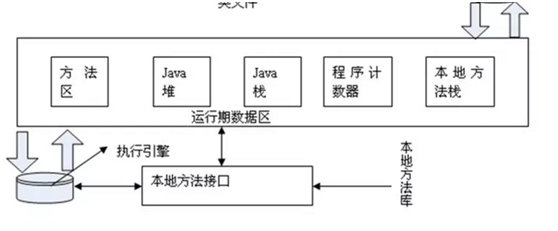
# Jvm

编译

 **-----------------------------------------------------------------------------------------------------**

好处：1.避免类重复加载，所有类由父先加载，子可以直接使用；2.安全，避免其它api试图替代系统api

**Bootstrap ClassLoader**

**Extension ClassLoader**

**Application ClassLoader**

双亲委派机制

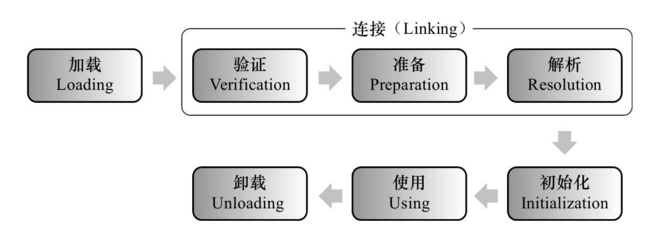
classload

Object.class

Obeject.java

## 类加载过程

JVM类加载机制分为五个部分：加载，验证，准备，解析，初始化



加载：查找并加载类的二进制数据，这个阶段会在堆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入口

验证：这一阶段的主要目的是为了确保Class文件的字节流中包含的信息是否符合当前虚拟机的要求，并且不会危害虚拟机自身的安全

准备：为类变量（即**静态变量）**分配内存，并将其初始化为默认值

此时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Java堆中。

比如一个类变量定义为：

|  |  |
| --- | --- |
| 1 | public static int v = 8080; |

实际上变量v在准备阶段过后的初始值为0而不是8080，将v赋值为8080的putstatic指令是程序被编译后，存放于类构造器<client>方法之中  
但是注意如果声明为：

|  |  |
| --- | --- |
| 1 | public static final int v = 8080; |

在编译阶段会为v生成ConstantValue属性，在准备阶段虚拟机会根据ConstantValue属性将v赋值为8080。

Ps: 静态代码块是在初始化阶段执行

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。   
a、符号引用：以一组符号来描述所引用的目标，符号可以是任何形式字面量，只要使用时无歧义地定位到目标就行。   
b、 直接引用：直接引用是直接指向目标的指针、相对偏移量或是一个能间接定位到目标的句柄。引用的目标已经在内存中存在。   
虚拟机实现可以根据需要来判断到底在类被加载器加载时就对常量池中的符号引用进行解析，还是等到一个符号引用将要被使用时才去解析它。解析动作主要针对类或接口、字段、类方法、接口方法、方法类型、方法句柄和调用点限定符7类符号引用进行。

初始化：真正执行类中定义的Java程序代码，是执行类构造器<client>方法的过程。

扩展了解：

**JVM初始化步骤**

 1、假如这个类还没有被加载和连接，则程序先加载并连接该类

 2、假如该类的直接父类还没有被初始化，则先初始化其直接父类

 3、假如类中有初始化语句，则系统依次执行这些初始化语句

**类初始化时机：只有当对类的主动使用的时候才会导致类的初始化，类的主动使用包括以下六种：**

– 创建类的实例，也就是new的方式

– 访问某个类或接口的静态变量，或者对该静态变量赋值

– 调用类的静态方法

– 反射（如Class.forName(“com.shengsiyuan.Test”)）

– 初始化某个类的子类，则其父类也会被初始化

– Java虚拟机启动时被标明为启动类的类（Java Test），直接使用java.exe命令来运行某个主类

**注意以下几种情况不会执行类初始化：**

* 通过子类引用父类的静态字段，只会触发父类的初始化，而不会触发子类的初始化。
* 定义对象数组，不会触发该类的初始化。
* 常量在编译期间会存入调用类的常量池中，本质上并没有直接引用定义常量的类，不会触发定义常量所在的类。
* 通过类名获取Class对象，不会触发类的初始化。
* 通过Class.forName加载指定类时，如果指定参数initialize为false时，也不会触发类初始化，其实这个参数是告诉虚拟机，是否要对类进行初始化。
* 通过ClassLoader默认的loadClass方法，也不会触发初始化动作。

