**？？怎么处理警告：编码 GBK 的不可映射字符**

这是因为：由于[JDK](https://www.baidu.com/s?wd=JDK&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1YLuH7BmyckmWIBuWbLuWN-0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EnWm4njDdPW6" \t "_blank)是国际版的，在编译的时候，如果我们没有用-encoding参数指定我们的[JAVA](https://www.baidu.com/s?wd=JAVA&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1YLuH7BmyckmWIBuWbLuWN-0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EnWm4njDdPW6" \t "_blank)源程序的编码格式，则[java](https://www.baidu.com/s?wd=java&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1YLuH7BmyckmWIBuWbLuWN-0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EnWm4njDdPW6" \t "_blank)c.exe首先获得我们[操作系统](https://www.baidu.com/s?wd=%E6%93%8D%E4%BD%9C%E7%B3%BB%E7%BB%9F&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1YLuH7BmyckmWIBuWbLuWN-0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EnWm4njDdPW6)默认采用的编码格式，也即在编译[java](https://www.baidu.com/s?wd=java&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1YLuH7BmyckmWIBuWbLuWN-0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EnWm4njDdPW6)程序时，若我们不指定源程序文件的编码格式，[JDK](https://www.baidu.com/s?wd=JDK&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1YLuH7BmyckmWIBuWbLuWN-0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EnWm4njDdPW6" \t "_blank)首先获得[操作系统](https://www.baidu.com/s?wd=%E6%93%8D%E4%BD%9C%E7%B3%BB%E7%BB%9F&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1YLuH7BmyckmWIBuWbLuWN-0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EnWm4njDdPW6" \t "_blank)的file.encoding参数(它保存的就是[操作系统](https://www.baidu.com/s?wd=%E6%93%8D%E4%BD%9C%E7%B3%BB%E7%BB%9F&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1YLuH7BmyckmWIBuWbLuWN-0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EnWm4njDdPW6" \t "_blank)默认的编码格式，如WIN2k，它的值为[GBK](https://www.baidu.com/s?wd=GBK&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1YLuH7BmyckmWIBuWbLuWN-0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EnWm4njDdPW6" \t "_blank))，然后[JDK](https://www.baidu.com/s?wd=JDK&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1YLuH7BmyckmWIBuWbLuWN-0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EnWm4njDdPW6" \t "_blank)就把我们的java源程序从file.encoding编码格式转化为JAVA内部默认的[UNICODE](https://www.baidu.com/s?wd=UNICODE&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1YLuH7BmyckmWIBuWbLuWN-0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EnWm4njDdPW6" \t "_blank)（一个英文字母占用两个字节）格式放入内存中。然后，javac把转换后的unicode格式的文件进行编译成.class类文件，此时.class文件是[UNICODE](https://www.baidu.com/s?wd=UNICODE&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1YLuH7BmyckmWIBuWbLuWN-0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EnWm4njDdPW6" \t "_blank)编码的，它暂放在内存中，紧接着，JDK将此以UNICODE编码的编译后的class文件保存到我们的操作系统中形成我们见到的.class文件。对我们来说，我们最终获得的.class文件是内容以UNICODE编码格式保存的类文件，它内部包含我们源程序中的中文字符串，只不过此时它己经由file.encoding格式转化为UNICODE格式了。当我们不加设置就编译时，相当于使用了参数：javac -encoding [gbk](https://www.baidu.com/s?wd=gbk&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1YLuH7BmyckmWIBuWbLuWN-0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EnWm4njDdPW6) XX.java，当然就会出现不兼容的情况。

**有时也不会出现警告，那么可能是正确转换了编码格式，或者我们设置了某些选项**。

解决办法是：应该使用-encoding参数指明编码方式：javac -encoding UTF-8 XX.java，这下没警告了，运行也正确了在JCreator 4中设置：菜单：Configure --> Options --> JDK Tools --> Compiler，选中<Default>，然后选Edit，Parameters里面，最前面添加：-encoding UTF-8。  
Parameters原来的  
默认值为：-classpath "$[ClassPath]" -d "$[OutputPath]" $[ModJavaFiles]  
修改后为：-encoding UTF-8 -classpath "$[ClassPath]" -d "$[OutputPath]" $[ModJavaFiles]

