# 抽象（abstract）

不能是静态的（static），不能是本地方法（native），不能被synchronized修饰。

抽象方法需要子类重写，而静态无法被重写，相互矛盾。

本地方法是实现的方法，抽象方法时没有实现的。互相矛盾。

synchronized和方法的实现有关，抽象方法不涉及实现细节。互相矛盾。

# 静态变量和实例变量的区别

静态变量称为类变量，它属于类，不属于类的任何一个对象，一个类不管创建多少个对象，静态变量在内存中有且仅有一个拷贝。

实例变量，必须依赖于某一实例，需要先创建对象后通过对象访问到它。静态变量可以实现让多个对象共享内存。

# ==与equals

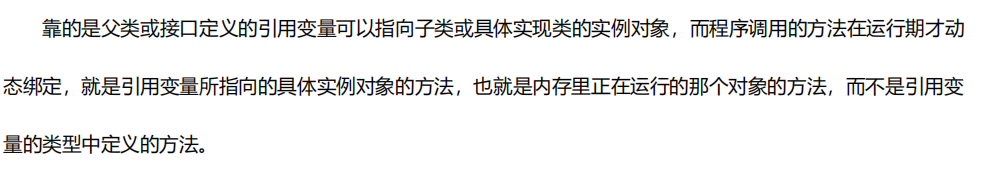
==：如果是基本数据类型，比较的是数值是否相等，如果是引用数据类型，则比较的是对象的地址值是否相同。

Equals：比较的是内容是否相同。

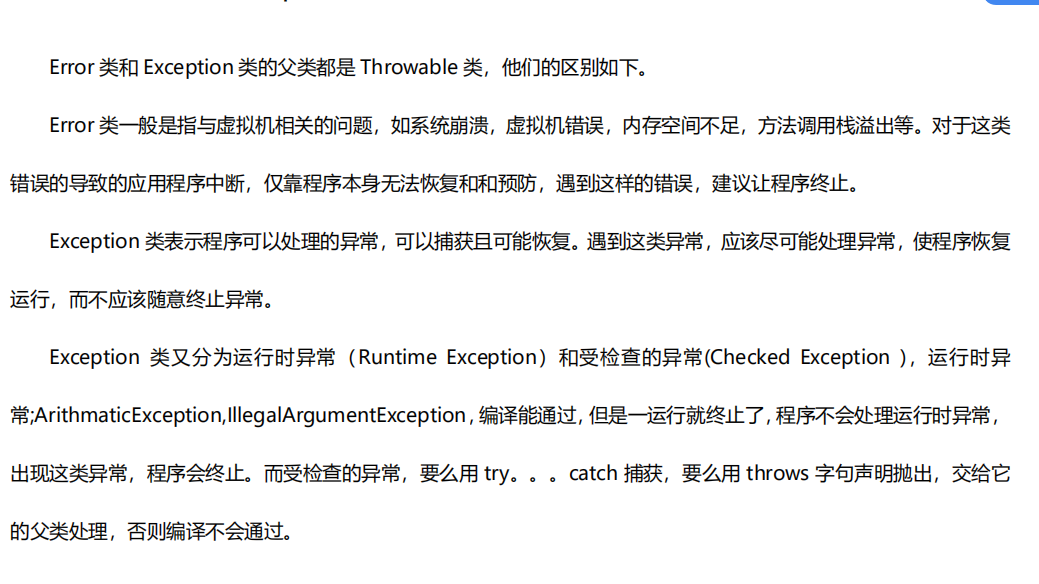
# String

String类型的数据存放在内存中。String对象建立之后不能改变，所以对于每一个不同的字符串，都需要一个String对象表示。这时需要考虑使用StringBuffer。

