# Java

## Hashmap和hashtable区别

1 Hashmap是实现了map集合的接口，而hashtable实现了map集合的接口，继承了Dictionary抽象类。

2 hashmap是非线程安全的，hashtable是线程安全的。

Hashtable的方法里面都加了synchronized关键字进行同步。因此在单线程情况下，hashmap效率比hashtable高，尽量使用hashmap。

3 hashmap的键值对允许空值，但是hashtable不允许存放空值。

4 hashmap的迭代器，也就是Iterator是fail-fast迭代器，而hashtable的enumerator迭代器不是fail-fast。因此，当有其他线程改变了hashmap的结构（添加或者删除元素），将会抛出并发修改异常ConcurrentModificationException。但是迭代器本身的remove方法移除元素时不会抛出该异常。该行为不一定会发生，要看jvm。

5 hashtable采用的是开放定址法解决哈希冲突，而hashmap采用的是链接法解决。Dictionary类采用的是 链接法。

6 hashmap的初始容量为16，而hashtable的初始容量为11，两者的填充因子默认都是0.75

Hashmap扩容时是2\*capacity 而hashtable扩容时是capacity\*2+1

7 两者计算的hash方法不同。

Hashtable计算hash是直接使用key的hashcode对table数组长度进行取模

**int** hash = key.hashCode();

**int** index = (hash & 0x7FFFFFFF) % tab.length;

hashmao计算hash时对key的hashcode进行了二次hash，以获得更好的散列值，然后对数组的长度进行取模。

可以使用hashmap做缓存

多线程时一般使用concurrenthashmap，它用的是分段锁，因此效率比hashtable高。

## Hashmap和treemap区别

HashMap通过hashcode对其内容进行快速查找，而 TreeMap中所有的元素都保持着某种固定的顺序，如果你需要得到一个有序的结果你就应该使用TreeMap（HashMap中元素的排列顺序是不固定的）。

Hashmap继承了AbstractMap抽象类，Treemap继承了AbstractMap抽象类，还实现了SortedMap接口。SortedMap接口为映像的视图（子集），包括两个端点提供了访问方法。除了排序是作用于映射的键以外，处理SortedMap和处理SortedSet一样。添加到SortedMap实现类的元素必须实现Comparable接口，否则您必须给它的构造函数提供一个Comparator接口的实现。TreeMap类是它的唯一一份实现。

Hashmap是基于哈希表的，使用hashmap要求添加的 键 类明确定义了hashcode和equals方法，为了优化hashmap空间的使用，可以调优初始容量和负载因子。

（1）HashMap（）： 构建一个空的哈希映像

　　（2）HashMap（Map m）： 构建一个哈希映像，并且添加映像m的所有映射

　　（3）HashMap（int initialCapacity）： 构建一个拥有特定容量的空的哈希映像

（4）HashMap（int initialCapacity, float loadFactor）： 构建一个拥有特定容量和加载因子的空的哈希映像

TreeMap是基于红黑树的实现，TreeMap没有调优选项，因为该树总是处于平衡状态。

（1）TreeMap（）：构建一个空的映像树

　　（2）TreeMap（Map m）： 构建一个映像树，并且添加映像m中所有元素

　　（3）TreeMap（Comparator c）： 构建一个映像树，并且使用特定的比较器对关键字进行排序

　　（4）TreeMap（SortedMap s）： 构建一个映像树，添加映像树s中所有映射，并且使用与有序映像s相同的比较器排序

Hashmap适应于在map中插入 删除 定位元素

Treemap适应于按顺序遍历键

Hashmap比treemap快一点，因为哈希表和树的结构使然，哈希表最好时O(1) 树O(lonN)

## 红黑树

<http://blog.csdn.net/z702143700/article/details/49079107>

BST二叉查找树最好的时间复杂度是O(logN) 最坏的时间复杂度O(N) 不一定平衡

插入时不需要改变树中原有的结构

平衡二叉查找树（AVL） 是严格平衡的BST（平衡因子不超过1） 不会出现最差的情况单支树 最坏情况下都是O（logN）级别

AVL树在插入时，最多需要进行一次旋转，其时间负载度在O(logN)左右

AVL树在删除时代价较大，删除后需要检查从删，除节点开始到根节点路径上的所有节点的平衡因子。删除时最多需要O（logN）次旋转，故总的复杂度O(logN)+O(logN)

R—B Tree。是一棵特殊的二叉树，是自平衡的二叉查找树。不是像AVL一样严格平衡。但是平衡性能比BST好。

AVL严格的平衡策略以牺牲建立查找结构（插入和删除）的代价，换来较小的查找时间复杂度。折中一下——RBT

特性：每个节点是黑色或者红色

根节点是黑色

叶节点（NULL）是黑色

如果一个节点是红色，则它的子节点必须是黑色

从一个节点到该节点的子孙节点的所有路径上包含相同数目的黑节点。

主要用来存储有序的数据，时间复杂度O(logN)，删除时，需要一个查找的代价，然后改变一下树的形态。删除操作时间复杂度最大为O(logN)

查找时，由于最长路径不会超过最短路径的2倍，说明红黑树虽然不像AVL一样严格平衡，但是平衡性能比BST好。查找代价基本维持在O(logN)。

插入时，需要旋转和变色操作。但是只要保证RBT平衡即可，因此插入节点最多只需要两次旋转，和AVL插入一样，变色操作需要O(logN)，但是变色操作十分简单，代价小。

删除时，RBT删除的代价比AVL好很多，删除一个节点最多需要3次旋转。

**一棵含有n个节点的红黑树的高度至多为2log(n+1)**.

## 如何让hashmap变同步

Map map = Collections.synchronizedMap(hashMap);

Collections.synchronizedMap()实现原理是Collections定义了一个SynchronizedTMBap的内部类，这个类实现了Map接口，在调用方法时使用synchronized来保证线程同步,当然了实际上操作的还是我们传入的HashMap实例，简单的说就是Collections.synchronizedMap()方法帮我们在操作HashMap时自动添加了synchronized来实现线程同步，类似的其它Collections.synchronizedXX方法也是类似原理

## Hashmap和hashset区别

Hashset实现了set集合的接口，不允许有重复的值，将对象存储在hashset之前，需要确保对象已经重写了equals和hashcode方法，这样才能 比较两个对象的值是否相等，确保set中没有存储相等的对象。

Hashmap实现了map集合接口，对键值对进行映射。Map中不允许重复的键。

Hashmap存储的是键值对，hashset存储的是对象。

Hashmap使用put方法将元素放入map中，hashset使用add方法将元素放入set中。

Hashmap中使用键对象计算hashcode的值，hashset中使用成员对象来计算hashcode值。

Hashmap比较快，因为使用唯一的键来获取对象，hashset比hashmap慢。

Hashset内部使用hashmap实现，只不过hashset里面的hashmap所有的value都是同一个object而已。**private** **transient** HashMap<E,Object> map;只是包含了hashmap的key。

## Fail-fast机制

快速失败，是java集合中的一种错误检测机制。当多个线程对集合进行结构上的改变操作时，就有可能会产生fail-fast机制。例如存在两个线程1和2，线程1通过Iterator在遍历集合中元素时，线程2修改了集合的结构（添加或者删除元素），这个时候就会抛出ConcurrentModificationException异常，从而产生了fail-fast机制。

并发修改异常ConcurrentModificationException：方法检测到对象的并发修改时，但是不允许这种修改，就会抛出该异常。单线程和多线程时都有可能产生这种异常。

分析ArrayList源码可知，迭代器在调用next() 和remove()方法时，都会调用checkForComodification()方法，该方法主要就是检测modCount和expectedModCount是否相等。如果不相等则抛出异常，从而产生了fail-fast机制。

modCount用来记录集合修改的次数，每修改一次（添加或者删除），modCount++。

**具体分析如下：**

**线程1在遍历集合A。调用了集合的iterator方法。**

**public** Iterator<E> iterator() {

**return** **new** Itr();

}

**Itr是ArrayList的内部类，实现了Iterator接口**

**private** **class** Itr **implements** Iterator<E> {

**int** cursor;

**int** lastRet = -1;

**int** expectedModCount = modCount;

**………省略**

@SuppressWarnings("unchecked")

**public** E next() {

checkForComodification();

**………省略**

}

**public** **void** remove() {

**if** (lastRet < 0)

**throw** **new** IllegalStateException();

checkForComodification();

**………省略**

}

@Override

@SuppressWarnings("unchecked")

**public** **void** forEachRemaining(Consumer<? **super** E> consumer) {

**………省略**

checkForComodification();

}

**final** **void** checkForComodification() {

**if** (modCount != expectedModCount)

**throw** **new** ConcurrentModificationException();

}

}

**由上述可知，在调用remove和next方法时都会调用checkForComodification()方法。**

**看看checkForComodification()方法的实现，代码如下：**

**final** **void** checkForComodification() {

**if** (modCount != expectedModCount)

**throw** **new** ConcurrentModificationException();

}

**这里进行了modCount 和expectedModCount的比较。如果不相等，则抛出ConcurrentModificationException异常。**

**下面分析一下什么时候才会不相等？**

**private** **class** Itr **implements** Iterator<E> {

**int** expectedModCount = modCount;

**在新建itr对象时，会把当前的modCount的值传递给expectedModCount，之后, expectedModCount的值就不会再改变。因此下面要分析一下modCount什么时候会发生改变？以add方法为例**

**public** **void** add(**int** index, E e) {

rangeCheckForAdd(index);

checkForComodification();

parent.add(parentOffset + index, e);