## 内存策略

1.方法区：静态变量和方法，存放了每个Class的结构信息，包括常量池、字段描述、方法描述等等。

2.堆区：new的对象以及对象的成员变量

特点——动态创建，gc回收，内存空间大，但是效率不及栈区

3.栈区：方法内的变量（局部变量）以及方法本身

在线程创建时创建，它的**生命期是跟随线程的生命期**，线程结束栈内存也就释放

特点—内存空间小，效率高

## 内存泄漏

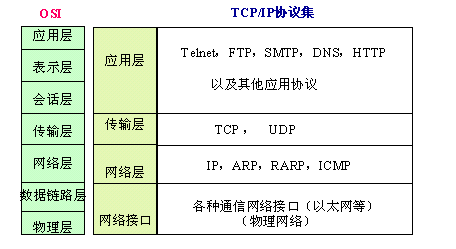
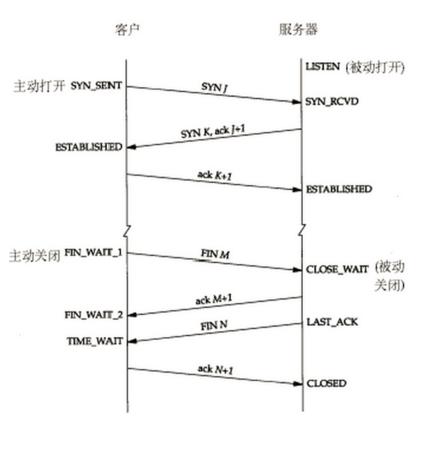
根本原因解释：长生命周期的对象持有了短生命周期对象的引用

# 网络

## 简介

Android中的网络通信基于Http实现，http基于TCP\IP通信模型，TCP/IP则需要实现socket编程，socket编程离不开IO

## Tcp/ip



Tcp3次握4次次手

## C:\Users\User\AppData\Local\Temp\1536311938(1).pngSocket

Socket是什么？？

Socket是应用层与TCP/IP协议族通信的中间软件抽象层，它是一组接口。在设计模式中，Socket其实就是一个门面模式，它把复杂的TCP/IP协议族隐藏在Socket接口后面，对用户来说，一组简单的接口就是全部，让Socket去组织数据，以符合指定的协议

简单来讲就是供开发人员使用的一组可以将数据从客户端运输到服务端的一组接口（反过来亦可）

创建ServerSocket

Client

OutputStream

关闭Socket

Close资源

OutputStream

InputStream

Socket=serverSocket.accept()

