1. synchronized同步方法
2. 非线程安全

非线程安全问题会在多个线程对同一个对象中的实例变量进行并发访问时发生，结果就是获取到已经被修改过的数据，即“脏读”。

1. 方法内的变量是线程安全的

非线程安全问题针对的是实例变量被并发访问的情况，而存在于方法内部的私有变量是不会被其他方法访问到的，因此不存在线程安全问题。注意，存在于方法内部的私有变量，会被不同的实例对象调用，这个实例对象在使用该方法的期间，使用该方法内的私有变量，此私有变量只会被此时的实例对象使用，而不会被其他实例对象所使用，因此，方法内的私有变量是线程安全的。

1. 多个对象创建多个锁

我们之前讨论的都是多个线程访问一个实例变量的问题，一个实例变量会创建一个锁。那么如果有多个实例变量呢？是不是就会创建多个锁呢？

是的，没错！当我们创建多个实例变量时，就会创建多个锁，每个锁之间的执行过程都是独立异步的，因此多个锁执行的顺序问题也就是异步的，而不是同步的。

（四） synchronized方法与非synchronized方法

当一个线程首先调用某个实例对象的一个synchronized方法时，该线程会持有这个实例对象的方法的线程锁，如果其他线程想要调用该实例对象的该synchronized方法，就需要排队。但是，如果其他线程想要调用的是该实例对象的一个非synchronized方法，就可以直接异步调用，无需排队等待！！！

（五）synchronized锁重入

关键字synchronized具有锁重入的功能，所谓的锁重入就是当一个线程得到一个对象锁之后，再次请求该对象锁时是可以再次得到该对象的锁的。主要在以下的场景中使用：

1. 在一个synchronized方法/块内部调用本类的其他synchronized方法/块时，是可以得到对象锁的。
2. 子类可以通过重入锁调用父类中的同步方法。

（六） 锁的释放

当线程执行的代码出现异常时，它所持有的的锁就会被自动释放，排队的其他线程就可以获得对象锁。

（七） 同步的继承性

同步是不能继承的，如果父类中的某个方法是同步方法，子类在重写该方法时是不会自动继承同步的，需要自己在子类方法上添加synchronized关键字。

1. synchronized同步代码块
2. 改进

之前我们都是将synchronized关键字应用在方法上，这样一个方法就会一直被某个线程占用，将会消耗大量时间，效率不高。这里我们使用synchronized代码块来解决，仅仅将方法中的一部分代码用synchronized关键字框起来，表示synchronized同步代码块。

1. synchronized同步代码块的使用

在方法中用synchronized关键字框住代码即可，如下所示：

package thread;  
  
public class Task {  
 public void doLongTime(){  
 for(int i = 0; i < 100; ++i){  
 System.*out*.println("no threadName = " + Thread.*currentThread*().getName() + "i= " + i);  
 }  
 System.*out*.println(" ");  
 synchronized (this){  
 for(int i = 0; i <100; ++i){  
 System.*out*.println("sy threadName = " + Thread.*currentThread*().getName() + " i = " +i);  
 }  
 }  
 }  
}

可以看到，方法doLongTime()并不是同步方法，方法内部的代码块才是同步的，而其他代码都是异步执行的。

1. synchronized(this)代码块间的同步性

当一个线程访问一个对象的一个synchronized(this)同步代码块时，其他线程对于该对象中的所有其他synchronized(this)同步代码块的访问将会被阻塞，这其实是因为这些同步代码块所使用的的对象监视器都是this。因此，使用同一个对象监视器的同步代码块在同一时间只有一个线程可以执行。同时，它还能够阻塞其他synchronized同步方法的调用，因为同步方法的对象监视器也是this。

1. 将任意对象作为对象监视器

可以使用实例变量及方法的参数作为对象监视器，使用的格式如下：

synchronized(对象监视器){}

当多个线程持有同一个对象监视器时，同一时间只有一个线程可以执行该同步代码块的代码。这种将非this作为对象监视器的同步代码块，它与其他的同步方法是异步的，因为对象监视器不一样，一个是非this对象，一个是this对象，因此可以提高运行效率。

同时，如果将一个非this对象作为对象监视器，那么也能够保证该非this对象不会被脏读，因为该同步代码块在同一时间里只能被一个线程执行，这样，能够保证该非this对象的线程安全性。

结论：

使用非this对象x作为对象监视器时：

1. 当多个线程同时执行synchronized(x){}同步代码块时是同步效果。
2. 当其他线程执行x对象中synchronized同步方法时呈同步效果。
3. 当其他线程执行x对象方法里面的synchronized(this)代码块时也是呈现同步效果。
4. 当其他线程调用非synchronized方法时，是异步效果。
5. 静态同步synchronized方法

当关键字synchronized应用于static静态方法时，这是对该静态方法所在的Class类进行加锁，Class锁能够对这个类的所有对象实例起作用。

1. 同步synchronized(Class)代码块

作用与synchronized static方法的作用一样，即锁住该Class类，如下所示：

public class Service{

public static void printA(){

synchronized(Service.class){}

}

}

1. String常量池特性

在寻找对象监视器时，注意String类型在JVM中具有常量池缓存的特性，当使用两个相同的String字符串作为对象监视器时，可能会导致其他线程拿不到锁。

1. 多线程死锁问题

不同的线程等待着根本不可能被释放的锁，从而导致所有的任务都无法完成。死锁问题是我们在编写程序时出现的问题，避免出现双方持有对方的锁或者等待着对方释放锁的情况，这样哪一方都不可能继续执行下去。

1. 内置类的同步问题

内置类是一个类中的类，分为静态和非静态的。当我们在调用静态内置类中的同步方法或者同步代码块时，原则与其他类中是一样的。如果两个同步方法持有的对象监视器不一样，那么两个方法的执行过程就是异步的。如果一个内部类InnerClass1的一个方法method1中的同步代码块的对象监视器是另一个内部类InnerClass2，那么这个同步代码块与InnerClass2中的同步方法或者this同步代码块就是同步的。

1. 锁对象的改变

我们知道，当多个线程同时持有同样的锁对象时，这些线程之间是同步的。若这些线程是在不同的时刻获得这个锁对象，依然是同步的。但是，如果这个锁对象在赋予不同的线程时被修改了，其他的线程获取到的就不是这个锁对象了，那么这些线程就不是同步的了。注意，如果改变的只是该对象的属性，其他线程获取的依然是这个锁对象，那么这些线程依然是同步的。