//修改modCount的值

**this**.modCount = parent.modCount;

**this**.size++;

}

**观察源码可知，add remove clear方法，只要涉及到修改集合结构时，就会改变modCount的值。**

**继续刚才线程1的执行，假设这时线程2执行了add方法，向集合中添加了一个元素，此时modCount++，线程1接着遍历，在执行到next函数时，调用checkForComodification方法比较expectedModCount和modCount的值，发现不相等了，就会抛出并发修改异常。从而产生了fail-fast机制。**

**如何解决fail-fast呢？或者解决并发修改异常呢？**

**可以使用同步来解决，在客户端调用会改变modCount值的方法时，加synchronized，或者直接使用Collections.synchronizedList类。**

**还可以使用jdk5提供的CopyOnWriteArrayList类。**

**CopyOnWriteArrayList仅仅实现了List集合接口，并没有继承AbstractList抽象类。ArrayList的iterator()方法是继承了AbstractList,但是CopyOnWriteArrayList是自己实现了iterator。最主要的原因是CopyOnWriteArrayList的Iterator实现类中没有checkForComodification方法，所以不会抛出并发修改异常。**

**那CopyOnWriteArrayList实现的原理是什么？以add方法为例**

**public** **boolean** add(E e) {

**final** ReentrantLock lock = **this**.lock;

lock.lock();

**try** {

Object[] elements = getArray();

**int** len = elements.length;

Object[] newElements = Arrays.*copyOf*(elements, len + 1);

newElements[len] = e;

setArray(newElements);

**return** **true**;

} **finally** {

lock.unlock();

}

}

**关键就在于**

Object[] newElements = Arrays.*copyOf*(elements, len + 1);

newElements[len] = e;

setArray(newElements);

**这三行代码。它是先对原先的数组进行复制，然后在复制之后的数组上进行添加元素，最后在改变原有数据的引用即可。怪不得该类叫做CopyOnWriteArrayList，无疑是先复制再进行写操作！**

## ArrayList和LinkedList区别

ArrayList是实现了基于数组的数据结构（动态数组，在java中，使用的是新建一个数组进行扩容，然后copy原来数组中内容，实现数组可增长），LinkedList是基于链表的数据结构。（双向循环链表）

对于随机访问的get和set操作，ArrayList性能优于LinkedList，因为LinkedList要移动指针。

对于增加和删除操作，LinkedList性能较好，因为ArrayList要移动数据。

## Object中的方法

### Clone方法

<http://blog.csdn.net/wangyanguiyiyang/article/details/49800493>

protected [Object](http://blog.csdn.net/uniquewonderq/article/details/46803261) **clone**() throws [CloneNotSupportedException](http://blog.csdn.net/uniquewonderq/article/details/46803261)

创建并返回此对象的一个副本。

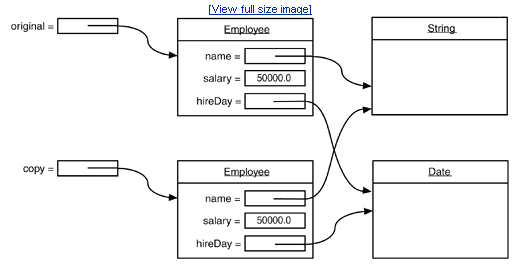
修饰符为protected，保证了只有在该类 同一个包下 以及子类可以访问。但是如果子类和基类不在同一个包下，子类在进行访问的时候，只能访问自身从基类继承过来的protected方法，不能访问基类实例中受保护的clone方法。如果子类和基类在同一个包下，那么protected就相当于public。

原来对象修改某些属性后，clone之后的新对象的属性并不会发生变化。（如果属性值是一个引用，这就例外了，因为都指向同一块内存区域，这就是浅表复制。如果是深表复制，那么在拷贝的时候会把引用所指向的那个对象也拷贝过来，不是拷贝那个引用本身了，这样就可以做到原有对象和拷贝之后的新对象完全独立，一方的改变不会影响另外一方）。

**Clone之后的对象和之前的对象的类型是一样的(getClass).**

**浅表拷贝：**

在对某个对象拷进行clone时，对其是一无所知的，仅仅是简单的执行对属性的copy。



上述的拷贝过程，由于属性hireDay是一个引用（对象），那么在拷贝的时候只会把这个引用拷贝过来，新的对象和原来的对象都指向Date这块内存了。

那么如何实现深表拷贝呢？一种方法是通过对那些非基本数据类型的属性进行特殊的处理，重新定义clone方法。如下：

Class Employee implements Cloneable {

**public** object clone() throws CloneNotSupportedException {

Emploee clonedEmploee = (Employee)super.clone();

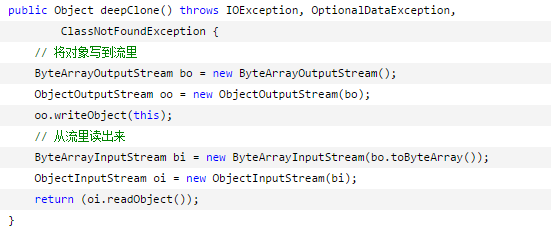
clonedEmploee.hireDay = (Date)hireDay.clone();

return clonedEmploee;

}

}

另外一种方法就是通过序列化，转换为流的方式进行读写。



在使用的时候要注意是使用浅表拷贝还是深表拷贝。

**调用clone方法的对象所属的类必须实现Cloneable接口，重写clone方法，并设置访问修饰符为public，方法中调用super.clone()方法得到需要复制的对象。**

**http://www.cnblogs.com/dolphin0520/p/3700693.html**

### equals方法

**boolean equals(Object obj)**

指示某个其他对象是否与此对象“相等”

==运算符判断两个引用是否指向同一个对象。

Object类中的equal方法：

public boolean equals(Object obj)

{

return (this == obj);

}

判断调用equals方法的引用和传进来的引用是否一致，也就是是否指向同一个对象。

**特点**

**自反性**（reflexive）：任何非空引用x，x.equals(x)返回为true。

**对称性**（symmetric）：任何非空引用x和y，x.equals(y)返回true当且仅当y.equals(x)返回true。

**传递性**（transitive）：任何非空引用x和y，如果x.equals(y)返回true，并且y.equals(z)返回true，那么x.equals(z)返回true。

**一致性**（consistent）：两个非空引用x和y，x.equals(y)的多次调用应该保持一致的结果，（前提条件是在多次比较之间没有修改x和y用于比较的相关信息）。

**约定**：对于任何非空引用x，x.equals(null)应该返回为false。

**并且覆写equals()方法时，应该同时覆写hashCode()方法，反之亦然。（满足hashcode的常规协定，也就是对于两个相等的对象返回的hashcode值应该一样，对于两个不相等的对象，返回的hashcode值不一定要不同）**

### hashCode方法

**int hashCode()**

返回该对象的哈希码值。

当重写了equals方法，必须重写hashcode方法。

这个方法返回一个整型值（hash code value），如果两个对象被equals()方法判断为相等，那么它们就应该拥有同样的hash code。Object类的hashCode()方法为不同的对象返回不同的值，Object类的hashCode值表示的是对象的地址。

hashCode 的常规协定是：

1.在 Java 应用程序执行期间，在对同一对象多次调用 hashCode 方法时，必须一致地返回相同的整数，前提是将对象进行 equals 比较时所用的信息没有被修改。从某一应用程序的一次执行到同一应用程序的另一次执行，该整数无需保持一致。

2.如果根据 equals(Object) 方法，两个对象是相等的，那么对这两个对象中的每个对象调用 hashCode 方法都必须生成相同的整数结果。

3.如果根据 equals(java.lang.Object) 方法，两个对象不相等，那么对这两个对象中的任一对象上调用 hashCode 方法不要求一定生成不同的整数结果。但是，程序员应该意识到，为不相等的对象生成不同整数结果可以提高哈希表的性能。

### toString方法

**String toString()**

**返回该对象的字符串表示。**

当打印引用，如调用System.out.println()时，会自动调用对象的toString()方法，打印出引用所指的对象的toString()方法的返回值，因为每个类都直接或间接地继承自Object，因此每个类都有toString()方法。

Object类中的toString()方法定义如下：

public String toString()

{

return getClass().getName() + "@" + Integer.toHexString(hashCode());

}

### finalize方法

**protected void finalize( )**

关键字protected是防止在该类之外定义的代码访问finalize()标识符。

垃圾回收器准备释放内存时，回收某个对象时，会先调用该对象的finalize方法。

1 对象不一定会被回收

2 垃圾回收不是析构函数

3 垃圾回收只与内存有关

4 垃圾回收和finalize都是靠不住的，只要JVM还没有到快耗尽内存的地步，它是不会浪费时间进行垃圾回收的。

有时当撤消一个对象时，需要完成一些操作。例如，如果一个对象正在处理的是非Java 资源，如文件句柄或window 字符字体，这时你要确认在一个对象被撤消以前要保证这些资源被释放。为处理这样的状况，Java 提供了被称为收尾（finalization ）的机制。使用该机制你可以定义一些特殊的操作，这些操作在一个对象将要被垃圾回收程序释放时执行。

在 Java 中，当你创建一个对象时，Java 虚拟机（JVM）为该对象分配内存、调用构造函数并开始跟踪你使用的对象。当你停止使用一个对象（就是说，当没有对该对象有效的引用时），JVM 通过垃圾回收器将该对象标记为释放状态。

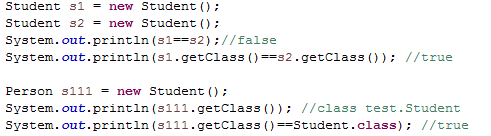
当垃圾回收器将要释放一个对象的内存时，它调用该对象的finalize() 方法（如果该对象定义了此方法）。垃圾回收器以独立的低优先级的方式运行，只有当其他线程挂起等待该内存释放的情况出现时，它才开始运行释放对象的内存。（事实上，你可以调用System.gc() 方法强制垃圾回收器来释放这些对象的内存。）