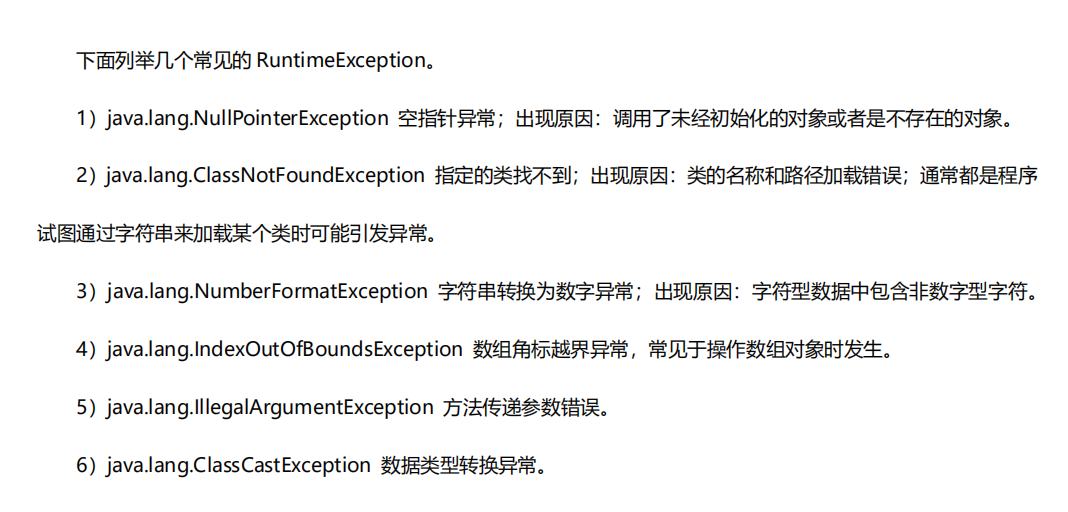
# 多态机制



# Error和Exception



# RuntimeException



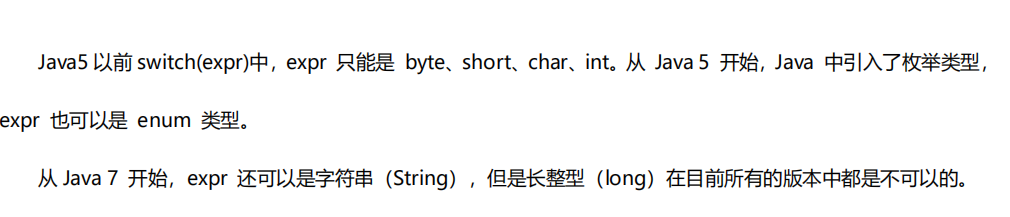
# JavaSE常用API

## Math.round

Math.round(11.5)等于12，Math.round(-11.5)等于 -11

四舍五入的原理是在参数上加0.5，然后再取整。

## Switch



## length()和length属性

数组没有length()方法，只有length属性。

String有length()方法。但是在js中通过length属性得到长度。刚好和java相反。

## StringBuilder和StringBuffer和String

由快到慢，StringBuilder非线程安全，StringBuffer线程安全。

String s = “abc”;

String ss = “f”+s+”addd”+5;

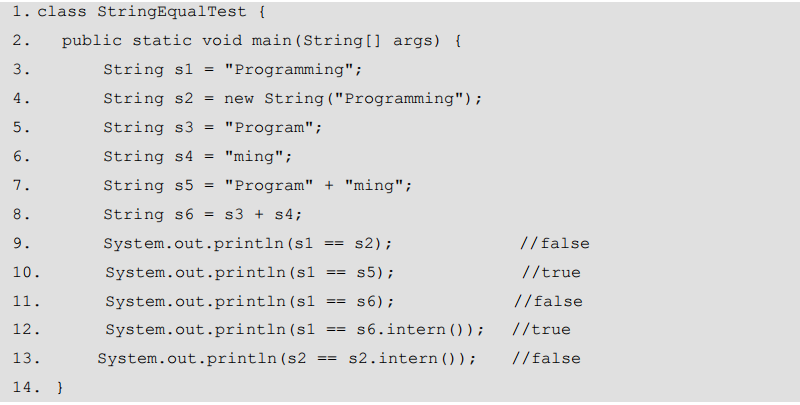
其实在编译的时候，java还是将+转换成StringBuilder。然后进行拼接，最后返回给ss。

