它支持面向缓冲的，基于通道的I/O操作方法。 NIO提供了与传统BIO模型中的 Socket 和 ServerSocket 相对应的 SocketChannel 和 ServerSocketChannel 两种不同的套接字通道实现,两种通道都支持阻塞和非阻塞两种模式。阻塞模式使用就像传统中的支持一样，比较简单，但是性能和可靠性都不好；非阻塞模式正好与之相反。对于低负载、低并发的应用程序，可以使用同步阻塞I/O来提升开发速率和更好的维护性；对于高负载、高并发的（网络）应用，应使用 NIO 的非阻塞模式来开发

1.15.3.1. Channel双向通道

1.15.3.2. Buffer缓冲

Relationship:

1.15.3.3. Selector选择器

1.15.3.3.1. 多路复用

1.15.4. AIO (Asynchronous I/O):异步非阻塞的IO模型

AIO 也就是 NIO 2。在 Java 7 中引入了 NIO 的改进版 NIO 2,它是异步非阻塞的IO模型。异步 IO 是基于事件和回调机制实现的，也就是应用操作之后会直接返回，不会堵塞在那里，当后台处理完成，操作系统会通知相应的线程进行后续的操作。AIO 是异步IO的缩写，虽然 NIO 在网络操作中，提供了非阻塞的方法，但是 NIO 的 IO 行为还是同步的。对于 NIO 来说，我们的业务线程是在 IO 操作准备好时，得到通知，接着就由这个线程自行进行 IO 操作，IO操作本身是同步的。查阅网上相关资料，我发现就目前来说 AIO 的应用还不是很广泛，Netty 之前也尝试使用过 AIO，不过又放弃了。

1.16. Math.round(1.5)的返回值是2，Math.round(-1.5)的返回值是-1

1.16.1. 四舍五入的原理是在参数上加0.5然后做向下取整

1.17. Java如何将字符串反转

1.17.1. 利用 StringBuffer 或 StringBuilder 的 reverse 成员方法

1.17.1.1. new StringBuilder(str).reverse().toString();

1.17.2. 利用 String 的 toCharArray 方法先将字符串转化为 char 类型数组

1.17.3. 利用 String 的 CharAt 方法取出字符串中的各个字符

1.18. String类中常用的方法

1.18.1. 获取字符串长度

1.18.1.1. str.length()

1.18.2. 获取字符串某一位置的字符

1.18.2.1. str.charAt(4)

1.18.3. 获取字符串的子串

1.18.3.1. str.substring(2,5)

1.18.4. 字符串的比较

public int compareTo(String str)
//该方法是对字符串内容按字典顺序进行大小比较，
//通过返回的整数值指明当前字符串与参数字符串的大小关系。
//若当前对象比参数大则返回正整数，反之返回负整数，相等返回0。

public int compareToIgnoreCase (String str)
//与compareTo方法相似，但忽略大小写。

public boolean equals(Object obj)
//比较当前字符串和参数字符串，在两个字符串相等的时候返回true，否则返回false。

public boolean equalsIgnoreCase(String str)
//与equals方法相似，但忽略大小写。

1.18.5. 查找子串在字符串中的位置

public int indexOf(String str)
//用于查找当前字符串中字符或子串，返回字符或
//子串在当前字符串中从左边起首次出现的位置，若没有出现则返回-1。

public int indexOf(String str, intfromIndex)
//改方法与第一种类似，区别在于该方法从fromIndex位置向后查找。

public int lastIndexOf(String str)
//该方法与第一种类似，区别在于该方法从字符串的末尾位置向前查找。

public int lastIndexOf(String str, intfromIndex)
//该方法与第二种方法类似，区别于该方法从fromIndex位置向前查找。

1.18.5.1. str.indexOf('a')

1.18.5.1.1. str.indexOf('a',2)

1.18.5.2. str.lastIndexOf('a')

1.18.5.2.1. str.lastIndexOf('a',2)

1.18.6. 字符串中字符的大小写转换

1.18.6.1. toLowerCase()

1.18.6.2. toUpperCase()

1.18.7. 字符串两端去空格

1.18.7.1. trim()

1.18.8. 将字符串分割成字符串数组

1.18.8.1. split(String str)

1.18.9. 基本类型转换为字符串

1.18.9.1. String.valueOf(12.99)

1.18.10. 替换字符串

1.18.10.1. replace(char oldChar, charnewChar)

1.18.10.2. replaceFirst(String regex,String replacement)

1.18.10.3. replaceAll(String regex,String replacement)

1.19. File类常用方法

1.19.1. 创建功能

1.19.1.1. public boolean createNewFile() throws IOException 创建新文件

1.19.1.2. public boolean mkdirs() 创建新的目录，若父目录不存在，会自动创建

1.19.1.3. public boolean renameTo(File dest) 重命名文件

1.19.2. 判断功能

1.19.2.1. public boolean isFile() 判断是否是文件

1.19.2.2. public boolean isDirectory() 判断是否是目录

1.19.2.3. public boolean exists() 判断文件或者目录是否存在

1.19.2.4. public boolean canRead() 判断文件是否可读

1.19.2.5. public boolean canWrite() 判断文件是否可写

1.19.2.6. public boolean isHidden() 判断文件是否隐藏

1.19.3. 获取功能

1.19.3.1. public String getAbsolutePath() 获取绝对路径

1.19.3.2. public String getPath() 获取相对路径

1.19.3.3. public String getName() 获取文件或目录名

1.19.3.4. public long length() 获取文件大小

1.19.3.5. public long lastModified() 获取文件最后一次修改的时间（单位，毫秒）

1.19.4. 高级获取功能

1.19.4.1. public String[] list() 获取路径表示目录下的所有文件和目录名称

1.19.4.2. public String[] list(FilenameFilter filter) 获取满足过滤器FilenameFilter条件的所有目录或文件

1.19.4.3. public File[] listFiles() 获取路径表示目录下的所有文件和目录对象（文件类型）

1.19.4.4. public File[] listFiles(FilenameFilter filter) 获取满足过滤器FilenameFilter条件的所有目录或文件对象

1.20. a+=b 和a=a+b

1.20.1. 数据类型一样时，无区别

1.20.2. 数据类型不同时，+=数据类型自动转换

1.21. i++ 和 ++i的区别

++在前: 先自增然后在做运算, ++在后先做运算在自增

1.21.1. i++是先赋值，然后再自增

1.21.1.1. ++i是先自增，后赋值

1.21.2. 运算符优先问题

```
public static void main(String[] args) {
    int x = 10;
    int y = 10;
    System.out.println("x+y="+x+y);
    System.out.println("x*y="+x*y);
    System.out.println("x/y="+x/y);
}
```

第一个打印：x+y=1010 ,第二个打印x*y=100,第三个打印x/y=1;

1.21.2.1. 临时变量区陷阱

```
@Test
public void testCount() {
    int count = 0;
    int y =0;
    for (int i = 1; i <= 10; i++) {
        count = count++;
    }
    System.out.println(count);
}
```

循环的时候首先把count的值（注意是值，不是引用）拷贝到一个临时变量区。
然后对count变量加1。
返回临时变量区的值，注意这个值是0，没修改过。

1.21.3. i++ 和 ++i语句的执行过程有多个操作组成，不是原子操作，因此不是线程安全的

1.22. 求字符串字母重复数

1.22.1. TreeMap集合获取字符串中每一个字母出现的次数

1.22.2. set集合

1.22.3. 遍历

1.23. ==和hashCode和equals方法

1.23.1. equals()才能确认是否真的相等

1.23.1.1. Object类中默认的实现方式是 : return this == obj 。只有this 和 obj引用同一个对象，才会返回true。

1.23.1.2. 重写equals

1.23.1.2.1. 自反性: x.equals(x) 一定是true

1.23.1.2.2. 对null: x.equals(null) 一定是false

1.23.1.2.3. 对称性: x.equals(y) 和 y.equals(x)结果一致

1.23.1.2.4. 传递性: a 和 b equals , b 和 c equals，那么 a 和 c也一定equals。

1.23.1.2.5. 一致性: 在某个运行时期间，2个对象的状态的改变不会不影响equals的决策结果

1.23.2. hashCode()用来缩小寻找范围

1.23.2.1. 重写equals，也必须重写hashCode

1.23.2.2. 参与equals函数的字段，也必须都参与hashCode的计算

1.23.2.2.1. 等价的(调用equals返回true)对象必须产生相同的散列码。不等价的对象，不要求产生的散列码不相同

1.23.2.2.2. equals中衡量相等的字段参入散列运算，每一个重要字段都会产生一个hash分量，为最终的hash值做出贡献

1.23.3. ==判断对象地址是否相等

2. java 容器

2.1. 集合容器

2.1.1. Map

采用键值对<key,value>存储元素，key键唯一

2.1.1.1. HashMap

2.1.1.1.1. 利用哈希算法根据hashCode()配置存储地址

2.1.1.1.2. null可以作为键，这样的键只有一个

2.1.1.1.3. 非线程安全的

2.1.1.1.3.1. 写入操作丢失

修改覆盖

2.1.1.2. Hashtable

2.1.1.2.1. key和value都不允许出现null值

2.1.1.2.2. 加了synchronized，线程安全

2.1.1.3. LinkedHashMap

2.1.1.3.1. 基于HashMap和双向链表/红黑树实现的

2.1.1.3.1.1. 有序

插入顺序

插入的是什么顺序，读出来的就是什么顺序

访问顺序

访问了一个key，这个key就跑到了最后面

2.1.1.4. TreeMap

2.1.1.4.1. 基于红黑树的NavigableMap 实现

2.1.1.4.1.1. 红黑树属于平衡二叉树

平均高度log(n)，最坏情况高度不会超过2log(n)

2.1.1.4.1.2. 红黑树能够以O(log2(N))的时间复杂度进行搜索、插入、删除操作

2.1.1.4.1.3. 任何不平衡都会在3次旋转之内解决

2.1.1.4.2. Key值是要求实现java.lang.Comparable

2.1.1.4.2.1. 迭代的时候TreeMap默认是按照Key值升序排序

2.1.1.5. SortedMap接口

2.1.1.5.1. 键的总体排序 的 Map

2.1.2. ConcurrentHashMap

2.1.2.1. 高效且线程安全

2.1.3. Collection

2.1.3.1. Set

具有唯一性，无序性

2.1.3.1.1. HashSet

2.1.3.1.2. LinkedHashSet

2.1.3.1.3. TreeSet

2.1.3.2. List

元素可重复性，有序性
String的长度限制: 底层是char 数组 长度 Integer.MAX_VALUE 线程安全的

2.1.3.2.1. ArrayList

2.1.3.2.2. Vector 队列（线程安全）

2.1.3.2.2.1. Stack 栈（线程安全）

2.1.3.2.3. LinkedList

2.1.3.3. Queue

2.1.3.3.1. PriorityQueue

2.2. 泛型擦除

泛型，即“参数化类型”。
创建集合时就指定集合元素的类型，该集合只能保存其指定类型的元素，避免使用强制类型转换。
Java编译器生成的字节码是不包涵泛型信息的，泛型类型信息将在编译处理是被擦除，这个过程即类型擦除。泛型擦除可以简单的理解为将泛型java代码转换为普通java代码，只不过编译器更直接点，将泛型java代码直接转换成普通java字节码。
类型擦除的主要过程如下：
1）.将所有的泛型参数用其最左边界（最顶级的父类型）类型替换。
2）.移除所有的类型参数。

2.2.1. 泛型和Object的区别

2.2.1.1. 泛型声明

2.2.1.1.1. public T doSomeThing(T t){return t; }

2.2.1.1.1.1. 泛型引用

不再需要强制转换，编译时自动检查类型安全，避免隐性的类型转换异常

2.2.1.2. Object声明

2.2.1.2.1. public Object doSomeThing(Object obj){ return obj; }

2.2.1.2.1.1. Object引用

可能发生类型转换异常（ClassCastException）

2.3. HashMap,HashSet,HashTable的区别

*HashMap**HashSet* HashMap实现了Map接口HashSet实现了Set接口
HashMap储存键值对HashSet仅仅存储对象使用put()方法将元素放入map
中使用add()方法将元素放入set中HashMap中使用键对象来计算hashcode
值HashSet使用成员对象来计算hashcode值，对于两个对象来说hashcode
可能相同，所以equals()方法用来判断对象的相等性，如果两个对象不同
的话，那么返回falseHashMap比较快，因为是使用唯一的键来获取对象
HashSet较HashMap来说比较慢

2.3.1. HashSet

HashSet实现了Set接口，它不允许集合中有重复的值，当我们提到HashSet时，第一件事情就是在将对象存储在HashSet之前，要先确保对象重写equals()和hashCode()方法，这样才能比较对象的值是否相等，以确保set中没有储存相等的对象。如果我们没有重写这两个方法，将会使用这个方法的默认实现。
public boolean add(Object o)方法用来在Set中添加元素，当元素值重复时则会立即返回false，如果成功添加的话会返回true。

2.3.2. HashMap

HashMap实现了Map接口，Map接口对键值对进行映射。Map中不允许重复的键。Map接口有两个基本的实现，HashMap和TreeMap。TreeMap保存了对象的排列次序，而HashMap则不能。HashMap允许键和值为null。HashMap是非synchronized的，但collection框架提供方法能保证HashMap synchronized，这样多个线程同时访问HashMap时，能保证只有一个线程更改Map。

HashMap在底层数据结构上采用了数组 + 链表 + 红黑树，通过散列映射来存储键值对数据因为在查询上使用散列码（通过键生成一个数字作为数组下标，这个数字就是hash code）所以在查询上的访问速度比较快，HashMap最多允许一对键值对的Key为Null，允许多对键值对的value为Null。它是非线程安全的。在排序上面是无序的。
在jdk1.7中的HashMap是位桶+链表实现在jdk1.8中的HashMap是位桶+链表+红黑树实现未超过8个节点时，是位桶+链表实现，在节点数超过8个节点时，是位桶+红黑树实现。