### getClass方法

public final Class<?> **getClass**()

返回一个对象的运行时类。

例子：Student类继承Person类



如果知道一个实例，那么你可以通过实例的“getClass()”方法获得该对象的类型类，如果你知道一个类型，那么你可以使用“.class”的方法获得该类型的类型类。

### notify方法

唤醒在此对象监视器上等待的单个线程。

### notifyAll方法

唤醒在此对象监视器上等待的所有线程。

### wait方法

void wait() //导致当前的线程等待，直到其他线程调用此对象的 notify() 方法或 notifyAll() 方法。

void wait(long timeout) //导致当前的线程等待，直到其他线程调用此对象的 notify() 方法或notifyAll() 方法，或者超过指定的时间量。

void wait(long timeout, int nanos) //导致当前的线程等待，直到其他线程调用此对象的notify() 方法或 notifyAll() 方法，或者其他某个线程中断当前线程，或者已超过某个实际时间量。

## 权限修饰符

                   类内部     package内       子类         其他   
public           **允许         允许                允许         允许** protected       **允许         允许               允许**         不允许   
default           **允许         允许**                不允许     不允许   
private           **允许         不允许**           不允许     不允许

对于protected的成员或方法，要分子类和超类是否在同一个包中。与基类不在同一个包中的子类，只能访问自身从基类继承而来的受保护成员，而不能访问基类实例本身的受保护成员。在相同包时,protected和public是一样的

## final finally finalize区别

final

修饰符（关键字）如果一个类被声明为final，意味着它不能再派生出新的子类，不能作为父类被继承。因此一个类不能既被声明为 abstract的，又被声明为final的。将变量或方法声明为final，可以保证它们在使用中不被改变。被声明为final的变量必须在声明时给定初值，而在以后的引用中只能读取，不可修改。被声明为final的方法不可以重写，但是可以重载。

finally

异常处理时提供 finally 块来执行任何清除操作。如果抛出一个异常，那么相匹配的 catch 子句就会执行，然后控制就会进入 finally 块（如果有的话）。一般异常处理块需要。

finalize

方法名。Java 技术允许使用 finalize() 方法在垃圾收集器将对象从内存中清除出去之前做必要的清理工作。这个方法是由垃圾收集器在确定这个对象没有被引用时对这个对象调用的。它是在 Object 类中定义的，因此所有的类都继承了它。子类覆盖 finalize() 方法以整理系统资源或者执行其他清理工作。finalize() 方法是在垃圾收集器删除对象之前对这个对象调用的。

Java中所有类都从Object类中继承finalize()方法。

当垃圾回收器(garbage colector)决定回收某对象时，就会运行该对象的finalize()方法。值得C++程序员注意的是，finalize()方法并不能等同与析构函数。Java中是没有析构函数的。C++的析构函数是在对象消亡时运行的。由于C++没有垃圾回收，对象空间手动回收，所以一旦对象用不到时，程序员就应当把它delete()掉。所以析构函数中经常做一些文件保存之类的收尾工作。但是在Java中很不幸，如果内存总是充足的，那么垃圾回收可能永远不会进行，也就是说filalize()可能永远不被执行，显然指望它做收尾工作是靠不住的。

那么finalize()究竟是做什么的呢？它最主要的用途是回收特殊渠道申请的内存。Java程序有垃圾回收器，所以一般情况下内存问题不用程序员操心。但有一种JNI(Java Native Interface)调用non-Java程序（C或C++），finalize()的工作就是回收这部分的内存。

## Jdk8新特性

### java语言新特性

#### 1 lambda表达式

Lambda表达式（也称为闭包）是整个Java 8发行版中最受期待的在Java语言层面上的改变，Lambda允许把函数作为一个方法的参数（函数作为参数传递进方法中），或者把代码看成数据：函数式程序员对这一概念非常熟悉。在JVM平台上的很多语言（Groovy，[Scala](http://www.javacodegeeks.com/tutorials/scala-tutorials/)，……）从一开始就有Lambda，但是Java程序员不得不使用毫无新意的匿名类来代替lambda。

在最简单的形式中，一个lambda可以由用逗号分隔的参数列表、–>符号与函数体三部分表示。例如：

Arrays.asList( "a", "b", "d" ).forEach( e -> System.out.println( e ) );

参数e的类型是由编译器推测出来的。同时，你也可以通过把参数类型与参数包括在括号中的形式直接给出参数的类型：

Arrays.asList( "a", "b", "d" ).forEach( ( String e ) -> System.out.println( e ) );

在某些情况下lambda的函数体会更加复杂，这时可以把函数体放到在一对花括号中，就像在Java中定义普通函数一样。例如：

Arrays.asList( "a", "b", "d" ).forEach( e -> {

    System.out.print( e );

    System.out.print( e );

} );

Lambda可以引用类的成员变量与局部变量（如果这些变量不是final的话，它们会被隐含的转为final，这样效率更高）。例如，下面两个代码片段是等价的：

String separator = ",";

Arrays.asList( "a", "b", "d" ).forEach(

    ( String e ) -> System.out.print( e + separator ) );

和

final String separator = ",";

Arrays.asList( "a", "b", "d" ).forEach(

    ( String e ) -> System.out.print( e + separator ) );

Lambda可能会返回一个值。返回值的类型也是由编译器推测出来的。如果lambda的函数体只有一行的话，那么没有必要显式使用return语句。下面两个代码片段是等价的：

Arrays.asList( "a", "b", "d" ).sort( ( e1, e2 ) -> e1.compareTo( e2 ) );

和

Arrays.asList( "a", "b", "d" ).sort( ( e1, e2 ) -> {

    int result = e1.compareTo( e2 );

    return result;

} );

Lambda是Java 8最大的卖点。它具有吸引越来越多程序员到Java平台上的潜力，并且能够在纯Java语言环境中提供一种优雅的方式来支持函数式编程。

例子：

**public** **class** Lambda {

/\*

\* 使用匿名内部类，然后重写方法

\*/

@Test

**public** **void** test1() {

Integer[] arr = {2,3,1,55,2,12};

Arrays.*sort*(arr, **new** Comparator<Integer>() {

@Override

**public** **int** compare(Integer i, Integer j) {

**return** j-i;

}

});

**for**(Integer i : arr) {

System.***out***.print(i+" ");

}

}

/\*

\* 在jdk8中不需要匿名内部类了，可以采用函数式编程，允许函数作为方法的参数，直接把程序看成数据

\*/

@Test

**public** **void** test2() {

Integer[] arr = {2,3,1,55,2,12};

// Arrays.sort(arr,(a,b)->{

// return b-a;

// });

//这里编译器可以自动推测出参数a和b的类型

Arrays.*sort*(arr,(a,b)->b-a);

**for**(Integer i : arr) {

System.***out***.print(i+" ");

}

}

}

#### 2 函数式接口

语言设计者投入了大量精力来思考如何使现有的函数友好地支持lambda。最终采取的方法是：增加函数式接口的概念。每一个lambda表达式都对应一种类型，通常是接口类型。函数式接口就是一个仅仅只有一个方法的普通接口。像这样的接口，可以被隐式转换为lambda表达式。每一个该类型的lambda表达式都会被匹配到这个抽象方法。

java.lang.Runnable与java.util.concurrent.Callable是函数式接口最典型的两个例子。在实际使用过程中，函数式接口是容易出错的：如有某个人在接口定义中增加了另一个方法，这时，这个接口就不再是函数式的了，并且编译过程也会失败。为了克服函数式接口的这种脆弱性并且能够明确声明接口作为函数式接口的意图，Java 8增加了一种特殊的注解@FunctionalInterface（Java 8中所有类库的已有接口都添加了@FunctionalInterface注解）。让我们看一下这种函数式接口的定义：

@FunctionalInterface

public interface Functional {

    void method();

}

需要记住的一件事是：默认方法与静态方法并不影响函数式接口的契约，可以任意使用：

@FunctionalInterface

public interface FunctionalDefaultMethods {

    void method();

    default void defaultMethod() {

    }

}

例子

**public** **class** Functionalinterface{

/\*

\* 将lambda表达式映射到一个单方法的接口上。

\*/

Converter<String,Integer> con = (from)->Integer.*valueOf*(from);

Integer i = con.convert("123");

}

@FunctionalInterface

**interface** Converter<F,T> {

T convert(F from);

}

#### 3 接口的默认方法与静态方法

Java 8用默认方法与静态方法这两个新概念来扩展接口的声明。默认方法使接口有点像Traits（Scala中特征(trait)类似于Java中的Interface，但它可以包含实现代码，也就是目前Java8新增的功能），但与传统的接口又有些不一样，它允许在已有的接口中添加新方法，而同时又保持了与旧版本代码的兼容性。

默认方法与抽象方法不同之处在于抽象方法必须要求实现，但是默认方法则没有这个要求。相反，每个接口都必须提供一个所谓的默认实现，这样所有的接口实现者将会默认继承它（如果有必要的话，可以覆盖这个默认实现）。让我们看看下面的例子：

private interface Defaulable {

    // Interfaces now allow default methods, the implementer may or

    // may not implement (override) them.

    default String notRequired() {

        return "Default implementation";

    }

}

private static class DefaultableImpl implements Defaulable {

}

private static class OverridableImpl implements Defaulable {

    @Override

    public String notRequired() {

        return "Overridden implementation";