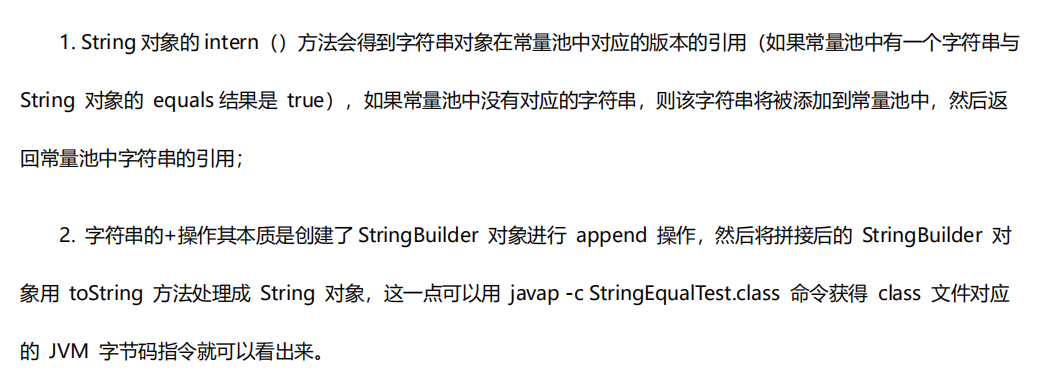
### +和StringBuilder的区别：

如果连接字符串行表达式很简单，那么他们是没有多大的差别的。但是如果使用循环连接字符串的话：

1. +号虽然转换成了StringBuilder，但是创建StringBuilder对象的位置却在for循环语句的内部，意思就是每执行一次循环就会创建一个StringBuilder对象。虽然Java有垃圾回收机制，但是回收器的时间是不定的。如果不断产生这样的垃圾，会占用大量的资源。
2. 直接用StringBuilder先创建一个对象，然后再循环内部使用拼接，这样一个对象就可以搞定。