关闭Socket

Close

InputStream

创建Socket

以上是一个socket请求的流程：

**Socket通信的步骤**

**① 创建ServerSocket和Socket**

**② 打开连接到Socket的输入/输出流**

**③ 按照协议对Socket进行读/写操作**

**④ 关闭输入输出流、关闭Socket**

1. **客户端：**

**① 创建Socket对象，指明需要连接的服务器的地址和端口号**

**② 连接建立后，通过输出流想服务器端发送请求信息**

**③ 通过输入流获取服务器响应的信息**

**④ 关闭响应资源**

**服务器端：**

**① 创建ServerSocket对象，绑定监听端口**

**② 通过accept()方法监听客户端请求**

**③ 连接建立后，通过输入流读取客户端发送的请求信息**

**④ 通过输出流向客户端发送乡音信息**

**⑤ 关闭相关资源**

## IO

### BIO(阻塞式IO----block io)

如果按照以上方式实现网络数据交互，假如一个客户端请求：

Accept() block

Read() block

Write() block

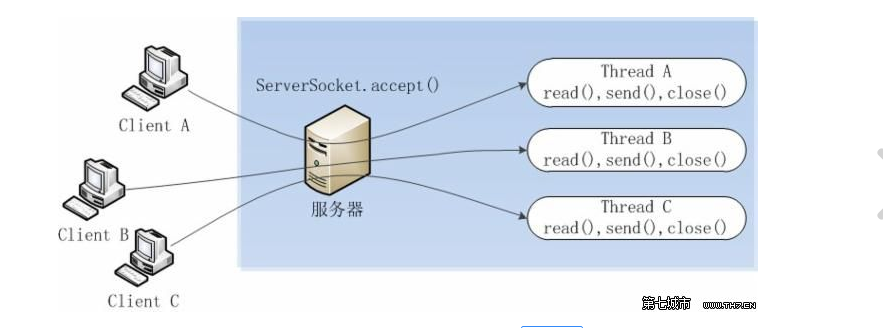
可以发现，上面三个步骤都是阻塞的。

关于阻塞（block）:执行IO或同步代码块时，线程会产生阻塞，失去CPU。

其中IO操作只有开始和结束需要cpu,数据的准备（比如从硬盘到核心）并不需要cpu.所以执行IO时线程会阻

那么，上图中的阻塞会产生什么影响呢？

当多个客户端同时发起请求时，如果依然是单线程的话，那么上面步骤中任意一步发生阻塞，都会导致整个请求阻塞，其它客户端无法建立连接。解决方法就是多线程，如图：



在建立连接后，就创建一个线程处理IO事件；

一个连接就创建一个线程即我们所说的BIO

这种io的优点：

1. 结构简单，容易管理
2. 便于开发低并发小请求

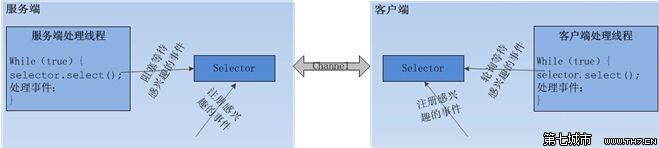
缺点：

当高并发发生时，会创建大量线程

1. 线程创建与销毁的对内存以及性能的消耗
2. 线程频繁切换对性能的影响
3. 无效请求对性能的浪费

所以这种情况需要一种新的方式——NIO

### NIO



因为IO是独立于cpu的，IO只有开始和结束需要cpu（线程）操作，过程是不需要的，虽然IO不占用cpu,但是在IO执行完成之前，线程只能等待，无法执行后续代码，所以导致线程会阻塞，失去cpu，直到IO完成。基于这种情况，有了BIO形式的通信，它的缺点上面说了很多，所以在很多场景下是不适用的，这时就有了一种新的通信方式NIO

#### NIO基础

Java NIO(New IO)是一个可以替代标准Java IO API的IO API（从Java 1.4开始)，Java NIO提供了与标准IO不同的IO工作方式。

**Java NIO: Channels and Buffers（通道和缓冲区）**

标准的IO基于字节流和字符流进行操作的，而NIO是基于通道（Channel）和缓冲区（Buffer）进行操作，数据总是从通道读取到缓冲区中，或者从缓冲区写入到通道中。

通道类型：

* + FileChannel：从文件中读写数据。
  + DatagramChannel：能通过UDP读写网络中的数据。
  + SocketChannel：能通过TCP读写网络中的数据。
  + ServerSocketChannel：可以监听新进来的TCP连接，像Web服务器那样。对每一个新进来的连接都会创建一个SocketChannel。