如果是NETBEANS，可以参考  
在项目 ---右键 -- 属性 -- 生成 -- 编译 --其他编译器选项 里加入 -encoding utf-8

# MongoDB的ObjectId



**TimeStamp**  
前4位是一个unix的时间戳（可以在mongo客户端使用命令-*ObjectId("4df2dcec2cdcd20936a8b817")*.getTimestamp()来获得当初创建此文档的时间），是一个int类别，我们将上面的例子中的objectid的前4位进行提取“4df2dcec”，然后再将他们按照十六进制转为十进制：“1307761900”，这个数字就是一个时间戳，为了让效果更佳明显，我们将这个时间戳转换成我们习惯的时间格式

$ date -d '1970-01-01 UTC 1307761900  sec'  -u  
2011年 06月 11日 星期六 03:11:40 UTC

前4个字节其实隐藏了文档创建的时间，并且时间戳处在于字符的最前面，这就意味着ObjectId大致会按照插入进行排序，这对于某些方面起到很大作用，如作为索引提高搜索效率等等。使用时间戳还有一个好处是，某些客户端驱动可以通过ObjectId解析出该记录是何时插入的，这也解答了我们平时快速连续创建多个Objectid时，会发现前几位数字很少发现变化的现实，因为使用的是当前时间，很多用户担心要对服务器进行时间同步，其实这个时间戳的真实值并不重要（同步时间与否都会保持数据的唯一性），只要其总不停增加就好。

**Machine**  
接下来的三个字节，就是 2cdcd2 ,这三个字节是所在主机的唯一标识符，一般是机器主机名的散列值（key-value值），这样就确保了不同主机生成不同的机器hash值，确保在分布式中不造成冲突，这也就是在同一台机器生成的objectid中间的字符串都是一模一样的原因。

**pid**  
上面的Machine是为了确保在不同机器产生的objectid不冲突，而pid就是为了在同一台机器不同的mongodb进程产生了objectid不冲突，接下来的0936两位就是产生objectid的进程标识符。

**increment**  
前面的九个字节是保证了一秒内不同机器不同进程生成objectid不冲突，这后面的三个字节a8b817，是一个自动增加的计数器，用来确保在同一秒内产生的objectid也不会发现冲突，允许256的3次方等于16777216条记录的唯一性。

**客户端生成**  
mongodb产生objectid还有一个更大的优势，就是mongodb可以通过自身的服务来产生objectid，也可以通过客户端的驱动程序来产生，如果你仔细看文档你会感叹，mongodb的设计无处不在的使用空间换时间的思想，比如objectid是轻量级，但服务端产生也必须开销时间，所以能从服务器转移到客户端驱动程序完成的就尽量的转移，必须将事务扔给客户端来完成，减低服务端的开销，另还有一点原因就是扩展应用层比扩展数据库层要变量得多。

**??? 将TimeSpan形式的字符串转换为TimeSpan类型**

string timeSpanStr=”1.02:12:22”;

方法一：

string tspan = "1.02:12:22";

TimeSpan ts = TimeSpan.Parse(tspan);

方法二：

int day = 0;

int hour = 0;

int minute = 0;

int second = 0;

string[] arry1 = tspan.Split(':');

string[] arry2 = arry1[0].Split('.');

day = Convert.ToInt32(arry2[0]);

hour = Convert.ToInt32(arry2[1]);

minute = Convert.ToInt32(arry1[1]);

second = Convert.ToInt32(arry1[2]);

TimeSpan ts = new TimeSpan(day,hour,minute,second);

**???Upserting in Mongo DB and the Id problem**

在程序外面可以使用ObjectId生成id来减轻服务器的负担：

ObjectId id = ObjectId.GenerateNewId();

**每一篇文档都继承自聚合根，这个聚合根引入了一个生成id的机制以防止id一样**

The id in AggregateRoot（聚合根） was introduced to correct the problem when retrieving data from MongoCollection to List and the generation was introduced so the id-s are different.