### 测试





## 日期和时间 java8新特性time包

### Calendar

这个类在util包下，

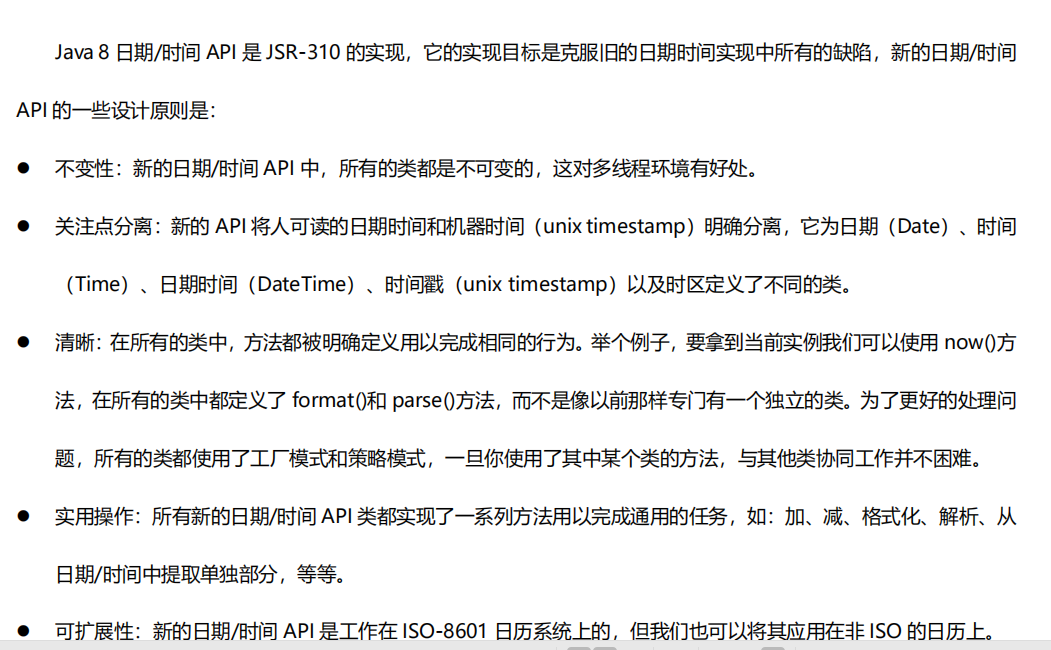
### LocalDate

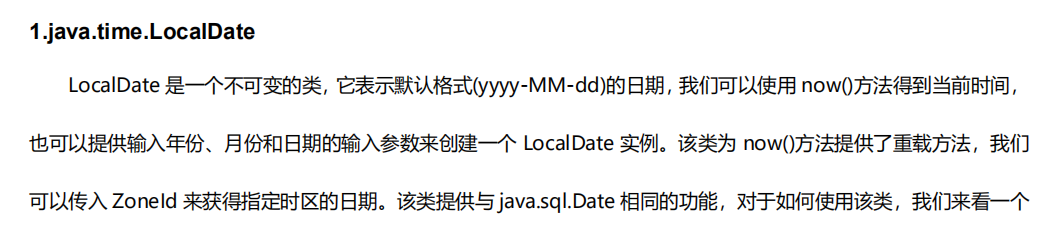
这个在time包下。

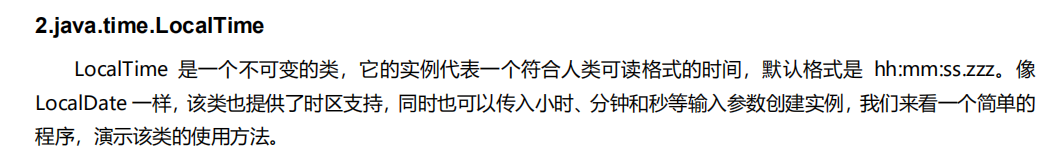
### SimpleDateFormat

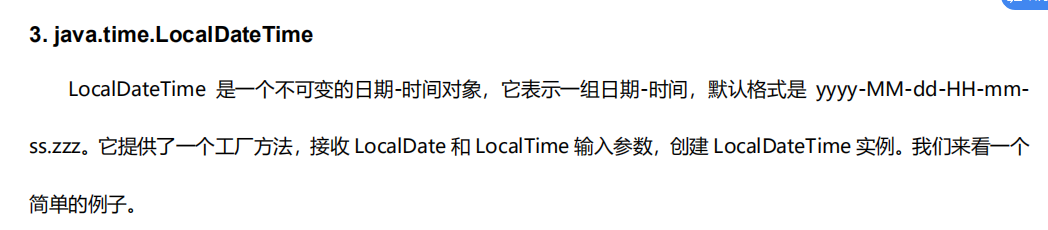
在text包下

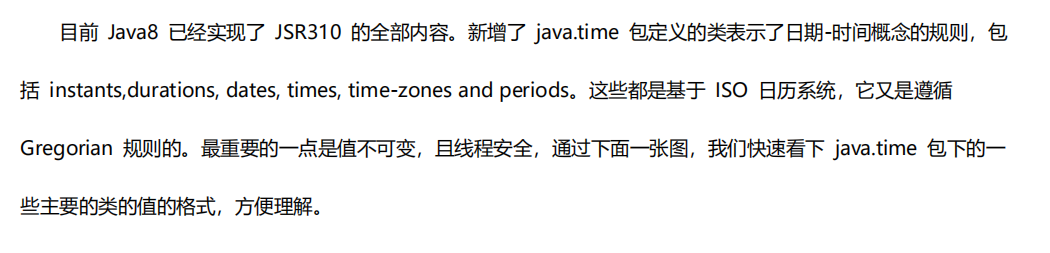
### Java8日期/时间特性











# Java 数据类型

## 基本数据类型

|  |  |  |
| --- | --- | --- |
| 四类 | 八种 | 字节数 |
| 整形 | byte | 1 |
| short | 2 |
| int | 4 |
| long | 8 |
| 浮点型 | float | 4 |
| double | 8 |
| 字符型 | char | 2 |
| 布尔型 | boolean | 1 |

## 类型转换

short s = 1;

1. s = s + 1;
2. s +=1;

第一种方式必须强制转换，short+int 结果是int类型的。要想成立要把结果转换一下。s = (short) (s +1);

第二种方式有隐含的强制转换。

## 装箱拆箱

Integer f1 = 100; f2 = 100;

F3 = 150; f4 = 150;

1. f1 == f2 true
2. f3 == f4 false

Integer f1 = 100 当这个整型字面量的值在-128到127之间，那么就不会new新的Integer对象，而是直接引用常量池中的Integer对象。

## 基本数据类型存放位置

在方法中声明的变量，属于局部变量，当程序调用方法时，系统都会为该方法创建一个方法栈，其中方法中的变量就在方法栈中。当方法结束后，方法栈就会销毁。