    }

}

Defaulable接口用关键字default声明了一个默认方法notRequired()，Defaulable接口的实现者之一DefaultableImpl实现了这个接口，并且让默认方法保持原样。Defaulable接口的另一个实现者OverridableImpl用自己的方法覆盖了默认方法。

Java 8带来的另一个有趣的特性是接口可以声明（并且可以提供实现）静态方法。例如

private interface DefaulableFactory {

    // Interfaces now allow static methods

    static Defaulable create( Supplier< Defaulable > supplier ) {

        return supplier.get();

    }

}

public static void main( String[] args ) {

    Defaulable defaulable = DefaulableFactory.create( DefaultableImpl::new );

    System.out.println( defaulable.notRequired() );

    defaulable = DefaulableFactory.create( OverridableImpl::new );

    System.out.println( defaulable.notRequired() );

}

控制台输出

Default implementation

Overridden implementation

在JVM中，默认方法的实现是非常高效的，并且通过字节码指令为方法调用提供了支持。默认方法允许继续使用现有的Java接口，而同时能够保障正常的编译过程。这方面好的例子是大量的方法被添加到java.util.Collection接口中去：stream()，parallelStream()，forEach()，removeIf()，……

尽管默认方法非常强大，但是在使用默认方法时我们需要小心注意一个地方：在声明一个默认方法前，请仔细思考是不是真的有必要使用默认方法，因为默认方法会带给程序歧义，并且在复杂的继承体系中容易产生编译错误。更多详情请参考

#### 4 方法引用

方法引用提供了非常有用的语法，可以直接引用已有Java类或对象（实例）的方法或构造器。与lambda联合使用，方法引用可以使语言的构造更紧凑简洁，减少冗余代码。

下面，我们以定义了4个方法的Car这个类作为例子，区分Java中支持的4种不同的方法引用。

public static class Car {

    public static Car create( final Supplier< Car > supplier ) {

        return supplier.get();

    }

    public static void collide( final Car car ) {

        System.out.println( "Collided " + car.toString() );

    }

    public void follow( final Car another ) {

        System.out.println( "Following the " + another.toString() );

    }

    public void repair() {

        System.out.println( "Repaired " + this.toString() );

    }

}

Java 8 允许你使用 :: 关键字来传递方法或者构造函数引用

第一种方法引用是构造器引用，它的语法是Class::new，或者更一般的Class< T >::new。

请注意构造器没有参数。

第二种方法引用是静态方法引用，它的语法是Class::static\_method。请注意这个方法接受一个Car类型的参数。

cars.forEach( Car::collide );

第三种方法引用是特定类的任意对象的方法引用，它的语法是Class::method。请注意，这个方法没有参数。

cars.forEach( Car::repair );

最后，第四种方法引用是特定对象的方法引用，它的语法是instance::method。请注意，这个方法接受一个Car类型的参数。

final Car police = Car.create( Car::new );

cars.forEach( police::follow );

控制台输出：

Collided com.javacodegeeks.java8.method.references.MethodReferences$Car@7a81197d

Repaired com.javacodegeeks.java8.method.references.MethodReferences$Car@7a81197d

Following the com.javacodegeeks.java8.method.references.MethodReferences$Car@7a81197d

#### 5 重复注解

先解释一下什么是注解吧。注解Annotation，是jdk5之后引入的一个特性，与类、接口和枚举是在同一个层次，它们都被称作java的一个类型（TYPE）。可以声明在包、类、方法、字段、局部变量和方法参数等的前面，用于对这些元素进行说明、注释。

作用：

生成文档。常见@return @param @see

跟踪代码的依赖性。实现替代配置文件的功能。

在编译时进行格式检查。@override放在方法前，如果这个方法不是覆盖了超类的方法，则在编译时就会报错。

元注解的作用就是负责注解其他注解。Java5.0定义了4个标准的meta-annotation类型，它们被用来提供对其它 annotation类型作说明。Java5.0定义的元注解：  
　　　　1.@Target,  
　　　　2.@Retention,  
　　　　3.@Documented,  
　　　　[4.@Inherited](mailto:4.@Inherited)

@Target说明了Annotation所修饰的对象范围：Annotation可被用于 packages、types（类、接口、枚举、Annotation类型）、类型成员（方法、构造方法、成员变量、枚举值）、方法参数和本地变量（如循环变量、catch参数）。在Annotation类型的声明中使用了target可更加明晰其修饰的目标。

　**作用：用于描述注解的使用范围（即：被描述的注解可以用在什么地方）**

**取值(ElementType)有：**

　　　　1.CONSTRUCTOR:用于描述构造器  
　　　　2.FIELD:用于描述域  
　　　　3.LOCAL\_VARIABLE:用于描述局部变量  
　　　　4.METHOD:用于描述方法  
　　　　5.PACKAGE:用于描述包  
　　　　6.PARAMETER:用于描述参数  
　　　　7.TYPE:用于描述类、接口(包括注解类型) 或enum声明

**@Retention**定义了该Annotation被保留的时间长短：某些Annotation仅出现在源代码中，而被编译器丢弃；而另一些却被编译在class文件中；编译在class文件中的Annotation可能会被虚拟机忽略，而另一些在class被装载时将被读取（请注意并不影响class的执行，因为Annotation与class在使用上是被分离的）。使用这个meta-Annotation可以对 Annotation的“生命周期”限制。

**作用：表示需要在什么级别保存该注释信息，用于描述注解的生命周期（即：被描述的注解在什么范围内有效）**

**取值（RetentionPoicy）有：**

　　　　1.SOURCE:在源文件中有效（即源文件保留）  
　　　　2.CLASS:在class文件中有效（即class保留）  
　　　　3.RUNTIME:在运行时有效（即运行时保留）

　　Retention meta-annotation类型有唯一的value作为成员，它的取值来自java.lang.annotation.RetentionPolicy的枚举类型值。

**@**Documented用于描述其它类型的annotation应该被作为被标注的程序成员的公共API，因此可以被例如javadoc此类的工具文档化。Documented是一个标记注解，没有成员。

@Inherited 元注解是一个标记注解，@Inherited阐述了某个被标注的类型是被继承的。如果一个使用了@Inherited修饰的annotation类型被用于一个class，则这个annotation将被用于该class的子类。

　　注意：@Inherited annotation类型是被标注过的class的子类所继承。类并不从它所实现的接口继承annotation，方法并不从它所重载的方法继承annotation。

　　当@Inherited annotation类型标注的annotation的Retention是RetentionPolicy.RUNTIME，则反射API增强了这种继承性。如果我们使用java.lang.reflect去查询一个@Inherited annotation类型的annotation时，反射代码检查将展开工作：检查class和其父类，直到发现指定的annotation类型被发现，或者到达类继承结构的顶层。

**自定义注解：**

　　使用@interface自定义注解时，自动继承了java.lang.annotation.Annotation接口，由编译程序自动完成其他细节。在定义注解时，不能继承其他的注解或接口。@interface用来声明一个注解，其中的每一个方法实际上是声明了一个配置参数。方法的名称就是参数的名称，返回值类型就是参数的类型（返回值类型只能是基本类型、Class、String、enum）。可以通过default来声明参数的默认值。

**定义注解格式：**  
　　public @interface 注解名 {定义体}

**注解参数的可支持数据类型：**

　　　　1.所有基本数据类型（int,float,boolean,byte,double,char,long,short)  
　　　　2.String类型  
　　　　3.Class类型  
　　　　4.enum类型  
　　　　5.Annotation类型  
　　　　6.以上所有类型的数组

　　Annotation类型里面的参数该怎么设定:   
　　第一,只能用public或默认(default)这两个访问权修饰.例如,String value();这里把方法设为defaul默认类型；　 　  
　　第二,参数成员只能用基本类型byte,short,char,int,long,float,double,boolean八种基本数据类型和 String,Enum,Class,annotations等数据类型,以及这一些类型的数组.例如,String value();这里的参数成员就为String;　　  
　　第三,如果只有一个参数成员,最好把参数名称设为"value",后加小括号.例:下面的例子FruitName注解就只有一个参数成员。

下面进入正题！

自从Java 5引入了注解机制，这一特性就变得非常流行并且广为使用。然而，使用注解的一个限制是相同的注解在同一位置只能声明一次，不能声明多次。Java 8打破了这条规则，引入了重复注解机制，这样相同的注解可以在同一地方声明多次。

重复注解机制本身必须用@Repeatable注解。事实上，这并不是语言层面上的改变，更多的是编译器的技巧，底层的原理保持不变。

在没有重复注解之前，比如在spring中想通过@PropertySource多次引入属性文件，这样在jdk8之前，下面的做法是错误的：

@PropertySource("classpath:config.properties")

@PropertySource("file:application.properties")

public class MainApp {

}

上面的代码无法在 Java 7 下通过编译，错误是: Duplicate annotation

在jdk8之前，我们可以自己声明一个新的注解，来包装@ PropertySource

@Retention(RetentionPolicy.RUNTIME)

public @interface PropertySources {

**PropertySource**[] **value**();

}

然后

@PropertySources({

@PropertySource("classpath:config.properties"),

@PropertySource("file:application.properties")

})

public class MainApp {

}

获取注解的内容

PropertySources **annotation** = MainApp.**class**.getAnnotation(PropertySources.**class**);

**for** (PropertySource propertySource: **annotation**.value()){

System.**out**.println(propertySource.value());

}

输出内容

classpath:config.properties

file:application.properties

在jdk8中，引入了注解的注解@Repeatable用来标识某个注解是可被重复使用的。基于这里的例子，如果要实现可重复的注解必须要满足两个条件

1. 前面的那个 **@PropertySources** 实现仍然是必须的，且实现是一样的，用以作为 **@PropertySource**  的容器注解
2. 用 **@Repeatable(PropertySources.class)**  注解 **@PropertySource**