FileChannel比较特殊，它可以与通道进行数据交互， 不能切换到非阻塞模式，套接字通道可以切换到非阻塞模式；

缓冲区类型：

ByteBuffer/MappedByteBuffer/CharBuffer/DoubleBuffer/FloatBuffer/IntBuffer/LongBuffer/

ShortBuffer/

**Java NIO: Selectors（选择器）**

Java NIO引入了选择器的概念，选择器用于监听多个通道的事件（比如：连接打开，数据到达）。因此，单个的线程可以监听多个数据通道。

*选择器：相当于一个观察者，用来监听通道感兴趣的事件，一个选择器可以绑定多个通道；*

 通道向选择器注册时，需要指定感兴趣的事件，选择器支持以下事件：

* + SelectionKey.OP\_CONNECT SelectionKey.OP\_ACCEPT
  + SelectionKey.OP\_READ SelectionKey.OP\_WRITE

 　　如果你对不止一种事件感兴趣，那么可以用“位或”操作符将常量连接起来，如下：

　　 int interestSet = SelectionKey.OP\_READ | SelectionKey.OP\_WRITE;

 　　通道向选择器注册时，会返回一个 SelectionKey对象，具有如下属性

* + interest集合 ready集合
  + Channel　　Selector
  + 附加的对象（可选）

也可以使用以下四个方法获取已就绪事件，返回值为boolean：

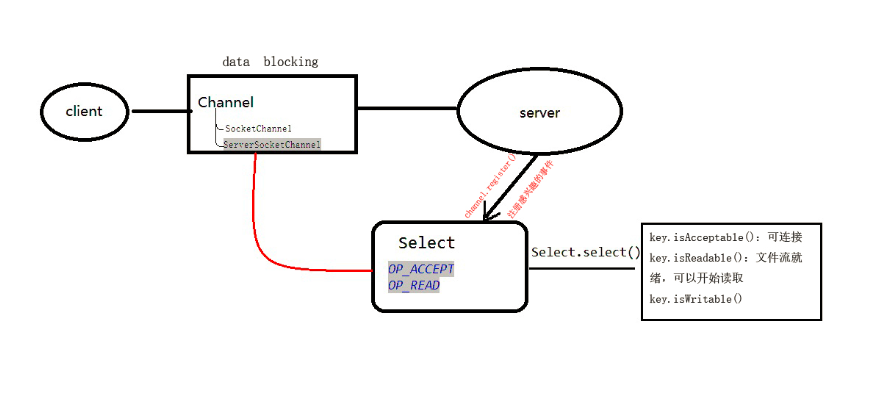
selectionKey.isAcceptable(); selectionKey.isConnectable(); selectionKey.isReadable(); selectionKey.isWritable();

代码中常用以上方法来区分Select的事件，可以根据这点作不同的操作

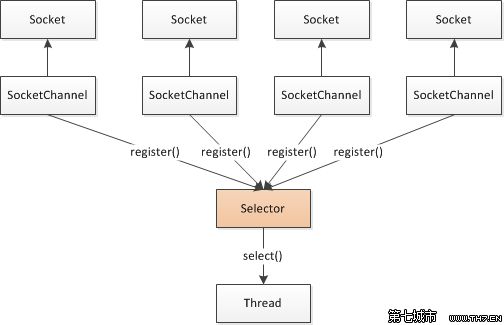
**Java NIO: Non-blocking IO（非阻塞IO）**

Java NIO可以让你非阻塞的使用IO，例如：当线程从通道读取数据到缓冲区时，线程还是可以进行其他事情。当数据被写入到缓冲区时，线程可以继续处理它。从缓冲区写入通道也类似。

在nio下一个客户端请求服务器的简单流程图：



服务器大致需要处理的三件事情：客户端的连接，IO输入，IO输出。但是这时候这三个过程不再像上面BIO图中那样都是block的了。只有IO或连接事件就绪,我们才去处理，这样cpu只要处理逻辑操作就行，不必阻塞等待。



当有多个客户端同时请求的时候，通过ServerSocketChannel向Select注册SelectionKey.OP\_ACCEPT，然后通过轮询获取，如果是连接事件，则获取SocketChannel并注册SelectionKey.*OP\_READ|SelectionKey.OP\_WRITE这样该Select同样就可以获取到这个请求的事件了。*

/\*\*

\* The Server accepts connections in non-blocking fashion. A connection when

\* established, a socket is created, which is registered for read/write with the

\* selector. Reading/Writing is performed on this socket when the selector

\* unblocks. This program works exactly the same way as MultiJabberServer.