2.3.2.1. 哈希冲突

2.3.2.1.1. 链地址法

2.3.2.1.1.1. 链表

2.3.2.2. JDK1.7

2.3.2.2.1. 位桶+链表

2.3.2.3. JDK1.8

2.3.2.3.1. 位桶+链表+红黑树

未超过8个节点时，是位桶+链表实现，在节点数超过8个节点时，是位桶+红黑树实现。

2.3.3. HashTable

2.3.4. ConcurrentHashMap

ConcurrentHashMap是基于segment数组内嵌哈希表的ConcurrentMap接口实现，线程安全。

ConcurrentHashMap的数据结构（数组+链表+红黑树），桶中的结构可能是链表，也可能是红黑树，红黑树是为了提高查找效率。

ConcurrentHashMap的主要实现原理就是使用segments将原本唯一的hashtable分段, 增加并发能力。

ConcurrentHashMap使用两次哈希算法（single-word Wang/Jenkins哈希算法的一个变种）尽量将元素均匀的分布到不同的segment的hashtable中：第一次哈希算法找到segment；第二次哈希算法找到hashentry；这两次哈希算法要求能尽量将元素均匀的分布到不同的segment的hashtable中, ConcurrentHashMap使用的是single-word Wang/Jenkins哈希算法的一个变种

2.4. HashMap、LinkedHashMap、TreeMap的区别

2.4.1. HashMap

2.4.1.1. 数组+链表/红黑树

2.4.1.1.1. 取值无顺序

2.4.1.1.1.1. 最多一条记录的key为null

绝大多数无需排序的情况

2.4.2. LinkedHashMap

2.4.2.1. 数组+链表/红黑树

2.4.2.1.1. 取值按插入的顺序/按修改的顺序，根据accessOrder控制

2.4.2.1.1.1. 最多一条记录的key为null

需要插入的顺序和取出的顺序一样的情况

2.4.3. TreeMap

2.4.3.1. 红黑树

2.4.3.1.1. 插入时按key的自然顺序或者自定义顺序

2.4.3.1.1.1. 当为key的自然顺序存储时key不能为null

需要按照key的自然顺序甚至于自定义顺序的情况下

2.5. Collection和Collections有什么区别

2.5.1. java.util.Collection 是一个集合接口

2.5.1.1. 提供了对集合对象进行基本操作的通用接口方法

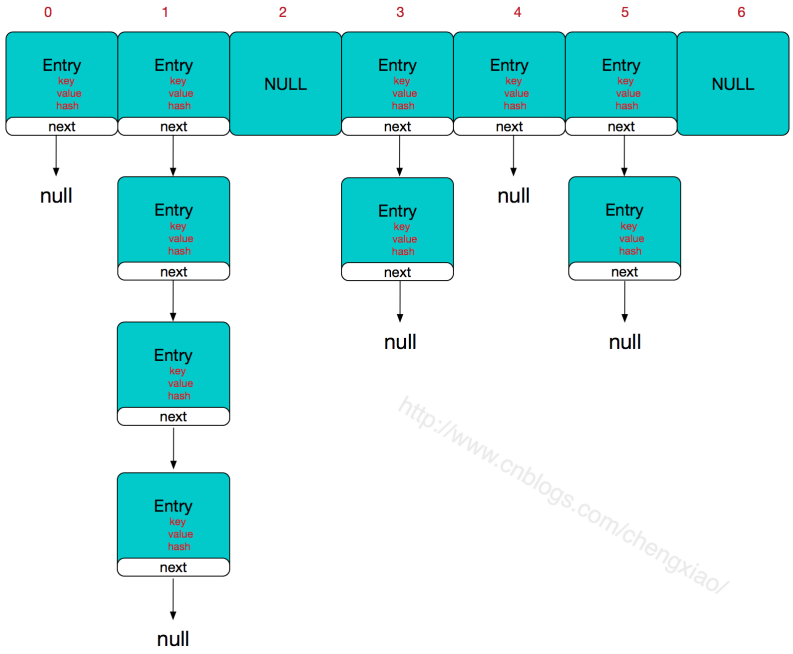
2.5.2. java.util.Collections 是一个包装类

2.5.2.1. 各种有关集合操作的静态方法

2.5.2.2. 不能实例化，就像一个工具类，服务于Java的Collection框架

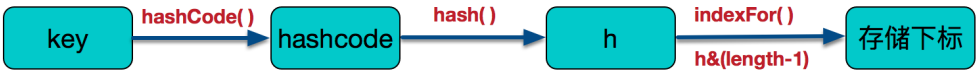
2.6. HashMap 的实现原理

2.6.1. Entry数组



简单来说，HashMap由数组+链表组成的，数组是HashMap的主体，链表则是主要为了解决哈希冲突而存在的，如果定位到的数组位置不含链表（当前entry的next指向null）,那么对于查找，添加等操作很快，仅需一次寻址即可；如果定位到的数组包含链表，对于添加操作，其时间复杂度为O(n)，首先遍历链表，存在即覆盖，否则新增；对于查找操作来讲，仍需遍历链表，然后通过key对象的equals方法逐一比对查找。所以，性能考虑，HashMap中的链表出现越少，性能才会越好。

2.6.2. 存储位置

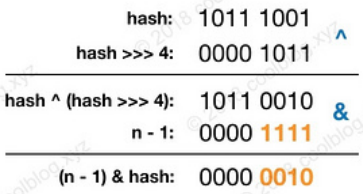


2.6.3. 源码

2.6.3.1. static int indexFor(int h, int length) { return h & (length-1);}

2.6.3.1.1. static final int hash(Object key) { int h; return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16); }

2.6.3.1.1.1.



2.6.3.2. 由于和（length-1）运算，length 绝大多数情况小于2的16次方。所以始终是hashcode 的低16位参与运算。要是高16位也参与运算，会让得到的下标更加散列

2.6.3.2.1. 所以(h >>> 16)得到他的高16位与hashCode()进行^运算

2.6.3.2.2. 因为&和|都会使得结果偏向0或者1 ,并不是均匀的概念,所以用异或

2.7. HashSet 的实现原理

2.7.1. 不允许有重复元素

2.7.1.1. HashSet的值存放于HashMap的key上

2.7.2. HashSet底层由HashMap实现，无序

2.7.2.1. HashSet中的元素都存放在HashMap的key上面

2.7.2.2. value统一为无意义静态常量private static final **Object PRESENT = new Object();**

2.8. ArrayList 和 LinkedList 的区别

2.8.1. Arraylist基于动态数组

2.8.1.1. 最大的数组容量是Integer.MAX_VALUE-8

2.8.1.1.1. 对于空出的8位

2.8.1.1.1.1. ①存储Headerwords

2.8.1.1.1.2. ②避免一些机器内存溢出，减少出错几率，所以少分配

2.8.1.1.1.3. ③最大还是能支持到Integer.MAX_VALUE

2.8.2. Linkedlist基于链表的动态数组

2.8.2.1. 数据添加删除效率高，只需要改变指针指向即可

2.8.2.2. 但是访问数据的平均效率低，需要对链表进行遍历

2.9. 数组和list集合之间转换的各种方法

2.9.1. 数组转List

```
package listtoArray;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class ArrayToList {

□public static void main(String[] args) {

□□//数组转list
□□String[] str=new String[] {"hello","world"};
□□//方式一：使用for循环把数组元素加进list
□□List<String> list=new ArrayList<String>();
□□for (String string : str) {
□□□list.add(string);
□□}
□□System.out.println(list);

□□//方式二：
□□List<String> list2=new
ArrayList<String>(Arrays.asList(str));
□□System.out.println(list2);

□□//方式三：
□□//同方法二一样使用了asList()方法。这不是最
好的，
□□//因为asList()返回的列表的大小是固定的。
```

□□//事实上，返回的列表不是java.util.ArrayList
类，而是定义在java.util.Arrays中一个私有静态类java.util.Arrays.ArrayList
□□//我们知道ArrayList的实现本质上是一个数
组，而asList()返回的列表是由原始数组支持的固定大小的列表。
□□//这种情况下，如果添加或删除列表中的元素，
程序会抛出异常UnsupportedOperationException。
□□//java.util.Arrays.ArrayList类具有 set()，
get()，contains()等方法，但是不具有添加add()或删除remove()方法,所以调
用add()方法会报错。
□□List<String> list3 = Arrays.asList(str);
□□//list3.remove(1);
□□//boolean contains = list3.contains("s");
□□//System.out.println(contains);
□□System.out.println(list3);

□□//方式四：使用Collections.addAll()
□□List<String> list4=new
ArrayList<String>(str.length);
□□Collections.addAll(list4, str);
□□System.out.println(list4);

□□//方式五：使用Stream中的Collector收集器
□□//转换后的List 属于 java.util.ArrayList 能进行
正常的增删查操作
□□List<String>
list5=Stream.of(str).collect(Collectors.toList());
□□System.out.println(list5);
□}
}

2.9.1.1. 使用Stream中的Collector收集器

2.9.1.1.1. List list5=Stream.of(str).collect(Collectors.toList());

2.9.2. List转数组

```
package listtoArray;

import java.util.ArrayList;
import java.util.List;

public class ListToArray {
```

```
□public static void main(String[] args) {
□□//list转数组
□□List<String> list=new ArrayList<String>();
□□list.add("hello");
□□list.add("world");

□□//方式一：使用for循环
□□String[] str1=new String[list.size()];
□□for(int i=0;i<list.size();i++) {
□□□str1[i]=list.get(i);
□□}
□□for (String string : str1) {
□□□System.out.println(string);
□□}

□□//方式二：使用toArray()方法
□□//list.toArray(T[] a); 将list转化为你所需要类
型的数组
□□String[] str2=list.toArray(new
String[list.size()]);
□□for (String string : str2) {
□□□System.out.println(string);
□□}

□□//错误方式：易错 list.toArray()返回的是
Object[]数组，怎么可以转型为String
□□//ArrayList<String> list3=new
ArrayList<String>();
□□//String strings[]=(String [])list.toArray();
□}

}
```

2.9.2.1. 使用toArray()方法

2.9.2.1.1. String[] str2=list.toArray(new String[list.size()]);

2.10. ArrayList与Vector的区别

2.10.1. 同步性

2.10.1.1. Vector是线程安全的

2.10.1.1.1. 方法之间是线程同步

2.10.1.2. ArrayList是线程序不安全的

2.10.1.2.1. 方法之间是线程不同步

2.10.2. 数据增长

2.10.2.1. Vector默认增长为原来的两倍

2.10.2.2. ArrayList增长为原来的1.5倍

2.11. Array 和 ArrayList 有何区别

2.11.1. ArrayList想象成一种“会自动扩增容量的Array”

2.11.2. Array大小固定，ArrayList的大小是动态变化的

2.11.3. Array可以包含基本类型和对象类型，ArrayList只能包含对象类型

2.12. Queue队列是一个典型的先进先出（FIFO）的容器

2.12.1. 向队列中添加一个元素 offer()和add()的区别

2.12.1.1. 如果想在一个满的队列中加入一个新元素

2.12.1.1.1. offer() 方法会返回 false

2.12.1.1.2. add() 方法就会抛出一个 unchecked 异常

2.12.2. 不移除的情况下返回队头 peek()和element()的区别

2.12.2.1. peek()方法在队列为空时返回null

2.12.2.2. element()方法会抛出NoSuchElementException异常

2.12.3. 移除并且返回队头 poll()和 remove()的区别

2.12.3.1. poll()在队列为空时返回null

2.12.3.2. remove()会抛出NoSuchElementException异常

2.13. 迭代器Iterator

2.13.1. 迭代器模式

2.13.1.1. 使得序列类型的数据结构的遍历行为与被遍历的对象分离

2.13.2. Iterable

2.13.2.1. 实现这个接口的集合对象支持迭代，是可以迭代的

2.13.2.1.1. 可以配合foreach使用

2.13.3. Iterator：迭代器

Iterator迭代器包含的方法:
hasNext()：如果迭代器指向位置后面还有元素，则返回 true，否则返回 false
next()：返回集合中Iterator指向位置后面的元素
remove()：删除集合中Iterator指向位置后面的元素

2.13.3.1. 提供迭代机制的对象，具体如何迭代是这个Iterator接口规范的

2.13.4. foreach和Iterator的关系

2.13.4.1. foreach中调用集合remove会导致原集合变化导致错误

2.13.4.1.1. 应该用迭代器的remove方法

2.13.5. for循环和迭代器Iterator对比

2.13.5.1. for循环中的get()方法，采用随机访问的方法

2.13.5.1.1. ArrayList对随机访问比较快

2.13.5.2. iterator中的next()方法，采用顺序访问的方法

2.13.5.2.1. LinkedList则是顺序访问比较快

2.13.6. Iterator 和 ListIterator 的区别

2.13.6.1. ListIterator迭代器

ListIterator迭代器包含的方法：
add(E e)：将指定的元素插入列表，插入位置为迭代器当前位置之前
hasNext()：以正向遍历列表时，如果列表迭代器后面还有元素，则返回true，否则返回false
hasPrevious()：如果以逆向遍历列表，列表迭代器前面还有元素，则返回true，否则返回false
next()：返回列表中ListIterator指向位置后面的元素
nextIndex()：返回列表中ListIterator所需位置后面元素的索引
previous()：返回列表中ListIterator指向位置前面的元素
previousIndex()：返回列表中ListIterator所需位置前面元素的索引
remove()：从列表中删除next()或previous()返回的最后一个元素（有点拗口，意思就是对迭代器使用hasNext()方法时，删除ListIterator指向位置后面的元素；当对迭代器使用hasPrevious()方法时，删除ListIterator指向位置前面的元素）
set(E e)：从列表中将next()或previous()返回的最后一个元素返回的最后一个元素更改为指定元素e