@Retention(RetentionPolicy.RUNTIME)

@Repeatable(PropertySources.class)

public @interface PropertySource {

**String** **value**();

}

**@Repeatable** 是 Java 8 开始提供的， 现在由它来告诉 Java **@PropertySources** 是 **@PropertySource** 的容器注解，而不需要用嵌套的方式来使用它们，因此我们就能使用本文最初的注解形式，也就是

@PropertySource("classpath:config.properties")

@PropertySource("file:application.properties")

public class MainApp {}

通过反射获得注解的内容

PropertySource[] propertySources = MainApp.**class**.getAnnotationsByType(PropertySource.**class**);

**for** (PropertySource propertySource: propertySources ){

System.out.println(propertySource.value()); //获得所有 @PropertySource 的内容

}

解释 **MainApp.class.getAnnotationsByType(PropertySource.class)** 的执行过程：

1. 如果能找到 **@Repeatable** 关联的容器注解类 **@PropertySources** , 就获得 **@PropertySources**的所有 value(类型为 **@PropertySource** ) 值组成的数组
2. 如果未有关联的容器注解类，则返回 **@PropertySource** 本身组成的数组(只有一个元素), 此时和 **new PropertySource[]{MainApp.class.getAnnotation(PropertySource.class}** 一样的。

#### 6 更好的类型推测机制

Java 8在类型推测方面有了很大的提高。在很多情况下，编译器可以推测出确定的参数类型，这样就能使代码更整洁。让我们看一个例子：

public class Value< T > {

    public static< T > T defaultValue() {

        return null;

    }

    public T getOrDefault( T value, T defaultValue ) {

        return ( value != null ) ? value : defaultValue;

    }

}

public class TypeInference {

    public static void main(String[] args) {

        final Value< String > value = new Value<>();

        value.getOrDefault( "22", Value.defaultValue() );

    }

}

Value.defaultValue()的参数类型可以被推测出，所以就不必明确给出。在Java 7中，相同的例子将不会通过编译，正确的书写方式是 Value.< String >defaultValue()。

#### 7 扩展注解的支持

Java 8扩展了注解的上下文。现在几乎可以为任何东西添加注解：局部变量、泛型类、父类与接口的实现，就连方法的异常也能添加注解。下面演示几个例子：

public class Annotations {

    @Retention( RetentionPolicy.RUNTIME )

    @Target( { ElementType.TYPE\_USE, ElementType.TYPE\_PARAMETER } )

    public @interface NonEmpty {

    }

    public static class Holder< @NonEmpty T > extends @NonEmpty Object {

        public void method() throws @NonEmpty Exception {

        }

    }

    @SuppressWarnings( "unused" )

    public static void main(String[] args) {

        final Holder< String > holder = new @NonEmpty Holder< String >();

        @NonEmpty Collection< @NonEmpty String > strings = new ArrayList<>();

    }

}

ElementType.TYPE\_USE和ElementType.TYPE\_PARAMETER是两个新添加的用于描述适当的注解上下文的元素类型。在Java语言中，注解处理API也有小的改动来识别新增的类型注解。

### java编译器新特性

#### 1 参数名字

很长一段时间里，Java程序员一直在发明不同的方式使得方法参数的名字能保留在Java字节码中，并且能够在运行时获取它们（比如，Paranamer类库）。最终，在Java 8中把这个强烈要求的功能添加到语言层面（通过反射API与Parameter.getName()方法）与字节码文件（通过新版的javac的–parameters选项）中。

public class ParameterNames {

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

        Method method = ParameterNames.class.getMethod( "main", String[].class );

        for( final Parameter parameter: method.getParameters() ) {

            System.out.println( "Parameter: " + parameter.getName() );

        }

    }

}

如果不使用–parameters参数来编译这个类，然后运行这个类，会得到下面的输出：

Parameter: arg0

如果使用–parameters参数来编译这个类，程序的结构会有所不同（参数的真实名字将会显示出来）：

Parameter: args

### Java类库新特性

Java 8 通过增加大量新类，扩展已有类的功能的方式来改善对并发编程、函数式编程、日期/时间相关操作以及其他更多方面的支持。

#### Optional类

#### Stream

最新添加的[Stream API](http://www.javacodegeeks.com/2014/05/the-effects-of-programming-with-java-8-streams-on-algorithm-performance.html)（java.util.stream） 把真正的函数式编程风格引入到Java中。这是目前为止对Java类库最好的补充，因为Stream API可以极大提供Java程序员的生产力，让程序员写出高效率、干净、简洁的代码。

Stream API极大简化了集合框架的处理（但它的处理的范围不仅仅限于集合框架的处理，这点后面我们会看到）。让我们以一个简单的Task类为例进行介绍：

public class Streams  {

    private enum Status {

        OPEN, CLOSED

    };

    private static final class Task {

        private final Status status;

        private final Integer points;

        Task( final Status status, final Integer points ) {

            this.status = status;

            this.points = points;

        }

        public Integer getPoints() {

            return points;

        }

        public Status getStatus() {

            return status;

        }

        @Override

        public String toString() {

            return String.format( "[%s, %d]", status, points );

        }

    }

}

Task类有一个分数的概念（或者说是伪复杂度），其次是还有一个值可以为OPEN或CLOSED的状态.让我们引入一个Task的小集合作为演示例子：

final Collection< Task > tasks = Arrays.asList(

    new Task( Status.OPEN, 5 ),

    new Task( Status.OPEN, 13 ),

    new Task( Status.CLOSED, 8 )

);

我们下面要讨论的第一个问题是所有状态为OPEN的任务一共有多少分数？在Java 8以前，一般的解决方式用foreach循环，但是在Java 8里面我们可以使用stream：一串支持连续、并行聚集操作的元素。

final long totalPointsOfOpenTasks = tasks

    .stream()

    .filter( task -> task.getStatus() == Status.OPEN )

    .mapToInt( Task::getPoints )

    .sum();

System.out.println( "Total points: " + totalPointsOfOpenTasks );

控制台输出

Total points: 18

这里有几个注意事项。第一，task集合被转换化为其相应的stream表示。然后，filter操作过滤掉状态为CLOSED的task。下一步，mapToInt操作通过Task::getPoints这种方式调用每个task实例的getPoints方法把Task的stream转化为Integer的stream。最后，用sum函数把所有的分数加起来，得到最终的结果。

在继续讲解下面的例子之前，关于stream有一些需要注意的地方（详情[在这里](http://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html#StreamOps)）.stream操作被分成了中间操作与最终操作这两种。

中间操作返回一个新的stream对象。中间操作总是采用惰性求值方式，运行一个像filter这样的中间操作实际上没有进行任何过滤，相反它在遍历元素时会产生了一个新的stream对象，这个新的stream对象包含原始stream  
中符合给定谓词的所有元素。

像forEach、sum这样的最终操作可能直接遍历stream，产生一个结果或副作用。当最终操作执行结束之后，stream管道被认为已经被消耗了，没有可能再被使用了。在大多数情况下，最终操作都是采用及早求值方式，及早完成底层数据源的遍历。

stream另一个有价值的地方是能够原生支持并行处理。让我们来看看这个算task分数和的例子。

final double totalPoints = tasks

   .stream()

   .parallel()

   .map( task -> task.getPoints() ) // or map( Task::getPoints )

   .reduce( 0, Integer::sum );

System.out.println( "Total points (all tasks): " + totalPoints );

这个例子和第一个例子很相似，但这个例子的不同之处在于这个程序是并行运行的，其次使用reduce方法来算最终的结果。  
下面是这个例子在控制台的输出：

Total points (all tasks): 26.0

经常会有这个一个需求：我们需要按照某种准则来对集合中的元素进行分组。Stream也可以处理这样的需求，下面是一个例子：

final Map< Status, List< Task > > map = tasks

    .stream()

    .collect( Collectors.groupingBy( Task::getStatus ) );

System.out.println( map );

这个例子的控制台输出如下

{CLOSED=[[CLOSED, 8]], OPEN=[[OPEN, 5], [OPEN, 13]]}

让我们来计算整个集合中每个task分数（或权重）的平均值来结束task的例子。

final Collection< String > result = tasks

    .stream()                                        // Stream< String >

    .mapToInt( Task::getPoints )                     // IntStream

    .asLongStream()                                  // LongStream

    .mapToDouble( points -> points / totalPoints )   // DoubleStream

    .boxed()                                         // Stream< Double >

    .mapToLong( weigth -> ( long )( weigth \* 100 ) ) // LongStream

    .mapToObj( percentage -> percentage + "%" )      // Stream< String>

    .collect( Collectors.toList() );                 // List< String >

System.out.println( result );

控制台输出

[19%, 50%, 30%]

最后，就像前面提到的，Stream API不仅仅处理Java集合框架。像从文本文件中逐行读取数据这样典型的I/O操作也很适合用Stream API来处理。下面用一个例子来应证这一点。

final Path path = new File( filename ).toPath();

try( Stream< String > lines = Files.lines( path, StandardCharsets.UTF\_8 ) ) {

    lines.onClose( () -> System.out.println("Done!") ).forEach( System.out::println );

}

对一个stream对象调用onClose方法会返回一个在原有功能基础上新增了关闭功能的stream对象，当对stream对象调用close()方法时，与关闭相关的处理器就会执行。