\*/

**public** **class** MultiJabberServer1 {

**public** **static** **final** **int** *PORT* = 8080;

**public** **static** **void** main(String[] args) **throws** IOException {

// Channel read from data will be in ByteBuffer form

// written by PrintWriter.println(). Decoding of this

// byte stream requires character set of default encoding.

String encoding = System.*getProperty*("file.encoding");

// Had to initialized here since we do not wish to create

// a new instance of Charset everytime it is required

// Charset cs = Charset.forName(

// System.getProperty("file.encoding"));

Charset cs = Charset.*forName*(encoding);

ByteBuffer buffer = ByteBuffer.*allocate*(16);

SocketChannel ch = **null**;

ServerSocketChannel ssc = ServerSocketChannel.*open*();

Selector sel = Selector.*open*();

**try** {

ssc.configureBlocking(**false**);

// Local address on which it will listen for connections

// Note: Socket.getChannel() returns null unless a channel

// is associated with it as shown below.

// i.e the expression (ssc.socket().getChannel() != null) is true

ssc.socket().bind(**new** InetSocketAddress(*PORT*));

// Channel is interested in OP\_ACCEPT events

SelectionKey key = ssc.register(sel, SelectionKey.*OP\_ACCEPT*);

System.*out*.println("Server on port: " + *PORT*);

**while** (**true**) {

sel.select();

Iterator it = sel.selectedKeys().iterator();

**while** (it.hasNext()) {

SelectionKey skey = (SelectionKey) it.next();

it.remove();

**if** (skey.isAcceptable()) {

ch = ssc.accept();

System.*out*.println("Accepted connection from:"

+ ch.socket());

ch.configureBlocking(**false**);

ch.register(sel, SelectionKey.*OP\_READ*);

} **else** {

// Note no check performed if the channel

// is writable or readable - to keep it simple

ch = (SocketChannel) skey.channel();

ch.read(buffer);

CharBuffer cb = cs.decode((ByteBuffer) buffer.flip());

String response = cb.toString();

System.*out*.print("Echoing : " + response);

ch.write((ByteBuffer) buffer.rewind());

**if** (response.indexOf("END") != -1)

ch.close();

buffer.clear();

}

}

}

} **finally** {

**if** (ch != **null**)

ch.close();

ssc.close();

sel.close();

}

}

} // /:

这个过程太难通过文字和画图理解，还是看代码吧。Read the fucking source code

源码：https://blog.csdn.net/qiaoyl113/article/details/82692669

[**java NIO原理及实例**](https://www.cnblogs.com/tengpan-cn/p/5809273.html)**：https://www.cnblogs.com/tengpan-cn/p/5809273.html**

NIO全解析：http://ifeve.com/socket-channel/

[Android系列之网络](http://www.cnblogs.com/smyhvae/p/4004983.html): <https://www.cnblogs.com/smyhvae/p/4004983.html>

[**Socket通信原理**](https://www.cnblogs.com/wangcq/p/3520400.html)**:**https://www.cnblogs.com/wangcq/p/3520400.html

**java多线程/磁盘IO过程详解:** **https://blog.csdn.net/wenxindiaolong061/article/details/79741027**

**处理大并发之一 对异步非阻塞的理解:** **https://www.cnblogs.com/-900401/p/4015048.html**

# Thread

## 一．线程的状态



线程从创建到最终的消亡，要经历若干个状态。一般来说，线程包括以下这几个状态：创建(new)、就绪(runnable)、运行(running)、阻塞(blocked)、time waiting、waiting、消亡（dead）。