2.13.6.2. 使用范围不同

2.13.6.2.1. Iterator可以应用于所有的集合，Set、List和Map和这些集合的子类型

2.13.6.2.2. ListIterator只能用于List及其子类型

2.13.6.3. ListIterator功能更强大，可以增删改查

Iterator 和 ListIterator 的不同点
1.使用范围不同，Iterator可以应用于所有的集合，Set、List和Map和这些集合的子类型。而ListIterator只能用于List及其子类型。
2.ListIterator有add方法，可以向List中添加对象，而Iterator不能。
3.ListIterator和Iterator都有hasNext()和next()方法，可以实现顺序向后遍历，但是ListIterator有hasPrevious()和previous()方法，可以实现逆向（顺序向前）遍历。Iterator不可以。
4.ListIterator可以定位当前索引的位置，nextIndex()和previousIndex()可以实现。Iterator没有此功能。
5.都可实现删除操作，但是ListIterator可以实现对象的修改，set()方法可以实现。Iterator仅能遍历，不能修改。

2.14. 怎么确保一个集合不能被修改

2.14.1. 使用Collections包下的Collections.unmodifiableCollection(Collection c)方法

2.14.1.1. 改变集合的任何操作都会抛出 Java. lang. UnsupportedOperationException 异常

2.15. List去重方式

2.15.1. 利用java8的stream去重

2.15.1.1. List uniqueList = list.stream().distinct().collect(Collectors.toList());

2.15.2. Set集合不允许重复元素

2.15.3. 遍历去重

3. 并发编程

3.1. JMM

3.1.1. 原子性

3.1.1.1. 一个或者多个操作在 CPU 执行的过程中不被中断的特性

3.1.1.1.1. Atomic开头的原子类

3.1.1.1.1.1. CAS 原理

CAS 包含 3 个参数，CAS(V, E, N)。V 表示需要更新的变量，E 表示变量当前期望值，N 表示更新为的值。只有当变量 V 的值等于 E 时，变量 V 的值才会被更新为 N。如果变量 V 的值不等于 E ，说明变量 V 的值已经被更新过，当前线程什么也不做，返回更新失败。
当多个线程同时使用 CAS 更新一个变量时，只有一个线程可以更新成功，其他都失败。失败的线程不会被挂起，可以继续重试 CAS，也可以放弃操作。
CAS 操作的原子性是通过 CPU 单条指令完成而保障的。JDK 中是通过 Unsafe 类中的 API 完成的。
在并发量很高的情况，会有大量 CAS 更新失败，所以需要慎用。

3.1.1.1.1.2. AtomicLong >> LongAdder

AtomicLong 是基于 CAS 方式自旋更新

LongAdder 是把 value 分成若干cell

AtomicLong 是基于 CAS 方式自旋更新的；LongAdder 是把 value 分成若干cell，并发量低的时候，直接 CAS 更新值，成功即结束。并发量高的情况，CAS更新某个cell值和需要时对cell数据扩容，成功结束；更新失败自旋 CAS 更新 cell值。取值的时候，调用 sum() 方法进行每个cell累加。
AtomicLong 包含有原子性的读、写结合的api；LongAdder 没有原子性的读、写结合的api，能保证结果最终一致性。
低并发场景AtomicLong 和 LongAdder 性能相似，高并发场景 LongAdder 性能优于 AtomicLong。

3.1.2. 可见性

各个线程对主内存共享变量操作，需要先将之拷贝到线程各自的工作内存中，然后进行操作，最后写回主内存。
主内存的某一共享变量值发生改变，将改变后的值推送给所有该变量的副本。

3.1.2.1. 一个线程对共享变量的修改，另外一个线程能够立刻看到

3.1.2.1.1. volatile

3.1.3. 有序性

3.1.3.1. 程序执行的顺序按照代码的先后顺序执行

3.1.3.1.1. Happens-Before 规则

3.1.3.1.1.1. 程序次序规则

在一个线程内，按照程序控制流顺序，书写在前面的操作先行发生于书写在后面的操作

3.1.3.1.1.2. 管程锁定规则

一个unlock操作先行发生于后面对同一个锁的lock操作

3.1.3.1.1.3. volatile变量规则

对一个volatile变量的写操作先行发生于后面对这个变量的读操作

3.1.3.1.1.4. 线程启动规则

Thread对象的start()方法先行发生于此线程的每一个动作

3.1.3.1.1.5. 线程终止规则

线程中的所有操作都先行发生于对此线程的终止检测

3.1.3.1.1.6. 线程中断规则

对线程interrupt()方法的调用先行发生于被中断线程的代码检测到中断事件的发生

3.1.3.1.1.7. 对象终结规则

一个对象的初始化完成(构造函数执行结束)先行发生于它的finalize()方法的开始

3.2. JVM

3.2.1. GC

3.2.1.1. 避免使用finalize()对象终结方法

3.3. 锁

偏向锁
轻量级锁
重量级锁
适用场景
只有一个线程进入同步块
虽然很多线程，但是没有冲突：多条线程进入同步块，但是线程进入时间错开因而并未争抢锁
发生了锁争抢的情况：多条线程进入同步块并争用锁
本质
取消同步操作
CAS操作代替互斥同步

互斥同步
优点
不阻塞，执行效率高（只有第一次获取偏向锁时需要CAS操作，后面只是比对ThreadId）
不会阻塞
不会空耗CPU
缺点
适用场景太局限。若竞争产生，会有额外的偏向锁撤销的消耗
长时间获取不到锁空耗CPU
阻塞，上下文切换，重量级操作，消耗操作系统资源

3.3.1. 无锁

无锁：没有对资源进行锁定，所有的线程都能访问并修改同一个资源，但同时只有一个线程能修改成功，其他修改失败的线程会不断重试直到修改成功。

3.3.1.1. 偏向锁

偏向锁：对象的代码一直被同一线程执行，不存在多个线程竞争，该线程在后续的执行中自动获取锁，降低获取锁带来的性能开销。偏向锁，指的就是偏向第一个加锁线程，该线程是不会主动释放偏向锁的，只有当其他线程尝试竞争偏向锁才会被释放。
偏向锁的撤销，需要在某个时间点上没有字节码正在执行时，先暂停拥有偏向锁的线程，然后判断锁对象是否处于被锁定状态。如果线程不处于活动状态，则将对象头设置成无锁状态，并撤销偏向锁；
如果线程处于活动状态，升级为轻量级锁的状态。

3.3.1.1.1. 轻量级锁

轻量级锁：轻量级锁是指当锁是偏向锁的时候，被第二个线程 B 所访问，此时偏向锁就会升级为轻量级锁，线程 B 会通过自旋的形式尝试获取锁，线程不会阻塞，从而提高性能。
当前只有一个等待线程，则该线程将通过自旋进行等待。但是当自旋超过一定的次数时，轻量级锁便会升级为重量级锁；当一个线程已持有锁，另一个线程在自旋，而此时又有第三个线程来访时，轻量级锁也会升级为重量级锁。

3.3.1.1.1.1. 重量级锁

重量级锁：指当有一个线程获取锁之后，其余所有等待获取该锁的线程都会处于阻塞状态。
重量级锁通过对象内部的监视器（monitor）实现，而其中 monitor 的本质是依赖于底层操作系统的 Mutex Lock 实现，操作系统实现线程之间的切换需要从用户态切换到内核态，切换成本非常高。

3.4. 死锁

线程死锁是指由于两个或者多个线程互相持有所需要的资源，导致这些线程一直处于等待其他线程释放资源的状态，无法前往执行，如果线程都不主动释放所占有的资源，将产生死锁。
当线程处于这种僵持状态时，若无外力作用，它们都将无法再向前推进。
产生原因：
持有系统不可剥夺资源，去竞争其他已被占用的系统不可剥夺资源，形成程序僵死的竞争关系。（不可剥夺资源如打印机等）
持有资源的锁，去竞争锁已被占用的其他资源，形成程序僵死的争关系。
信号量使用不当。
...

3.4.1. 四个必要条件

3.4.1.1. 互斥条件

3.4.1.1.1. 一段时间内某资源仅为一个线程所占有，其他请求该资源的线程只能等待

3.4.1.2. 不剥夺条件

3.4.1.2.1. 线程所获得的资源在未使用完毕之前，不能被其他线程强行夺走

3.4.1.2.1.1. 只能由获得该资源的线程自己主动释放

3.4.1.3. 请求和保持条件

3.4.1.3.1. 线程已经保持了至少一个资源，但又提出了新的资源请求

3.4.1.3.1.1. 而该资源已被其他线程占有，此时请求线程被阻塞

对自己已获得的资源保持不放

3.4.1.4. 循环等待条件

3.4.1.4.1. 存在一种线程资源的循环等待链

3.4.1.4.1.1. 每一个线程已获得的资源同时被链中下一个线程所请求

3.4.2. 避免死锁的方式

3.4.2.1. 加锁顺序

3.4.2.1.1. 线程按照一定的顺序加锁

3.4.2.2. 加锁时限

3.4.2.2.1. 线程尝试获取锁的时候加上一定的时限

3.4.2.2.1.1. 超过时限则放弃对该锁的请求，并释放自己占有的锁

3.4.2.3. 死锁检测

3.5. 自旋锁

3.5.1. 线程反复检查锁变量是否可用

自旋锁：线程获取锁的时候，如果锁被其他线程持有，则当前线程将循环等待，直到获取到锁。
自旋锁等待期间，线程的状态不会改变，线程一直是用户态并且是活动的(active)。
自旋锁如果持有锁的时间太长，则会导致其它等待获取锁的线程耗尽CPU。
自旋锁本身无法保证公平性，同时也无法保证可重入性。
基于自旋锁，可以实现具备公平性和可重入性质的锁。
TicketLock:采用类似银行排号叫好的方式实现自旋锁的公平性，但是由于不停的读取serviceNum，每次读写操作都必须在多个处理器缓存之间进行缓存同步，这会导致繁重的系统总线和内存的流量，大大降低系统整体的性能。
CLHLock和MCSLock通过链表的方式避免了减少了处理器缓存同步，极大的提高了性能，区别在于CLHLock是通过轮询其前驱节点的状态，而MCS则是查看当前节点的锁状态。
CLHLock在NUMA架构下使用会存在问题。在没有cache的NUMA系统架构中，由于CLHLock是在当前节点的前一个节点上自旋,NUMA架构中处理器访问本地内存的速度高于通过网络访问其他节点的内存，所以CLHLock在NUMA架构上不是最优的自旋锁。

3.5.1.1. TicketLock锁主要解决公平性问题

3.5.1.1.1. CLH锁是一种基于链表的可扩展、高性能、公平的自旋锁

3.5.1.1.1.1. MCSLock则对本地变量的节点进行循环

3.5.2. CAS实现

3.5.2.1. CAS有3个操作数，内存值V，旧的预期值A，要修改的新值B

3.5.2.1.1. 当且仅当预期值A和内存值V相同时，将内存值V修改为B，否则什么都不做

3.5.2.1.1.1. 深入原理

CAS票据

小马

3.6. 锁优化

3.6.1. JVM优化

3.6.1.1. 优化的方向就是减少线程的阻塞

3.6.1.1.1. 在 Java SE 1.6 中，锁一共有4个状态

3.6.1.1.1.1. 无锁状态，偏向锁状态，轻量级锁状态，重量级锁状态

状态会随着竞争情况逐渐升级

锁升级之后不能降级

3.6.1.1.2. 锁消除

3.6.1.1.2.1. JIT 编译器对一些没有必要同步的代码却同步了的锁进行消除

逃逸分析

就是观察某一个变量是否会逃出某一个作用域

3.6.1.1.3. 锁粗化

3.6.1.1.3.1. 如果一系列连续操作都对同一个对象反复加锁和解锁

JVM将会把加锁同步的范围扩展（粗化）到整个操作序列的外部

3.6.2. 程序优化

3.6.2.1. 减小锁的持有时间

3.6.2.1.1. 减小锁的粒度

3.6.2.1.1.1. 使用读写锁替换独占锁

锁分离

无锁

CAS

3.7. volatile

JVM提供的轻量级同步机制

3.7.1. 作用

3.7.1.1. 只能作用于变量

3.7.1.1.1. 表示变量在 CPU 的寄存器中是不确定的，必须从主存中读取

3.7.2. 特点

3.7.2.1. 保证可见性

3.7.2.2. 不保证原子性

3.7.2.3. 禁止指令重排

3.7.2.4. 线程阻塞

3.8. synchronized同步锁

3.8.1. 作用

3.8.1.1. 修饰一个代码块

3.8.1.1.1. 同步语句块

####### 3.8.1.1.1.1. 其作用的范围是大括号{}括起来的代码

####### 3.8.1.1.1.2. 作用的对象是调用这个代码块的对象

3.8.1.2. 修饰一个方法

3.8.1.3. 修饰一个静态的方法

3.8.1.4. 修饰一个类

3.8.2. 特点

3.8.2.1. 保证线程间的有序性、原子性和可见性

3.8.2.2. 线程阻塞

3.8.3. 原理

3.8.3.1. 字节码加标识

3.8.3.1.1. 同步代码块是通过 monitorenter 和 monitorexit 指令获取线程的执行权

3.8.3.1.2. 同步方法通过加 ACC_SYNCHRONIZED 标识实现线程的执行权的控制

3.9. ReadWriteLock 读写锁

3.9.1. 一种实现ReentrantReadWriteLock

特点：
包含一个 ReadLock 和 一个 WriteLock 对象
读锁与读锁不互斥；读锁与写锁，写锁与写锁互斥
适合对共享资源有读和写操作，写操作很少，读操作频繁的场景
可以从写锁降级到读锁。获取写锁->获取读锁->释放写锁
无法从读锁升级到写锁
读写锁支持中断
写锁支持Condition；读锁不支持Condition

3.9.1.1. synchronized 和 ReentrantLock 的区别

synchronized 竞争锁时会一直等待；ReentrantLock 可以尝试获取锁，并得到获取结果
synchronized 获取锁无法设置超时；ReentrantLock 可以设置获取锁的超时时间
synchronized 无法实现公平锁；ReentrantLock 可以满足公平锁，即先等待先获取到锁
synchronized 控制等待和唤醒需要结合加锁对象的 wait() 和 notify()、notifyAll()；ReentrantLock 控制等待和唤醒需要结合 Condition 的 await() 和 signal()、signalAll() 方法
synchronized 是 JVM 层面实现的；ReentrantLock 是 JDK 代码层面实现
synchronized 在加锁代码块执行完或者出现异常，自动释放锁；
ReentrantLock 不会自动释放锁，需要在 finally{} 代码块显示释放