 Every document inherits from AggregateRoot（每一篇插入的文档都继承自聚合跟）, so this thing is generated on receiving every object.

**在外面设置id，插入时可以但是更新时是不可以的**

Looks like you might be explicitly setting the Id value for both inserts and updates. That's fine for inserts, all new objects need an \_id value, however for updates you're not allowed to change the value of \_id on an existing document after it's created.

**总的来说，在操作mongo库时，插入时可以指定自定义的id，但是在更新时是不能更改id的**

**???什么是线程安全和线程不安全**

首先要明白线程的工作原理，jvm有一个main   memory(主线程)，而每个线程有自己的working   memory（线程），一个线程对一个variable进行操作时，都要在自己的working   memory里面建立一个copy，操作完之后再写入main   memory。多个线程同时操作同一个variable，就可能会出现不可预知的结果。根据上面的解释，很容易想出相应的scenario（情景）。   
而用synchronized的关键是建立一个monitor，这个monitor可以是要修改的variable也可以其他你认为合适的object比如method，然后通过给这个monitor加锁来实现线程安全，每个线程在获得这个锁之后，要执行完load到workingmemory   －>   use&assign   －>   store到mainmemory   的过程，才会释放它得到的锁。这样就实现了所谓的线程安全—就是所谓的使用多线程时加锁机制，从而对一个变量操作时不会导致冲突。

------------------------------------------------------

什么是线程安全?线程安全是怎么完成的(原理)?   
线程安全就是说多线程访问同一代码，不会产生不确定的结果。编写线程安全的代码是依靠线程同步。

------------------------------------------------------

在接口方式中，线程有一个共享的数据成员（实现接口Runnable），即： private   int   count   =10;   
而在继承方式中（继承Thread类），线程之间没有共享的成员，而是各线程各自有一个私有成员，即： private   int   count   =10;   
楼主的代码刚好是一个好例子，说明了何时需要考虑线程同步，并在一定程序上说明了怎样才能编写出线程安全的代码。   
在多线程环境中，当各线程不共享数据的时候，那么一定是线程安全的。问题是这种情况并不多见，在多数情况下需要共享数据，这时就需要进行适当的同步控制了。

--------------------------------------------------------

线程安全一般都涉及到synchronized（同步的） 就是一段代码同时只能有一个线程来操作，不然中间过程可能会产生不可预制的结果

---------------------------------------------------------

如果你的代码所在的进程中有多个线程在同时运行，而这些线程可能会同时运行这段代码。如果每次运行结果和单线程运行的结果是一样的，而且其他的变量的值也和预期的是一样的，就是线程安全的。

举例 比如一个 ArrayList 类，在添加一个元素的时候，它可能会有两步来完成：1. 在 Items[Size] 的位置存放此元素；2. 增大 Size 的值。

在单线程运行的情况下，如果 Size = 0，添加一个元素后，此元素在位置 0，而且 Size=1； 而如果是在多线程情况下，比如有两个线程，线程 A 先将元素存放在位置 0。但是此时 CPU 调度线程A暂停，线程 B 得到运行的机会。线程B也向此 ArrayList 添加元素，因为此时 Size 仍然等于 0 （注意哦，我们假设的是添加一个元素是要两个步骤哦，而线程A仅仅完成了步骤1），所以线程B也将元素存放在位置0。然后线程A和线程B都继续运行，都增加 Size 的值。 那好，现在我们来看看 ArrayList 的情况，元素实际上只有一个，存放在位置 0，而 Size 却等于 2。这就是“线程不安全”了。