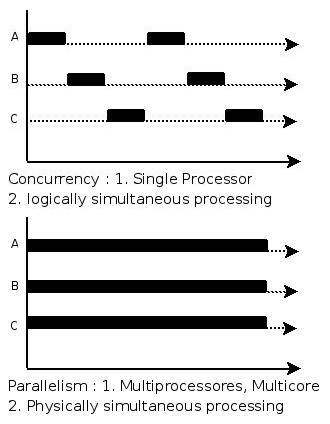
当需要新起一个线程来执行某个子任务时，就创建了一个线程。但是线程创建之后，不会立即进入就绪状态，因为线程的运行需要一些条件（比如内存资源，在前面的JVM内存区域划分一篇博文中知道程序计数器、Java栈、本地方法栈都是线程私有的，所以需要为线程分配一定的内存空间），只有线程运行需要的所有条件满足了，才进入就绪状态。

当线程进入就绪状态后，不代表立刻就能获取CPU执行时间，也许此时CPU正在执行其他的事情，因此它要等待。当得到CPU执行时间之后，线程便真正进入运行状态。

线程在运行状态过程中，可能有多个原因导致当前线程不继续运行下去，比如用户主动让线程睡眠（睡眠一定的时间之后再重新执行）、用户主动让线程等待，或者被同步块给阻塞，此时就对应着多个状态：time waiting（睡眠或等待一定的事件）、waiting（等待被唤醒）、blocked（阻塞）。

当由于突然中断或者子任务执行完毕，线程就会被消亡。

# 二．上下文切换

对于单核CPU来说（对于多核CPU，此处就理解为一个核），CPU在一个时刻只能运行一个线程，当在运行一个线程的过程中转去运行另外一个线程，这个叫做线程上下文切换（对于进程也是类似）。

**左图一描述了单核下多线程并发运行的形式（与下图多核并行区别）**

一般来说，线程上下文切换过程中会记录程序计数器、CPU寄存器状态等数据。

对于线程的上下文切换实际上就是 存储和恢复CPU状态的过程，它使得线程执行能够从中断点恢复执行。

虽然多线程可以使得任务执行的效率得到提升，但是由于在线程切换时同样会带来一定的开销代价（前文提到的bio就有该缺陷），并且多个线程会导致系统资源占用的增加，所以在进行多线程编程时要注意这些因素

## 三．常用方法

1）start方法

start()用来启动一个线程，当调用start方法后，系统才会开启一个新的线程来执行用户定义的子任务，在这个过程中，会为相应的线程分配需要的资源。

2）run方法

run()方法是不需要用户来调用的，当通过start方法启动一个线程之后，当线程获得了CPU执行时间，便进入run方法体去执行具体的任务。注意，继承Thread类必须重写run方法，在run方法中定义具体要执行的任务。

3）sleep 交出cpu,不释放锁，不多解释

4）yield方法

调用yield方法会让当前线程交出CPU权限，让CPU去执行其他的线程。它跟sleep方法类似，同样不会释放锁。但是yield不能控制具体的交出CPU的时间，另外，yield方法只能让拥有相同优先级的线程有获取CPU执行时间的机会。

注意，调用yield方法并不会让线程进入阻塞状态，而是让线程重回就绪状态，它只需要等待重新获取CPU执行时间，这一点是和sleep方法不一样的。

5）join方法 见博客；用wait方法实现，由于wait方法会让线程释放对象锁，所以join方法同样会让线程释放对一个对象持有的锁。

6）interrupt方法

interrupt，顾名思义，即中断的意思。单独调用interrupt方法可以使得处于阻塞状态的线程抛出一个异常，也就说，它可以用来中断一个正处于阻塞状态的线程；另外，通过interrupt方法和isInterrupted()方法来停止正在运行的线程。