3.10. Lock

3.10.1. 代码实现

3.10.1.1. lock()以阻塞方式获得锁，且阻塞态会忽略interrupt方法

3.10.1.2. lockInterruptibly（）不同于lock（）不会忽略interrupt方法

3.10.1.3. tryLock()以非阻塞方式获得锁，可以加时间参数

3.10.2. Lock和synchronized

3.10.2.1. 相同点：Lock能完成synchronized所实现的所有功能

3.10.2.2. 不同点：Lock有比synchronized更精确的线程语义和更好的性能

3.10.2.2.1. Lock的锁定是通过代码实现的

3.10.2.2.1.1. 程序员手工释放，并且必须在finally从句中释放

3.10.2.2.2. synchronized是在JVM层面上实现的

3.10.2.2.2.1. 自动释放锁

3.10.2.2.3. Lock锁的范围有局限性，块范围

3.10.2.2.4. synchronized可以锁住块、对象、类

3.11. 进程和线程的区别

程序是一组指令的有序集合，它本身没有任何运行的含义，它只是一个静态的实体。而进程可以请求资源和调度，是一个动态的概念。
进程是计算机中的程序关于某数据集合上的一次运行活动，是系统进行资源分配和调度的基本单位，是操作系统结构的基础。
程序是指令、数据及其组织形式的描述，进程是程序的实体，一个程序可能有多个进程。
线程有时被称为轻量级进程，是程序执行流的最小单元。线程是程序中一个单一的顺序控制流程。进程内一个相对独立、可调度的执行单元，是系统独立调度和分派CPU的基本单位，也指运行中的程序的调度单位。
整个outlook应用程序代码是一个程序；打开一个outlook是一个进程，打开一个word是另一个进程；而发邮件是outlook进程的一个线程，收邮件又是另一个线程。

3.11.1. Process/进程

3.11.1.1. 是程序的实体

3.11.1.1.1. 程序是指令、数据及其组织形式的描述

3.11.1.2. 是计算机中的程序关于某数据集合上的一次运行活动

3.11.1.3. 是系统进行资源分配和调度的基本单位

3.11.1.4. 是操作系统结构的基础

3.11.1.5. 是线程的容器

3.11.1.6. 进程的特点

3.11.1.6.1. 独立性

3.11.1.6.2. 动态性

3.11.1.6.3. 并发性

3.11.1.7. 通过进程调起计算机、文本编辑器等

3.11.2. 线程

3.11.2.1. 是依附于进程而存在的，每一个线程必须有父进程

3.11.2.2. 线程拥有自己的堆栈、程序计数器和局部变量，线程和其他线程共享进程的系统资源

3.11.2.3. 进程不能共享内存，而线程之间可以轻松地共享内存

3.12. 多进程

3.12.1. 特点

3.12.1.1. 内存隔离，单个进程的异常不会导致整个应用的崩溃，方便调试

3.12.1.2. 进程间调用、通信和切换的开销大

3.12.1.3. 常使用在目标子功能间交互少的场景，弱相关性的、可扩展到多机分布（Nginx负载均衡）的场景

3.12.2. 进程的分类

3.12.2.1. 守护进程

3.12.2.1.1. Daemon Thread(守护线程)

3.12.2.1.1.1. 为其他线程的运行提供服务

GC线程

3.12.2.2. 非守护进程

3.12.2.2.1. User Thread(用户线程)

3.12.3. 创建进程的方式

3.12.3.1. 使用Runtime的exec(String cmdarray[])方法，在虚拟机实例中创建

3.12.3.1.1. 任何进程只会运行在一个虚拟机实例当中（底层源码采用 单例模式）

3.12.3.2. 使用ProcessBuilder的start()方法创建操作系统进程

3.13. 多线程

3.13.1. 多线程的意义

3.13.1.1. 发挥处理器最大性能

3.13.2. 创建线程的方式

3.13.2.1. 继承Thread类

3.13.2.2. 实现Runnable接口

3.13.2.2.1. 避免多继承局限

3.13.2.2.1.1. 可以更好的体现共享的概念

3.13.2.3. 实现Callable接口

3.13.2.4. 通过线程池启动多线程

3.13.2.5. runnable 和 callable 有什么区别

3.13.2.5.1. Runnable 接口 run 方法无返回值

3.13.2.5.1.1. 只能抛出运行时异常，且无法捕获处理

3.13.2.5.2. Callable 接口 call 方法有返回值，支持泛型

3.13.2.5.2.1. 允许抛出异常，可以获取异常信息

3.13.3. 如何进行信息交互

3.13.3.1. void notify()

3.13.3.1.1. 随机唤醒在此对象的等待池（监视器）上等待的单个线程，进入锁池

3.13.3.2. void notifyAll()

3.13.3.2.1. 唤醒在此对象的等待池（监视器）上等待的所有线程，进入锁池

3.13.3.3. void wait()

3.13.3.3.1. 导致当前的线程等待，直到其他线程调用此对象的notify()方法或notifyAll()方法

3.13.3.4. void wait(long timeout)

3.13.3.4.1. 同上+或者超过指定的时间量

3.13.3.5. void wait(long timeout, int nanos)

3.13.3.5.1. 同上+或者其他某个线程中断当前线程

3.13.3.6. sleep和wait的区别

3.13.3.6.1. sleep()方法是Thread类中方法

3.13.3.6.1.1. 线程不会释放对象锁

3.13.3.6.2. wait()方法是Object类中的方法

3.13.3.6.2.1. 线程会放弃对象锁，进入等待此对象的等待锁定池

3.13.4. ThreadLocal 线程本地存储

3.13.4.1. 每个线程可以访问自己内部 ThreadLocalMap 对象内的 value

3.13.4.1.1. 例如：每个线程分配一个 JDBC 连接 **Connection**

3.14. 线程池

3.14.1. 创建

3.14.1.1. newFixedThreadPool

3.14.1.1.1. 定长线程池，每当提交一个任务就创建一个线程，直到达到线程池的最大数量

3.14.1.2. newScheduledThreadPool

3.14.1.2.1. 定长线程池，可执行周期性的任务

3.14.1.2.1.1. 能够根据需要安排在给定延迟后运行命令或者定期地执行

3.14.1.3. newCachedThreadPool

3.14.1.3.1. 可缓存的线程池，如果线程池的容量超过了任务数，自动回收空闲线程

3.14.1.3.1.1. 任务增加时可以自动添加新线程，线程池的容量不限制

3.14.1.4. newSingleThreadExecutor

3.14.1.4.1. 单线程的线程池，线程异常结束，会创建一个新的线程，能确保任务按提交顺序执行

3.14.1.5. newSingleThreadScheduledExecutor

3.14.1.5.1. 单线程可执行周期性任务的线程池

3.14.1.6. newWorkStealingPool

3.14.1.6.1. 任务窃取线程池，不保证执行顺序，适合任务耗时差异较大

3.14.1.6.1.1. 默认创建的并行 level 是 CPU 的核数。主线程结束，即使线程池有任务也会立即停止

3.14.1.7. ForkJoinTask

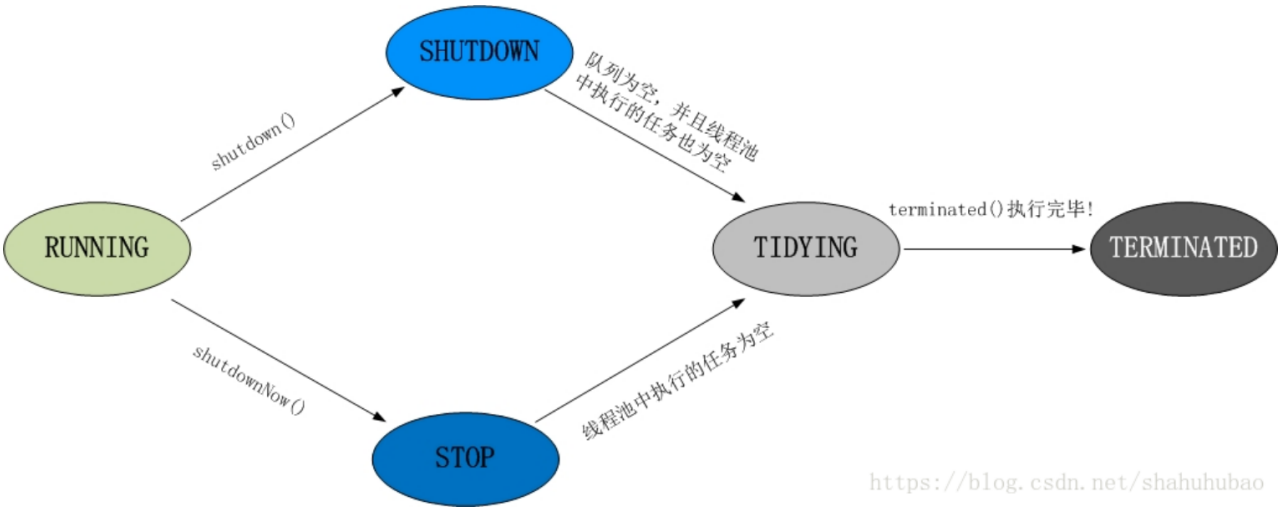
3.14.1.7.1. 解决 CPU 负载不均衡的问题

3.14.1.7.1.1. 使用分治法充分利用多核CPU

任务切分

结果合并

3.14.2. 状态



3.14.3. 方法

3.14.3.1. execute()

3.14.3.1.1. 参数 Runnable

3.14.3.1.1.1. 无返回值

3.14.3.2. submit()