**安全性：**

线程安全性不是一个非真即假的命题。 Vector 的方法都是同步的，并且 Vector 明确地设计为在多线程环境中工作。但是它的线程安全性是有限制的，即在某些方法之间有状态依赖(类似地，如果在迭代过程中 Vector 被其他线程修改，那么由 Vector.iterator() 返回的 iterator会抛出ConcurrentModifiicationException)。

对于 Java 类中常见的线程安全性级别，没有一种分类系统可被广泛接受，不过重要的是在编写类时尽量记录下它们的线程安全行为。

Bloch 给出了描述五类线程安全性的分类方法：不可变、线程安全、有条件线程安全、线程兼容和线程对立。只要明确地记录下线程安全特性，那么您是否使用这种系统都没关系。这种系统有其局限性 -- 各类之间的界线不是百分之百地明确，而且有些情况它没照顾到 -- 但是这套系统是一个很好的起点。这种分类系统的核心是调用者是否可以或者必须用外部同步包围操作(或者一系列操作)。下面几节分别描述了线程安全性的这五种类别。

***不可变***

**不可变的对象一定是线程安全的**，并且永远也不需要额外的同步[1] 。因为一个不可变的对象只要构建正确，其外部可见状态永远也不会改变，永远也不会看到它处于不一致的状态。Java 类库中大多数基本数值类如 Integer 、 String 和 BigInteger 都是不可变的。

需要注意的是，对于Integer，该类不提供add方法，加法是使用+来直接操作。而+操作是不具线程安全的。这是提供原子操作类AtomicInteger的原因。

***线程安全***

线程安全的对象具有在上面“线程安全”一节中描述的属性 -- 由类的规格说明所规定的约束在对象被多个线程访问时仍然有效，不管运行时环境如何排线程都不需要任何额外的同步。这种线程安全性保证是很严格的 -- 许多类，如 Hashtable 或者 Vector 都不能满足这种严格的定义。

***有条件的***

有条件的线程安全类对于单独的操作可以是线程安全的，但是某些操作序列可能需要外部同步。条件线程安全的最常见的例子是遍历由 Hashtable 或者 Vector 或者返回的迭代器 -- 由这些类返回的 fail-fast 迭代器假定在迭代器进行遍历的时候底层集合不会有变化。为了保证其他线程不会在遍历的时候改变集合，进行迭代的线程应该确保它是独占性地访问集合以实现遍历的完整性。通常，独占性的访问是由对锁的同步保证的 -- 并且类的文档应该说明是哪个锁(通常是对象的内部监视器(intrinsic monitor))。

如果对一个有条件线程安全类进行记录，那么您应该不仅要记录它是有条件线程安全的，而且还要记录必须防止哪些操作序列的并发访问。用户可以合理地假设其他操作序列不需要任何额外的同步。

***线程兼容***

线程兼容类不是线程安全的，但是可以通过正确使用同步而在并发环境中安全地使用。这可能意味着用一个 synchronized 块包围每一个方法调用，或者创建一个包装器对象，其中每一个方法都是同步的(就像 Collections.synchronizedList() 一样)。也可能意味着用 synchronized 块包围某些操作序列。为了最大程度地利用线程兼容类，如果所有调用都使用同一个块，那么就不应该要求调用者对该块同步。这样做会使线程兼容的对象作为变量实例包含在其他线程安全的对象中，从而可以利用其所有者对象的同步。

许多常见的类是线程兼容的，如集合类 ArrayList 和 HashMap 、 java.text.SimpleDateFormat 、或者 JDBC 类 Connection 和 ResultSet 。

***线程对立***

线程对立类是那些不管是否调用了外部同步都不能在并发使用时安全地呈现的类—也即线程对立禁止使用多线程。线程对立很少见，当类修改静态数据，而静态数据会影响在其他线程中执行的其他类的行为，这时通常会出现线程对立。线程对立类的一个例子是调用 System.setOut() 的类。