### 方法中

基本数据类型时，变量名及值都是放在栈中。

引用变量时，变量名放在栈中，但是变量值放在堆内存中。

### 类中

变量名以及值都是放在堆中。

# JavaIO

## 序列化与反序列化

序列化就是将对象进行流化，可以对流化的数据进行读写操作，也可以将流化后的对象传输与网络之间。序列化是为了解决在对对象流进行读写操作时所引发的问题。

序列化的实现：实现Serializable接口，该接口没有要实现的方法。实现此接口只是为了标注该对象是可以被序列化的。

# Java集合

## HashMap排序

|  |
| --- |
| private static HashMap<Integer, User> sort(Map<Integer, User> hashMap) {  Set<Entry<Integer, User>> entrySet = hashMap.entrySet();  List<Entry<Integer, User>> list = new ArrayList<>(entrySet);  Collections.sort(list,new Comparator<Entry<Integer, User>>() {  @Override  public int compare(Entry<Integer, User> o1, Entry<Integer, User> o2) {  return o1.getValue().getAge() - o2.getValue().getAge();  }  });  LinkedHashMap<Integer, User> linked = new LinkedHashMap<Integer, User>();  for (Entry<Integer, User> entry : list) {  linked.put(entry.getKey(), entry.getValue());  }  return linked;  } |

## 集合的安全性

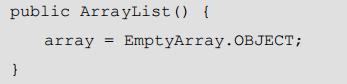
ArrayList，HashSet，HashMap都不是线程安全的。Vector和HashTable是线程安全的。