3.14.3.2.1. 参数（Runnable）或（Runnable 和 结果 T）或 **(Callable)**

3.14.3.2.1.1. 返回值 Future 调用get方法时，可以捕获处理异常

4. Java异常

4.1. throwable类

4.1.1. Error类

4.1.1.1. 一般是指与虚拟机相关的问题

4.1.1.2. 这类错误的导致的中断，仅靠程序本身无法恢复，遇到这样的错误，建议让程序终止

4.1.1.3. NoClassDefFoundError

4.1.1.3.1. 发生在JVM在动态运行时，根据你提供的类名，在classpath中找不到对应的类进行加载

4.1.2. Exception类

4.1.2.1. 表示程序可以处理的异常，可以捕获且可能恢复

4.1.2.2. 遇到这类异常，应该尽可能处理异常，使程序恢复运行，而不应该随意终止异常

4.1.2.3. Unchecked Exception

4.1.2.3.1. 指的是程序的瑕疵或逻辑错误，并且在运行时无法恢复

4.1.2.3.2. RuntimeException

4.1.2.3.2.1. NullPointerException/空指针异常

在为空的对象中调用方法就会出现**NullPointerException**

4.1.2.3.2.2. IndexOutOfBoundsException 数组下标越界异常

4.1.2.3.2.3. NegativeArraySizeException 负数组长度异常

4.1.2.3.2.4. ArithmeticException 数学计算异常

4.1.2.3.2.5. ClassCastException 类型强制转换异常

4.1.2.3.2.6. SecurityException 违背安全原则异常

4.1.2.4. Checked Exception

4.1.2.4.1. 代表程序不能直接控制的无效外界情况

4.1.2.4.2. ClassNotFoundException

4.1.2.4.2.1. 编译的时候在classpath中找不到对应的类而发生的错误

4.1.2.5. NoSuchMethodException 方法未找到异常

4.1.2.6. IOException 输入输出异常

4.1.2.7. NumberFormatException 字符串转换为数字异常

4.1.2.8. EOFException 文件已结束异常

4.1.2.9. FileNotFoundException 文件未找到异常

4.1.2.10. SQLException 操作数据库异常

4.2. Throws

4.2.1. 作用在方法的声明上，表示如果抛出异常，则由该方法的调用者来进行异常处理

4.2.2. 方法会抛出会抛出某种类型的异常，让它的使用者知道捕获异常的类型

4.2.2.1. 出现异常是一种可能性，但不一定会发生异常

4.3. Throw

4.3.1. 真实抛出一个异常

4.4. try-catch-finally

4.4.1. catch 和 finally 都可以被省略，但是不能同时省略

4.4.2. finally 一定会执行，atch 中的 return 会等 finally 中的代码执行完之后，才会执行

5. Java Web和网络

5.1. Servlet

5.1.1. Jsp

5.1.1.1. 内置对象

5.1.1.1.1. 1 request javax.servlet.http.HttpServletRequest

5.1.1.1.1.1. 客户端的请求信息：Http协议头信息、Cookie、请求参数等

5.1.1.1.2. 2 response javax.servlet.http.HttpServletResponse

5.1.1.1.2.1. 用于服务端响应客户端请求，返回信息

5.1.1.1.3. 3 pageContext javax.servlet.jsp.PageContext

5.1.1.1.3.1. 页面的上下文

5.1.1.1.4. 4 session javax.servlet.http.HttpSession

5.1.1.1.4.1. 客户端与服务端之间的会话

5.1.1.1.5. 5 application javax.servlet.ServletContext

5.1.1.1.5.1. 用于获取服务端应用生命周期的信息

5.1.1.1.6. 6 out javax.servlet.jsp.JspWriter

5.1.1.1.6.1. 用于服务端传输内容到客户端的输出流

5.1.1.1.7. 7 config javax.servlet.ServletConfig

5.1.1.1.7.1. 初始化时，Jsp 引擎向 Jsp 页面传递的信息

5.1.1.1.8. 8 page java.lang.Object

5.1.1.1.8.1. 指向 Jsp 页面本身

5.1.1.1.9. 9 exception java.lang.Throwable

5.1.1.1.9.1. 页面发生异常，产生的异常对象

5.1.1.2. 作用域

5.1.1.2.1. page 当前页面作用域

5.1.1.2.1.1. 该作用域中存放的属性值，只能在当前页面中取出

5.1.1.2.2. request 请求作用域

5.1.1.2.2.1. 范围是从请求创建到请求消亡这段时间，一个请求可以涉及的多个页面

5.1.1.2.3. session 会话作用域

5.1.1.2.3.1. 范围是一段客户端和服务端持续连接的时间

用户在会话有效期内多次请求所涉及的页面

5.1.1.2.4. application 全局作用域

5.1.1.2.4.1. 范围是服务端Web应用启动到停止，整个Web应用中所有请求所涉及的页面

5.2. 客户端禁止 cookie，session 还能用吗

5.2.1. 一般默认情况下服务器存储 session 的 sessionid 是通过 cookie 存到浏览器里

5.2.1.1. session失效

5.2.2. 客户端禁止 cookie后，继续使用session方法

5.2.2.1. 通过url重写，把 sessionid 作为参数追加的原 url 中

5.2.2.2. 服务器的返回数据中包含 sessionid，浏览器发送请求时，携带 sessionid 参数

5.2.2.3. 通过 Http 协议其他 header 字段，服务器每次返回时设置该 header 字段信息

5.2.2.3.1. 通过 Http 协议其他 header 字段，服务器每次返回时设置该 header 字段信息

5.3. http 响应码

5.3.1. 301 Moved Permanently

5.3.1.1. 被请求的资源已永久移动到新位置

5.3.1.1.1. 并且将来任何对此资源的引用都应该使用本响应返回的若干个 URI 之一

5.3.2. 302 Found

5.3.2.1. 请求的资源现在临时从不同的 URI 响应请求

5.3.3. 404 Not Found

5.3.3.1. 请求失败，请求所希望得到的资源未被在服务器上发现

5.4. forward 和 redirect 的区别

5.4.1. 浏览器 url 地址显示不同

5.4.1.1. 服务端通过 forward 返回，浏览器 url 地址不会发生变化

5.4.1.1.1. 服务器通过 redirect 返回，浏览器会重新请求，url 地址会发生变化

5.4.2. 前后台两者页面跳转的处理方式不同

5.4.2.1. forward 跳转，是服务端进行页面跳转加载（include）新页面，直接返回到浏览器

5.4.2.1.1. redirect 跳转，是服务端返回新的 url 地址，浏览器二次发出 url 请求

5.4.3. 参数携带情况不一样

5.4.3.1. forward 跳转页面，会携带请求的参数到新的页面

5.4.3.1.1. redirect 跳转页面，属于一次全新的 http 请求，无法携带上一次请求的参数

5.4.4. http 请求次数不同

5.4.4.1. forward 1次；redirect 2次

5.5. TCP 传输控制协议

5.5.1. 面向连接，即必须在双方建立可靠连接之后，才会收发数据

5.5.1.1. 信息包头 20 个字节

5.5.2. 建立可靠连接需要经过3次握手

5.5.2.1. 断开连接需要经过4次挥手

5.5.2.1.1. 需要维护连接状态

5.5.3. 拥有流量控制及拥塞控制的机制

5.5.3.1. 报文头里面的确认序号、累计确认及超时重传机制能保证不丢包、不重复、按序到达

5.5.4. tcp 为什么要三次握手，两次不行吗

5.5.4.1. 两次握手只能保证单向连接是畅通的

5.5.4.2. 只有经过第三次握手，才能确保双向都可以接收到对方的发送的数据

5.5.5. tcp 粘包

5.5.5.1. 发送方发送的多个数据包，到接收方缓冲区首尾相连，粘成一包，被接收

5.5.5.2. 原因

5.5.5.2.1. TCP 协议默认使用 Nagle 算法可能会把多个数据包一次发送到接收方

5.5.5.2.1.1. 应用程读取缓存中的数据包的速度小于接收数据包的速度

缓存中的多个数据包会被应用程序当成一个包一次读

5.5.5.3. 处理方法

5.5.5.3.1. 发送方使用 TCP_NODELAY 选项来关闭 Nagle 算法

5.5.5.3.1.1. 数据包增加开始符和结束，应用程序读取、区分数据包

5.5.5.3.2. 在数据包的头部定义整个数据包的长度

5.5.5.3.2.1. 应用程序先读取数据包的长度，然后读取整个长度的包字节数据，保证读取的是单个包

5.6. UDP 用户数据报协议

5.6.1. 不建立可靠连接，无需维护连接状态

5.6.1.1. 信息包头 8 个字节

5.6.2. 接收端，UDP 把消息段放在队列中，应用程序从队列读消息

5.6.2.1. 不受拥挤控制算法的调节

5.6.3. 传送数据的速度受应用软件生成数据的速率、传输带宽、源端和终端主机性能的限制

5.6.3.1. 面向数据报，不保证接收端一定能收到

5.7. TCP和UDP的区别

5.7.1. TCP 面向连接；UDP 不建立可靠连接

5.7.1.1. TCP 信息包头 20 个字节；UDP 8 个字节

5.7.2. TCP 只能一对一的传输；UDP 支持一对一、一对多、多对一、多对多的传输

5.7.3. TCP 需要维护连接状态；UDP 无

5.7.3.1. TCP 拥有流量控制及拥塞控制的机制；UDP 不关注网络状况

5.7.4. TCP 通过流模式传输数据；UDP 通过数据报模式传输数据

5.7.4.1. TCP 保证数据正确性、不丢包、不重复、有序性；UDP 只最大交付，不保证可靠性

5.7.5. TCP 适合网络负担不大、可靠性要求高的场景

5.7.5.1. UDP 适合网络负担重响应高、客户端较多、可靠性要求不高的场景

5.8. OSI 的七层模型

OSI七层模型
功能
数据格式
对应的网络协议
TCP/IP四层概念模型
应用层
提供为应用软件而设的接口，以设置与另一应用软件之间的通信
数据ATPU
DNS、HTTP、FTP、IMAP4、POP3、SSH、TELNET…
应用层
表达层
把数据转换为能与接收者的系统格式兼容并适合传输的格式
数据PTPU

会话层
负责在数据传输中设置和维护计算机网络中两台计算机之间的通信连接
数据DTPU

传输层
把传输表头（TH）加至数据以形成数据包。传输表头包含了所使用的协议等发送信息
数据组织成数据段Segment
TCP、UDP、PPTP、TLS/SSL…
传输层
网络层
决定数据的路径选择和转寄，将网络表头（NH）加至数据包，以形成分组。网络表头包含了网络数据
分割和重新组合数据包Packet
IP（v4・v6）、ICMP（v6）、IGMP、Ipsec…
网络层
数据链路层
负责网络寻址、错误侦测和改错。当表头和表尾被加至数据包时，会形成帧。数据链表头（DLH）是包含了物理地址和错误侦测及改错的方法。数据链表尾（DLT）是一串指示数据包末端的字符串
将比特信息封装成数据帧Frame
Wi-Fi（IEEE 802.11）、ARP、WiMAX（IEEE 802.16）、PPP、PPPoE、L2TP…
数据链路层
物理层

在局部局域网上传送数据帧（data frame），它负责管理计算机通信设备和网络媒体之间的互通。包括了针脚、电压、线缆规范、集线器、中继器、网卡、主机接口卡等
传输比特（bit）流

5.9. get 和 post 请求的区别

5.9.1. get 长度有限制

5.9.1.1. 因为浏览器和 web 服务器限制了 URL 的长度

5.9.1.1.1. 超出了最大长度，大部分的服务器直接截断，有些服务器会报414错误

5.9.2. GET 是通过 URL 方式请求，可以直接看到，明文传输

5.9.2.1. POST 参数通过 header 传输，同样是明文，但不会显示在 URL 中

5.9.3. 浏览器会缓存和记录 GET 请求及参数，不缓存 POST 的请求的参数

5.10. 实现跨域请求

5.10.1. jsonp

5.10.1.1. 核心是动态添加

5.10.1.1.1. 只能 get 方式，易受到 XSS攻击

5.10.2. CORS 跨域资源共享

5.10.2.1. 忽略 cookie，浏览器版本有一定要求

5.10.3. 代理跨域请求

5.10.3.1. 前端向发送请求，经过额外的代理服务器，请求需要的服务器资源

5.10.4. Html5 postMessage 方法

5.10.4.1. 浏览器版本要求，部分浏览器要配置放开跨域限制

5.10.5. 修改 document.domain 跨子域

5.10.6. 基于 Html5 websocket 协议

5.11. 安全

5.11.1. SQL 注入

5.11.1.1. 程序没有有效过滤用户的输入，使攻击者成功的向服务器提交恶意的 SQL 脚本

5.11.2. XSS 攻击，即跨站脚本攻击

5.11.2.1. 攻击者往 web 页面里插入恶意的 HTML 代码（Javascript、css、html 标签等）

5.11.2.1.1. 当用户浏览该页面时，嵌入其中的 HTML 代码会被执行

####### 5.11.2.1.1.1. 如盗取用户 cookie 执行一系列操作，破坏页面结构、重定向到其他网站等

5.11.2.2. 对输入输出数据进行转义、过滤处理

5.11.2.2.1. 前端对 html 标签属性、css 属性赋值的地方进行校验

5.11.3. CSRF 跨站点请求伪造

避免方法：
CSRF 漏洞进行检测的工具，如 CSRFTester、CSRF Request Builder...
验证 HTTP Referer 字段
添加并验证 token
添加自定义 http 请求头
敏感操作添加验证码
使用 post 请求

5.11.3.1. CSRF 攻击者在用户已经登录目标网站之后，诱使用户访问一个攻击页面

5.11.3.1.1. 利用目标网站对用户的信任，以用户身份在攻击页对目标网站发起伪造用户操作的请求

6. 架构

6.1. 通信方式

6.1.1. WebServices

6.1.1.1. SOAP协议

6.1.1.1.1. 采用HTTP协议传送XML文件

6.1.1.1.1.1. 大数据大文件(GB级)传输

压缩该XML文件，然后上传到FTPServer上，服务端再下载解压缩然后读取

6.1.1.2. 应用

6.1.1.2.1. 跨越防火墙通信

6.1.1.2.1.1. 应用程序集成

软件复用

6.1.2. Socket

6.1.2.1. TCP/IP的传输层协议

6.1.2.1.1. 流传输，不支持面向对象

6.1.2.2. 应用

6.1.2.2.1. 高性能大数据的传输

6.2. SOA架构和微服务架构的区别

6.2.1. SOA 面向服务的架构

6.2.1.1. 一种设计方法，其中包含多个服务， 服务之间通过相互依赖最终提供一系列的功能

6.2.1.1.1. 一个服务 通常以独立的形式存在与操作系统进程中。各个服务之间 通过网络调用

6.2.2. 微服务架构

6.2.2.1. SOA 的升华

6.2.2.1.1. 单个业务系统会拆分为多个可以独立开发、设计、运行的小应用

6.2.2.1.1.1. 小应用之间通过服务完成交互和集成

6.2.2.2. = 80%的SOA服务架构思想 + 100%的组件化架构思想 + 80%的领域建模思想

6.2.3. 区别

6.2.3.1. SOA

6.2.3.1.1. 大块业务逻辑

6.2.3.1.1.1. 通常松耦合

任何类型架构

着重中央管理

确保应用能够交互操作

6.2.3.2. 微服务

6.2.3.2.1. 单独任务或小块业务逻辑

6.2.3.2.1.1. 总是松耦合

小型、专注于功能交叉团队

着重分散管理

执行新功能、快速拓展开发团队

6.3. SLA 服务等级协议

6.3.1. 在一定开销下为保障服务的性能和可用性

6.3.1.1. 3个9、4个9，即99.9%、99.99%

6.3.1.1.1. 9越多代表全年服务可用时间越长服务更可靠，停机时间越短

6.3.1.1.1.1. 1年 = 365天 = 8760小时

99.999 = 8760 * 0.00001 = 0.0876小时 = 0.0876 * 60 = 5.26分钟

6.4. 单点登录

6.4.1. JSONP实现，可跨域

6.4.1.1. 通过页面重定向实现安全

6.4.1.1.1. 使用独立登录系统，称为用户中心

7. 设计模式

7.1. 创建型

7.1.1. 工厂模式与抽象工厂模式 （Factory Pattern）

7.1.1.1. 简单工厂模式

定义一个用于创建对象的接口，让子类决定实例化哪一个类

使用参数或者配置文件等事先定义好的变量，然后利用分支判断初始化具体产品类并返回；不符合“开发-封闭”原则，每次增加产品，都需要修改类方法。工厂类单一，不用维护大量的工厂类；

7.1.1.1.1. 一个工厂类根据传入的参量决定创建出那一种产品类的实例

7.1.1.2. 抽象工厂模式

为创建一组相关或相互依赖的对象提供一个接口，而且无需指定他们的具体类
区别在于产品，如果产品单一，最合适用工厂模式，但是如果有多个业务品种、业务分类时，通过抽象工厂模式产生需要的对象是一种非常好的解决方式。再通俗深化理解下：工厂模式针对的是一个产品等

