Java知识

1. **List,Set,Map**

1、List（有序、可重复）

List里存放的对象是有序的，同时也是可以重复的，List关注的是索引，拥有一系列和索引相关的方法，查询速度快。因为往list集合里插入或删除数据时，会伴随着后面数据的移动，所以插入删除数据速度慢。

2、Set（无序、不能重复）

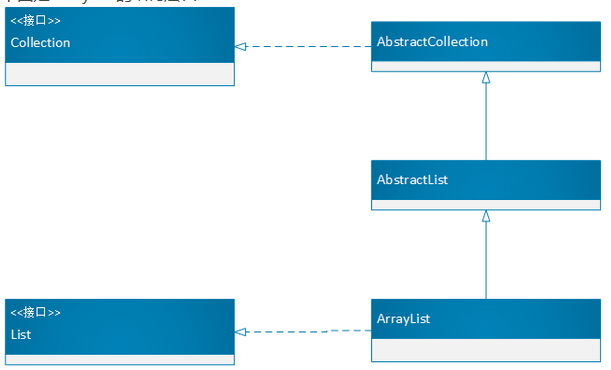
Set里存放的对象是无序，不能重复的，集合中的对象不按特定的方式排序，只是简单地把对象加入集合中。

3、Map（键值对、键唯一、值不唯一）

Map集合中存储的是键值对，键不能重复，值可以重复。根据键得到值，对map集合遍历时先得到键的set集合，对set集合进行遍历，得到相应的值。



1. **ArrayList**



ArrayList是Java集合框架中的一个重要的类，它继承于AbstractList，实现了List接口，是一个长度可变的集合，提供了增删改查的功能。集合中允许null的存在。ArrayList类还是实现了RandomAccess接口，可以对元素进行快速访问。实现了Serializable接口，说明ArrayList可以被序列化，还有Cloneable接口，可以被复制。和Vector不同的是，ArrayList不是线程安全的。

以数组实现。节约空间，但数组有容量限制。超出限制时会增加50%容量，用System.arraycopy()复制到新的数组，因此最好能给出数组大小的预估值。默认第一次插入元素时创建大小为10的数组。

其次可以快速查找：在物理内存上采用顺序存储结构，因此可根据索引快速的查找元素。因此按数组下标访问元素—get(i)/set(i,e) 的性能很高，这是数组的基本优势。直接在数组末尾加入元素—add(e)的性能也高，但如果按下标插入、删除元素—add(i,e), remove(i), remove(e)，则要用System.arraycopy()来移动部分受影响的元素数据，性能就变差了，这是基本劣势。

Get方法：

public E get(int index) {

rangeCheck(index);

return elementData(index);

}

//直接通过数组下标获取，速度很快

@SuppressWarnings("unchecked")

E elementData(int index) {

return (E) elementData[index];

}

效率：对于指定下标的查找，时间复杂度为O(1)；通过给定值进行查找，需要遍历数组，逐一比对给定关键字和数组元素，时间复杂度为O(n)。

注：数组进行扩容时，会将老数组中的元素重新拷贝一份到新的数组中，每次数组容量的增长都会进行判断，最低都会增长1.5，如果容量过大，则会赋值最大的Interage值。这种操作的代价是很高的，

因此在实际使用时，我们应该尽量避免数组容量的扩张。当我们可预知要保存的元素的多少时，要在构造ArrayList实例时，就指定其容量，以避免数组扩容的发生。或者根据实际需求，通过调用ensureCapacity方法来手动增加ArrayList实例的容量。

1. **LinkedList**

LinkedList基于双向链表实现，可以作为栈、队列或者双端队列使用。

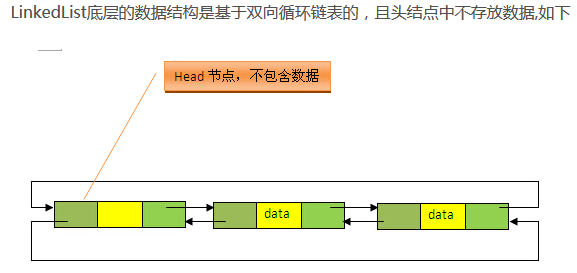
LinkedList继承了AbstractSequentialList，实现了get等方法；

LinkedList实现了Deque接口，可以作为双端队列使用；

LinkedList实现Cloneable接口重写了接口定义的clone()方法，可以使用clone()复制链表。

LinkedList实现 java.io.Serializable接口使LinkedList支持序列化。

LinkedList是个双向循环列表



原理：

LinkedList中之定义了两个属性：

private transient Entry<E> header = new Entry<E>(null, null, null);

private transient int size = 0;

header是双向链表的头节点，它是双向链表节点所对应的类Entry的实例。Entry中包含成员变量： previous, next, element。其中，previous是该节点的上一个节点，next是该节点的下一个节点，element是该节点所包含的值。

size是双向链表中节点实例的个数。

Entry类代码:

private static class Entry<E> {

E element;

Entry<E> next;

Entry<E> previous;

Entry(E element, Entry<E> next, Entry<E> previous) {

this.element = element;

this.next = next;

this.previous = previous;

}

}

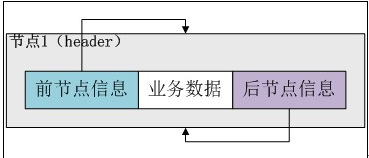
LinkedList提供两个构造方法：

第一个构造方法不接受参数，将header实例的previous和next全部指向header实例（注意，这个是一个双向循环链表，如果不是循环链表，空链表的情况应该是header节点的前一节点和后一节点均为null），这样整个链表其实就只有header一个节点，用于表示一个空的链表。执行完构造函数后，header实例自身形成一个闭环。

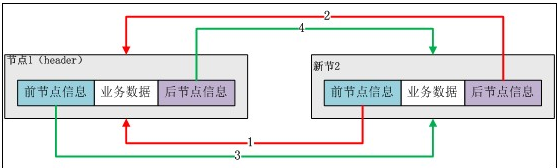
第二个构造方法接收一个Collection参数c，调用第一个构造方法构造一个空的链表，之后通过addAll将c中的元素全部添加到链表中。

添加节点后示意图：

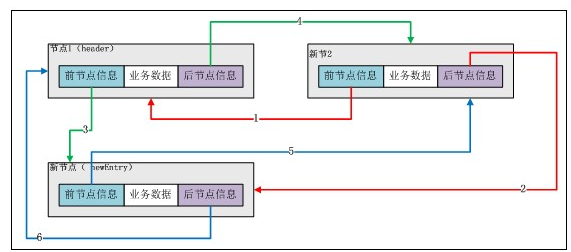
初始化



添加一个节点



添加第二个节点



删除：

由于删除了某一节点因此调整相应节点的前后指针信息，如下：

e.previous.next = e.next;//预删除节点的前一节点的后指针指向预删除节点的后一个节点。

e.next.previous = e.previous;//预删除节点的后一节点的前指针指向预删除节点的前一个节点。

清空预删除节点：

e.next = e.previous = null;

e.element = null;

交给gc完成资源回收，删除操作结束。

查找：

public E get(int index) {

checkElementIndex(index);

return node(index).item;

}

Node<E> node(int index) {

// assert isElementIndex(index);

// 若index < 双向链表长度的1/2,则从前先后查找;

// 否则，从后向前查找。

if (index < (size >> 1)) {

Node<E> x = first;

for (int i = 0; i < index; i++)

x = x.next;

return x;

} else {

Node<E> x = last;

for (int i = size - 1; i > index; i--)

x = x.prev;

return x;

}

}

效率：

随机插入和删除操作比 ArrayList 更加高效，由于其为基于链表的，所以随机访问的效率要比 ArrayList 差。

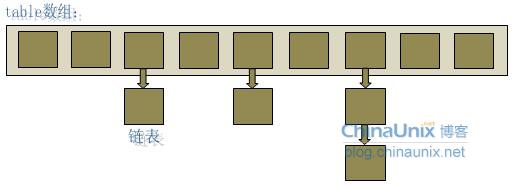
与ArrayList比较而言，LinkedList的删除动作不需要“移动”很多数据，从而效率更高。

遍历速度慢

1. **HashMap**

HashMap是基于哈希表的Map接口的非同步实现。此实现提供所有可选的映射操作，并允许使用null值和null键。此类不保证映射的顺序，特别是它不保证该顺序恒久不变。

HashMap实际上是一个“链表散列”的数据结构，即数组和链表的结合体。



HashMap的功能是通过“键(key)”能够快速的找到“值”。下面我们分析下HashMap存数据的基本流程：

1、 当调用put(key,value)时，首先获取key的hashcode，int hash = key.hashCode();

2、 再把hash通过一下运算得到一个int h.

hash ^= (hash >>> 20) ^ (hash >>> 12);

int h = hash ^ (hash >>> 7) ^ (hash >>> 4);

为什么要经过这样的运算呢？这就是HashMap的高明之处。先看个例子，一个十进制数32768(二进制1000 0000 0000 0000)，经过上述公式运算之后的结果是35080(二进制1000 1001 0000 1000)。看出来了吗？或许这样还看不出什么，再举个数字61440(二进制1111 0000 0000 0000)，运算结果是65263(二进制1111 1110 1110 1111)，现在应该很明显了，它的目的是让“1”变的均匀一点，散列的本意就是要尽量均匀分布。那这样有什么意义呢？看第3步。

3、 得到h之后，把h与HashMap的承载量（HashMap的默认承载量length是16，可以自动变长。在构造HashMap的时候也可以指定一个长 度。这个承载量就是上图所描述的数组的长度。）进行逻辑与运算，即 h & (length-1)，这样得到的结果就是一个比length小的正数，我们把这个值叫做index。其实这个index就是索引将要插入的值在数组中的 位置。第2步那个算法的意义就是希望能够得出均匀的index，这是HashTable的改进，HashTable中的算法只是把key的 hashcode与length相除取余，即hash % length，这样有可能会造成index分布不均匀。还有一点需要说明，HashMap的键可以为null，它的值是放在数组的第一个位置。

4、 我们用table[index]表示已经找到的元素需要存储的位置。先判断该位置上有没有元素（这个元素是HashMap内部定义的一个类Entity， 基本结构它包含三个类，key，value和指向下一个Entity的next）,没有的话就创建一个Entity<K,V>对象，在 table[index]位置上插入，这样插入结束；如果有的话，通过链表的遍历方式去逐个遍历，看看有没有已经存在的key，有的话用新的value替 换老的value；如果没有，则在table[index]插入该Entity，把原来在table[index]位置上的Entity赋值给新的 Entity的next，这样插入结束。

解决哈希冲突方法：开放定址法（发生冲突，继续寻找下一块未被占用的存储地址），再散列函数法，链地址法，而HashMap即是采用了链地址法（也叫拉链法），也就是数组+链表的方式。

扩展问题：

要同时复写equals方法和hashCode方法。

按照散列函数的定义，如果两个对象相同，即obj1.equals(obj2)=true，则它们的hashCode必须相同，但如果两个对象不同，则它们的hashCode不一定不同。

如果两个不同对象的hashCode相同，这种现象称为冲突，冲突会导致操作哈希表的时间开销增大，所以尽量定义好的hashCode()方法，能加快哈希表的操作。

如果相同的对象有不同的hashCode，对哈希表的操作会出现意想不到的结果（期待的get方法返回null），

再回头看看前面提到的为什么覆盖了equals方法之后一定要覆盖hashCode方法，很简单，比如，String a = new String(“abc”);String b = new String(“abc”);如果不覆盖hashCode的话，那么a和b的hashCode就会不同，把这两个类当做key存到HashMap中的话就 会出现问题，就会和key的唯一性相矛盾。

效率：在哈希表中进行添加，删除，查找等操作，性能十分之高，不考虑哈希冲突的情况下，仅需一次定位即可完成，时间复杂度为O(1)

有冲突的情况：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，其时间复杂度依然为O(1)，因为最新的Entry会插入链表头部，只需要简单改变引用链即可，而对于查找操作来讲，此时就需要遍历链表，然后通过key对象的equals方法逐一比对查找。

加载因子：hashmap默认的长度是16，加载因子是0.75，加载因子主要是为了减少冲突而设置为0.75，这是经过验证的。0.75表示hashmap有75%的地址已经存放了数据，这时就需要hashmap进行扩容，扩容过程跟AarryList差不多，这个开销是很大的。因此如果能够预估大小，最好设置map大小，减少扩容，加载因子也不能太大，太大容易哈希冲突

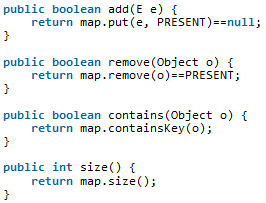
1. **HashSet**

HashSet是基于HashMap来实现的，操作很简单，更像是对HashMap做了一次“封装”，而且只使用了HashMap的key来实现各种特性。

HashSet实现了Set接口，它不允许集合中出现重复元素。当我们提到HashSet时，第一件事就是在将对象存储在HashSet之前，要确保重写hashCode（）方法和equals（）方法，这样才能比较对象的值是否相等，确保集合中没有储存相同的对象。

源码：





1. **HashTable**

和[HashMap](http://www.cnblogs.com/skywang12345/p/3310835.html)一样，Hashtable 也是一个散列表，它存储的内容是键值对(key-value)映射。Hashtable 继承于Dictionary，实现了Map、Cloneable、java.io.Serializable接口。Hashtable 的函数都是同步的，这意味着它是线程安全的。它的key、value都不可以为null。此外，Hashtable中的映射不是有序的。

Hashtable 的实例有两个参数影响其性能：初始容量 和 加载因子。容量 是哈希表中桶的数量，初始容量就是哈希表创建时的容量。注意，哈希表的状态为 open：在发生“哈希冲突”的情况下，单个桶会存储多个条目，这些条目必须按顺序搜索。加载因子是对哈希表在其容量自动增加之前可以达到多满的一个尺度。初始容量和加载因子这两个参数只是对该实现的提示。关于何时以及是否调用 rehash 方法的具体细节则依赖于该实现。

通常，默认加载因子是 0.75, 这是在时间和空间成本上寻求一种折衷。加载因子过高虽然减少了空间开销，但同时也增加了查找某个条目的时间，这点跟hashmap是一样的。

Hashtable也是通过"拉链法"实现的哈希表，这点与hashmap类似

Hashtable实现同步主要是在方法上加了synchronized

Hashtable 与 HashMap 的简单比较：

1. HashTable 基于 Dictionary 类，而 HashMap 是基于 AbstractMap。Dictionary 是任何可将键映射到相应值的类的抽象父类，而 AbstractMap 是基于 Map 接口的实现，它以最大限度地减少实现此接口所需的工作。

2. HashMap 的 key和value都允许为null，而 Hashtable 的 key 和 value 都不允许为 null。HashMap 遇到 key 为 null 的时候，调用 putForNullKey 方法进行处理，而对 value 没有处理；Hashtable遇到 null，直接返回 NullPointerException。