7）以下是关系到线程属性的几个方法：

1）getId

用来得到线程ID

2）getName和setName

用来得到或者设置线程名称。

3）getPriority和setPriority

用来获取和设置线程优先级。

4）setDaemon和isDaemon

用来设置线程是否成为守护线程和判断线程是否是守护线程。

守护线程和用户线程的区别在于：守护线程依赖于创建它的线程，而用户线程则不依赖。举个简单的例子：如果在main线程中创建了一个守护线程，当main方法运行完毕之后，守护线程也会随着消亡。而用户线程则不会，用户线程会一直运行直到其运行完毕。在JVM中，像垃圾收集器线程就是守护线程。

Thread类有一个比较常用的静态方法currentThread()用来获取当前线程。

## 四．并发与线程安全

看JLS（java语言规范）对线程工作内存的描述，线程的working memory只是cpu的寄存器和高速缓存的抽象描述。

JVM的内存模型，怎么会扯到cpu上去呢？在此，我认为很有必要阐述下，先抛开java虚拟机不谈，我们都知道，现在的计算机，cpu在计算的时候，并不总是从内存读取数据，它的数据读取顺序优先级是：寄存器－高速缓存－内存。线程耗费的是CPU，线程计算的时候，原始的数据来自内存，在计算过程中，有些数据可能被频繁读取，这些数据被存储在寄存器和高速缓存中，当线程计算完后，这些缓存的数据在适当的时候应该写回内存。当个多个线程同时读写某个内存数据时，就会产生多线程并发问题，涉及到三个特性：原子性，有序性，可见性。在《线程安全总结》这篇文章中，为了理解方便，我把原子性和有序性统一叫做“多线程执行有序性”。支持多线程的平台都会面临这种问题，运行在多线程平台上支持多线程的语言应该提供解决该问题的方案。

## 五．线程的特性

**1.原子性：**  何谓原子性操作，即为最小的操作单元，比如i=1，就是一个原子性操作，这个过程只涉及一个赋值操作。又如i++就不是一个原子操作，它相当于语句i=i+1；这里包括读取i，i+1，结果写入内存三个操作单元。因此如果操作不符合原子性操作，那么整个语句的执行就会出现混乱，导致出现错误的结果，从而导致线程安全问题。

**2.可见性**：可见性是值一个线程对共享变量的修改，对于另一个线程来说是否是可以看到的。java线程通信是通过共享内存的方式进行通信的，而我们又知道，为了加快执行的速度，线程一般是不会直接操作内存的，而是操作缓存。实际上，线程操作的是自己的工作内存，而不会直接操作主内存。如果线程对变量的操作没有刷写主内存的话，仅仅改变了自己的工作内存的变量的副本，那么对于其他线程来说是不可见的。而如果另一个变量没有读取主内存中的新的值，而是使用旧的值的话，同样的也可以列为不可见。对于jvm来说，主内存是所有线程共享的java堆，而工作内存中的共享变量的副本是从主内存拷贝过去的，是线程私有的局部变量，位于java栈中。

**3.有序性**

有序性是指程序在执行的时候，程序的代码执行顺序和语句的顺序是一致的。为什么会出现不一致的情况呢？这是由于重排序的缘故。在Java内存模型中，允许编译器和处理器对指令进行重排序，但是重排序过程不会影响到单线程程序的执行，却会影响到多线程并发执行的正确性。

## 六．线程的安全

**1.** **synchronized关键字**

以上安全性问题都可以通过加锁实现线程安全。但是效率比较低

**2.volatile关键字**

1. 一般只需要使用volatile关键字，就能实现内存的可见性了
2. volatile关键字可以保证变量的操作是不会被重排序的

但是volatile关键字无法保证原子性

**3.使用J.U.C下的atomic来实现原子操作**

1. atomic系列的类在是J.U.C包下的一系列类。它主要包括四类：基本类型，数组类型，属性原子修改器类型，引用类型。
2. 基本类型的类主要包括AtomicInteger、AtomicLong、AtomicBoolean等；数组类型主要包括AtomicIntegerArray、AtomicLongArray、AtomicReferenceArray；属性原子修改器类型主要包括AtomicIntegerFieldUpdater、AtomicLongFieldUpdater、AtomicReferenceFieldUpdater ；引用类型主要包括AtomicReference、AtomicStampedRerence、AtomicMarkableReference。
3. **乐观锁**