级结构 ，抽象工厂模式针对的是面向多个产品等级结构的。

7.1.1.2.1. 创建相关或依赖对象的家族，而无需明确指定具体类

7.1.1.3. 工厂方法模式

7.1.1.3.1. 定义一个创建对象的接口，让子类决定实例化那个类

7.1.2. 单例模式 （Singleton Pattern）

7.1.2.1. 某个类只能有一个实例，提供一个全局的访问点

7.1.3. 建造者模式 （Builder Pattern）

7.1.3.1. 封装一个复杂对象的构建过程，并可以按步骤构造

7.1.4. 原型模式 （Prototype Pattern）

7.1.4.1. 通过复制现有的实例来创建新的实例

7.2. 结构型

7.2.1. 适配器模式 （Adapter Pattern）

7.2.1.1. 将一个类的方法接口转换成客户希望的另外一个接口

7.2.2. 组合模式 （Composite Pattern）

7.2.2.1. 将对象组合成树形结构以表示“”部分-整体“”的层次结构

7.2.3. 装饰器模式 （Decorator Pattern）

7.2.3.1. 动态的给对象添加新的功能

7.2.4. 代理模式 （Proxy Pattern）

7.2.4.1. 为其他对象提供一个代理以便控制这个对象的访问

7.2.5. 享元模式 （Flyweight Pattern）

7.2.5.1. 通过共享技术来有效的支持大量细粒度的对象

7.2.6. 桥接模式 （Bridge Pattern）

7.2.6.1. 将抽象部分和它的实现部分分离，使它们都可以独立的变化

7.2.7. 外观模式 （Facade Pattern）

7.2.7.1. 对外提供一个统一的方法，来访问子系统中的一群接口

7.2.8. 过滤器模式 （Filter、Criteria Pattern）

7.2.8.1. 使用不同的标准来过滤一组对象，通过逻辑运算以解耦的方式把它们连接起来

7.2.8.1.1. 制定不同的规则来对一组对象进行过滤，然后对过滤结果进行分组

7.3. 行为型

7.3.1. 迭代器模式（Iterator Pattern）

7.3.1.1. 一种遍历访问聚合对象中各个元素的方法，不暴露该对象的内部结构

7.3.2. 观察者模式（Observer Pattern）

7.3.2.1. 对象间的一对多的依赖关系

7.3.3. 模板模式（Template Pattern）

7.3.3.1. 定义一个算法结构，而将一些步骤延迟到子类实现

7.3.4. 命令模式（Command Pattern）

7.3.4.1. 将命令请求封装为一个对象，使得可以用不同的请求来进行参数化

7.3.5. 解释器模式（Interpreter Pattern）

7.3.5.1. 给定一个语言，定义它的文法的一种表示，并定义一个解释器

7.3.6. 中介者模式（Mediator Pattern）

7.3.6.1. 用一个中介对象来封装一系列的对象交互

7.3.7. 策略模式（Strategy Pattern）

7.3.7.1. 定义一系列算法，把他们封装起来，并且使它们可以相互替换

7.3.8. 状态模式（State Pattern）

7.3.8.1. 允许一个对象在其对象内部状态改变时改变它的行为

7.3.9. 备忘录模式（Memento Pattern）

7.3.9.1. 在不破坏封装的前提下，保持对象的内部状态

7.3.10. 责任链模式（Chain of Responsibility Pattern）

7.3.10.1. 将请求的发送者和接收者解耦，使的多个对象都有处理这个请求的机会

7.3.11. 空对象模式（Null Object Pattern）

7.3.11.1. 通过实现一个默认的无意义对象来避免null值出现

7.3.11.1.1. 去掉对象null控制判断

8. 数据库

8.1. ACID

8.1.1. Atomicity 原子性：事务不可分割，不可约简

8.1.2. Consistency 一致性：事务开始前和事务结束后，数据库的完整性没有被破坏

8.1.3. Isolation 隔离性：数据库允许多个并发事务同时对其数

据进行读写和修改的能力

8.1.3.1. 可以防止多个事务并发执行时由于交叉执行而导致数据的不一致

8.1.4. Durability 持久性：事务处理结束后，对数据的修改永久化，即使系统故障也不会丢失

8.2. 隔离级别

为什么要有事务隔离级别，因为事务隔离级别越高，在并发下会产生的问题就越少，但同时付出的性能消耗也将越大，因此很多时候必须在并发性和性能之间做一个权衡。所以设立了几种事务隔离级别，以便让不同的项目可以根据自己项目的并发情况选择合适的事务隔离级别，对于在事务隔离级别之外会产生的并发问题，在代码中做补偿。

8.2.1. 读未提交

8.2.2. 读已提交

8.2.2.1. 脏读

8.2.2.1.1. 事务A读到了事务B还没有提交的数据

8.2.2.1.1.1. 读取未提交数据

8.2.3. 可重复读

8.2.3.1. 不可重复读

8.2.3.1.1. 在一个事务里面读取了两次某个数据，读出来的数据不一致

8.2.3.1.1.1. 前后多次读取，数据内容不一致

8.2.4. 串行化

8.2.4.1. 幻读

8.2.4.1.1. 在一个事务里面的操作中发现了未被操作的数据

8.2.4.1.1.1. 前后多次读取，数据总量不一致

8.3. 索引

索引是对数据库表中一列或多列的值进行排序的一种结构。

8.3.1. 概述

8.3.1.1. 索引是对一列或多列的值进行排序的一种结构，（B树，B+树）为了提高查询效率而引入，是独立于表的对象，存放在与表不同的表空间（TABLESPACE）中（整棵树存储在硬盘，读写时将某个节点加载到内存）。索引记录中存有索引关键字和指向表中数据的指针（地址）。

8.3.2. 索引的类型

8.3.2.1. FULLTEXT 全文索引

8.3.2.2. HASH 哈希索引

8.3.2.3. BTREE B树索引

8.3.2.4. B+TREE B+树索引

8.3.2.5. RTREE R树索引

8.3.3. 索引的种类

8.3.3.1. 普通索引

8.3.3.1.1. 加速查询

8.3.3.2. 唯一索引

8.3.3.2.1. 加速查询 + 列值唯一 + 可以为null

8.3.3.3. 主键索引

8.3.3.3.1. 加速查询 + 列值唯一 + 不可为null + 表中只有一个

8.3.3.3.1.1. 类似新华字典正文内容本身就是一种按照一定规则排列的目录

8.3.3.3.1.2. 每个表只能有一个聚集索引，因为目录只能按照一种方法进行排序 。

8.3.3.4. 组合索引

8.3.3.4.1. 多列值组成一个索引，专用于组合搜索，效率大于索引合并

8.3.3.5. 全文索引

8.3.3.5.1. 对文本的内容进行分词，进行搜索

8.3.3.6. 聚集索引

8.3.3.6.1. 叶子节点存的是整行数据，直接通过这个聚集索引的键值找到某行

8.3.3.6.2. 数据的物理存放顺序与索引顺序是一致，即：只要索引是相邻的，那么对应的数据一定也是相邻地存放在磁盘上

8.3.3.6.3. 数据行和相邻的键值紧凑地存储在一起，因为无法同时把数据行存放在两个不同的地方，所以一个表只能有一个聚集索引

8.3.3.7. 非聚集索引

8.3.3.7.1. 这种目录纯粹是目录，正文纯粹是正文的排序方式

8.3.3.7.2. 叶子节点存的是字段的值，通过这个非聚集索引的键值找到对应的聚集索引字段的值，再通过聚集索引键值找到表的某行，类似oracle通过键值找到rowid，再通过rowid找到行

8.3.4. 什么数据应该添加索引？

Where子句中经常使用的字段应该创建索引
分组字段或者排序字段应该创建索引
两个表的连接字段应该创建索引
存储空间固定的字段更适合选作索引的关键字 text类型的字段相比，

char类型更适合
占用存储空间少的字段更适合选作索引的关键字 与字符串相比，整数字段更适合
文章长数据索引应该考虑使用短索引，如果前面的字段多数都是不同的，那就指定一个前缀长度。

8.3.4.1. where后常用字段

8.3.4.1.1. 分组字段

8.3.4.1.1.1. 排序字段

表连接字段

8.3.5. 索引失效

8.3.5.1. 索引失效

8.3.5.1.1. or必全有索引

8.3.5.1.1.1. like只有%放右边才生效

有类型转换

有运算

有函数

数据少全表扫描更快

8.3.6. 怎么验证 mysql 的索引是否满足需求

8.3.6.1. 使用explain查看 SQL是如何执行查询语句的

8.3.6.1.1. explain select * from table where type = 1

8.4. 一张自增表里面总共有 7 条数据，删除了最后 2 条数据，重启 mysql 数据库，又插入了一条数据，此时 id 是几？

8.4.1. 表类型若是 MyISAM，则 id 为8

8.4.1.1. 不提供事务支持，也不支持行级锁和外键

8.4.1.2. 当执行插入和更新语句时，即执行写操作时需要锁定这个表，导致效率降低

8.4.1.3. 但它保存了表的行数，当进行那条语句时可以直接读取已经保存的值而不用扫描全表

8.4.1.3.1. 当表的读操作远大于写操作是，并且不需要事务支持的，可用MyISAM

8.4.2. 表类型若是 InnoDB，则 id 为6

8.4.2.1. mysql5.5默认InnoDB引擎，表只会把自增主键的最大id记录在内存中

8.4.2.1.1. 重启之后会导致最大id丢失

8.4.2.2. 提供了对数据 ACID 事务的支持，且提供行级锁和外键的约束

8.4.2.2.1. 设计目标就是处理大数据容量的数据库系统

8.4.2.3. InnoDB会在内存中建立缓冲池，用于缓冲数据和索引

8.4.2.3.1. 但是不支持全文搜索，且启动慢，不会保存表行数

8.4.2.3.1.1. 当进行 select count(*) from table 指令时，需进行扫描全表

8.4.2.4. 由于锁的粒度小，写操作不会锁定全表，所以在并发度较高的场景下使用会提升效率

8.5. InnoDB支持表锁和行锁，默认为行锁

8.5.1. 表级锁：开销小，加锁快，不会出现死锁

8.5.1.1. 锁定力度大，发生锁冲突概率高，并发量最低

8.5.2. 行级锁：开销大，加锁慢，会出现死锁

8.5.2.1. 锁力度小，发生锁冲突概率小，并发度最高

8.6. 乐观锁和悲观锁

8.6.1. 乐观锁：每次去拿数据都认为别人不会修改，所以不上锁

8.6.1.1. 但在提交更新时会判断在此期间是否有人更新数据

8.6.2. 悲观锁：每次去拿数据都认为别人修改了，所以每次都上锁

8.6.2.1. 这样别人拿回被阻止，直到此锁被释放

8.7. 获取当前数据库版本

8.7.1. 使用select version（）获取

8.8. char 和 varchar 的区别

8.8.1. char（n）：固定长度类型

8.8.1.1. 效率高但是占用空间

8.8.1.1.1. 适合存储密码的md5值，固定长度的

8.8.2. varchar（n）：长度可变

8.8.2.1. 存储值是每个值占用的字节加上一个用来记录其长度的字节的长度

8.8.2.1.1. 空间考虑用varchar，效率上用char

8.9. 内连接、左连接、右连接有什么区别

8.9.1. 内连接：inner join；把匹配的关联数据显示

8.9.2. 左连接：left join；把左边的表全显示，右边的表显示出符合条件的数据

8.9.3. 右连接：right join；与左连接相反

9. MYSQl优化

9.1. 概述

9.1.1. 设计数据库时：数据库表、字段的设计，存储引擎

9.1.2. 索引、存储过程

9.1.3. 横向扩展：MySQL集群、负载均衡、读写分离

9.1.4. SQL语句的优化

9.2. 字段设计

9.2.1. 尽量使用整型表示字符串

9.2.2. 尽可能选择小的数据类型和指定短的长度

9.2.3. 尽可能使用 not null

9.2.4. 单表字段不宜过多

9.2.5. 可以预留字段

9.2.6. 范式 Normal Format

9.2.6.1. 第一范式1NF：字段原子性

9.2.6.2. 第二范式：消除对主键的部分依赖

9.2.6.3. 第三范式：消除对主键的传递依赖

9.3. 索引

9.4. 查询缓存

9.4.1. windows上是my.ini，linux上是my.cnf

9.4.2. 0：不开启

9.4.3. 1：开启，默认缓存所有，SQL语句中增加select sql-no-cache来放弃缓存

9.4.4. 2：开启，默认都不缓存，SQL语句中增加select sql-cache来主动缓存（常用）

9.4.5. 缓存失效问题

9.4.5.1. 当数据表改动时，基于该数据表的任何缓存都会被删除

9.5. 分区

9.5.1. 只有检索字段为分区字段时，分区带来的效率提升才会比较明显

9.6. 水平分割和垂直分割

9.6.1. 水平分割：通过建立结构相同的几张表分别存储数据

9.6.2. 垂直分割：将经常一起使用的字段放在一个单独的表中，分割后的表记录之间是一一对应关系

9.7. 集群

横向扩展：从根本上（单机的硬件处理能力有限）提升数据库性能 。由此而生的相关技术：==读写分离、负载均衡==

9.7.1. 主从复制

9.7.2. 读写分离

9.7.3. 负载均衡

9.8. 典型SQL

9.8.1. 线上DDL

9.8.2. 数据库导入语句

9.8.3. 避免出现大页码的情况

9.8.4. select * 要少用

9.8.4.1. order by rand()不要用

9.8.5. 单表和多表查询

9.8.6. count(*)

9.9. 典型的服务器配置

9.9.1. max_connections，最大客户端连接数

9.9.1.1. show variables like 'max_connections'

9.9.2. table_open_cache，表文件句柄缓存

9.9.3. key_buffer_size，索引缓存大小

9.9.4. innodb_buffer_pool_size，Innodb存储引擎缓存池大小

9.9.5. innodb_file_per_table

9.10. 压测工具

9.10.1. mysqlslap

9.11. 问题排查

9.11.1. 使用 show processlist 命令查看当前所有连接信息

9.11.2. 使用 explain 命令查询SQL语句执行计划

9.11.3. 开启慢查询日志，查看慢查询的SQL

10. 时间复杂度

10.1. 数组 Array (T[])

10.1.1. 查询

10.1.1.1. O(n)

10.2. 链表 Linked list (LinkedList)

10.2.1. 查询

10.2.1.1. O(n)

10.3. hashmap

10.3.1. 查询

10.3.1.1. 元素均匀分布

10.3.1.1.1. O(1)

10.3.1.2. 哈希冲突，链表有 n 个元素

10.3.1.2.1. JDK1.8前

10.3.1.2.1.1. O(n)

10.3.1.2.2. JDK1.8后链表长度大于8时转换为红黑树

10.3.1.2.2.1. O(logn)

10.4. 排序

排序算法	平均时间复杂度	最好情况	最坏情况	空间复杂度	排序方式	稳定性
冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	In-place	不稳定
插入排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
希尔排序	$O(n \log n)$	$O(n \log^2 n)$	$O(n \log^2 n)$	$O(1)$	In-place	不稳定
归并排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Out-place	稳定
快速排序	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	In-place	不稳定
堆排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	In-place	不稳定
计数排序	$O(n + k)$	$O(n + k)$	$O(n + k)$	$O(k)$	Out-place	稳定
桶排序	$O(n + k)$	$O(n + k)$	$O(n^2)$	$O(n + k)$	Out-place	稳定
基数排序	$O(n \times k)$	$O(n \times k)$	$O(n \times k)$	$O(n + k)$	Out-place	稳定