3. Hashtable 方法是同步，而HashMap则不是。我们可以看一下源码，Hashtable 中的几乎所有的 public 的方法都是 synchronized 的，而有些方法也是在内部通过 synchronized 代码块来实现。所以有人一般都建议如果是涉及到多线程同步时采用 HashTable，没有涉及就采用 HashMap，但是在 Collections 类中存在一个静态方法：synchronizedMap()，该方法创建了一个线程安全的Map对象，并把它作为一个封装的对象来返回。也可以用ConcurrentHashMap，ConcurrentHashMap更高效。

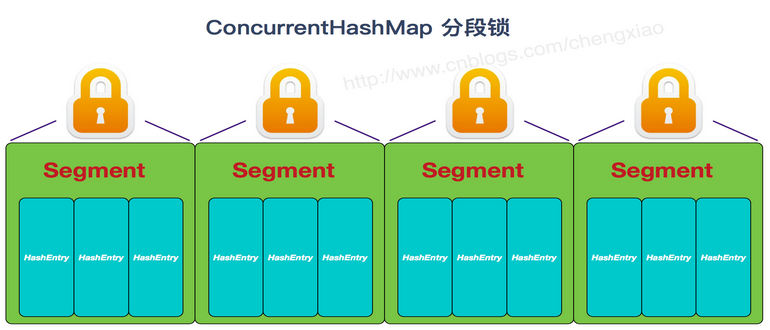
1. **ConcurrentHashMap**

ConcurrentHashMap是Java并发包中提供的一个线程安全且高效的HashMap实现，ConcurrentHashMap在并发编程的场景中使用频率非常之高。

原理：

数据结构 ConcurrentHashMap的目标是实现支持高并发、高吞吐量的线程安全的HashMap。当然不能直接对整个hashtable加锁，所以在ConcurrentHashMap中，数据的组织结构和HashMap有所区别。

一个ConcurrentHashMap由多个segment组成，每一个segment都包含了一个HashEntry数组的hashtable， 每一个segment包含了对自己的hashtable的操作，比如get，put，replace等操作，这些操作发生的时候，对自己的hashtable进行锁定。由于每一个segment写操作只锁定自己的hashtable，所以可能存在多个线程同时写的情况，性能无疑好于只有一个hashtable锁定的情况（**分段锁**）。

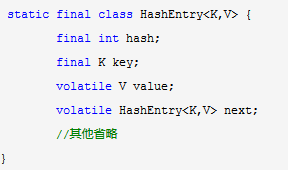
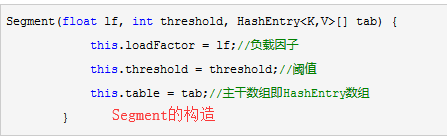


ConcurrentHashMap的主干是个Segment数组。

final Segment<K,V>[] segments;

Segment继承了ReentrantLock，所以它就是一种可重入锁（ReentrantLock)。在ConcurrentHashMap，一个Segment就是一个子哈希表，Segment里维护了一个HashEntry数组，并发环境下，对于不同Segment的数据进行操作是不用考虑锁竞争的。

所以，对于同一个Segment的操作才需考虑线程同步，不同的Segment则无需考虑。Segment类似于HashTable，一个Segment维护着一个HashEntry数组。HashEntry是目前我们提到的最小的逻辑处理单元了。一个ConcurrentHashMap维护一个Segment数组，一个Segment维护一个HashEntry数组。

Segment类似哈希表，那么一些属性就跟我们之前提到的HashMap差不离，比如负载因子loadFactor，比如阈值threshold等等。

ConcurrentHashMap的构造方法，初始化方法有三个参数，如果用户不指定则会使用默认值，initialCapacity为16，loadFactor为0.75（负载因子，扩容时需要参考），concurrentLevel为16。

Put:

1.定位segment并确保定位的Segment已初始化

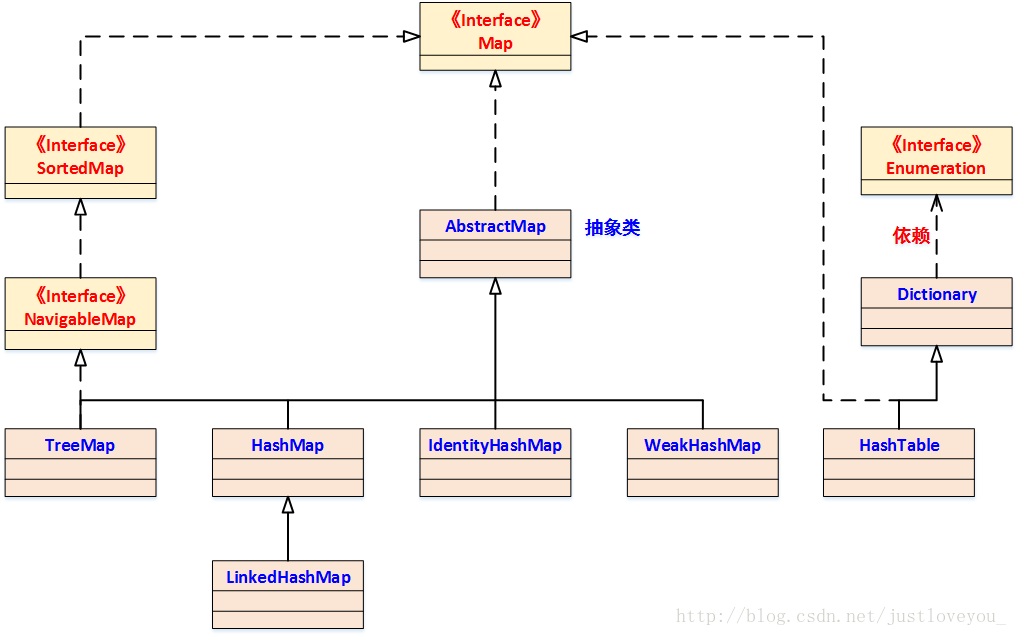
2.调用Segment的put方法

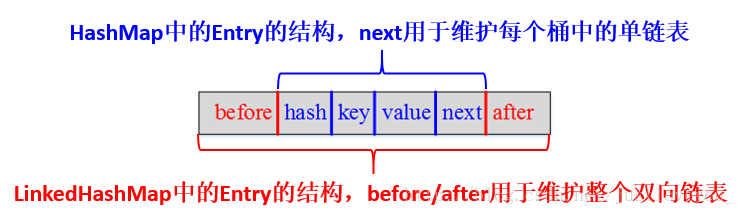
1. **LinkedHashMap**

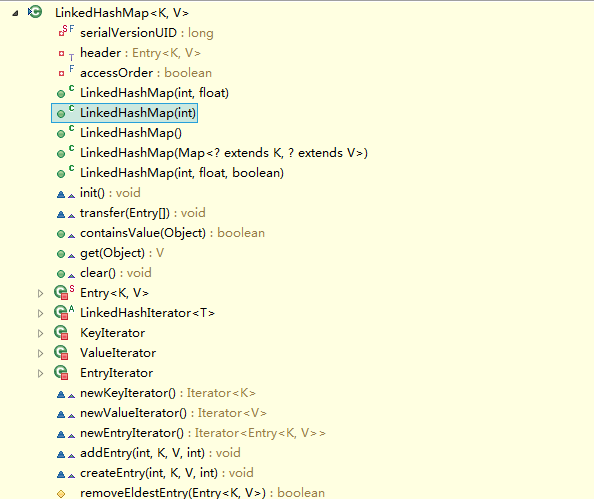
HashMap和双向链表合二为一即是LinkedHashMap。所谓LinkedHashMap，其落脚点在HashMap，因此更准确地说，它是一个将所有Entry节点链入一个双向链表的HashMap。由于LinkedHashMap是HashMap的子类，所以LinkedHashMap自然会拥有HashMap的所有特性。比如，LinkedHashMap的元素存取过程基本与HashMap基本类似，只是在细节实现上稍有不同。当然，这是由LinkedHashMap本身的特性所决定的，因为它额外维护了一个双向链表用于保持迭代顺序。

LinkedHashMap是有序的，线程不安全的。

HashMap是无序的，也就是说，迭代HashMap所得到的元素顺序并不是它们最初放置到HashMap的顺序。HashMap的这一缺点往往会造成诸多不便，因为在有些场景中，我们确需要用到一个可以保持插入顺序的Map。庆幸的是，JDK为我们解决了这个问题，它为HashMap提供了一个子类 —— LinkedHashMap。虽然LinkedHashMap增加了时间和空间上的开销，但是它通过维护一个额外的双向链表保证了迭代顺序。特别地，该迭代顺序可以是插入顺序，也可以是访问顺序。因此，根据链表中元素的顺序可以将LinkedHashMap分为：保持插入顺序的LinkedHashMap 和 保持访问顺序的LinkedHashMap，其中LinkedHashMap的默认实现是按插入顺序排序的。







看到LinkedHashMap中并没有什么操作数据结构的方法，也就是说LinkedHashMap操作数据结构（比如put一个数据），和HashMap操作数据的方法完全一样，无非就是细节上有一些的不同罢了。

1. **TreeMap**

TreeMap 是一个有序的key-value集合，它是通过红黑树（R-B-tree）实现的。

TreeMap 继承于AbstractMap，所以它是一个Map，即一个key-value集合。

TreeMap 实现了NavigableMap接口，意味着它支持一系列的导航方法。比如返回有序的key集合。

TreeMap 实现了Cloneable接口，意味着它能被克隆。

TreeMap 实现了java.io.Serializable接口，意味着它支持序列化。

TreeMap基于红黑树（Red-Black tree）实现。该映射根据其键的自然顺序进行排序，或者根据创建映射时提供的 Comparator 进行排序，具体取决于使用的构造方法。

TreeMap的基本操作 containsKey、get、put 和 remove 的时间复杂度是 log(n) 。

另外，TreeMap是非同步的。 它的iterator 方法返回的迭代器是fail-fastl的。

1. **Volatile**

Java语言提供了一种稍弱的同步机制，即volatile变量，用来确保将变量的更新操作通知到其他线程。当把变量声明为volatile类型后，编译器与运行时都会注意到这个变量是共享的，因此不会将该变量上的操作与其他内存操作一起重排序。volatile变量不会被缓存在寄存器或者对其他处理器不可见的地方，因此在读取volatile类型的变量时总会返回最新写入的值。

volatile修饰的变量不允许线程内部缓存和重排序，即直接修改内存。所以对其他线程是可见的。但是这里需要注意一个问题，volatile只能让被他修饰内容具有可见性，但不能保证它具有原子性。比如 volatile int a = 0；之后有一个操作 a++；这个变量a具有可见性，但是a++ 依然是一个非原子操作，也就是这个操作同样存在线程安全问题。

Java 语言提供了 volatile 和 synchronized 两个关键字来保证线程之间操作的有序性，volatile 是因为其本身包含“禁止指令重排序”的语义，synchronized 是由“一个变量在同一个时刻只允许一条线程对其进行 lock 操作”这条规则获得的，此规则决定了持有同一个对象锁的两个同步块只能串行执行。

在访问volatile变量时不会执行加锁操作，因此也就不会使执行线程阻塞，因此volatile变量是一种比sychronized关键字更轻量级的同步机制。

当对非 volatile 变量进行读写的时候，每个线程先从内存拷贝变量到CPU缓存中。如果计算机有多个CPU，每个线程可能在不同的CPU上被处理，这意味着每个线程可以拷贝到不同的 CPU cache 中。

而声明变量是 volatile 的，JVM 保证了每次读变量都从内存中读，跳过 CPU cache 这一步。

当一个变量定义为 volatile 之后，将具备两种特性：

1.保证此变量对所有的线程的可见性

2.禁止指令重排序优化。

性能：volatile 的读性能消耗与普通变量几乎相同，但是写操作稍慢，因为它需要在本地代码中插入许多内存屏障指令来保证处理器不发生乱序执行。

1. **JavaBean的创建规范**

我们在创建一个JavaBean的时候，一般情况下会写一个默认的空的构造函数以及一个带参的构造函数。空的构造函数用于以后操作的方便，特别是用到反射进行对象的创建的时候。

1. **回调函数的概念**

**所谓的回调，就是程序员A写了一段程序（程序a），其中预留有回调函数接口，并封装好了该程序。程序员B要让a调用自己的程序b中的一个方法，于是，他通过a中的接口回调自己b中的方法。**

下面是例子。

首先定义一个类Caller，按照上面的定义就是程序员A写的程序a，这个类里面保存一个接口引用。

public class Caller {

private MyCallInterface callInterface;

public Caller() {

}

//注册回调函数接口

public void setCallFunc(MyCallInterface callInterface) {

this.callInterface = callInterface;

}

public void call() {

//调用回调函数

callInterface.printName();

}

}

当然需要接口的定义，为了方便程序员B根据我的定义编写程序实现接口。

public interface MyCallInterface {

public void printName();

}

第三是定义程序员B写的程序b

public class Client implements MyCallInterface {

//回调函数

@Override

public void printName() {

System.*out*.println("This is the client printName method");

}

}

测试如下

public class Test {

public static void main(String[] args) {

Caller caller = new Caller();

caller.setCallFunc(new Client());

caller.call();

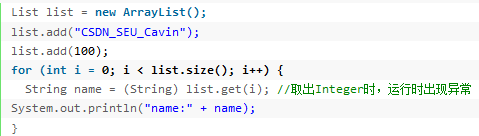
}

}

1. **泛型**

泛型，即“参数化类型”。一提到参数，最熟悉的就是定义方法时有形参，然后调用此方法时传递实参。那么参数化类型怎么理解呢？顾名思义，就是将类型由原来的具体的类型参数化，类似于方法中的变量参数，此时类型也定义成参数形式（可以称之为类型形参），然后在使用/调用时传入具体的类型（类型实参）。

泛型的本质是为了参数化类型（在不创建新的类型的情况下，通过泛型指定的不同类型来控制形参具体限制的类型）。也就是说在泛型使用过程中，操作的数据类型被指定为一个参数，这种参数类型可以用在类、接口和方法中，分别被称为泛型类、泛型接口、泛型方法。



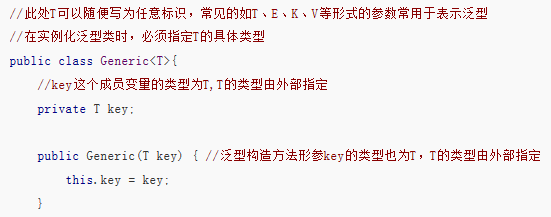
本例向list类型集合中加入了一个字符串类型的值和一个Integer类型的值。（这样合法，因为此时list默认的类型为Object类型）。在之后的循环中，由于忘记了之前在list中也加入了Integer类型的值或其他原因，运行时会出现java.lang.ClassCastException异常。为了解决这个问题，泛型应运而生。

Java泛型编程是JDK1.5版本后引入的。泛型让编程人员能够使用类型抽象，通常用于集合里面。只要在上例中将第1行代码改成如下形式，那么就会在编译list.add(100)时报错。

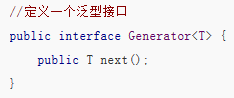
List<String> list = new ArrayList<String>();

通过List<String>，直接限定了list集合中只能含有String类型的元素，从而在上例中的第6行中，无须进行强制类型转换，因为集合能够记住其中元素的类型信息，编译器已经能够确认它是String类型了。