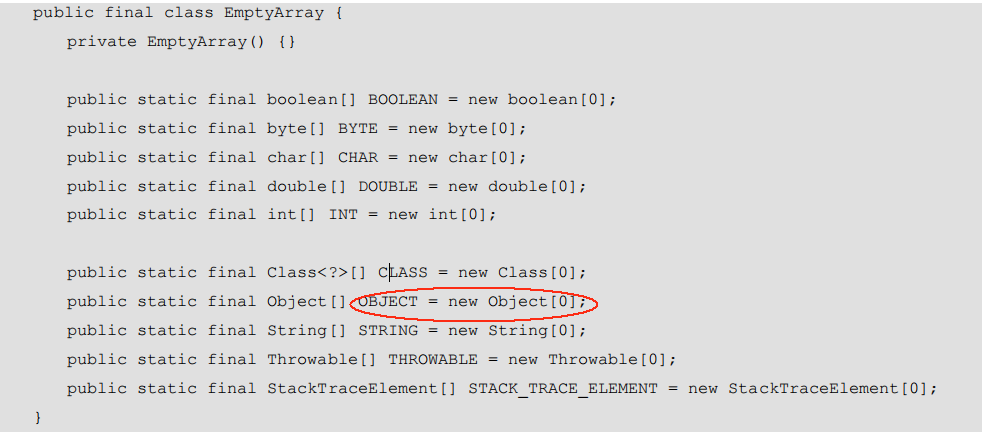
## ArrayList的内部原理

内部是Object[]数组实现的，

### 构造

#### 无参构造：





调用无参构造时其实内部使用了一个new Object[0]数组。

#### 有参构造1：

Public ArrayList(int capacity){

If(capacity<0){

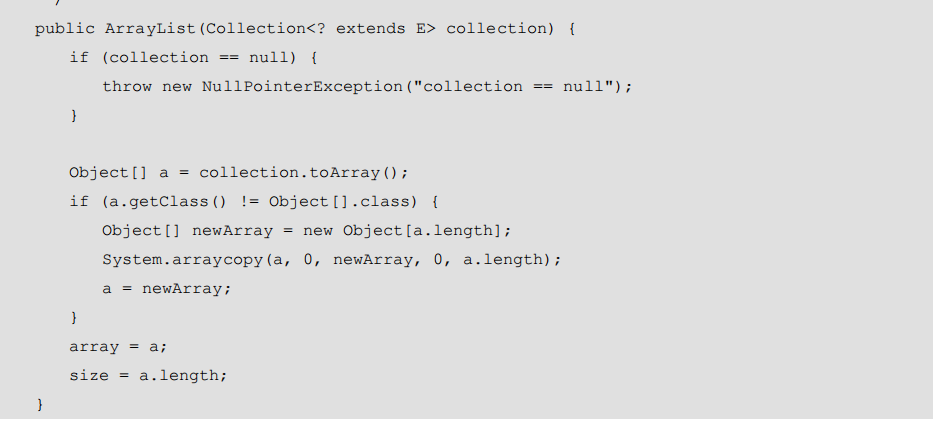
抛异常

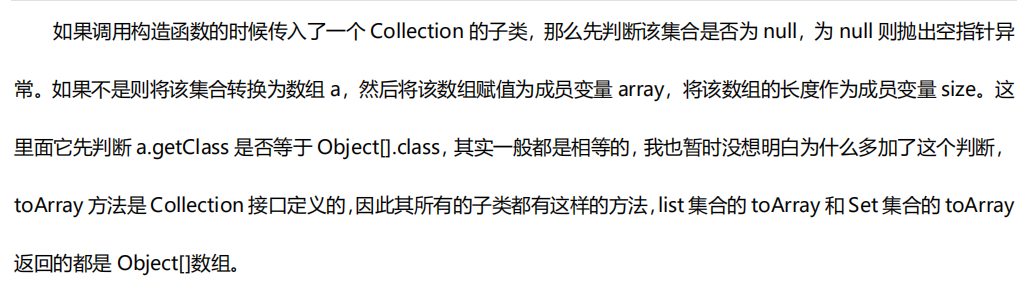
}

array = (capacity == 0? EmptyArray.OBJECT: new Object[capacity]);

}

#### 有参构造2





### Add方法

## 并发集合

集合分为普通集合、同步(线程安全)的集合、并发集合。

普通集合通常性能最高，但是不能保证多线程的安全性和并发的可靠性。

线程安全集合仅仅是给集合添加了synchronize同步锁。严重牺牲了性能。并且对并发的效率就更低了，

并发集合则通过复杂的策略不仅保证了多线程的安全又提高并发的效率。

concurrentHashMap基于concurrentLevel划分出了多个Segment来对key-value进行存储，从而避免每次put操作都得锁住整个数组。在默认情况下可允许16个线程并发无阻塞的操作集合对象。