Stream API、Lambda表达式与方法引用在接口默认方法与静态方法的配合下是Java 8对现代软件开发范式的回应。

#### Date/Time API

Java 8通过发布新的Date-Time API (JSR 310)来进一步加强对日期与时间的处理。对日期与时间的操作一直是Java程序员最痛苦的地方之一。标准的 java.util.Date以及后来的java.util.Calendar一点没有改善这种情况（可以这么说，它们一定程度上更加复杂）。

这种情况直接导致了Joda-Time——一个可替换标准日期/时间处理且功能非常强大的Java API的诞生。Java 8新的Date-Time API (JSR 310)在很大程度上受到Joda-Time的影响，并且吸取了其精髓。新的java.time包涵盖了所有处理日期，时间，日期/时间，时区，时刻（instants），过程（during）与时钟（clock）的操作。在设计新版API时，十分注重与旧版API的兼容性：不允许有任何的改变（从java.util.Calendar中得到的深刻教训）。如果需要修改，会返回这个类的一个新实例。

让我们用例子来看一下新版API主要类的使用方法。第一个是Clock类，它通过指定一个时区，然后就可以获取到当前的时刻，日期与时间。Clock可以替换System.currentTimeMillis()与TimeZone.getDefault()。

final Clock clock = Clock.systemUTC();

System.out.println( clock.instant() );

System.out.println( clock.millis() );

控制台输出

2014-04-12T15:19:29.282Z

1397315969360

我们需要关注的其他类是LocaleDate与LocalTime。LocaleDate只持有ISO-8601格式且无时区信息的日期部分。相应的，LocaleTime只持有ISO-8601格式且无时区信息的时间部分。LocaleDate与LocalTime都可以从Clock中得到

// Get the local date and local time

final LocalDate date = LocalDate.now();

final LocalDate dateFromClock = LocalDate.now( clock );

System.out.println( date );

System.out.println( dateFromClock );

// Get the local date and local time

final LocalTime time = LocalTime.now();

final LocalTime timeFromClock = LocalTime.now( clock );

System.out.println( time );

System.out.println( timeFromClock );

控制台输出

2014-04-12

2014-04-12

11:25:54.568

15:25:54.568

LocaleDateTime把LocaleDate与LocaleTime的功能合并起来，它持有的是ISO-8601格式无时区信息的日期与时间。下面是一个快速入门的例子。

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | // Get the local date/time  final LocalDateTime datetime = LocalDateTime.now();  final LocalDateTime datetimeFromClock = LocalDateTime.now( clock );    System.out.println( datetime );  System.out.println( datetimeFromClock ); |

下面是程序在控制台上的输出：

|  |  |
| --- | --- |
| 1  2 | 2014-04-12T11:37:52.309  2014-04-12T15:37:52.309 |

如果你需要特定时区的日期/时间，那么ZonedDateTime是你的选择。它持有ISO-8601格式具具有时区信息的日期与时间。下面是一些不同时区的例子：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | // Get the zoned date/time  final ZonedDateTime zonedDatetime = ZonedDateTime.now();  final ZonedDateTime zonedDatetimeFromClock = ZonedDateTime.now( clock );  final ZonedDateTime zonedDatetimeFromZone = ZonedDateTime.now( ZoneId.of( "America/Los\_Angeles" ) );    System.out.println( zonedDatetime );  System.out.println( zonedDatetimeFromClock );  System.out.println( zonedDatetimeFromZone ); |

下面是程序在控制台上的输出：

|  |  |
| --- | --- |
| 1  2  3 | 2014-04-12T11:47:01.017-04:00[America/New\_York]  2014-04-12T15:47:01.017Z  2014-04-12T08:47:01.017-07:00[America/Los\_Angeles] |

最后，让我们看一下Duration类：在秒与纳秒级别上的一段时间。Duration使计算两个日期间的不同变的十分简单。下面让我们看一个这方面的例子。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | // Get duration between two dates  final LocalDateTime from = LocalDateTime.of( 2014, Month.APRIL, 16, 0, 0, 0 );  final LocalDateTime to = LocalDateTime.of( 2015, Month.APRIL, 16, 23, 59, 59 );    final Duration duration = Duration.between( from, to );  System.out.println( "Duration in days: " + duration.toDays() );  System.out.println( "Duration in hours: " + duration.toHours() ); |

上面的例子计算了两个日期2014年4月16号与2014年4月16号之间的过程。下面是程序在控制台上的输出：

|  |  |
| --- | --- |
| 1  2 | Duration in days: 365  Duration in hours: 8783 |

对Java 8在日期/时间API的改进整体印象是非常非常好的。一部分原因是因为它建立在“久战杀场”的Joda-Time基础上，另一方面是因为用来大量的时间来设计它，并且这次程序员的声音得到了认可。

#### JavaScript引擎Nashorn

Nashorn，一个新的JavaScript引擎随着Java 8一起公诸于世，它允许在JVM上开发运行某些JavaScript应用。Nashorn就是javax.script.ScriptEngine的另一种实现，并且它们俩遵循相同的规则，允许Java与JavaScript相互调用。下面看一个例子：

|  |  |
| --- | --- |
| 1  2  3  4  5 | ScriptEngineManager manager = new ScriptEngineManager();  ScriptEngine engine = manager.getEngineByName( "JavaScript" );    System.out.println( engine.getClass().getName() );  System.out.println( "Result:" + engine.eval( "function f() { return 1; }; f() + 1;" ) ); |

下面是程序在控制台上的输出：

|  |  |
| --- | --- |
| 1  2 | jdk.nashorn.api.scripting.NashornScriptEngine  Result: 2 |

#### Base64

public class Base64s {

    public static void main(String[] args) {

        final String text = "Base64 finally in Java 8!";

        final String encoded = Base64

            .getEncoder()

            .encodeToString( text.getBytes( StandardCharsets.UTF\_8 ) );

        System.out.println( encoded );

        final String decoded = new String(

            Base64.getDecoder().decode( encoded ),

            StandardCharsets.UTF\_8 );

        System.out.println( decoded );

    }

}

程序在控制台上输出了编码后的字符与解码后的字符：

|  |  |
| --- | --- |
| 1  2 | QmFzZTY0IGZpbmFsbHkgaW4gSmF2YSA4IQ==  Base64 finally in Java 8! |

Base64类同时还提供了对URL、MIME友好的编码器与解码器（Base64.getUrlEncoder() / Base64.getUrlDecoder(), Base64.getMimeEncoder() / Base64.getMimeDecoder()）。

#### 并行（parallel）数组

Java 8增加了大量的新方法来对数组进行并行处理。可以说，最重要的是parallelSort()方法，因为它可以在多核机器上极大提高数组排序的速度。

public class ParallelArrays {

    public static void main( String[] args ) {

        long[] arrayOfLong = new long [ 20000 ];

        Arrays.parallelSetAll( arrayOfLong,

            index -> ThreadLocalRandom.current().nextInt( 1000000 ) );

        Arrays.stream( arrayOfLong ).limit( 10 ).forEach(

            i -> System.out.print( i + " " ) );

        System.out.println();

        Arrays.parallelSort( arrayOfLong );

        Arrays.stream( arrayOfLong ).limit( 10 ).forEach(

            i -> System.out.print( i + " " ) );

        System.out.println();

    }

}

#### 并发（Concurrency）

在新增Stream机制与lambda的基础之上，在java.util.concurrent.ConcurrentHashMap中加入了一些新方法来支持聚集操作。同时也在java.util.concurrent.ForkJoinPool类中加入了一些新方法来支持共有资源池（common pool）

新增的java.util.concurrent.locks.StampedLock类提供一直基于容量的锁，这种锁有三个模型来控制读写操作（它被认为是不太有名的java.util.concurrent.locks.ReadWriteLock类的替代者）。

在java.util.concurrent.atomic包中还增加了下面这些类：

* DoubleAccumulator
* DoubleAdder
* LongAccumulator
* LongAdder

### Java虚拟机（JVM）的新特性

PermGen空间被移除了，取而代之的是Metaspace（[JEP 122](http://openjdk.java.net/jeps/122)）。JVM选项**-XX:PermSize**与**-XX:MaxPermSize**分别被**-XX:MetaSpaceSize**与**-XX:MaxMetaspaceSize**所代替。

## 注解的实现原理

<http://blog.csdn.net/lylwo317/article/details/52163304>

<http://www.cnblogs.com/peida/archive/2013/04/26/3038503.html>

注解其实就是一个接口，是一个是实现了Annotation接口的特殊接口。当我们通过反射获得注解时，返回的是java运行时生成的动态代理对象$Proxy1，该类就是注解的具体实现类。

## 如何从100万个数找出最大的前100个数

1 堆排序

先取出前100个数（m），维护一个100个数的小根堆。然后读取后续的元素，如果当前元素小于堆顶元素，直接丢弃，如果大于，则替换堆顶元素，然后调整为小根堆。

最坏情况下是每次都需要替换掉堆顶的元素，因此维护堆的代价为(N-m)logm

建立一个小根堆的运行时间是mO(logm)

这个也可以简化，直接使用数组保存100个数，对于第101个数，如果大于数组中最小的数，则替换….

2 分块查找

先把100w个数分成100份，每份1w个数，分别找出每1w个数中最大的数，然后比较。找出100个最大的数中的最大的数和最小的数，取最大数的这组的第二大的数与最小的数进行比较...