泛型类：泛型类用于类的定义中，被称为泛型类。通过泛型可以完成对一组类的操作对外开放相同的接口。最典型的就是各种容器类，如：List、Set、Map。



泛型接口：泛型接口与泛型类的定义及使用基本相同。泛型接口常被用在各种类的生产器中。



通配符：

List<Integer> ex\_int= new ArrayList<Integer>();

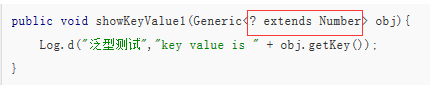
List<Number> ex\_num = ex\_int; //非法的

上述第2行会出现编译错误，因为Integer虽然是Number的子类，但List<Integer>不是List<Number>的子类型。

我们需要一个在逻辑上可以用来同时表示为List<Integer>和List<Number>的父类的一个引用类型，类型通配符应运而生。在本例中表示为List<?>即可。

类型通配符一般是使用？代替具体的类型实参，注意了，**此处’？’是类型实参，而不是类型形参** 。再直白点的意思就是，此处的？和Number、String、Integer一样都是一种实际的类型，可以把？看成所有类型的父类。是一种真实的类型。

**上下边界：**在使用泛型的时候，我们还可以为传入的泛型类型实参进行上下边界的限制，如：类型实参只准传入某种类型的父类或某种类型的子类。



泛型数组：不能创建一个确切的泛型类型的数组

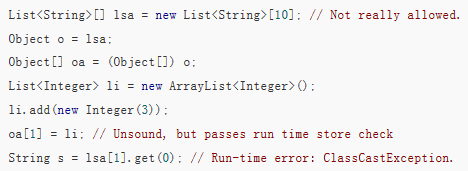
List<String>[] ls = new ArrayList<String>[10]; 不可以

下面是可以的

List<?>[] ls = new ArrayList<?>[10];

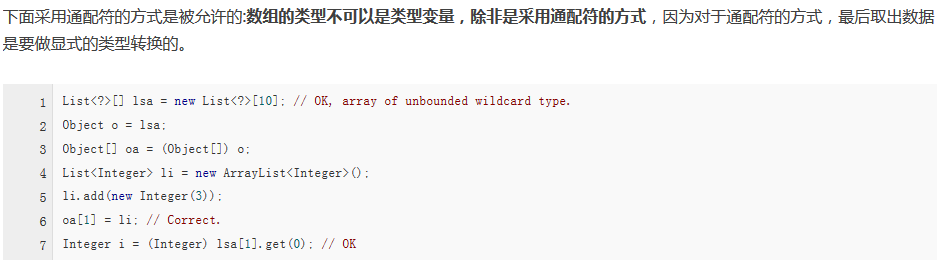
List<String>[] ls = new ArrayList[10];

下面使用Sun的一篇文档的一个例子来说明这个问题：



这种情况下，由于JVM泛型的擦除机制，在运行时JVM是不知道泛型信息的，所以可以给oa[1]赋上一个ArrayList而不会出现异常，但是在取出数据的时候却要做一次类型转换，所以就会出现ClassCastException，如果可以进行泛型数组的声明，上面说的这种情况在编译期将不会出现任何的警告和错误，只有在运行时才会出错。

而对泛型数组的声明进行限制，对于这样的情况，可以在编译期提示代码有类型安全问题，比没有任何提示要强很多。



泛型只在编译阶段有效

1. **反射案例**

反射概念：JAVA反射机制是在运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意一个方法和属性；这种动态获取的信息以及动态调用对象的方法的功能称为java语言的反射机制。

概念解释：java代码一般要进行编译成class文件才能运行，如果我们写代码时想让程序在运行时动态的去获取class文件中的类（jar包里面的类），这个时候就相当于是获取某个类的字节码，这个时候就要用到反射。

package com.gh.enums;

import java.util.ArrayList;

public class ClassTest {

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

//获得字节码的第一种方法 cn.itcast.reflect.HelloWorld

Class clazz = Class.*forName*("cn.itcast.reflect.HelloWorld");

System.*out*.println(clazz);

//已知类，获取字节码的第二种方法

Class clazz2 = String.class;

System.*out*.println(clazz2);

//已知对象 获取字节码的第三种方法

Object obj = new ArrayList();

Class clazz3 = obj.getClass();

System.*out*.println(clazz3);

}

}

案例1：见百度云盘->技术资料->java->java源码->day03\_reflect.zip

1. **单例模式**

package com.test.singleton;

/\*\*

\* 测试单例模式

\* @author Administrator

\*

\*/

public class SingleTon {

public static void main(String[] args) {

System.*out*.println(EagerSingleton.*getInstance*().toString());

System.*out*.println(EagerSingleton.*getInstance*().toString());

System.*out*.println(LazySingleton.*getInstance*().toString());

System.*out*.println(LazySingleton.*getInstance*().toString());

}

}

/\*\*

\* 饿汉式单例，不存在多线程同步问题

\* @author Administrator

\*

\*/

class EagerSingleton{

private static EagerSingleton *instance* = new EagerSingleton();

/\*\*

\* 私有化构造函数，不让用户通过构造函数实例化该类

\*/

private EagerSingleton(){}

public static EagerSingleton getInstance(){

return *instance*;

}

}

/\*\*

\* 懒汉式单例，在这里用synchronized解决多线程同步问题

\* @author Administrator

\*

\*/

class LazySingleton{

private static LazySingleton *instance* = null;

/\*\*

\* 私有化构造函数，不让用户通过构造函数实例化该类

\*/

private LazySingleton(){}

/\*\*

\* synchronized解决线程同步访问问题，比如多线程同步调用该方法的时候，一个线程进来发现instance为null 然后阻塞了，第二个线程进来发现instance也为null

\* 因此创建了一个instance实例，完了以后第一个线程就绪运行，又会创建一个instance实例。在这里用synchronized就可以解决该问题

\* @return

\*/

public static synchronized LazySingleton getInstance(){

if( *instance* == null ){

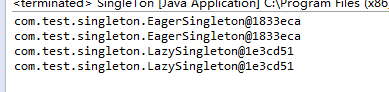
*instance* = new LazySingleton();

}

return *instance*;

}

}



1. **观察者模式**

/\*\*

\* 观察者设计模式示例

\* 第一步：实现一个需要被监听的类Person.

\* 第二步：实现一个监听接口PersonListener。

\* 第三步：在Person类中，提供一个方法用于注册PersonListener类，即registerListener

\* 第四步：必须要在Person类中维护PersonListener类的实例。

\* 第五步：在调用person.eat方法是，判断PersonListener是否为null,如果不为null则调用它的eating方法。

\* 第六步：在Main类中，实例化Person，并注册一个监听。

\*

\* @author wangjianme

\*/

public class Main {

public static void main(String[] args) {

final Person tom = new Person();

tom.registerListener(new PersonListener() {

public void eating() {

System.*err*.println("真想吃死呀：");

}

});

tom.eat();

}

}

/\*\*

\* 被监听的类,必须提供注册监听器的方法

\* @author wangjianme

\*/

class Person{

private PersonListener pl; //记录监听器对像

public void eat(){

if(pl!=null){

pl.eating();

}

System.*err*.println("吃个死");

}

/\*\*

\* 注册监听的方法

\*/

public void registerListener(PersonListener pl){

this.pl=pl;

}

}

/\*\*

\* 再设计一个监听器的接口，此接口需要用户加以实现

\*/

interface PersonListener{

void eating(); //提供一个用户必须实现的方法

}

1. **观察者模式添加事件源**

第一步：在前页的基础上继续添加一个PersonEvent类(注意我说是类不是接口)，代表事件对象。

第二步：给PersonEvent对象，添加一个Object属性，用以标识事件源对象。

第三步：修改PersonListener接口的eating方法，让它接收一个PersonEvent参数。

第四步：在Person类eat方法中，如果判断PersonListener属性不为空，则在调用eating方法，实例化PersonEvent并传给eating方法。

第五步：在main方法中，通过PersonEvent的getSource方法测试是否是同一个对像。

/\*\*

\* 观察者设计模式2

\* 添加事件源对像

\*/

public class ObserverDemo2 {

public static void main(String[] args) {

Person2 p = new Person2(); //初始化Person2对象

System.*err*.println(">>:"+p);

p.setListener(new Person2Listener() { //给Person2对象添加监听

public void eating(PersonEvent e) {

Person2 p = (Person2)e.getSource(); //获取数据源对象

System.*err*.println("事件源是："+p);

}

});

p.eat();

}

}

/\*\*

\* 需要被监听的对像

\* @author wangjianme

\*/

class Person2{

private Person2Listener listener;

public void eat(){

if(listener!=null){

/\*\*

\* 如果已经注册了监听器则在调用时传递事件源本身

\*/

listener.eating(new PersonEvent(this));

}

System.*err*.println("吃死一批");

}

/\*\*

\* 给Person2添加注册事件源的方法

\*/

public void setListener(Person2Listener pl){

this.listener=pl;

}

}

/\*\*

\* 事件监听者

\*/

interface Person2Listener{

/\*\*

\* 接收一个事件对像

\*/

public void eating(PersonEvent e);

}

/\*\*

\* 事件对像，维护事件源

\*/

class PersonEvent{

private Object source; //内部维护一个事件源

public PersonEvent(Object t){ //在构造时给它赋值

this.source=t;

}

public Object getSource() { //提供获取事件源的方法

return source;

}

}

1. **序列化对象存储与解序列化对象**

//文件输出流，到目的地student.txt文件，此处我们创建了一个缓冲流，这样更有效率，一般推荐使用缓冲流。如果文件不存在，那么则创建一个

BufferedOutputStream bos = new BufferedOutputStream(new FileOutputStream("student.txt"));

//对象输出流,此时的对象Student必须序列化

ObjectOutputStream os = new ObjectOutputStream(bos);

Student stu = new Student("gh", 11, "成都");

//将序列化的对象写到输出流中

os.writeObject(stu);

//关闭对象输出流

os.close();

//文件输入流，到文件student.txt文件去读取

BufferedInputStream bis = new BufferedInputStream(new FileInputStream("student.txt"));

//对象输入流

ObjectInputStream is = new ObjectInputStream(bis);

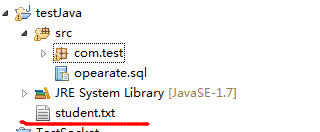
//将读取的对象转化为我们需要的对象

Student s = (Student) is.readObject();

//关闭对象输入流

is.close();

System.*out*.println(s);



具体代码见：百度云盘->技术资料->java->java源码->对象序列化与解序列化

1. **自动装箱和自动拆箱——java 1.5及以上支持**

概念：基本数据类型（int,double,short,long,float等等）与包装类间的转换

package com.gh.box;

public class TestBox {

public static void main(String[] args) {

// 5.0

//装箱，基本数据类型装箱成包装类

Integer i = 10;

//拆箱，包装类自动拆箱为基本数据类型

int j = i;

System.*out*.println(i + " : " + j);

// 1.4，为了兼容1.4及以下版本，一般选择下面这种写法

Integer ii = new Integer(20);

int jj = ii.intValue();

System.*out*.println(ii + " : " + jj);

Double d = new Double(2.0);

double b = d.doubleValue();

System.*out*.println("\*\*\*\*\*\*\*\*\*\*\*");

*print*(3.0f);

/\*

\* 3 int -- 1 int

\* 3 int -- 注释第一个 print(int i) --【 short -- int -- long -- float -- double】 -- 3 double

\* 3 int -- 注释 1 和 3 -- 2 Integer（自动装箱）

\* 3 int -- 注释 1、2和3 --抛错

\* \*/

}

public static void print(int i){

System.*out*.println("1 int");

}

public static void print(Integer i){

System.*out*.println("2 Integer");

}

public static void print(double i){

System.*out*.println("3 double");

}

public static void print(Double i){

System.*out*.println("4 Double");

}

public static void print(float i){

System.*out*.println("5 float");

}

}

代码见百度云盘->技术资料->java->源码->自动拆箱和装箱

1. **可变参数**

package com.gh.box;

public class Variable {

public static void main(String[] args) {

//Integer[] ins = {1,2,3,4,5};

//sum(ins);

//sum();

*sum*(1,2,3,4,5);

}

/\* 可变参数的格式：类型 ... 变量名

\* 使用：

\* \* 在方法体内，形参可变参数，将被当成数组使用

\* \* 实际参数的个数，将是形参可变参数，在方法体内使用的数组的长度

\* \* 实际参数是数组，数组将会被打散

\* \* 总结：可变参数只能放置在方法参数列表的最后一位

\* \* 一个方法中能否具有两个可变参数？不行的

\*

\*/

// public static void sum(Integer... ins,int s){}

// 该方法错误，不能这样写，可变参数一定要作为最后的参数。

public static void sum(int s,Integer... ins){ // m[5]

int sum = 0;

for(int i = 0 ; i < ins.length ; i++){

sum = sum + ins[i];

// sum += ins[i];

}

System.*out*.println(sum);

}

/\*\*

\* public static void sum(Integer[] ins){ }在方法体内，形参可变参数，将被当成数组使用，像这个方法的声明与下面

\* 这个方法的声明有冲突，就是因为形参被当成数组

\* public static void sum(Integer。。。 ins){ }

\*/

}

1. **抽象类的new，一般情况不能直接new抽象类，不过可以使用内部类**

package com.gh.pattern;

public abstract class Gender {

/\*\*

\* new一个抽象类Gender是不可以的，不过我们可以通过内部类来实现

\*/

public static Gender *male* = new Gender("男"){

//这就是一个内部类

public String getAge(){

return "25";

}

};

public static Gender *female* = new Gender("女"){

public String getAge() {

return "23";

}

};

private String value;

private Gender(String val){

this.value = val;

}

public String getValue(){

return this.value;

}

public abstract String getAge();

}

package com.gh.pattern;

public class GenderTest {

public static void main(String[] args) {

Gender male = Gender.*male*;

Gender female = Gender.*female*;

System.*out*.println(male.getAge());

System.*out*.println(female.getAge());

}

}

1. **枚举，多例一般用枚举实现**

概念：一些方法在运行时，它需要的数据不能是任意的，而必须是一定范围内的值，因此需要枚举。

特点：枚举也是一个类，枚举的实例默认是public static final的，枚举的构造方法默认是私有的。枚举的实例变量名，必须放在所有内容的最前面。意思就是声明枚举变量的前面不能存在方法或者其他什么的。

格式：

public enum Gender {

/\*\*

\* 在枚举变量名声明前不能存在方法声明或者其他的

\*/

*male*("男"),*female*("女");

private String value;

private Gender(String val){

this.value = val;

}

public String getValue(){

return this.value;

}

}

用法：

package com.gh.pattern;

public class GenderTest {

public static void main(String[] args) {

Gender male = Gender.*male*;

Gender female = Gender.*female*;

System.*out*.println(male.getAge());

System.*out*.println(female.getAge());

}

}

1. **关于this**

规则1：如果是属性则各是各的

规则2：如果是方法则是调用者的

注意以下代码的结果

public class Parent {

public String name="tom";

public void init() {

System.*out*.println("1 init parent");

this.demo();