10.5. Redis查询

10.5.1. O(1)

10.5.1.1. 大部分

10.5.2. O(N)

10.5.2.1. keys

10.5.2.1.1. hgetall hvals hkeys

10.5.2.1.1.1. hmget

hmset

11. Redis

11.1. 支持的数据类型

11.1.1. Redis采用Key-Value型的基本数据结构

11.1.2. String Key-Value

11.1.2.1. String是Redis的基础数据类型，Redis没有Int、Float、Boolean等数据类型的概念

11.1.3. Hash：key-filed-value

11.1.3.1. 相当于一个key 对应一个map

11.1.4. List是链表型的数据结构

11.1.5. Set是无序的，不可重复的String集合

11.1.6. Sorted Set是有序的、不可重复的String集合

11.2. 持久化（双开）

只打算用Redis 做缓存，可以关闭持久化
打算使用Redis 的持久化。建议RDB和AOF都开启。其实RDB更适合做数据的备份，留一后手。AOF出问题了，还有RDB。

11.2.1. 默认RDB指定的时间间隔内，执行指定次数的写操作，则将数据写入到磁盘中

11.2.1.1. 适合大规模的数据恢复，数据一致性和完整性较差

11.2.2. AOF每秒将写操作日志追加到AOF文件中

11.2.2.1. AOF文件大，数据完整性比RDB高

11.3. Redis和Memcache区别，优缺点对比

11.3.1. Memcached 是一个高性能的分布式内存对象缓存系统

11.3.1.1. 一个单一key-value内存Cache

11.3.1.1.1. 用于动态Web应用以减轻数据库负载，可缓存图片、视频等

11.3.2. Redis则是一个数据结构内存数据库

11.3.2.1. 支持数据持久化和数据恢复，允许单点故障

11.3.2.1.1. 可以在服务器端直接对数据进行丰富的操作，减少网络IO次数和数据体积

11.4. redis 为什么是单线程的

11.4.1. 因为Redis是基于内存的操作，CPU不是Redis的瓶颈

11.4.1.1. 避免了不必要的上下文切换和竞争条件，也不存在多进程或多线程切换而消耗 CPU

11.4.1.1.1. 使用多路I/O复用模型，非阻塞IO

11.4.1.1.1.1. Redis直接自己构建了VM 机制

11.5. 什么是缓存穿透？怎么解决

11.5.1. 缓存穿透，是指查询一个数据库一定不存在的数据

11.5.1.1. 解决方案

11.5.1.1.1. 对key的规范进行检测，拦截恶意攻击

11.5.1.1.2. 从数据库查询的对象为空，也放入缓存，设定较短的缓存过期时间，比如设置为60秒

11.5.2. 缓存雪崩，是指在某一个时间段，缓存集中过期失效

11.5.2.1. 不同分类商品，缓存不同周期。在同一分类中的商品，加上一个随机因子

11.5.3. 缓存击穿，是指一个key非常热点，大并发集中对这一个点进行访问

11.5.3.1. 当这个key在失效的瞬间，持续的大并发就穿破缓存，直接请求数据库

11.5.3.1.1. 可让缓存永不过期

11.6. Redis的Java客户端

11.6.1. Lettuce

11.6.1.1. springboot默认使用的客户端

11.6.1.1.1. 基于Netty框架的事件驱动的通信层，其方法调用是异步的

11.6.1.1.1.1. 线程安全同步，异步和响应使用，支持集群，Sentinel，管道和编码器

11.6.2. Redisson

11.6.2.1. 基于Netty实现，采用非阻塞IO，性能高

11.6.2.1.1. 支持异步请求

11.6.2.1.1.1. 支持连接池

不支持事务

支持读写分离，支持读负载均衡

可以与Spring Session集成，实现基于Redis的会话共享

11.6.3. Jedis

11.6.3.1. 轻量，简洁，便于集成和改造

11.6.3.1.1. 支持连接池

11.6.3.1.1.1. 支持pipelining、事务、LUA Scripting、Redis Sentinel、Redis Cluste

不支持读写分离，需要自己实现

11.7. 如何保证Redis与数据库的一致性

11.7.1. 更新的时候，先删除缓存，然后再更新数据库

11.7.1.1. 读的时候，先读缓存；如果没有的话，就读数据库，同时将数据放入缓存，并返回响应

11.7.1.1.1. 特别情况设置缓存失效时间

11.8. redis 怎么实现分布式锁

11.8.1. SET key value [EX seconds] [PX milliseconds] [NX|XX]

11.8.1.1. EX second :设置键的过期时间为second秒

11.8.1.1.1. NX :只在键不存在时,才对键进行设置操作

11.8.1.1.1.1. SET操作成功完成时,返回OK ,否则返回null

11.8.1.2. PX millisecond :设置键的过期时间为millisecond毫秒

11.8.1.2.1. XX:只在键已经存在时,才对键进行设置操作

11.8.2. 仅在单实例的场景下是安全的

11.8.2.1. 多个Client可能同时获取到了锁

11.8.2.1.1. 异步获取，节点异常

11.9. Redis的内存优化

11.9.1. 存储编码的优化

11.9.1.1. Redis存储的数据都使用redisObject结构体来封装

11.9.2. 共享对象池

11.9.2.1. 指Redis内部维护了[0-9999]的整数对象池，用于节约内存

11.9.2.1.1. 除了整数值对象，其它类型如list、hash、set和zset内部元素也可以使用整数对象池

11.9.3. 字符串优化

11.9.3.1. 控制键的数量，使用hash代替多个key value

11.9.3.1.1. 缩减键值对象，键值越短越好

11.10. 淘汰策略有哪些

11.10.1. volatile-lru:从设置了过期时间的数据集中

11.10.1.1. 选择最近最久未使用的数据释放

11.10.2. allkeys-lru:从数据集中(包括设置过期时间以及未设置过期时间的数据集中)

11.10.2.1. 选择最近最久未使用的数据释放

11.10.3. volatile-random:从设置了过期时间的数据集中

11.10.3.1. 随机选择一个数据进行释放

11.10.4. allkeys-random:从数据集中(包括了设置过期时间以及未设置过期时间)

11.10.4.1. 随机选择一个数据进行入释放

11.10.5. volatile-ttl：从设置了过期时间的数据集中

11.10.5.1. 选择马上就要过期的数据进行释放操作

11.10.6. noeviction：不删除任意数据(但redis还会根据引用计数器进行释放)

11.10.6.1. 这时如果内存不够时，会直接返回错误

11.11. Redis常见的性能问题

11.11.1. 避免master写内存快照

11.11.1.1. master AOF持久化，AOF文件过大会影响master重启时的恢复速度

11.11.1.1.1. 避免master调用BGREWRITEAOF重写AOF文件，出现短暂的服务暂停现象

11.11.2. redis主从复制的性能问题

11.11.2.1. 为了主从复制的速度和连接的稳定性，slave和master最好在同一个局域网内

12. 框架

12.1. Spring

12.1.1. bean生命周期

12.1.1.1. 实例化bean对象(通过构造方法或者工厂方法)

12.1.1.2. 设置对象属性(setter等)（依赖注入）

12.1.1.3. 如果Bean实现了BeanNameAware接口

12.1.1.3.1. 工厂调用Bean的setBeanName()方法传递Bean的ID

12.1.1.4. 如果Bean实现了BeanFactoryAware接口

12.1.1.4.1. 工厂调用setBeanFactory()方法传入工厂自身

12.1.1.5. 将Bean实例传递给Bean的前置处理器的postProcessBeforeInitialization方法

12.1.1.6. 调用Bean的初始化方法

12.1.1.7. 将Bean实例传递给Bean的后置处理器的postProcessAfterInitialization方法

12.1.1.8. 使用Bean

12.1.1.9. 容器关闭之前，调用Bean的销毁方法

12.1.2. bean作用域

12.1.2.1. singleton：单例模式

12.1.2.1.1. 整个Spring IoC容器中，使用 singleton 定义的 bean 只有一个实例

12.1.2.2. prototype：原型模式

12.1.2.2.1. 每次通过容器的getbean获取 prototype 定义的 bean 时，都产生一个新的 bean 实例

12.1.2.3. request

12.1.2.3.1. 每次 HTTP 请求将会产生不同的 bean 实例

12.1.2.4. session

12.1.2.4.1. 同一个 Session 共享一个 bean 实例

12.1.2.5. global-session

12.1.2.5.1. 同 session 作用域不同的是，所有的Session共享一个Bean实例

12.1.3. Shiro权限框架

12.1.3.1. 用户身份认证

12.1.3.2. 授权/访问控制

12.1.3.2.1. @RequiresPermissions("task:Vehicle:view")

12.1.3.2.1.1. 实例级访问控制

12.1.3.2.1.2. 权限字符串规则

12.1.3.3. 权限管理

12.1.3.3.1. 权限分配

12.1.3.3.2. 权限控制

12.1.3.3.2.1. 基于角色的访问控制

粗颗粒度

细颗粒度

12.1.3.3.2.2. 基于url拦截

12.1.4. aop

12.1.4.1. spring事务

12.1.4.1.1. 七种事务传播行为

PROPAGATION_REQUIRED
如果当前没有事务，就新建一个事务，如果已经存在一个事务中，加入到这个事务中。这是最常见的选择。
PROPAGATION_SUPPORTS
支持当前事务，如果当前没有事务，就以非事务方式执行。
PROPAGATION_MANDATORY
使用当前的事务，如果当前没有事务，就抛出异常。
PROPAGATION_REQUIRES_NEW
新建事务，如果当前存在事务，把当前事务挂起。
PROPAGATION_NOT_SUPPORTED
以非事务方式执行操作，如果当前存在事务，就把当前事务挂起。
PROPAGATION_NEVER
以非事务方式执行，如果当前存在事务，则抛出异常。
PROPAGATION_NESTED
如果当前存在事务，则在嵌套事务内执行。如果当前没有事务，则执行与PROPAGATION_REQUIRED类似的操作。

12.1.4.2. 通知类型

12.1.4.2.1. before 前置通知 在目标方法调用之前执行

12.1.4.2.2. after-returning后置通知 在目标方法调用之后执行，一旦目标方法产生异常不会执行

12.1.4.2.3. after 最终通知 在目标调用方法之后执行，无论目标方法是否产生异常，都会执行

12.1.4.2.4. after-throwing 异常通知 在目标方法产生异常时执行

12.1.4.2.5. around 环绕通知 在目标方法执行之前和执行之后都会执行

12.1.4.3. 实现方式

12.1.4.3.1. JDK动态代理

12.1.4.3.1.1. 当Bean实现接口时，Spring就会用JDK的动态代理

12.1.4.3.2. CGlib动态代理

12.1.4.3.2.1. 当Bean没有实现接口时，Spring使用CGlib实现

修改字节码

12.1.5. 注解+xml配置bean会冲突吗

12.1.5.1. 当出现两个相同名称实例，spring会覆盖其中一个，xml优先级高于注解

12.1.5.2. xml中同时配置两个相同id的bean，直接校验不通过报错

12.1.6. 自动化装配bean

12.1.6.1. 组件扫描

12.1.6.1.1. 自动装配

12.1.6.1.1.1. 第三方的类库无法自动装配

13. 手写代码

13.1. 懒汉式—静态内部类实现单例

```
public class Sg {
   private static class Sgh {
   private static final Sg ob = new Sg();
   }
   private Sg (){}
   public static final Sg getob() {
   return Sgh.ob;
   }
}
```

13.1.1. 饿汉式—静态常量方式（线程安全）

```
public class Singleton {
    private static Singleton instance = new Singleton();
    private Singleton (){}
    public static Singleton getInstance() {
    return instance;
    }
}
```

13.2. 将字符串转int类型

```
public static int parseInt(String s, int radix)
            throws NumberFormatException
  {

      if (s == null) {
          throw new NumberFormatException("null");
      }

      if (radix < Character.MIN_RADIX) {
          throw new NumberFormatException("radix " + radix +
                                          " less than Character.MIN_RADIX");
      }

      if (radix > Character.MAX_RADIX) {
          throw new NumberFormatException("radix " + radix +
                                          " greater than Character.MAX_RADIX");
      }

      int result = 0;
      boolean negative = false;
```

```
    int i = 0, len = s.length();
    int limit = -Integer.MAX_VALUE;
    int multmin;
    int digit;

    if (len > 0) {
        char firstChar = s.charAt(0);
        if (firstChar < '0') { // Possible leading "+" or "-"
            if (firstChar == '-') {
                negative = true;
                limit = Integer.MIN_VALUE;
            } else if (firstChar != '+')
                throw NumberFormatException.forInputString(s);

            if (len == 1) // Cannot have lone "+" or "-"
                throw NumberFormatException.forInputString(s);
            i++;
        }
        multmin = limit / radix;
        while (i < len) {
            // Accumulating negatively avoids surprises near MAX_VALUE
            digit = Character.digit(s.charAt(i++),radix);
            if (digit < 0) {
                throw NumberFormatException.forInputString(s);
            }
            if (result < multmin) {
                throw NumberFormatException.forInputString(s);
            }
            result *= radix;
            if (result < limit + digit) {
                throw NumberFormatException.forInputString(s);
            }
            result -= digit;
        }
    } else {
        throw NumberFormatException.forInputString(s);
    }
    return negative ? result : -result;
}
```

13.2.1. number = number*10 + str[i] -'0';

13.3. 因式分解

```
public static ArrayList<Integer> Factorization(int n) {
    ArrayList<Integer> list = new ArrayList<Integer>();
    for (int i = 2; i <= Math.sqrt(n); i++) {
        if (n % i == 0) {
            list.add(i);
            if (n / i != i) {
                list.add(n / i);
            }
        }
    }
    return list;
}
```

13.3.1. GCD最大公约数求解

```
public static GetGCD(int p, int q){
    if(q == 0)
        return p;
    int r = p % q;
    return GetGCD(q, r);//欧几里得算法digui
}
```