所谓乐观锁就是，每次不加锁而是假设没有冲突而去完成某项操作，如果因为冲突失败就重试，直到成功为止。乐观锁实现的机制就是CAS操作（Compare and Swap）

Lock和atomic使用了这种机制

1. **5.CAS与Synchronized的使用情景：**

1、对于资源竞争较少（线程冲突较轻）的情况，使用synchronized同步锁进行线程阻塞和唤醒切换以及用户态内核态间的切换操作额外浪费消耗cpu资源；而CAS基于硬件实现，不需要进入内核，不需要切换线程，操作自旋几率较少，因此可以获得更高的性能。

2、对于资源竞争严重（线程冲突严重）的情况，CAS自旋的概率会比较大，从而浪费更多的CPU资源，效率低于synchronized。

　　　补充： synchronized在jdk1.6之后，已经改进优化。synchronized的底层实现主要依靠Lock-Free的队列，基本思路是自旋后阻塞，竞争切换后继续竞争锁，稍微牺牲了公平性，但获得了高吞吐量。在线程冲突较少的情况下，可以获得和CAS类似的性能；而线程冲突严重的情况下，性能远高于CAS。

[**Java Thread 的使用**](https://www.cnblogs.com/renhui/p/6066852.html)**:**https://www.cnblogs.com/renhui/p/6066852.html

<https://blog.csdn.net/xiashuangyuan1/article/details/75644463>

1.usb数据读取

a.方式有同步和异步，当数据量小的时候（小于10M/s）通常都可以正常使用。当数据量大的时候，任何操作都可能引起数据量的减少。

b.数据量大时：通常通过usbrequest进行读取，增加队列中的请求，减少requestWait()方法的等待时间。

c.usbrequest方式read()方法中接收多种类型队列的问题

2.cpu资源优化：

-ps 和 -top指令结合ddms跟踪trace view查看线程所在核心以及cpu占用率。

1.调用优化：线程的轮询频率，线程之间的切换调度对cpu的消耗都很大，且非线性变化。20ms和15ms或许差别不大，但是6ms和1ms可能是天差地别。

2.方式优化：通过单例线程池替代while true。usb数据量每秒超过17M，通过线程不停读取，如果在处理过程中有些许耗时操作都会影响数据量。所以

起初是通过添加缓存队列，然后开另外一个线程不停轮询取出进行组包处理。cpu上升迅速，通过单例线程池替代while true，有效减少。

3.fragment入栈 和livedata /viewmodel结合时的问题

平时使用Fragment如果需要响应back按键时，可能会用到事务的addToBackStack()方法，将当前Fragment加入回退栈中。

需要注意的是一旦加入回退栈中，系统才会将片段放入由宿主 Activity 管理的返回栈，Fragment的生命周期也随之改变，具体表现为当使用Replace（）替换Fragment时，加入回退栈的Fragment不会被销毁，也就不再执行onDestory()方法。

这里对viewmodel和livedata的影响？

我们知道viewmodel的生命周期是大于Activity/Fragment的，如果Fragment没有onDestory，这种情况下ViewModel也不会被销毁。因此当通过回退栈返回，无论是否重新获取，ViewModel并未创建新的实例，还是之前的实例。

如果ViewModel里面使用了LiveData,而初始化工作是在onCreat()后执行的，那么LiveData的观察者就可能会被反复创建多个，这样一来就会存在多个观察者，逻辑被执行多次。