System.*out*.println(this.name);

}

public void demo() {

System.*out*.println("2 demo parent");

}

}

public class Son extends Parent {

public String name="jack";

public void init(){

super.init();

System.*out*.println("3 init son");

this.demo();

System.*out*.println(this.name);

}

public void demo() {

System.*out*.println("4 demo Son");

}

public static void main(String[] args) {

//当前运行类 Son

Son son = new Son();

son.init();

/\*\*

\* 运行结果是

\* 1 init parent

\* 4 demo Son

\* tom

\* 3 init son

\* 4 demo Son

\* jack

\*/

//init(son)

}

}

1. **Java项目的路径**

①File file = new File("");

System.*out*.println(file.getAbsolutePath());

\* 使用java命令，输出路径是，当前java命令停留的盘符

\* C:\Users\guohong\jsp\test\bin

\* 使用myeclipse或eclipse运行时

\* C:\Users\guohong\jsp\test

②//获取当前盘符

File file = new File("/");

System.*out*.println(file.getAbsolutePath());

\* C:\

③获得路径，使用类加载器

URL url = TestPath.class.getClassLoader().getResource("");

System.*out*.println(url);

\* file:/C:/Users/guohong/jsp/test/bin/

④不能使用这个

URL url = TestPath.class.getClassLoader().getResource("/");

System.*out*.println(url);

\* null

1. **产生随机数**

/\*\*

\* 产生随机数字

\*/

public static int rand(){

//定义种子数，意思就是产生0-99,999,999间的随机数

int n = 100000000;

Random rand = new Random();

int randInt = 0;

//产生随机数

randInt = rand.nextInt(n);

return randInt;

}

1. **从字符串中提取数字**

/\*\*

\* 从字符串中解析出数字

\* @param str 需要提取数字的字符串

\* @return 返回数字字符

\*/

private String getNumFromStr(String str){

String regEx="[^0-9]";

Pattern p = Pattern.*compile*(regEx);

Matcher m = p.matcher(str);

return m.replaceAll("").trim();

}

1. **正则判断数字个数范围**

String val = "123456789012345678900";

//数字范围在1-20个之间

String rex = "^\\d{1,20}$";

System.*out*.println(val.matches(rex));

1. **Java中的正则表达式写法注意事项**

①特殊字符的写法

比如|字符在java正则表达式中的写法为image而不是”\|”的写法，这里需要注意，js中的话是后面的写法。

例子：

1. //正则表达式，截取符合EGAAGCBZIAJDEJT|昭觉寺公交站|1路@24f5dad9格式的字符串if(str[i].matches("[A-Z]+\\|[^\\x00-\\xff]+\\|.+"))

1. **Java中克隆对象**

问题描述：比如我们在java对象使用中，有一个全局对象worker，这个全局对象包含许多字段属性，我们写了一个循环想每次为该对象的某些字段（而不是全部字段，原来的字段我们不想再重新设置，只想沿用原来的值）重新设值，然后加入一个list链表中，如果每次循环我们都去重新设置worker对象的值，然后加入list链表，这样问题就来了，你会发现链表中的所有值都是一样的，而且都是最后加进去的worker的值。

解决方法1：使用克隆方法，每次循环的时候，我们都去克隆worker对象，形成一个克隆体，然后每次都去修改克隆体的字段值，最后把克隆体对象加入到list链表中，这样就可以解决上面的问题。

/\*\*

\* 想要该对象可以克隆，那么需要实现Cloneable这个接口

\* @author Administrator

\*

\*/

public class CloneClass implements Cloneable{

/\*\*

\* 复写接口的clone()方法

\*/

@Override

public Object clone(){

CloneClass o = null;

try{

o = (CloneClass)super.clone();

}catch(CloneNotSupportedException e){

e.printStackTrace();

}

return o;

}

}

public class TestChar {

public static void main(String[] args) {

CloneClass cloneClass = new CloneClass();

Cloneable clone = (Cloneable) cloneClass.clone();

//结果为false，说明克隆的对象与本体不是指的同一个对象

System.*out*.println(cloneClass.equals(clone));

CloneClass cloneClass01 = new CloneClass();

CloneClass clone01 = new CloneClass();

clone01 = cloneClass01;

//结果为true,说明这样赋值的两个对象指向的引用是同一个地方，也就是同一个对象

System.*out*.println(cloneClass01.equals(clone01));

}

}

1. **将简单json格式的字符串转为java对象**

方法1：

第一步：新建一个对应json字符串格式的java对象

public class Seller {

/\*\*

\* 注意：以下的字段名称必须与json字符串中的名称一一对应，并且名称要相同

\* 要不然则无法获取到该字段的数据，程序不会报错

\*/

private String trace;

private String url;

private int num;

public Seller() {

}

public Seller(String trace, String url, int num) {

this.trace = trace;

this.url = url;

this.num = num;

}

public String getTrace() {

return trace;

}

public void setTrace(String trace) {

this.trace = trace;

}

public String getUrl() {

return url;

}

public void setUrl(String url) {

this.url = url;

}

public int getNum() {

return num;

}

public void setNum(int num) {

this.num = num;

}

@Override

public String toString() {

return "Seller [trace=" + trace + ", url=" + url + ", num=" + num + "]";

}

}

第二步：将json字符串转化为java对象

import java.util.List;

import net.sf.json.JSONArray;

import com.gh.test.bean.Seller;

public class Test {

public static void main(String[] args) {

String str = "[{'trace':'vertical\_seller','url':'http://localhost:8080/gh\_web','num':89}]";

JSONArray ja = JSONArray.*fromObject*(str);

// Map<String, Class<Property>> propertyMap = new HashMap<String, Class<Property>>();

List<Seller> list = JSONArray.~~toList~~(ja, Seller.class);

for (int i = 0; i < list.size(); i++) {

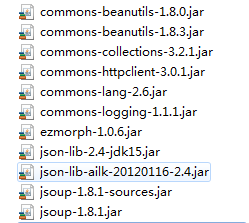
System.*out*.println(list.get(i).getNum());

}

}

}

所依赖的jar包：



方法2：使用阿里巴巴的一个开源框架，fastjson

注：fastjson在字符串的名称为中文时，解析有点问题，建议用第一种方法

第一步和方法1的第一步是一样的。

第二步：

import com.alibaba.fastjson.JSON;

Area area = JSON.parseObject(result, Area.class);

这样就把字符串result转化为Java对象了。

如果是maven工程需要在pom.xml文件的配置：

<dependency>

<groupId>com.alibaba</groupId>

<artifactId>fastjson</artifactId>

<version>1.1.26</version>

</dependency>

1. **将复杂json格式的字符串转为java对象**

第一步：新建一个对应json字符串格式的java对象。

import java.util.List;

public class Seller {

/\*\*

\* 注意：以下的字段名称必须与json字符串中的名称一一对应，并且名称要相同

\* 要不然则无法获取到该字段的数据，程序不会报错

\*/

private String trace;

private String url;

private int num;

private List<Property> property;

public Seller() {

}

public Seller(String trace, String url, int num, List<Property> property) {

this.trace = trace;

this.url = url;

this.num = num;

this.property = property;

}

public String getTrace() {

return trace;

}

public void setTrace(String trace) {

this.trace = trace;

}

public String getUrl() {

return url;

}

public void setUrl(String url) {

this.url = url;

}

public int getNum() {

return num;

}

public void setNum(int num) {

this.num = num;

}

public List<Property> getProperty() {

return property;

}

public void setProperty(List<Property> property) {

this.property = property;

}

@Override

public String toString() {

return "Seller [trace=" + trace + ", url=" + url + ", num=" + num

+ ", property=" + property + "]";

}

}

package com.gh.test.bean;

public class Property {

private String pname;

private String vname;

private String wordIcon;

public Property() {

}

public Property(String pname, String vname, String wordIcon) {

this.pname = pname;

this.vname = vname;

this.wordIcon = wordIcon;

}

public String getPname() {

return pname;

}

public void setPname(String pname) {

this.pname = pname;

}

public String getVname() {

return vname;

}

public void setVname(String vname) {

this.vname = vname;

}

public String getWordIcon() {

return wordIcon;

}

public void setWordIcon(String wordIcon) {

this.wordIcon = wordIcon;

}

public String toString() {

return "Property [pname=" + pname + ", vname=" + vname + ", wordIcon="

+ wordIcon + "]";

}

}

第二步：解析

import java.util.HashMap;

import java.util.List;

import java.util.Map;

import net.sf.json.JSONArray;

import com.gh.test.bean.Property;

import com.gh.test.bean.Seller;

public class Test {

public static void main(String[] args) {

String str = "["

+ "{"

+ "'trace':'vertical\_seller',"

+ "'url':'http://localhost:8080/gh\_web',"

+ "'num':89,"

+ "property:"

+ "[{"

+ "'pname':'abcd',"

+ "'vname':'efgh',"

+ "wordIcon:'图标'"

+ "}]"

+ "}"

+ "]";

JSONArray ja = JSONArray.*fromObject*(str);

Map<String, Class<Property>> propertyMap = new HashMap<String, Class<Property>>();

propertyMap.put("property", Property.class);

List<Seller> list = JSONArray.~~toList~~(ja, Seller.class, propertyMap);

for (int i = 0; i < list.size(); i++) {

System.*out*.println(list.get(i));

}

}

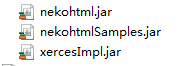
}

运行结果：

Seller [trace=vertical\_seller, url=http://localhost:8080/gh\_web, num=89, property=[Property [pname=abcd, vname=efgh, wordIcon=图标]]]

1. **使用Nekohtml解析html文档**

依赖的jar包：



实例：

import java.io.IOException;

import org.apache.html.dom.HTMLDocumentImpl;

import org.cyberneko.html.parsers.DOMFragmentParser;

import org.w3c.dom.DocumentFragment;

import org.w3c.dom.Node;

import org.w3c.dom.html.HTMLDocument;

import org.xml.sax.SAXException;

public class TestBrand {

public static void main(String[] args) {

try {

getBrand();

} catch (SAXException e) {

e.printStackTrace();

} catch (IOException e) {

e.printStackTrace();

}

}

public static void print(Node node, String indent) {

//如果是文本节点，则打印出文本节点的信息

if (node.getNodeType() == Node.*TEXT\_NODE*) {

System.*out*.println(indent + node.getNodeValue());

}

Node child = node.getFirstChild();

while (child != null) {

*print*(child, indent + " ");

child = child.getNextSibling();

}

}

public static void getBrand() throws SAXException, IOException{

//获取html文档的url地址，这个地方也可以换成指定的html文件

String strUrl = "http://quote.eastmoney.com/center/BKList.html#trade\_0\_0?sortRule=0";

DOMFragmentParser parser = new DOMFragmentParser();

HTMLDocument document = new HTMLDocumentImpl();

DocumentFragment fragment = document.createDocumentFragment();

parser.parse(strUrl, fragment);

*print*(fragment, "");

}

}

1. **Java字符编码，以及编码转换**

概念：不管对于何种编码的字符，最终存在存储器上的都是一个字节一个字节的。因此本质上对字符的转码就是对字节的转码。

我们能实实在在看见的是字符，而不是字节，因此当一个字符用gbk编码的时候，比如‘你好’这个字符，你想把它转成utf-8的编码，那么就先将字符串用utf-8转为字节数组，然后再用utf-8将字节数组转为字符串，这样就进行了一次转码。

String str="//必须要使用原生的response";

temp = str.getBytes("utf-8");

1. **日期Date的替代对象Calendar，获取日期建议使用后者**

在进行日期获取的时候都用Calendar对象，而不用Date，因为Date对象已经过期。

/\*\*

\* 获取现在时间，格式为2015-02-03 14：13：43

\* @return返回当前时间的字符串形式

\*/

public static String getCurTimeSecond(){

//使用默认时区和语言环境获得一个日历

Calendar cale = Calendar.getInstance();

cale.add(Calendar.DATE, -1);

//将Calendar类型转换成Date类型

Date tasktime=cale.getTime();

//设置日期输出的格式

SimpleDateFormat df=new SimpleDateFormat("yyyy-MM-dd HH:mm:ss:SSS");

//格式化输出

return df.format(tasktime);

}

1. **URL转码汉字，%25E9%2587%2591%25E5%258D%258E转成金华使用URLDecoder或者URLEncoder进行编码或者解码**

/\*\*

\* 对中文进行url解码

\* @param str

\* @return

\* @throws UnsupportedEncodingException

\*/

public static String decoderURL(String str) throws UnsupportedEncodingException{

str = URLDecoder.*decode*(str, "utf-8");

str = URLDecoder.*decode*(str, "utf-8");

return str;

}

/\*\*

\* 对中文进行url编码

\* @param str

\* @return

\*/

public static String encoderURL(String str) {

try {

str = URLEncoder.*encode*(str, "utf-8");

str = URLEncoder.*encode*(str, "utf-8");

} catch (UnsupportedEncodingException e) {

e.printStackTrace();

}

return str;

}

1. **Maven打jar包**

问题描述：由于eclipse自带的jar打成工具在面对log4j时，又使用hadoop的jar时，打成的可运行jar包后，无法找到log4j的配置文件，这个是与hadoop的jar包有关（具体什么原因有待查询），因此这个时候要么给log4j的配置文件读取设置相对路径，要么使用maven打成jar包。

步骤1：在pom.xml文件中配置maven打包插件

<build>

<finalName>hxz-spider</finalName>

<defaultGoal>package</defaultGoal>

<plugins>

<!-- 打包jar文件时，配置manifest文件，加入lib包的jar依赖  -->

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-jar-plugin</artifactId>

<configuration>

<archive>

<manifest>

<addClasspath>true</addClasspath>

<classpathPrefix>lib/</classpathPrefix>

<mainClass>com.yp.check.CheckSpider</mainClass>

</manifest>

</archive>

</configuration>

</plugin>

    <!-- 拷贝依赖的jar包到lib目录  -->

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-dependency-plugin</artifactId>

<executions>

<execution>

<id>copy</id>

<phase>package</phase>

<goals>

<goal>copy-dependencies</goal>

</goals>

<configuration>

<outputDirectory>${project.build.directory}/lib</outputDirectory>

</configuration>

</execution>

</executions>

</plugin>

<!-- 解决资源文件的编码问题  -->

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-resources-plugin</artifactId>