13.3.1.1. 大数加法

```
public static void main(String[] args) {

  String a="8888899999999888";
  String b="88888888888888";
  String str=new BigInteger(a).add(new BigInteger(b)).toString();
  System.out.println(str);

 }

/**
     * 用字符串模拟两个大数相加
     * @param n1 加数1
     * @param n2 加数2
     * @return   相加结果
     */
    public static String add2(String n1,String n2)
    {
```

```
        StringBuffer result = new StringBuffer();

        //1、反转字符串
        n1 = new StringBuffer(n1).reverse().toString();
        n2 = new StringBuffer(n2).reverse().toString();

        int len1 = n1.length();
        int len2 = n1.length();
        int maxLen = len1 > len2 ? len1 : len2;
        boolean nOverFlow = false; //是否越界
        int nTakeOver = 0 ; //溢出数量

        //2.把两个字符串补齐，即短字符串的高位用0补齐
        if(len1 < len2)
        {
           for(int i = len1 ; i < len2 ; i++)
           {
           n1 += "0";
           }
        }
        else if (len1 > len2)
        {
           for(int i = len2 ; i < len1 ; i++)
           {
           n2 += "0";
           }
        }

        //3.把两个正整数相加，一位一位的加并加上进位
        for(int i = 0 ; i < maxLen ; i++)
        {
           int nSum = Integer.parseInt(n1.charAt(i) +"") +
Integer.parseInt(n2.charAt(i) +"");

           if(nSum >= 10)
           {
           if(i == (maxLen - 1))
           {
              nOverFlow  = true;
           }
           nTakeOver = 1;
           result.append(nSum - 10);
           }
           else
           {
           nTakeOver = 0;
```

13.3.1.1.1. 大数乘法法

```
mport java.util.Scanner;

/**
 * 大数相乘
 * @author Ant
 *
 */
public class BigMultiply {

    /**
     * 大数相乘基本思想，输入字符串，转成char数组，转成int数组。采用分治思想，每一位的相乘;<br>
     * 公式：AB*CD = AC (BC+AD) BD , 然后从后到前满十进位(BD, (BC+AD),AC)。
     * @param num1
     * @param num2
     */
    public String multiply(String num1, String num2){
        //把字符串转换成char数组
        char chars1[] = num1.toCharArray();
        char chars2[] = num2.toCharArray();

        //声明存放结果和两个乘积的容器
        int result[] = new int[chars1.length + chars2.length];
        int n1[] = new int[chars1.length];
        int n2[] = new int[chars2.length];

        //把char转换成int数组，为什么要减去一个'0'呢？因为要减去0的ascii码得到的就是实际的数字
        for(int i = 0; i < chars1.length;i++)
            n1[i] = chars1[i]-'0';
        for(int i = 0; i < chars2.length;i++)
            n2[i] = chars2[i]-'0';

        //逐个相乘，因为你会发现。AB*CD = AC(BC+AD)BD , 然后进位。
        for(int i =0 ; i < chars1.length; i++){
            for(int j =0; j < chars2.length; j++){
                result[i+j]+=n1[i]*n2[j];
            }
        }
```

```
            //满10进位，从后往前满十进位
            for(int i =result.length-1; i > 0 ;i--){
                result[i-1] += result[i] / 10;
                result[i] = result[i] % 10;
            }

            //转成string并返回
            String resultStr = "";
            for(int i = 0; i < result.length-1; i++){
                resultStr+=""+result[i];
            }
            return resultStr;
        }

        public static void main(String[] args) {
            BigMultiply bm = new BigMultiply();
            System.out.println("-----输入两个大数------");
            Scanner scanner = new Scanner(System.in);
            String num1 = scanner.next();
            String num2 = scanner.next();
            String result = bm.multiply(num1, num2);
            System.out.println("相乘结果为："+result);
            scanner.close();
        }
    }
```

13.4. 二分查找

13.4.1. while循环，非递归

```
    public static int BinarySearch(int[] arr,int key){
        int low = 0;
        int high = arr.length - 1;
        int middle = 0;

        if(key < arr[low] || key > arr[high] || low > high){
            return -1;
        }

        while(low <= high){
            middle = (low + high) / 2;
```

```
            if(arr[middle] > key){
                high = middle - 1;
            }else if(arr[middle] < key){
                low = middle + 1;
            }else{
                return middle;
            }
        }

        return -1;□□//not found, return -1.
    }
```

13.4.1.1. 递归

```
    public static int recursionBinarySearch(int[] arr,int key,int low,int
high){
        // out of range, not found
        if(key < arr[low] || key > arr[high] || low > high){
            return -1;
        }

        int mid = (low + high) / 2;
        if(arr[mid] > key){
            return recursionBinarySearch(arr, key, low, mid - 1);
        }else if(arr[mid] < key){
            return recursionBinarySearch(arr, key, mid + 1, high);
        }else {
            return mid;
        }

    }
```

13.5. 生兔子

13.5.1. 斐波那契数列

13.5.1.1. 从第3项开始，每一项都等于前两项之和

13.5.1.1.1. 递归

```
public class Rabbit {

    public static void main(String[] args) {
```

```
    System.out.println("请输入月份");
    Scanner sc = new Scanner(System.in);
    int m = sc.nextInt();
    for (int i = 1; i <= m; i++) {
      System.out.println("第" + i + "个月的兔子为：" + f(i));
    }
  }

  public static int f(int x) {
    if (x == 1 || x == 2)
      return 1;
    else
      return f(x - 1) + f(x - 2);
  }
}
```

13.5.1.1.1.1. 非递归

```
public class Rabbit1 {
  public static void main(String[] args) {
    int  f1 = 1;
    int  f2=1;
    System.out.println("请输入月份");
    Scanner sc=new Scanner(System.in);
    int m=sc.nextInt();
     int temp = 0;
    for (int i = 3; i <=m; i++) {
      if (m==1||m==2)
        f2=1;
      else
        temp=f2;
        f2=f1+f2;
        f1=temp;
    }
    System.out.println(f2);
  }
}
```

13.6. 冒泡排序

13.6.1. 基本实现

```
public void BubbleSort(int arr[],int n){
□int temp;
□//有N个数，所以要跑N趟
□for(int i = 0;i<n;++i)
□□//每一趟比较从右往左，得到的是最小值
□□//每一趟比较从左往右，得到的是最大值
□□for(int j = n-1;j>i;--j)
□□□//如果右边的数《左边的，那就
交换位置
□□□if(arr[j]<arr[j-1]){
□□□□temp = arr[j];
□□□□arr[j] = arr[j-1];
□□□□arr[j-1] = temp;
□□□}
}
```

13.6.1.1. 加标志改进

```
public static void bubbleSort2(int [] a, int n){
    int j, k = n;
    //发生了交换就为true, 没发生就为false
    boolean flag = true;
    while (flag){
        flag=false;//每次开始排序前，都设置flag为未排序过
        for(j=1; j<k; j++){
            //前面的数字大于后面的数字就交换
            if(a[j-1] > a[j]){
                int temp;
                temp = a[j-1];
                a[j-1] = a[j];
                a[j]=temp;
                //表示交换过数据;
                flag = true;
            }
        }
        k--;
    }
}
```

13.6.1.1.1. 标志有具体位置

```
public static void bubbleSort3(int [] a, int n){
    int j , k;
```

```
            int flag = n ;
            while (flag > 0){
                k = flag; //k 来记录遍历的尾边界
                flag = 0;
                for(j=1; j<k; j++){
                    if(a[j-1] > a[j]){
                        int temp;
                        temp = a[j-1];
                        a[j-1] = a[j];
                        a[j]=temp;
                        //表示交换过数据;
                        flag = j;
                    }
                }
            }
        }
```

13. 7. 人民币转大写

```
package com.test;

import java.math.BigDecimal;

/**
 *
 *
 * 数字转换为汉语中人民币的大写<br>
 *
 */
public class NumberToCN {
    /**
     * 汉语中数字大写
     */
    private static final String[] CN_UPPER_NUMBER = { "零", "壹", "贰", "叁",
"肆",
            "伍", "陆", "柒", "捌", "玖" };
    /**
     * 汉语中货币单位大写，这样的设计类似于占位符
     */
    private static final String[] CN_UPPER_MONETRAY_UNIT = { "分", "角",
"元",
            "拾", "佰", "仟", "万", "拾", "佰", "仟", "亿", "拾", "佰", "仟", "兆", "拾",
```

```
            "佰", "仟" };
    /**
     * 特殊字符：整
     */
    private static final String CN_FULL = "整";
    /**
     * 特殊字符：负
     */
    private static final String CN_NEGATIVE = "负";
    /**
     * 金额的精度，默认值为2
     */
    private static final int MONEY_PRECISION = 2;
    /**
     * 特殊字符：零元整
     */
    private static final String CN_ZEOR_FULL = "零元" + CN_FULL;

    /**
     * 把输入的金额转换为汉语中人民币的大写
     *
     * @param numberOfMoney
     *            输入的金额
     * @return 对应的汉语大写
     */
    public static String number2CNMontrayUnit(BigDecimal
numberOfMoney) {
        StringBuffer sb = new StringBuffer();
        // -1, 0, or 1 as the value of this BigDecimal is negative, zero, or
        // positive.
        int signum = numberOfMoney.signum();
        // 零元整的情况
        if (signum == 0) {
            return CN_ZEOR_FULL;
        }
        // 这里会进行金额的四舍五入
        long number =
numberOfMoney.movePointRight(MONEY_PRECISION)
                .setScale(0, 4).abs().longValue();
        // 得到小数点后两位值
        long scale = number % 100;
        int numUnit = 0;
        int numIndex = 0;
        boolean getZero = false;
        // 判断最后两位数，一共有四中情况：00 = 0, 01 = 1, 10, 11
```

13.8. 第一个只出现一次的字符

```
public class Solution {
    public int FirstNotRepeatingChar(String str) {
        int[] words = new int[58];
        for(int i = 0;i<str.length();i++){
           words[((int)str.charAt(i))-65] += 1;
        }
        for(int i=0;i<str.length();i++){
            if(words[((int)str.charAt(i))-65]==1)
                return i;
        }
        return -1;
    }
}
```

13.9. 链表是否有环路

```
public boolean hasCycle(ListNode h) {
//声明两个节点从头开始遍历节点
ListNode quick = h;
ListNode slow = h;
//当快指针能够走到头表示无环
while(quick!=null&&quick.next!=null){
quick = quick.next.next;
slow = slow.next;
if(quick==slow){
return true;
}
}
return false;
}
```

13.9.1. 找出该链表的环的入口结点

```
    public ListNode EntryNodeOfLoop(ListNode pHead)
        {
            if(pHead == null || pHead.next == null)
                return null;
            HashSet<ListNode> set = new HashSet<ListNode>();
            for(;pHead != null;pHead = pHead.next){
                if(!set.add(pHead)){
```

```
            return pHead;
        }
    }
    return null;
}
```

13.9.1.1. 综合

```
public T getLinkCircleVal(){
Entry slow = this.head.next;
Entry fast = this.head.next;
// 使用快慢指针解决该问题
    while(fast != null && fast.next != null){
        slow = slow.next;
        fast = fast.next.next;
        if(slow == fast){
            break;
        }
    }

    if(fast == null){
        return null;
    } else {
                // fast从第一个节点开始走，slow从快慢指针相交的地方开始
走，它们相遇的时候，就是环的入口节点
        fast = this.head.next;
        while(fast != slow){
            fast = fast.next;
            slow = slow.next;
        }
        return slow.data;
    }
}
```

14. 手写SQL

14.1. sql比较两个相同结构表的不同记录

14.1.1. select * from t2 where not exists(select * from t1 where t1.pid = t2.pid);

14.2. 外连接查询(表名1 ： a表名2 ： b)

14.2.1. select a.a, a.b, a.c, b.c, b.d, b.f from a LEFT OUTER JOIN b ON a.a = b.c

14.3. 分组排序查最大值

14.3.1. select * from gt where grade in(select max(grade) from gt group by subject);

14.4. 查询出每门课都大于80 分的学生姓名

14.4.1. select distinct na from t where na not in (select distinct na from t where fs

14.4.1.1. select name from table group by name having min(fenshu)>80

14.5. 删除除了自动编号不同，其他都相同的学生冗余信息

14.5.1. delete tb where id not in(select min(id) from tb

group by 学号，姓名，其他）

14.6. team 表，里面只有一个字段name，一共有4 条纪录，分别是a,b,c,d，对应四个球队

14.6.1. 现在四个球队进行比赛，用一条sql 语句显示所有可能的比赛组合

14.6.1.1. select a.name, b.name from team a, team b where a.name

14.7. 复制表(只复制结构，源表名：a新表名：b）

14.7.1. select * into b from a where 11

14.7.1.1. select top 0 * into b from a

14.7.1.1.1. DDL（Data Definition Language）：数据定义语言，操作数据库对象：库、表、列等

14.7.1.1.2. DML（Data Manipulation Language）：数据操作语言，增删改数据库中的数据；

14.7.1.1.3. DCL（Data Control Language）：数据控制语言，用来设置访问权限和安全级别；

14.7.1.1.4. DQL（Data Query Language）：数据查询语言，用来查询数据库中的数据。

14.8. 拷贝表(拷贝数据, 源表名：a目标表名：b)

14.8.1. insert into b(a, b, c) select d,e,f from a;

14.9. 显示文章、提交人和最后回复时间

14.9.1. select a.title,a.username,b.adddate from table a, (select max(adddate) adddate from table where table.title=a.title) b

14.10. 日程安排提前五分钟提醒

14.10.1. select * from 日程安排 where datediff('minute',f 开始时间,getdate())>5

14.11. 两张关联表，删除主表中已经在副表中没有的信息

14.11.1. Delete from zb where not exists (select * from fb where zb.infid=fb.infid)

14.12. 有两个表A 和B ，均有key 和value 两个字段

14.12.1. 如果B 的key 在A 中也有，就把B 的value 换为A 中对应的value

14.12.1.1. update b set b.value=(select a.value from a where a.key=b.key) where b.id in(select b.id from b,a where b.key=a.key);

14.13. 修饰查询结果，显示成绩是否合格

14.13.1. 使用decode函数比较大小

14.13.1.1. select name ,score
,decode（sign(score-60),-1,'fail','pass'）as mark from **course**