3 利用快速排序的思想，使用分区函数

1. 递归对所有数据分成[a,b）b（b,d]两个区间，(b,d]区间内的数都是大于[a,b)区间内的数   
   (2) 对(b,d]重复(1)操作，直到最右边的区间个数小于100个。注意[a,b)区间不用划分   
   (3) 返回上一个区间，并返回此区间的数字数目。接着方法仍然是对上一区间的左边进行划分，分为[a2,b2）b2（b2,d2]两个区间，取（b2,d2]区间。如果个数不够，继续(3)操作，如果个数超过100的就重复1操作，直到最后右边只有100个数为止。 递归对所有数据分成[a,b）b（b,d]两个区间，(b,d]区间内的数都是大于[a,b)区间内的数   
   (2) 对(b,d]重复(1)操作，直到最右边的区间个数小于100个。注意[a,b)区间不用划分   
   (3) 返回上一个区间，并返回此区间的数字数目。接着方法仍然是对上一区间的左边进行划分，分为[a2,b2）b2（b2,d2]两个区间，取（b2,d2]区间。如果个数不够，继续(3)操作，如果个数超过100的就重复1操作，直到最后右边只有100个数为止。

## NIO

<http://www.importnew.com/19816.html>

## 同步 异步 阻塞 非阻塞

**同步和异步关注的是消息通信机制。主要是针对客户端。**

同步：当客户端发出一个功能调用时，在没有得到结果之前，该调用不返回。也就是说必须一件一件的事情去做，等一件做完了才能去做下一件。

异步：当客户端发出一个功能调用时，这个调用就直接返回了，不管有没有结果。调用处理完成后，被调用者会通过状态、通知和回调来通知调用者。

同步异步主要是针对客户端，但是必须配合服务器端才能实现。同步异步是客户端自己控制，但是服务器端阻塞非阻塞，客户端就不需要关心了。

**阻塞非阻塞关注的是在等待调用结果时的状态。主要针对服务器端。**

阻塞：在服务器端被调用者调用结果返回之前，当前线程会被挂起。调用线程只有在得到结果之后才会返回。

非阻塞：在不能立即得到结果之前，该调用不会阻塞当前线程。（减少了线程切换的开销）

以下面的例子说明：

老张爱喝茶，废话不说，煮开水。  
出场人物：老张，水壶两把（普通水壶，简称水壶；会响的水壶，简称响水壶）。  
1 老张把水壶放到火上，立等水开。（同步阻塞）  
老张觉得自己有点傻  
2 老张把水壶放到火上，去客厅看电视，时不时去厨房看看水开没有。（同步非阻塞）  
老张还是觉得自己有点傻，于是变高端了，买了把会响笛的那种水壶。水开之后，能大声发出嘀~~~~的噪音。  
3 老张把响水壶放到火上，立等水开。（异步阻塞）  
老张觉得这样傻等意义不大  
4 老张把响水壶放到火上，去客厅看电视，水壶响之前不再去看它了，响了再去拿壶。（异步非阻塞）  
老张觉得自己聪明了。

所谓同步异步，只是对于水壶而言。  
普通水壶，同步；响水壶，异步。  
虽然都能干活，但响水壶可以在自己完工之后，提示老张水开了。这是普通水壶所不能及的。  
同步只能让调用者去轮询自己（情况2中），造成老张效率的低下。

所谓阻塞非阻塞，仅仅对于老张而言。  
立等的老张，阻塞；看电视的老张，非阻塞。  
情况1和情况3中老张就是阻塞的，媳妇喊他都不知道。虽然3中响水壶是异步的，可对于立等的老张没有太大的意义。所以一般异步是配合非阻塞使用的，这样才能发挥异步的效用。

对应到java中：

**同步阻塞 IO ：**

在此种方式下，用户进程在发起一个 IO 操作以后，必须等待 IO 操作的完成，只有当真正完成了 IO 操作以后，用户进程才能运行。 JAVA传统的 IO 模型属于此种方式！

client在调用read（）方法时，stream里没有数据可读，线程停止向下执行，直至stream有数据。

阻塞：体现在这个线程不能干别的了，只能在这里等着  
同步：是体现在消息通知机制上的，即stream有没有数据是需要我自己来判断的。

**同步非阻塞 IO:**

在此种方式下，用户进程发起一个 IO 操作以后，就可以返回做其它事情，但是用户进程需要时不时的询问 IO 操作是否就绪，这就要求用户进程不停的去询问，从而引入不必要的 CPU 资源浪费。（并不会把当前线程挂起，可以通过循环，但是浪费cpu资源）其中目前 JAVA 的 NIO 就属于同步非阻塞 IO。

调用read方法后，如果stream没有数据，方法就返回，然后这个线程就就干别的去了。

非阻塞：体现在，这个线程可以去干别的，不需要一直在这等着  
同步：体现在消息通知机制，这个线程仍然要定时的读取stream，判断数据有没有准备好，client采用循环的方式去读取，可以看出CPU大部分被浪费了

**异步阻塞 IO ：**

http://www.linuxidc.com/Linux/2012-11/74321.htm

此种方式下是指应用发起一个 IO 操作以后，不等待内核 IO 操作的完成，等内核完成 IO 操作以后会通知应用程序，这其实就是同步和异步最关键的区别，同步必须等待或者主动的去询问 IO 是否完成，那么为什么说是阻塞的呢？因为此时是通过 select 系统调用来完成的，而 select 函数本身的实现方式是阻塞的，**而采用 select 函数有个好处就是它可以同时监听多个文件句柄，从而提高系统的并发性！**

**异步非阻塞 IO:**

https://my.oschina.net/joshuashaw/blog/650117

在此种模式下，用户进程只需要发起一个 IO 操作然后立即返回，等 IO 操作真正的完成以后，应用程序会得到 IO 操作完成的通知，此时用户进程只需要对数据进行处理就好了，不需要进行实际的 IO 读写操作，因为 真正的 IO读取或者写入操作已经由 内核完成了。目前 Java 中还没有支持此种 IO 模型。

服务端调用read()方法，若stream中无数据则返回，程序继续向下执行。当stream中有数据时，操作系统会负责把数据拷贝到用户空间，然后通知这个线程，这里的消息通知机制就是异步！而不是像NIO那样，自己起一个线程去监控stream里面有没有数据！

## 多线程实现的三种方式

<http://blog.csdn.net/andymu077/article/details/52312330>

JAVA多线程实现方式主要有三种：继承Thread类、实现Runnable接口、使用ExecutorService、Callable、Future实现有返回结果的多线程。其中前两种方式线程执行完后都没有返回值，只有最后一种是带返回值的。

### 继承Thread类

继承Thread类的方法尽管被我列为一种多线程实现方式，但Thread本质上也是实现了Runnable接口的一个实例，它代表一个线程的实例，并且，启动线程的唯一方法就是通过Thread类的start()实例方法。start()方法是一个native方法，它将启动一个新线程，并执行run()方法。这种方式实现多线程很简单，通过自己的类直接extend Thread，并复写run()方法，就可以启动新线程并执行自己定义的run()方法。例如：  
public class MyThread extends Thread {  
　　public void run() {  
　　 System.out.println("MyThread.run()");  
　　}  
}  
在合适的地方启动线程如下：  
MyThread myThread1 = new MyThread();  
MyThread myThread2 = new MyThread();  
myThread1.start();  
myThread2.start();

### 实现Runnable接口

public class MyThread extends OtherClass implements Runnable {  
　　public void run() {  
　　 System.out.println("MyThread.run()");  
　　}  
}

实现Runnable接口，重写run方法。

如果自己的类已经extends另一个类，就无法直接extends Thread，此时，必须实现一个Runnable接口。

在main函数中，new一个Thread对象，然后把Runnable对象作为参数传递给Thread，同一个Runnable对象可以作为多个线程的参数进行传递，且这些线程共享Runnable对象的实例变量。

MyThread myThread = new MyThread();  
Thread thread = new Thread(myThread);  
thread.start();

事实上，当传入一个Runnable target参数给Thread后，Thread的run()方法就会调用target.run()，参考JDK源代码：  
public void run() {  
　　if (target != null) {  
　　 target.run();  
　　}  
}s

### 实现Callable接口

ExecutorService、Callable、Future这个对象实际上都是属于Executor框架中的功能类。想要详细了解Executor框架的可以访问http://www.javaeye.com/topic/366591 ，这里面对该框架做了很详细的解释。返回结果的线程是在JDK1.5中引入的新特征，确实很实用，有了这种特征我就不需要再为了得到返回值而大费周折了，而且即便实现了也可能漏洞百出。

可返回值的任务必须实现Callable接口，类似的，无返回值的任务必须Runnable接口。执行Callable任务后，可以获取一个Future的对象，在该对象上调用get就可以获取到Callable任务返回的Object了，再结合线程池接口ExecutorService就可以实现传说中有返回结果的多线程了。

import java.util.concurrent.\*;

import java.util.Date;

import java.util.List;

import java.util.ArrayList;

/\*\*

\* Java线程：有返回值的线程

\*

\* @author wb\_qiuquan.ying

\*/

@SuppressWarnings("unchecked")

public class Test {

public static void main(String[] args) throws ExecutionException,

InterruptedException {

System.out.println("----程序开始运行----");

Date date1 = new Date();

int taskSize = 5;

// 创建一个线程池

ExecutorService pool = Executors.newFixedThreadPool(taskSize);

// 创建多个有返回值的任务

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

for (int i = 0; i < taskSize; i++) {

Callable c = new MyCallable(i + " ");

// 执行任务并获取Future对象

Future f = pool.submit(c);

// System.out.println(">>>" + f.get().toString());

list.add(f);

}

// 关闭线程池

pool.shutdown();

// 获取所有并发任务的运行结果

for (Future f : list) {

// 从Future对象上获取任务的返回值，并输出到控制台

System.out.println(">>>" + f.get().toString());

}

Date date2 = new Date();

System.out.println("----程序结束运行----，程序运行时间【"