<version>2.3</version>

<configuration>

<encoding>UTF-8</encoding>

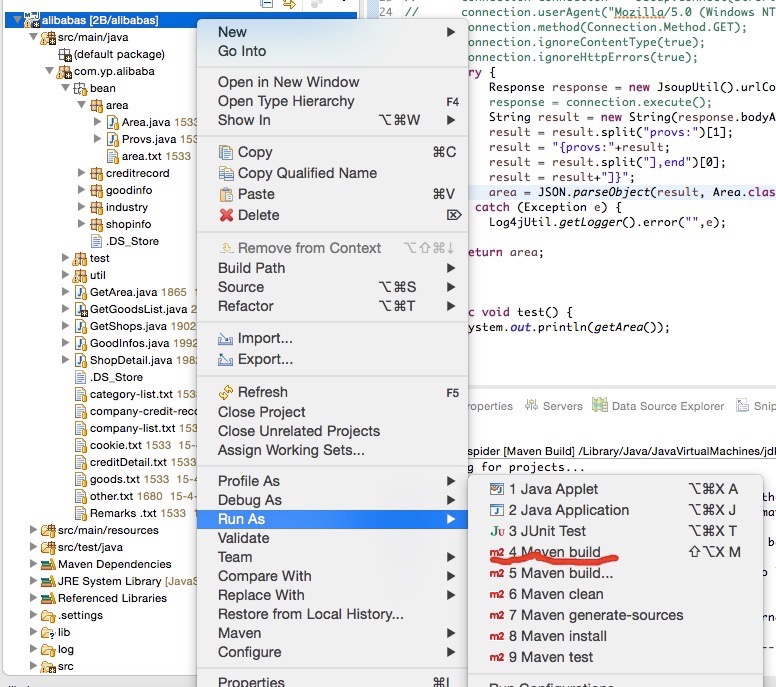
</configuration>

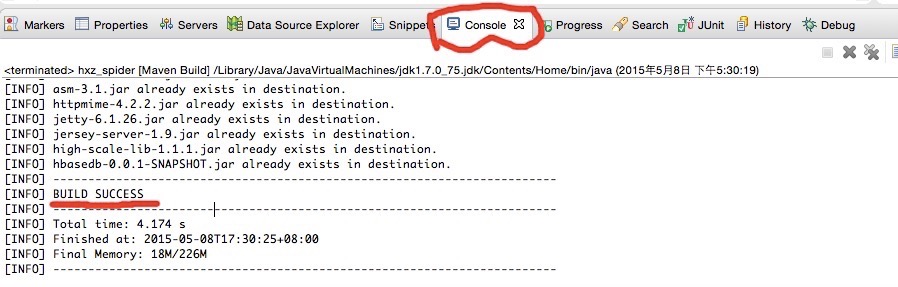
</plugin>

</plugins>

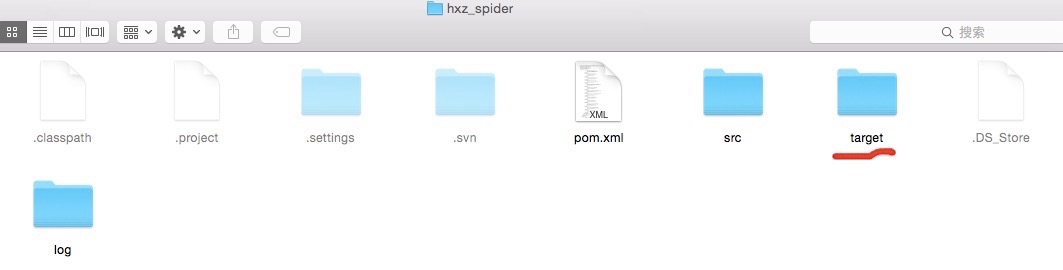
</build>

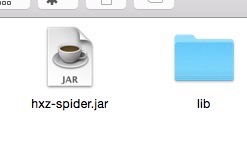
步骤2：使用run as中的maven bulid,将工程build一下





步骤3：这时我们就可以看到打成的jar包，jar包在你打包的工程的target文件夹下，比如你的工程名是test，那么jar就在test工程文件的target文件夹下。





这时运行jar包就可以了。

1. **MD5加密问题**

在不同环境下实现MD5加密结果一致，由于不同环境的默认编码不一致，因此不指定MD5加密的编码，在不同环境下加密出来的结果就不一样，所以在加密的时候一定要指定加密的编码。

问题描述：在windows下的eclipse下进行MD5加密，用junit的test运行是没问题的，但是在tomcat上运行每次都有问题，加密的结果不一样，参数都是相同的。这就是编码的问题

import java.security.MessageDigest;

public class MD5Utils {

public static String encrypt(String msg,String key){

return md5(md5(msg)+key);

}

public static String md5(String msg) {

try {

MessageDigest md = MessageDigest.getInstance("md5");

//指定MD5加密的编码

byte md5[] = md.digest(msg.getBytes("utf-8"));

String strRet = "";

for(int i = 0; i < md5.length; i++){

strRet = strRet + String.format("%02x", md5[i]);

}

return strRet;

} catch (Exception e) {

return "";

}

}

}

1. **根据汉字首字母排序**

问题描述：根据汉字首字母排序

缺点：此排序不能区分多音字，比如重庆市，应该是按C排序，结果却是按Z排序。

方法1：使用链表排序

//这里将map.entrySet()转换成list

List<Map.Entry<Integer,String>> list = new ArrayList<Map.Entry<Integer,String>>(treeMap.entrySet());

//然后通过比较器来实现排序

Collections.*sort*(list,new Comparator<Map.Entry<Integer,String>>() {

//关于Collator。

private Collator collator = Collator.*getInstance*();

//升序排序

@Override

public int compare(Entry<Integer, String> o1,

Entry<Integer, String> o2) {

//把字符串转换为一系列比特，它们可以以比特形式与 CollationKeys 相比较

CollationKey key1 = collator.getCollationKey(o1.getValue().toLowerCase());

//要想不区分大小写进行比较用o1.toString().toLowerCase()

CollationKey key2 = collator.getCollationKey(o2.getValue().toLowerCase());

//返回的分别为1,0,-1 分别代表大于，等于，小于。要想按照字母降序排序的话 加个“-”号

return key1.compareTo(key2);

}

});

for(Map.Entry<Integer,String> mapping:list){

linkMap.put(mapping.getKey(),mapping.getValue());

}

方法2：使用数组排序

Comparator<Object> com=Collator.*getInstance*(java.util.Locale.*CHINA*);

String[] vals = new String[provinceMap.size()];

Map<Integer, String> linkMap = new LinkedHashMap<Integer, String>();

int i = 0;

for (Map.Entry<Integer, String> entry : provinceMap.entrySet()) {

vals[i++] = entry.getValue();

}

Arrays.*sort*(vals,com);

for(String val:vals){

for (Map.Entry<Integer, String> entry : provinceMap.entrySet()) {

if(val.equals(entry.getValue())){

linkMap.put(entry.getKey(), val);

break;

}

}

}

1. **Java构造函数中为什么不能同时调用super和this函数？**

原因：在构造的时候只需要调用父类的super()作为初始化父类一次，如果super(...)和this(...)同时出现的话，那么就会出现初始化父类两次的不安全操作，因为当super(...)和this(...)同时出现的时候，在调用完了super(..)之后还会执行this(..)，而this(...)中又会自动调用super（）,这就造成了调用两次super()的结果(这是假设super(...)和this(...)同时出现时候，super(...)在this(...)前面，反过来出现也一样)

1. **Java堆栈问题**

**堆：**

堆的优势是可以动态地分配内存大小，生存期也不必事先告诉编译器，Java的垃圾收集器会自动收走这些不再使用的数据。缺点是，由于要在运行时动态分配内存，存取速度较慢。

堆内容被所有线程共享，内存回收是Java虚拟机自己处理

基本数据的包装类存在于堆中

包装类数据，如Integer, String, Double等将相应的基本数据类型包装起来的类。这些类数据全部存在于堆中，Java用new()语句来显示地告诉编译器，在运行时才根据需要动态创 建，因此比较灵活，但缺点是要占用更多的时间。

类成员变量

new的对象

**String对象：**

String是一个特殊的包装类数据。即可以用String str = new String("abc");的形式来创建，也可以用String str = "abc"；的形式来创建(作为对比，在JDK 5.0之前，你从未见过Integer i = 3;的表达式，因为类与字面值是不能通用的，除了String。而在JDK 5.0中，这种表达式是可以的！因为编译器在后台进行Integer i = new Integer(3)的转换)。前者是规范的类的创建过程，即在Java中，一切都是对象，而对象是类的实例，全部通过new()的形式来创建。Java 中的有些类，如DateFormat类，可以通过该类的getInstance()方法来返回一个新创建的类，似乎违反了此原则。其实不然。该类运用了单 例模式来返回类的实例，只不过这个实例是在该类内部通过new()来创建的，而getInstance()向外部隐藏了此细节。那为什么在String str = "abc"；中，并没有通过new()来创建实例，是不是违反了上述原则？其实没有。

关于String str = "abc"的内部工作。Java内部将此语句转化为以下几个步骤：

（1）先定义一个名为str的对String类的对象引用变量：String str

（2）在栈中查找有没有存放值为"abc"的地址，如果没有，则开辟一个存放字面值为"abc"的地址，接着创建一个新的String类的对象o，并 将o的字符串值指向这个地址，而且在栈中这个地址旁边记下这个引用的对象o。如果已经有了值为"abc"的地址，则查找对象o，并返回o的地址。

（3）将str指向对象o的地址

值得注意的是，一般String类中字符串值都是直接存值的。但像String str = "abc"；这种场合下，其字符串值却是保存了一个指向存在栈中数据的引用！

String str1 = new String("abc");

String str2 = "abc";

System.out.println(str1==str2); //false

创建了两个引用。创建了两个对象。两个引用分别指向不同的两个对象。

以上两段代码说明，只要是用new()来新建对象的，都会在堆中创建，而且其字符串是单独存值的，即使与栈中的数据相同，也不会与栈中的数据共享。

**栈：**

存取速度比堆快，仅次于直接位于CPU中的寄存器。只属于某个线程，方法执行完后，会马上回收栈内存。

栈是一个先进后出的数据结构,通常用于保存方法(函数)中的参数,局部变量。在java中,所有基本类型和引用类型都在栈中存储.栈中数据的生存空间一般在当前scopes内(就是由{...}括起来的区域)。

一种是基本类型(primitive types), 共有8种，即int, short, long, byte, float, double, boolean, char(注意，并没有string的基本类型)。这种类型的定义是通过诸如int a = 3; long b = 255L;的形式来定义的，称为自动变量。值得注意的是，自动变量存的是字面值，不是类的实例，即不是类的引用，这里并没有类的存在。如int a = 3; 这里的a是一个指向int类型的引用，指向3这个字面值。这些字面值的数据，由于大小可知，生存期可知(这些字面值固定定义在某个程序块里面，程序块退出 后，字段值就消失了)，出于追求速度的原因，就存在于栈中。（栈数据共享，比如int a=3,b=3;在栈中只会存在一个3；a,b的引用都会指向3）

特别注意的是，这种字面值的引用与类对象的引用不同。假定两个类对象的引用同时指向一个对象，如果一个对象引用变量修改了这个对象的内部状态，那么另 一个对象引用变量也即刻反映出这个变化。相反，通过字面值的引用来修改其值，不会导致另一个指向此字面值的引用的值也跟着改变的情况。如上例，我们定义完 a与b的值后，再令a=4；那么，b不会等于4，还是等于3。在编译器内部，遇到a=4；时，它就会重新搜索栈中是否有4的字面值，如果没有，重新开辟地 址存放4的值；如果已经有了，则直接将a指向这个地址。因此a值的改变不会影响到b的值。

存放局部变量

自定义对象的引用

缺点：存在栈中的数据大小与生存期必须是确定的，缺乏灵活性。

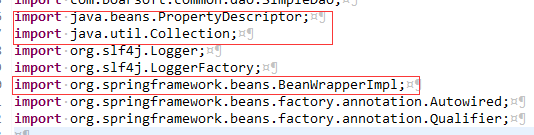
静态变量的存放有单独的区域存放

1. **转化对象**

描述：将某个对象根据字段名相同的，转化到另外一个对象中，这两个对象属于不同类

示例：

// excludes需要忽略的字段，忽略的字段不会转化



public <T> T transfer(Object source, Class<T> targetClass, String[] excludes) {

Object result = null;

try {

result = targetClass.newInstance();

BeanWrapperImpl e = new BeanWrapperImpl(source);

BeanWrapperImpl bwt = new BeanWrapperImpl(result);

PropertyDescriptor[] pds = e.getPropertyDescriptors();

PropertyDescriptor[] arr$ = pds;

int len$ = pds.length;

for (int i$ = 0; i$ < len$; ++i$) {

PropertyDescriptor pd = arr$[i$];

Class pc = pd.getPropertyType();

if (!source.getClass().getPackage().equals(pc.getPackage()) && !Collection.class.isAssignableFrom(pc)) {

String s = pd.getName();

if (Util.*inArray*(s, excludes) < 0 && e.isReadableProperty(s) && bwt.isWritableProperty(s)) {

bwt.setPropertyValue(s, e.getPropertyValue(s));

}

}

}

} catch (Exception arg13) {

this.log.error("Error on transfer", arg13);

}

return result;

}

public static int inArray(Object o, Object[] arr) {

if (arr != null && arr.length >= 1) {

for (int i = 0; i < arr.length; ++i) {

if (o == null) {

if (arr[i] == null) {

return i;

}

} else if (o.equals(arr[i])) {

return i;

}

}

return -1;

} else {

return -1;

}

}

1. **内部类引用外部类局部变量的问题**

外部类局部变量必须声明成final才能被内部类使用，原因如下：

因为该内部类出现在一个方法的内部，但实际编译时，内部类编译为Outer$1TimerPrint.class，这说明，外部类的这个方法和内部类是处于同一级别的。换句话说是非final变量和内部类的生命周期不一样！start被调用后，非final变量也会随之消失，就会出现内部类引用非法！

实际做法：java编译器的行为是这样的(前提条件是该变量在内部类中被引用)：

若定义为final，则java编译器则会在内部类TimerPrint内生成一个外部变量的拷贝，而且这样既可以保证内部类可以引用外部属性，又能保证值的唯一性。

例子：

final List<VisitedPages> visitedPagesList = new ArrayList<VisitedPages>();

String sql = "SELECT sum(uv),sum(pv),url,url\_name,sum(entry\_page\_times),sum(leave\_times),sum(contributed\_pv),sum(duration)"

+ " FROM stat\_basic\_info\_reach\_page WHERE site\_id=? AND class\_id=? AND date\_time>? AND date\_time<? "

+ "GROUP BY url ORDER BY pv";

jdbcTemplateHadoop.query(sql, new Object[]{vpRequest.getSiteId(),vpRequest.getClassId(),

vpRequest.getStartTime(),endTime},

new RowCallbackHandler() {

VisitedPages visitedPages;

public void processRow(ResultSet result) throws SQLException {

visitedPages = getVisitedPages(result);

visitedPagesList.add(visitedPages);

}

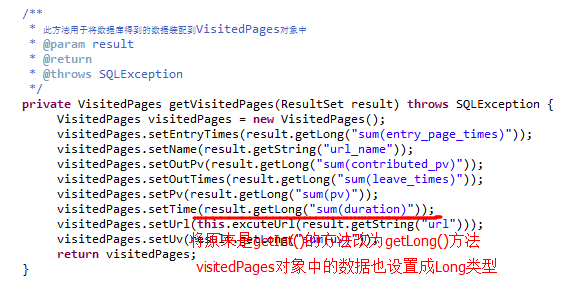
});