Set接口有两个实现类：

HashSet：底层是hashMap，不允许集合中有重复的值，使用该方式时需要提供重写equals()和hashCode()方法。

## 集合杂谈

List list = new ArrayList(); 方式一

ArrayList list = new ArrayyList(); 方式二

方式一，在创建对象以后上溯到list对象了，那他就是一个list对象了，只有list中的属性和方法了。方式二就是一个arrayList对象。

# 多线程-传统

当线程并发执行时，由于资源共享和线程协作，使用线程之间存在以下两种制约关系。

## 间接相互制约：

一个系统中的多个线程共同使用某种资源，如CPU，打印机等。一个在使用的时候另一个只能等待。

## 直接相互制约：

如线程B的执行必须等到线程A返回结果才能继续。

## 线程的互斥和同步

间接相互制约可以称为互斥，直接相互制约称为同步。

对于互斥也可以这样理解，线程A和线程B互斥访问某个资源，要么线程A等线程B执行完，要么线程B等待线程A执行完，这其实就是线程的同步了。

其实互斥是一种特殊的同步。

## ThreadLocal

ThreadLocal的作用和目的：用于实现线程内的数据共享，即对于相同的程序代码，多个模块在同一个线程中运行时要共享一份数据，而在另外线程中运行时共享另外一份数据。

## 多线程共享数据

### 多个线程行为一致，共同操作一个数据源

也就是每个线程执行的代码相同，可以使用一个Runnable对象，这个Runnable对象中有那个共享数据，例如卖票系统可以这么做。

### 多个线程行为不一致，共同操作一个数据源

每个线程执行的代码不同，这个时候需要不同的Runnable对象。例如银行存款。

# 多线程-并发库

Java1.5添加了一个新的包java.util.concurrent包。

java.util.concurrent 多线程并发库

java.util.concurrent.automic 多线程的原子性操作提供的工具类

java.util.concurrent.lock 多线程的锁机制

## Volatile关键字

1. volatile修改时的变量，线程在每次使用变量的时候，都会读取变量修改后的值，可见性，有序性。
2. Volatile没有原子性。
3. 当一个共享变量被volatile修饰时，它会保证修改的值立即被更新到主存。
4. 对变量的写操作不依赖当前值，如多线程执行a++，是无法通过volatile保证结果准确性。
5. 读写具有原子性，但是a++不具备原子性的。

Java虚拟机的内存模型中，有主内存和工作内存的概念，每个线程对应一个工作内存，并共享主内存数据。

对于普通变量，读操作会优先读取工作内存的数据，如果工作内存中不存在，则从主内存中拷贝一份数据到工作内存中，写操作只会修改工作内存的副本数据，这种情况下，其它线程就无法读取变量的最新值。

Volatile修饰的变量，读操作时，会将工作内存中对应的值设为无效，要求线程从主内存中读取数据，写操作时，jvm会将工作内存中对应的数据刷新到主内存中。

java.util.concurrent.lock包下有三大接口：

1. Lock接口，支持那些语义不同(重入，公平等)的锁规则，主要实现是ReentrantLock
2. ReadWriteLock ReentrantReadWriteLock
3. Condition接口

## 创建多线程的三种方式

继承Thread，实现Runnable接口，使用ExecutorService，Callable，Future实现有返回结果的多线程。

## Wait和sleep区别

1. Sleep方法属于Thread类中的方法，而wait方法来自于Object。
2. 调犯Sleep必须捕获异常，而wait，notify，notifyAll不需要捕获异常
3. Sleep方法在睡眠时间到达后会自动醒来进入到可运行状态，不会马上进入到运行状态，因为线程调度机制恢复线程的运行也需要时间。并且在睡眠期间不会释放锁，所以他不会影响其他线程的运行。在sleep的过程中可能被其他对象调用他的intterupt()，产生InterruptException异常，如果不捕获这个异常，县城就会异常终止。

## Synchronized和volatile关键字

撒旦