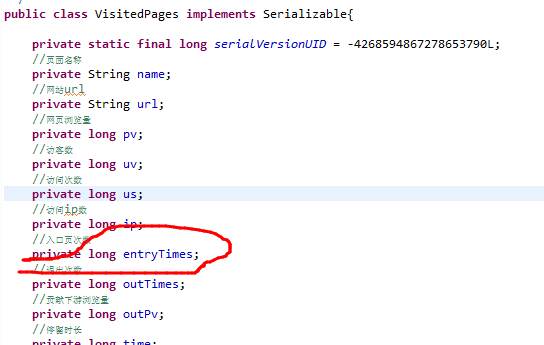
1. **利用Socket通信进行对象传递，对象的序列化问题**

注：客户端和服务端的通信对象所在的包名必须相同，而且类名页必须相同。然后通信的对象必须实现序列化接口

1. **设置对象数据报错org.springframework.dao.DataIntegrityViolationException**

问题描述：从数据查询到的数据放在对象中出现这个错误，out of range Integer





1. **对数字的商保留多少位的处理**

/\*\*

\* 此方法用于求占比

\* @param divisor除数

\* @param dividend被除数

\* @param accuracy精度，保留小数多少位

\* @return

\*/

public static String getRate(long divisor,long dividend,int accuracy){

double rate = 0;

NumberFormat numberFormat = NumberFormat.*getInstance*();

numberFormat.setMaximumFractionDigits(accuracy);

if(dividend!=0){

rate = (double)divisor/dividend;

}else{

rate = 0;

}

return numberFormat.format(rate);

}

1. **静态代理、动态代理、Cglib**

**Cglib:** CGLib采用了非常底层的字节码技术，其原理是通过字节码技术为一个类创建子类，并在子类中采用方法拦截的技术拦截所有父类方法的调用，顺势织入横切逻辑。

需要被代理的类，也就是父类，通过字节码技术创建这个类的子类，实现动态代理。

public class SayHello {

public void say(){

System.*out*.println("hello everyone");

}

**}**

getProxy(SuperClass.class)方法通过入参即父类的字节码，通过扩展父类的class来创建代理对象。intercept()方法拦截所有目标类方法的调用，obj表示目标类的实例，method为目标类方法的反射对象，args为方法的动态入参，proxy为代理类实例。proxy.invokeSuper(obj, args)通过代理类调用父类中的方法。

public class CglibProxy implements MethodInterceptor{

private Enhancer enhancer = new Enhancer();

public Object getProxy(Class clazz){

//设置需要创建子类的类

enhancer.setSuperclass(clazz);

enhancer.setCallback(this);

//通过字节码技术动态创建子类实例

return enhancer.create();

}

//实现MethodInterceptor接口方法

public Object intercept(Object obj, Method method, Object[] args,

MethodProxy proxy) throws Throwable {

System.*out*.println("前置代理");

//通过代理类调用父类中的方法

Object result = proxy.invokeSuper(obj, args);

System.*out*.println("后置代理");

return result;

}

**}**

**操作类：**

public class DoCGLib {

public static void main(String[] args) {

CglibProxy proxy = new CglibProxy();

//通过生成子类的方式创建代理类

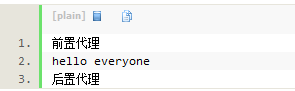
SayHello proxyImp = (SayHello)proxy.getProxy(SayHello.class);

proxyImp.say();

}

**}**

**结果：**



注：CGLib创建的动态代理对象性能比JDK创建的动态代理对象的性能高不少，但是CGLib在创建代理对象时所花费的时间却比JDK多得多，所以对于单例的对象，因为无需频繁创建对象，用CGLib合适，反之，使用JDK方式要更为合适一些。同时，由于CGLib由于是采用动态创建子类的方法，对于final方法，无法进行代理。

**静态代理：**由程序员创建或特定工具自动生成源代码，再对其编译。在程序运行前，代理类的.class文件就已经存在了。

/\*\*

\* 定义一个账户接口

\*

\* @author Administrator

\*

\*/

public interface Count {

// 查看账户方法

public void queryCount();

// 修改账户方法

public void updateCount();

**}**

/\*\*

\* 委托类(包含业务逻辑)

\*

\* @author Administrator

\*

\*/

public class CountImpl implements Count {

@Override

public void queryCount() {

System.*out*.println("查看账户方法...");

}

@Override

public void updateCount() {

System.*out*.println("修改账户方法...");

}

**}**

/\*\*

\* 这是一个代理类（增强CountImpl实现类）

\*

\* @author Administrator

\*

\*/

public class CountProxy implements Count {

private CountImpl countImpl;

/\*\*

\* 覆盖默认构造器

\*

\* @param countImpl

\*/

public CountProxy(CountImpl countImpl) {

this.countImpl = countImpl;

}

@Override

public void queryCount() {

System.*out*.println("事务处理之前");

// 调用委托类的方法;

countImpl.queryCount();

System.*out*.println("事务处理之后");

}

@Override

public void updateCount() {

System.*out*.println("事务处理之前");

// 调用委托类的方法;

countImpl.updateCount();

System.*out*.println("事务处理之后");

}

**}**

/\*\*

\*测试Count类

\*

\* @author Administrator

\*

\*/

public class TestCount {

public static void main(String[] args) {

CountImpl countImpl = new CountImpl();

CountProxy countProxy = new CountProxy(countImpl);

countProxy.updateCount();

countProxy.queryCount();

}

**}**

注：每一个代理类只能为一个接口服务，这样一来程序开发中必然会产生过多的代理，而且，所有的代理操作除了调用的方法不一样之外，其他的操作都一样，则此时肯定是重复代码。

**动态代理：**动态代理类的字节码在程序运行时由Java反射机制动态生成，无需程序员手工编写它的源代码。动态代理类不仅简化了编程工作，而且提高了软件系统的可扩展性，因为Java 反射机制可以生成任意类型的动态代理类。java.lang.reflect 包中的Proxy类和InvocationHandler 接口提供了生成动态代理类的能力。

JDK动态代理中包含一个类和一个接口：   
InvocationHandler接口：   
public interface InvocationHandler {

public Object invoke(Object proxy,Method method,Object[] args) throws Throwable;

}

参数说明：   
Object proxy：指被代理的对象。   
Method method：要调用的方法   
Object[] args：方法调用时所需要的参数

可以将InvocationHandler接口的子类想象成一个代理的最终操作类

Proxy类：

Proxy类是专门完成代理的操作类，可以通过此类为一个或多个接口动态地生成实现类，此类提供了如下的操作方法：

public static Object newProxyInstance(ClassLoader loader, Class<?>[] interfaces,

InvocationHandler h)

throws **IllegalArgumentException**

参数说明：

ClassLoader loader：类加载器

Class<?>[] interfaces：得到全部的接口

InvocationHandler h：得到InvocationHandler接口的子类实例

示例：

public interface BookFacade {

public void addBook();

**}**

public class BookFacadeImpl implements BookFacade {

@Override

public void addBook() {

System.*out*.println("增加图书方法。。。");

}

**}**

/\*\*

\* JDK动态代理代理类

\*

\* @author student

\*

\*/

public class BookFacadeProxy implements InvocationHandler {

private Object target;

/\*\*

\* 绑定委托对象并返回一个代理类

\* @param target

\* @return

\*/

public Object bind(Object target) {

this.target = target;

//取得代理对象

return Proxy.newProxyInstance(target.getClass().getClassLoader(),

target.getClass().getInterfaces(), this); //要绑定接口(这是一个缺陷，cglib弥补了这一缺陷)

}

@Override

/\*\*

\* 调用方法

\*/

public Object invoke(Object proxy, Method method, Object[] args)

throws Throwable {

Object result=null;

System.*out*.println("事物开始");

//执行方法

result=method.invoke(target, args);

System.*out*.println("事物结束");

return result;

}

**}**

public class TestProxy {

public static void main(String[] args) {

BookFacadeProxy proxy = new BookFacadeProxy();

BookFacade bookProxy = (BookFacade) proxy.bind(new BookFacadeImpl());

bookProxy.addBook();

}

**}**

**注：JDK的动态代理依靠接口实现，如果有些类并没有实现接口，则不能使用JDK代理**

1. **类加载器**
2. **类加载器工作机制**

类装载器就是寻找类的子节码文件并构造出类在JVM内部表示对象的组件。在Java中，类装载器把一个类装入JVM中，要经过以下步骤：

1）、装载：查找和导入Class文件；

2）、链接：执行校验、准备和解析步骤，其中解析步骤是可以选择的：

2.1）、校验：检查载入Class文件数据的正确性；

2.2）、准备：给类的静态变量分配存储空间；

2.3）、解析：将符号引用转成直接引用；

3）、初始化：对类的静态变量、静态代码块执行初始化工作。

JDK 默认提供了如下几种**ClassLoader**：

1）、Bootstrp loader

Bootstrp加载器是用C++语言写的，它是在Java虚拟机启动后初始化的，它主要负责加载%JAVA\_HOME%/jre/lib,-Xbootclasspath参数指定的路径以及%JAVA\_HOME%/jre/classes中的类。

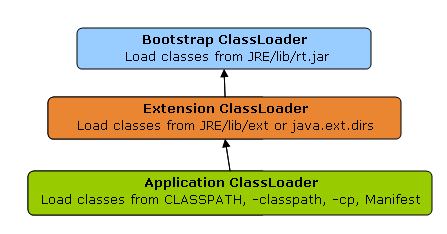
2）、ExtClassLoader

Bootstrp loader加载ExtClassLoader,并且将ExtClassLoader的父加载器设置为Bootstrp loader.ExtClassLoader是用Java写的，具体来说就是 sun.misc.Launcher$ExtClassLoader，ExtClassLoader主要加载%JAVA\_HOME%/jre/lib/ext，此路径下的所有classes目录以及java.ext.dirs系统变量指定的路径中类库。

3）、AppClassLoader

Bootstrp loader加载完ExtClassLoader后，就会加载AppClassLoader,并且将AppClassLoader的父加载器指定为 ExtClassLoader。AppClassLoader也是用Java写成的，它的实现类是 sun.misc.Launcher$AppClassLoader，另外我们知道ClassLoader中有个getSystemClassLoader方法,此方法返回的正是AppclassLoader.AppClassLoader主要负责加载classpath所指定的位置的类或者是jar文档，它也是Java程序默认的类加载器。

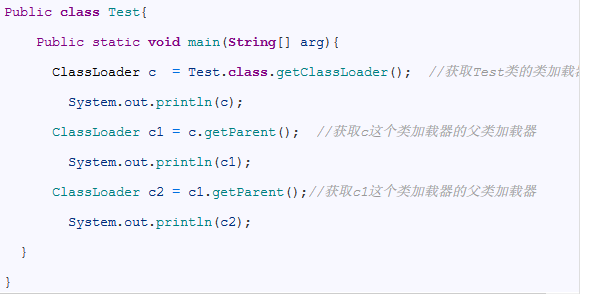
综上所述，它们之间的关系可以通过下图形象的描述：



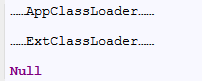
1. **类的全盘负责委托机制**

类装载器有载入类的需求时，会先请示其Parent使用其搜索路径帮忙载入，如果Parent 找不到,那么才由自己依照自己的搜索路径搜索类

下面举一个例子来说明，为了更好的理解，先弄清楚几行代码：



结果



可以看出Test是由AppClassLoader加载器加载的，AppClassLoader的Parent 加载器是 ExtClassLoader,但是ExtClassLoader的Parent为 null 是怎么回事呵，朋友们留意的话，前面有提到Bootstrap Loader是用C++语言写的，依java的观点来看，逻辑上并不存在Bootstrap Loader的类实体，所以在java程序代码里试图打印出其内容时，我们就会看到输出为null。

1. **java.lang.NoSuchMethodError解决办法**

但凡Java的开发者，想必遇到过java.lang.NoSuchMethodError的错误信息吧。究其源，这个错误基本上都是由JVM的"全盘负责委托机制"引发的问题：因为在类路径下放置了多个不同版本的类包，如commons-lang 2.x.jar和commons-lang3.x.jar都位于类路径中，代码中用到了commons-lang3.x类的某个方法，而这个方法在commons-lang2.x中并不存在，JVM加载类时碰巧又从commons-lang 2.x.jar中加载类，运行时就会抛出NoSuchMethodError的错误。

这种问题的排查是比较棘手的，特别是在Web应用的情况下，可作为类路径的系统目录比较多，特别在类包众多时，情况尤其复杂：你不知道JVM到底从哪个类包中加载类文件。不过笔者有一个一般人不告诉的易用小工具，现奉献出来：

在光盘根路径下有一个srcAdd.jsp的程序，你把它放到Web应用的根路径下，通过如下方式即可查看JVM从哪个类包加载指定类：

可以去网上搜源码来解决该问题

1. **Jdbc的MetaData元数据信息**

1. 获取数据库相关信息

获取数据库相关信息

|  |
| --- |
| public static void getDataBaseInfo() {  Connection conn =  getConnection();  ResultSet rs = null;  try{  DatabaseMetaData dbmd = conn.getMetaData();  System.*out*.println("数据库已知的用户: "+ dbmd.getUserName());  System.*out*.println("数据库的系统函数的逗号分隔列表: "+ dbmd.getSystemFunctions());  System.*out*.println("数据库的时间和日期函数的逗号分隔列表: "+ dbmd.getTimeDateFunctions());  System.*out*.println("数据库的字符串函数的逗号分隔列表: "+ dbmd.getStringFunctions());  System.*out*.println("数据库供应商用于 'schema' 的首选术语: "+ dbmd.getSchemaTerm());  System.*out*.println("数据库URL: " + dbmd.getURL());  System.*out*.println("是否允许只读:" + dbmd.isReadOnly());  System.*out*.println("数据库的产品名称:" + dbmd.getDatabaseProductName());  System.*out*.println("数据库的版本:" + dbmd.getDatabaseProductVersion());  System.*out*.println("驱动程序的名称:" + dbmd.getDriverName());  System.*out*.println("驱动程序的版本:" + dbmd.getDriverVersion());  System.*out*.println("数据库中使用的表类型");  rs = dbmd.getTableTypes();  while (rs.next()) {  System.*out*.println(rs.getString("TABLE\_TYPE"));  }  }catch (SQLException e){  e.printStackTrace();  } finally{  JdbcUtil.close(rs,conn);  }  } |

2. 获得数据库中所有Schemas

获得数据库中所有Schemas(对应于oracle中的Tablespace)

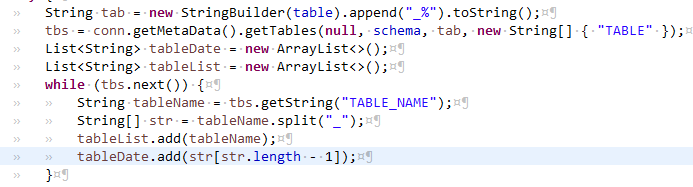
|  |
| --- |
| public static void getSchemasInfo(){  Connection conn =  getConnection();  ResultSet rs = null;  try{  DatabaseMetaData dbmd = conn.getMetaData();  rs = dbmd.getSchemas();  while (rs.next()){  String tableSchem = rs.getString("TABLE\_SCHEM");  System.*out*.println(tableSchem);  }  } catch (SQLException e){  e.printStackTrace();  } finally{  JdbcUtil.close(rs,conn);  }  } |

3. 获取数据库中所有的表信息

获取数据库中所有的表信息

|  |
| --- |
| public static void getTablesList() {  Connection conn = getConnection();  ResultSet rs = null;  try {  /\*\*  \* 设置连接属性,使得可获取到表的REMARK(备注)  \*/  ((OracleConnection) conn).setRemarksReporting(true);  DatabaseMetaData dbmd = conn.getMetaData();  String[] types = { "TABLE" };  rs = dbmd.getTables(null, null, "%", types);  while (rs.next()) {  String tableName = rs.getString("TABLE\_NAME"); // 表名  String tableType = rs.getString("TABLE\_TYPE"); // 表类型  String remarks = rs.getString("REMARKS"); // 表备注  System.*out*.println(tableName + " - " + tableType + " - " + remarks);  }  } catch (SQLException e) {  e.printStackTrace();  } finally {  JdbcUtil.close(rs, conn);  }  } |

4.进行表名模糊查询表



5.获取某表信息

|  |
| --- |
| public static void getTablesInfo() {  Connection conn = getConnection();  ResultSet rs = null;  try {  /\*\*  \* 设置连接属性,使得可获取到表的REMARK(备注)  \*/  ((OracleConnection) conn).setRemarksReporting(true);  DatabaseMetaData dbmd = conn.getMetaData();  /\*\*  \* 获取给定类别中使用的表的描述。 方法原型:ResultSet getTables(String catalog,String  \* schemaPattern,String tableNamePattern,String[] types); catalog -  \* 表所在的类别名称;""表示获取没有类别的列,null表示获取所有类别的列。 schema -  \* 表所在的模式名称(oracle中对应于Tablespace);""表示获取没有模式的列,null标识获取所有模式的列;  \* 可包含单字符通配符("\_"),或多字符通配符("%"); tableNamePattern -  \* 表名称;可包含单字符通配符("\_"),或多字符通配符("%"); types - 表类型数组; "TABLE"、"VIEW"、"SYSTEM  \* TABLE"、"GLOBAL TEMPORARY"、"LOCAL TEMPORARY"、"ALIAS" 和  \* "SYNONYM";null表示包含所有的表类型;可包含单字符通配符("\_"),或多字符通配符("%");  \*/  rs = dbmd.getTables(null, null, "CUST\_INTER\_TF\_SERVICE\_REQ", new String[] { "TABLE", "VIEW" });  while (rs.next()) {  String tableCat = rs.getString("TABLE\_CAT"); // 表类别(可为null)  String tableSchemaName = rs.getString("TABLE\_SCHEM");// 表模式（可能为空）,在oracle中获取的是命名空间,其它数据库未知  String tableName = rs.getString("TABLE\_NAME"); // 表名  String tableType = rs.getString("TABLE\_TYPE"); // 表类型,典型的类型是 "TABLE"、"VIEW"、"SYSTEM TABLE"、"GLOBAL  // TEMPORARY"、"LOCAL TEMPORARY"、"ALIAS" 和 "SYNONYM"。  String remarks = rs.getString("REMARKS"); // 表备注  System.*out*.println(  tableCat + " - " + tableSchemaName + " - " + tableName + " - " + tableType + " - " + remarks);  }  } catch (Exception ex) {  ex.printStackTrace();  } finally {  JdbcUtil.close(rs, conn);  }  } |

6. 获取表主键信息

|  |
| --- |
| public static void getPrimaryKeysInfo() {  Connection conn = getConnection();  ResultSet rs = null;  try {  DatabaseMetaData dbmd = conn.getMetaData();  /\*\*  \* 获取对给定表的主键列的描述 方法原型:ResultSet getPrimaryKeys(String catalog,String  \* schema,String table); catalog - 表所在的类别名称;""表示获取没有类别的列,null表示获取所有类别的列。 schema  \* - 表所在的模式名称(oracle中对应于Tablespace);""表示获取没有模式的列,null标识获取所有模式的列;  \* 可包含单字符通配符("\_"),或多字符通配符("%"); table - 表名称;可包含单字符通配符("\_"),或多字符通配符("%");  \*/  rs = dbmd.getPrimaryKeys(null, null, "CUST\_INTER\_TF\_SERVICE\_REQ");  while (rs.next()) {  String tableCat = rs.getString("TABLE\_CAT"); // 表类别(可为null)  String tableSchemaName = rs.getString("TABLE\_SCHEM");// 表模式（可能为空）,在oracle中获取的是命名空间,其它数据库未知  String tableName = rs.getString("TABLE\_NAME"); // 表名  String columnName = rs.getString("COLUMN\_NAME");// 列名  short keySeq = rs.getShort("KEY\_SEQ");// 序列号(主键内值1表示第一列的主键，值2代表主键内的第二列)  String pkName = rs.getString("PK\_NAME"); // 主键名称  System.*out*.println(tableCat + " - " + tableSchemaName + " - " + tableName + " - " + columnName + " - "  + keySeq + " - " + pkName);  }  } catch (SQLException e) {  e.printStackTrace();  } finally {  JdbcUtil.close(rs, conn);  }  } |

7. 获取表索引信息

|  |
| --- |
| public static void getIndexInfo() {  Connection conn = getConnection();  ResultSet rs = null;  try {  DatabaseMetaData dbmd = conn.getMetaData();  /\*\*  \* 获取给定表的索引和统计信息的描述 方法原型:ResultSet getIndexInfo(String catalog,String  \* schema,String table,boolean unique,boolean approximate) catalog -  \* 表所在的类别名称;""表示获取没有类别的列,null表示获取所有类别的列。 schema -  \* 表所在的模式名称(oracle中对应于Tablespace);""表示获取没有模式的列,null标识获取所有模式的列;  \* 可包含单字符通配符("\_"),或多字符通配符("%"); table - 表名称;可包含单字符通配符("\_"),或多字符通配符("%"); unique  \* - 该参数为 true时,仅返回唯一值的索引; 该参数为 false时,返回所有索引; approximate -  \* 该参数为true时,允许结果是接近的数据值或这些数据值以外的值;该参数为 false时,要求结果是精确结果;  \*/  rs = dbmd.getIndexInfo(null, null, "CUST\_INTER\_TF\_SERVICE\_REQ", false, true);  while (rs.next()) {  String tableCat = rs.getString("TABLE\_CAT"); // 表类别(可为null)  String tableSchemaName = rs.getString("TABLE\_SCHEM");// 表模式（可能为空）,在oracle中获取的是命名空间,其它数据库未知  String tableName = rs.getString("TABLE\_NAME"); // 表名  boolean nonUnique = rs.getBoolean("NON\_UNIQUE");// 索引值是否可以不唯一,TYPE为 tableIndexStatistic时索引值为 false;  String indexQualifier = rs.getString("INDEX\_QUALIFIER");// 索引类别（可能为空）,TYPE为 tableIndexStatistic 时索引类别为  // null;  String indexName = rs.getString("INDEX\_NAME");// 索引的名称 ;TYPE为 tableIndexStatistic 时索引名称为 null;  /\*\*  \* 索引类型： tableIndexStatistic - 此标识与表的索引描述一起返回的表统计信息 tableIndexClustered - 此为集群索引  \* tableIndexHashed - 此为散列索引 tableIndexOther - 此为某种其他样式的索引  \*/  short type = rs.getShort("TYPE");// 索引类型;  short ordinalPosition = rs.getShort("ORDINAL\_POSITION");// 在索引列顺序号;TYPE为 tableIndexStatistic 时该序列号为零;  String columnName = rs.getString("COLUMN\_NAME");// 列名;TYPE为 tableIndexStatistic时列名称为 null;  String ascOrDesc = rs.getString("ASC\_OR\_DESC");// 列排序顺序:升序还是降序[A:升序; B:降序];如果排序序列不受支持,可能为 null;TYPE为  // tableIndexStatistic时排序序列为 null;  int cardinality = rs.getInt("CARDINALITY"); // 基数;TYPE为 tableIndexStatistic 时,它是表中的行数;否则,它是索引中唯一值的数量。  int pages = rs.getInt("PAGES"); // TYPE为 tableIndexStatisic时,它是用于表的页数,否则它是用于当前索引的页数。  String filterCondition = rs.getString("FILTER\_CONDITION"); // 过滤器条件,如果有的话(可能为 null)。  System.*out*.println(tableCat + " - " + tableSchemaName + " - " + tableName + " - " + nonUnique + " - "  + indexQualifier + " - " + indexName + " - " + type + " - " + ordinalPosition + " - "  + columnName + " - " + ascOrDesc + " - " + cardinality + " - " + pages + " - "  + filterCondition);  }  } catch (SQLException e) {  e.printStackTrace();  } finally {  JdbcUtil.close(rs, conn);  }  } |

8. 获取表中列值信息

|  |
| --- |
| public static void getColumnsInfo() {  Connection conn = getConnection();  ResultSet rs = null;  try {  /\*\*  \* 设置连接属性,使得可获取到列的REMARK(备注)  \*/  ((OracleConnection) conn).setRemarksReporting(true);  DatabaseMetaData dbmd = conn.getMetaData();  /\*\*  \* 获取可在指定类别中使用的表列的描述。 方法原型:ResultSet getColumns(String catalog,String  \* schemaPattern,String tableNamePattern,String columnNamePattern) catalog -  \* 表所在的类别名称;""表示获取没有类别的列,null表示获取所有类别的列。 schema -  \* 表所在的模式名称(oracle中对应于Tablespace);""表示获取没有模式的列,null标识获取所有模式的列;  \* 可包含单字符通配符("\_"),或多字符通配符("%"); tableNamePattern -  \* 表名称;可包含单字符通配符("\_"),或多字符通配符("%"); columnNamePattern - 列名称;  \* ""表示获取列名为""的列(当然获取不到);null表示获取所有的列;可包含单字符通配符("\_"),或多字符通配符("%");  \*/  rs = dbmd.getColumns(null, null, "CUST\_INTER\_TF\_SERVICE\_REQ", null);  while (rs.next()) {  String tableCat = rs.getString("TABLE\_CAT"); // 表类别（可能为空）  String tableSchemaName = rs.getString("TABLE\_SCHEM"); // 表模式（可能为空）,在oracle中获取的是命名空间,其它数据库未知  String tableName\_ = rs.getString("TABLE\_NAME"); // 表名  String columnName = rs.getString("COLUMN\_NAME"); // 列名  int dataType = rs.getInt("DATA\_TYPE"); // 对应的java.sql.Types的SQL类型(列类型ID)  String dataTypeName = rs.getString("TYPE\_NAME"); // java.sql.Types类型名称(列类型名称)  int columnSize = rs.getInt("COLUMN\_SIZE"); // 列大小  int decimalDigits = rs.getInt("DECIMAL\_DIGITS"); // 小数位数  int numPrecRadix = rs.getInt("NUM\_PREC\_RADIX"); // 基数（通常是10或2） --未知  /\*\*  \* 0 (columnNoNulls) - 该列不允许为空 1 (columnNullable) - 该列允许为空 2  \* (columnNullableUnknown) - 不确定该列是否为空  \*/  int nullAble = rs.getInt("NULLABLE"); // 是否允许为null  String remarks = rs.getString("REMARKS"); // 列描述  String columnDef = rs.getString("COLUMN\_DEF"); // 默认值  int charOctetLength = rs.getInt("CHAR\_OCTET\_LENGTH"); // 对于 char 类型，该长度是列中的最大字节数  int ordinalPosition = rs.getInt("ORDINAL\_POSITION"); // 表中列的索引（从1开始）  /\*\*  \* ISO规则用来确定某一列的是否可为空(等同于NULLABLE的值:[ 0:'YES'; 1:'NO'; 2:''; ]) YES -- 该列可以有空值;  \* NO -- 该列不能为空; 空字符串--- 不知道该列是否可为空  \*/  String isNullAble = rs.getString("IS\_NULLABLE");  /\*\*  \* 指示此列是否是自动递增 YES -- 该列是自动递增的 NO -- 该列不是自动递增 空字串--- 不能确定该列是否自动递增  \*/  // String isAutoincrement = rs.getString("IS\_AUTOINCREMENT"); //该参数测试报错  System.*out*.println(tableCat + " - " + tableSchemaName + " - " + tableName\_ + " - " + columnName + " - "  + dataType + " - " + dataTypeName + " - " + columnSize + " - " + decimalDigits + " - "  + numPrecRadix + " - " + nullAble + " - " + remarks + " - " + columnDef + " - "  + charOctetLength + " - " + ordinalPosition + " - " + isNullAble);  }  } catch (SQLException ex) {  ex.printStackTrace();  } finally {  JdbcUtil.close(rs, conn);  }  } |

9. ResultSetMetaData结果信息

数据结果的类型信息(核心类)

|  |
| --- |
| @Test  public void rs2() throws Exception{  Connection con = DataSourceUtils.*getConn*();  //转到exam数据库中去  Statement st = con.createStatement();  st.execute("use exam");  //查询  String sql = "select \* from dept";  ResultSet rs = st.executeQuery(sql);  //对rs结果集进行分析    ResultSetMetaData rsmd=rs.getMetaData();  //获取有几个列    int cols = rsmd.getColumnCount();    System.*err*.println(cols);  //获取每一个字段名    List<String> colNames = new ArrayList<String>();//保存所有的字段    for(int i=0;i<cols;i++){    String colName = rsmd.getColumnName(i+1);    System.*err*.print(colName+"\t\t");    colNames.add(colName);    }    System.*err*.println();    //获取数据    while(rs.next()){    for(String nm:colNames){//遍历一行中的所列    String val = rs.getString(nm);    System.*err*.print(val+"\t\t");    }    System.*err*.println();    }  con.close();  } |

1. **Aspectjweaver切面编程**

**问题描述：**我们在执行某个类的方法时，希望执行这个类方法后或者前能再执行一段代码操作，但是原类我们已经无法修改，所以这个时候借助Aspectjweaver进行切面编程。

package com.boarsoft.boar.soagov.svc.biz;

import org.aspectj.lang.JoinPoint;

import org.aspectj.lang.annotation.After;

import org.aspectj.lang.annotation.Aspect;

import org.aspectj.lang.annotation.Pointcut;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Component;

import com.boarsoft.boar.soagov.svc.SvcProtocolBiz;

@Component

@Aspect

public class SvcProtocolAspect {

protected Logger log = LoggerFactory.*getLogger*(this.getClass());

@Autowired

protected SvcProtocolBiz svcProtocolBiz;

/\*\*

\* 设置切面方法的映射,此方法表示在执行类SvcBizImpl的delete方法时侵入

\*/

@Pointcut("execution(\* com.boarsoft.boar.soagov.svc.biz.SvcBizImpl.delete(String))")

public void deleteSvc() {

log.info("Point Cut com.boarsoft.boar.soagov.svc.biz.SvcBizImpl.delete");

}

/\*\*

\* svc\_info的删除切面，删除svc\_info时将svc\_protocol相关的数据也一并删除

\* 执行切面的具体方法，在执行类SvcBizImpl的delete方法后执行下面的代码

\* @param jp 类SvcBizImpl的delete方法的参数

\*/

@After("deleteSvc()")

public void onDeleteSvc(JoinPoint jp) {

String svcId = (String) jp.getArgs()[0];

log.info("After delete svcInfo {}, delete svc protocol of it", svcId);

svcProtocolBiz.delete(svcId, null);

}

}

1. **"title".equals(childName)与childName.equals("title")的区别**

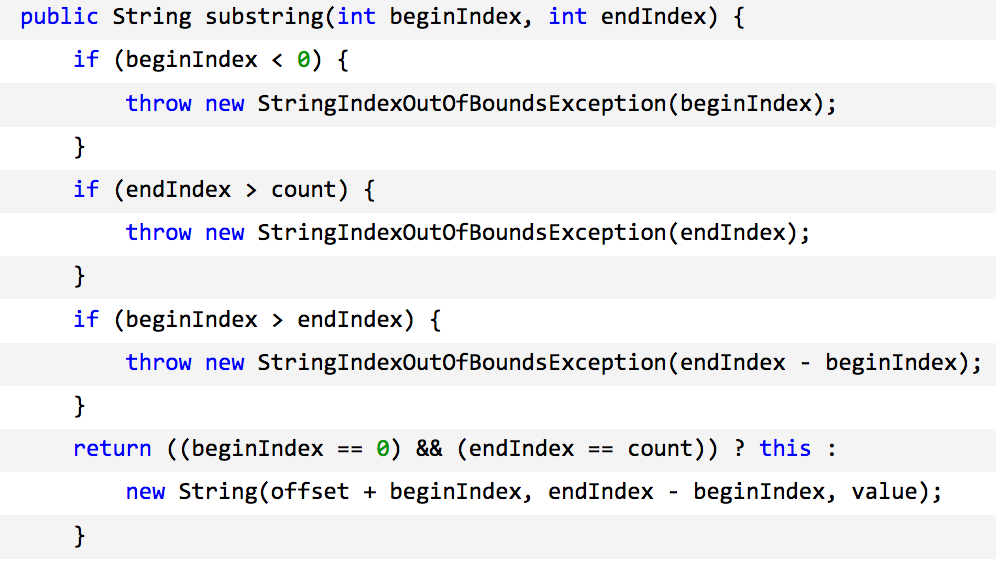
千万别写成后者，如果遇到title是null的时候程序就会报错，因此必须使用前者。

1. **String、StringBuilder、StringBuffer**

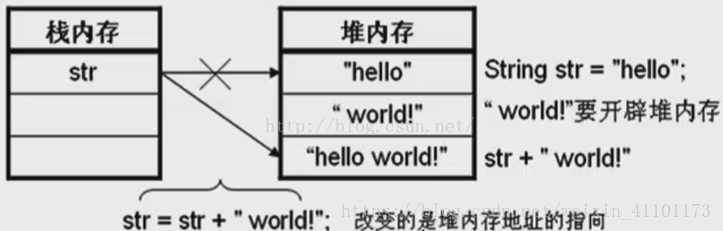
**String类：**

String类是final类，也即意味着String类不能被继承，并且它的成员方法都默认为final方法。在Java中，被final修饰的类是不允许被继承的，并且该类中的成员方法都默认为final方法。在早期的JVM实现版本中，被final修饰的方法会被转为内嵌调用以提升执行效率。而从Java SE5/6开始，就渐渐摈弃这种方式了。因此在现在的Java SE版本中，不需要考虑用final去提升方法调用效率。只有在确定不想让该方法被覆盖时，才将方法设置为final。

上面列举出了String类中所有的成员属性，从上面可以看出String类其实是通过char数组来保存字符串的。



从上面的方法可以看出，无论是substring、concat还是replace操作都不是在原有的字符串上进行的，而是重新生成了一个新的字符串对象。也就是说进行这些操作后，最原始的字符串并没有被改变。

****

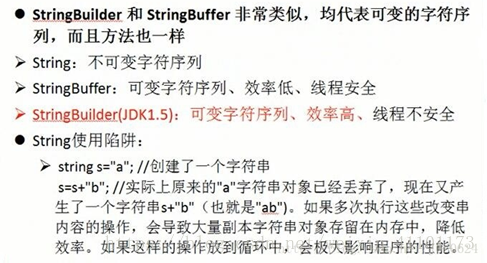
我们可以看到，初始String值为“hello”，然后在这个字符串后面加上新的字符串“world”，这个过程是需要重新在栈堆内存中开辟内存空间的，最终得到了“hello world”字符串也相应的需要开辟内存空间，这样短短的两个字符串，却需要开辟三次内存空间，不得不说这是对内存空间的极大浪费。

在这里要永远记住一点：

对String对象的任何改变都不影响到原对象，相关的任何change操作都会生成新的对象。

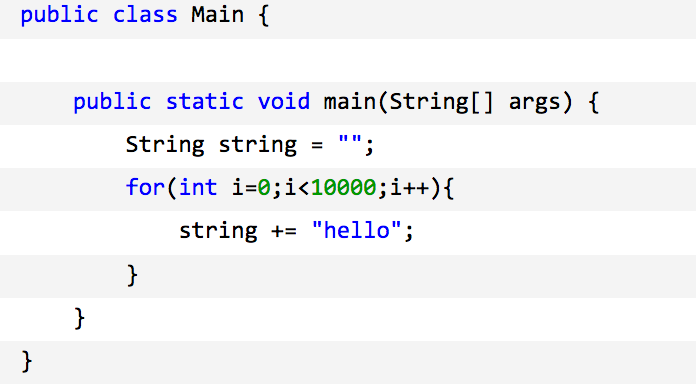
只有使用引号包含文本的方式创建的String对象之间使用“+”连接产生的新对象才会被加入字符串池中。对于所有包含new方式新建对象（包括null）的“+”连接表达式，它所产生的新对象都不会被加入字符串池中，对此我们不再赘述。因此我们提倡大家用引号包含文本的方式来创建String对象以提高效率，实际上这也是我们在编程中常采用的。

**StringBuilder类:**

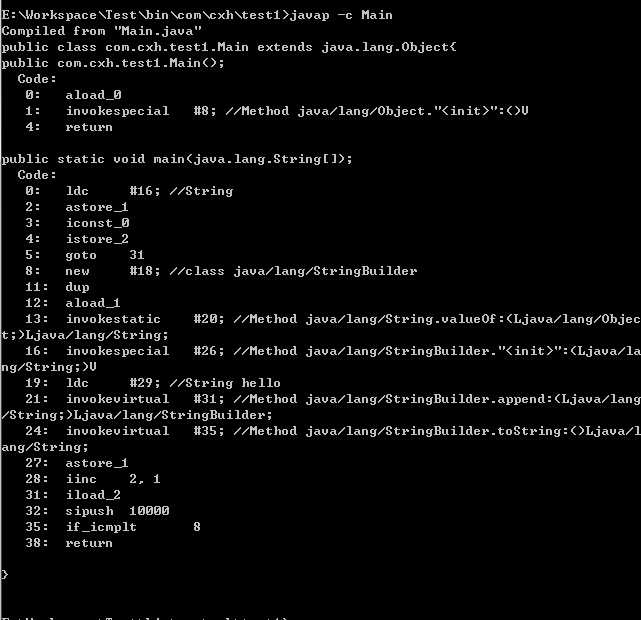
****

既然在Java中已经存在了String类，那为什么还需要StringBuilder和StringBuffer类呢？

那么看下面这段代码：



这句 string += "hello";的过程相当于将原有的string变量指向的对象内容取出与"hello"作字符串相加操作再存进另一个新的String对象当中，再让string变量指向新生成的对象。如果大家还有疑问可以反编译其字节码文件便清楚了：

****

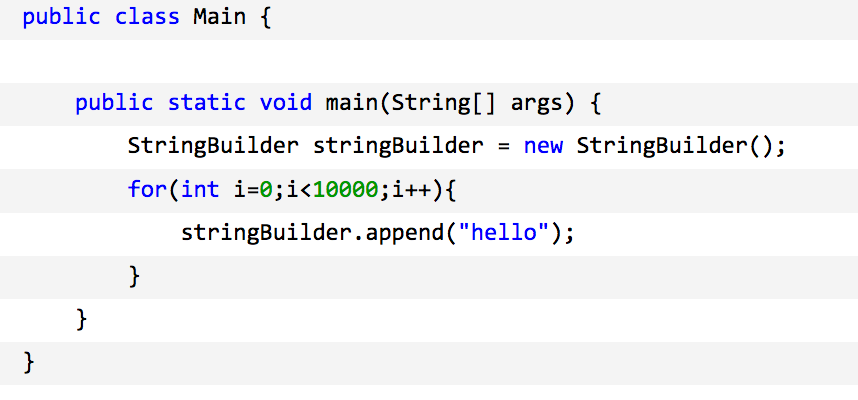
从这段反编译出的字节码文件可以很清楚地看出：从第8行开始到第35行是整个循环的执行过程，并且每次循环会new出一个StringBuilder对象，然后进行append操作，最后通过toString方法返回String对象。也就是说这个循环执行完毕new出了10000个对象，试想一下，如果这些对象没有被回收，会造成多大的内存资源浪费。从上面还可以看出：string+="hello"的操作事实上会自动被JVM优化成：

StringBuilder str = new StringBuilder(string);

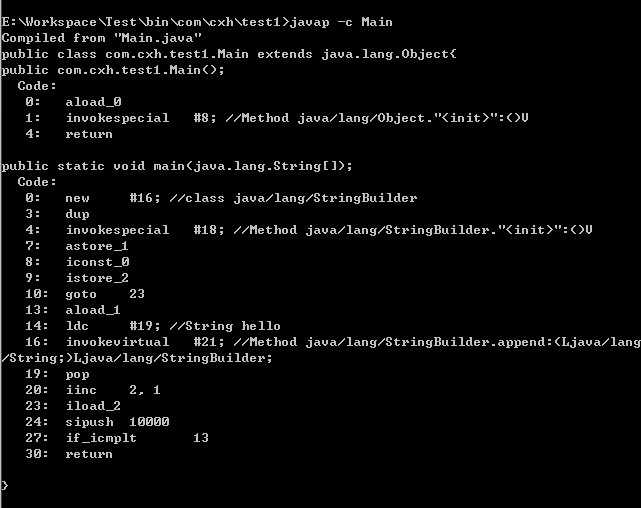
str.append("hello");

str.toString();

再看如下代码：



反编译字节码文件得到：

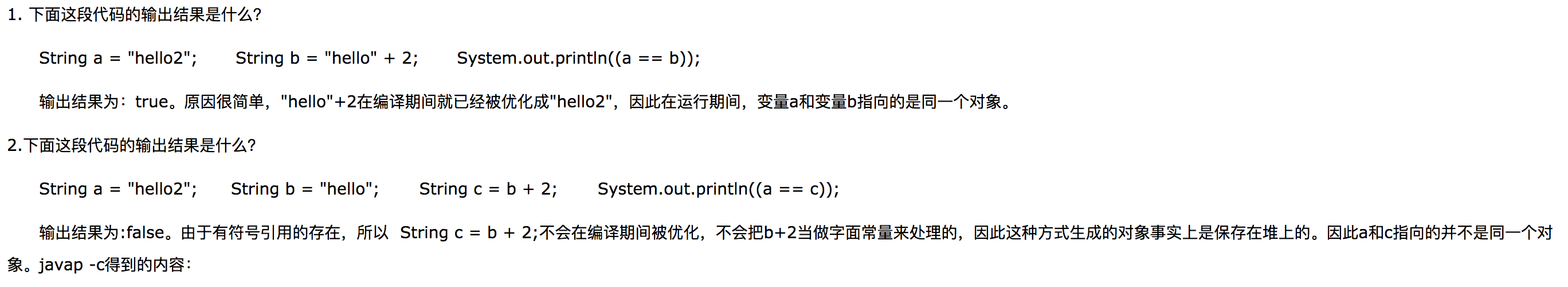
****

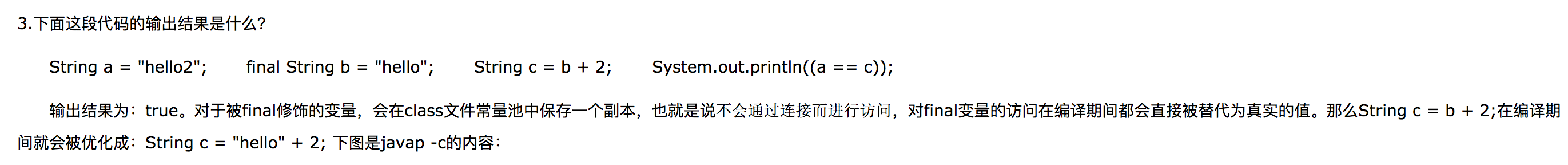
从这里可以明显看出，这段代码的for循环式从13行开始到27行结束，并且new操作只进行了一次，也就是说只生成了一个对象，append操作是在原有对象的基础上进行的。因此在循环了10000次之后，这段代码所占的资源要比上面小得多。这就是为什么要用StringBuilder的原因。

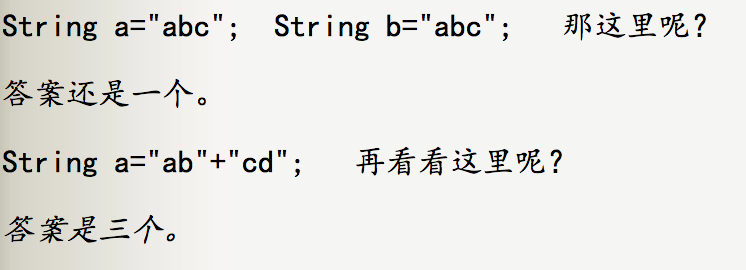
**StringBuffer类：**

那么有人会问既然有了StringBuilder类，为什么还需要StringBuffer类？查看源代码便一目了然，事实上，StringBuilder和StringBuffer类拥有的成员属性以及成员方法基本相同，区别是StringBuffer类的成员方法前面多了一个关键字：synchronized，不用多说，这个关键字是在多线程访问时起到安全保护作用的,也就是说StringBuffer是线程安全的。

**面试题：**







String str = new String("abc")创建了多少个对象？

这个问题在很多书籍上都有说到比如《Java程序员面试宝典》，包括很多国内大公司笔试面试题都会遇到，大部分网上流传的以及一些面试书籍上都说是2个对象，这种说法是片面的。

这道题目让人混淆的地方就是这里，这段代码在运行期间确实只创建了一个对象，即在堆上创建了"abc"对象。而为什么大家都在说是2个对象呢，这里面要澄清一个概念 该段代码执行过程和类的加载过程是有区别的。在类加载的过程中，确实在运行时常量池中创建了一个"abc"对象，而在代码执行过程中确实只创建了一个String对象。

因此，这个问题如果换成 String str = new String("abc")涉及到几个String对象？合理的解释是2个。

1. **啊啊啊**