+ (date2.getTime() - date1.getTime()) + "毫秒】");

}

}

class MyCallable implements Callable<Object> {

private String taskNum;

MyCallable(String taskNum) {

this.taskNum = taskNum;

}

public Object call() throws Exception {

System.out.println(">>>" + taskNum + "任务启动");

Date dateTmp1 = new Date();

Thread.sleep(1000);

Date dateTmp2 = new Date();

long time = dateTmp2.getTime() - dateTmp1.getTime();

System.out.println(">>>" + taskNum + "任务终止");

return taskNum + "任务返回运行结果,当前任务时间【" + time + "毫秒】";

}

}

## 线程和进程区别

并发并行同步互斥参考<http://blog.csdn.net/yaosiming2011/article/details/44280713>

进程是具有一定独立功能的程序关于某个数据集合上的一次运行活动，进程是是系统是进行资源分配和调度的一个独立的单位。

线程是进程的一个实体，或者说是一条执行路径，是CPU进行调度和分派的基本单位。它是比进程更小的能独立运行的基本单位，线程自己不拥有系统资源，只拥有一点在运行中必不可少的资源（如程序计数器 寄存器和栈），它可与同属于一个进程的其他线程共享进行所有的全部资源。

一个进程的多个线程可以并发的执行，一个线程可以创建和撤销另一个线程。

（并发：在一个时间段内同时发生 并行：在一个时间点上同时发生）

进程和线程主要差别在于它们是不同的操作系统资源管理方式。进程有独立的地址空间，一个进程崩溃之后，在保护模式下，不会对其他进程产生影响。而线程只是一个进程的不同执行路径。线程有自己的堆栈和局部变量，但是线程之间没有单独的地址空间，一个线程死掉之后，等于整个进程死掉，所以多进程的程序要比多线程的程序健壮，但是在进程切换时，耗费资源较大，效率差一些。但对于一些要求同时进行并且又要共享某些变量的并发操作，只能使用线程，不能用进程。

线程执行的开销比较小，但不利于资源的管理和维护，进程正好相反。

### 一、进程

进程：指在系统中能独立运行并作为资源分配的基本单位，它是由一组机器指令、数据和堆栈等组成的，是一个能独立运行的活动实体。

注意，进程一般有三个状态：就绪状态、执行状态和等待状态【或称阻塞状态】；进程只能由父进程建立，系统中所有的进程形成一种进程树的层次体系；挂起命令可由进程自己和其他进程发出，但是解除挂起命令只能由其他进程发出。

进程控制块（PCB）：PCB不但可以记录进程的属性信息，以便操作系统对进程进行控制和管理，而且PCB标志着进程的存在，操作系统根据系统中是否有该进程的进程控制块PCB而知道该进程存在与否。系统建立进程的同时就建立该进程的PCB，在撤销一个进程时，也就撤销其PCB，故进程的PCB对进程来说是它存在的具体的物理标志和体现。一般PCB包括以下三类信息：进程标识信息；处理器状态信息；进程控制信息。 由程序段、相关的数据段和PCB三部分构成了进程实体（又称进程印像），一般，我们把进程实体就简称为进程。

进程的特征： 1.动态性：进程的实质是程序的一次执行过程，进程是动态产生，动态消亡的。 2.并发性：任何进程都可以同其他进程一起并发执行。 3.独立性：进程是一个能独立运行的基本单位，同时也是系统分配资源和调度的独立单位。 4.异步性：由于进程间的相互制约，使进程具有执行的间断性，即进程按各自独立的、不可预知的速度向前推进。

### [二、线程](#%E4%BA%8C-%E7%BA%BF%E7%A8%8B)

[线程：线程是进程中的一个实体，作为系统调度和分派的基本单位。Linux下的线程看作轻量级进程。](#%E4%BA%8C-%E7%BA%BF%E7%A8%8B)

[线程的性质：](#%E4%BA%8C-%E7%BA%BF%E7%A8%8B)

[1.线程是进程内的一个相对独立的可执行的单元。若把进程称为任务的话，那么线程则是应用中的一个子任务的执行。](evernote-html-snippet://" \l "%E4%BA%8C-%E7%BA%BF%E7%A8%8B)

[2.由于线程是被调度的基本单元，而进程不是调度单元。所以，每个进程在创建时，至少需要同时为该进程创建一个线程。即进程中至少要有一个或一个以上的线程，否则该进程无法被调度执行。](evernote-html-snippet://" \l "%E4%BA%8C-%E7%BA%BF%E7%A8%8B)

[3.进程是被分给并拥有资源的基本单元。同一进程内的多个线程共享该进程的资源，但线程并不拥有资源，只是使用他们。](evernote-html-snippet://" \l "%E4%BA%8C-%E7%BA%BF%E7%A8%8B)

[4.线程是操作系统中基本调度单元，因此线程中应包含有调度所需要的必要信息，且在生命周期中有状态的变化。](evernote-html-snippet://" \l "%E4%BA%8C-%E7%BA%BF%E7%A8%8B)

[5.由于共享资源【包括数据和文件】，所以线程间需要通信和同步机制，且需要时线程可以创建其他线程，但线程间不存在父子关系。](evernote-html-snippet://" \l "%E4%BA%8C-%E7%BA%BF%E7%A8%8B)

[多线程使用的情形：前台和后台工作情况；异步处理工作情况；需要加快执行速度情况；组织复杂工作的情况；同时有多个用户服务请求的情况等。](#%E4%BA%8C-%E7%BA%BF%E7%A8%8B)

[线程机制的优点： 多线程运行在同一个进程的相同的地址空间内，和采用多进程相比有以下优点：](#%E4%BA%8C-%E7%BA%BF%E7%A8%8B)

* [1.创建和撤销线程的开销较之进程要少。创建线程时只需要建立线程控制表相应的表目，或有关队列，而创建进程时，要创建PCB表和初始化，进入有关进程队列，建立它的地址空间和所需资源等。](evernote-html-snippet://" \l "%E4%BA%8C-%E7%BA%BF%E7%A8%8B)
* [2.CPU在线程之间开关时的开销远比进程要少得多。因开关线程都在同一地址空间内，只需要修改线程控制表或队列，不涉及地址空间和其他工作。](evernote-html-snippet://" \l "%E4%BA%8C-%E7%BA%BF%E7%A8%8B)
* [3.线程机制也增加了通讯的有效性。进程间的通讯往往要求内核的参与，以提供通讯机制和保护机制，而线程间的通讯是在同一进程的地址空间内，共享主存和文件，无需内核参与。](evernote-html-snippet://" \l "%E4%BA%8C-%E7%BA%BF%E7%A8%8B)

### [三、进程和线程的区别](#%E4%B8%89-%E8%BF%9B%E7%A8%8B%E5%92%8C%E7%BA%BF%E7%A8%8B%E7%9A%84%E5%8C%BA%E5%88%AB)

[（1）调度： 在传统的操作系统中，CPU调度和分派的基本单位是进程。而在引入线程的操作系统中，则把线程作为CPU调度和分派的基本单位，进程则作为资源拥有的基本单位，从而使传统进程的两个属性分开，线程编程轻装运行，这样可以显著地提高系统的并发性。同一进程中线程的切换不会引起进程切换，从而避免了昂贵的系统调用，但是在由一个进程中的线程切换到另一进程中的线程，依然会引起进程切换。](#%E4%B8%89-%E8%BF%9B%E7%A8%8B%E5%92%8C%E7%BA%BF%E7%A8%8B%E7%9A%84%E5%8C%BA%E5%88%AB)

[（2）并发性： 在引入线程的操作系统中，不仅进程之间可以并发执行，而且在一个进程中的多个线程之间也可以并发执行，因而使操作系统具有更好的并发性，从而更有效地提高系统资源和系统的吞吐量。例如，在一个为引入线程的单CPU操作系统中，若仅设置一个文件服务进程，当它由于某种原因被封锁时，便没有其他的文件服务进程来提供服务。在引入线程的操作系统中，可以在一个文件服务进程设置多个服务线程。当第一个线程等待时，文件服务进程中的第二个线程可以继续运行；当第二个线程封锁时，第三个线程可以继续执行，从而显著地提高了文件服务的质量以及系统的吞吐量。](#%E4%B8%89-%E8%BF%9B%E7%A8%8B%E5%92%8C%E7%BA%BF%E7%A8%8B%E7%9A%84%E5%8C%BA%E5%88%AB)

[（3）拥有资源： 不论是引入了线程的操作系统，还是传统的操作系统，进程都是拥有系统资源的一个独立单位，他可以拥有自己的资源。一般地说，线程自己不能拥有资源（也有一点必不可少的资源），但它可以访问其隶属进程的资源，亦即一个进程的代码段、数据段以及系统资源（如已打开的文件、I/O设备等），可供同一个进程的其他所有线程共享。](#%E4%B8%89-%E8%BF%9B%E7%A8%8B%E5%92%8C%E7%BA%BF%E7%A8%8B%E7%9A%84%E5%8C%BA%E5%88%AB)

[（4）独立性： 在同一进程中的不同线程之间的独立性要比不同进程之间的独立性低得多。这是因为为防止进程之间彼此干扰和破坏，每个进程都拥有一个独立的地址空间和其它资源，除了共享全局变量外，不允许其它进程的访问。但是同一进程中的不同线程往往是为了提高并发性以及进行相互之间的合作而创建的，它们共享进程的内存地址空间和资源，如每个线程都可以访问它们所属进程地址空间中的所有地址，如一个线程的堆栈可以被其它线程读、写，甚至完全清除。](#%E4%B8%89-%E8%BF%9B%E7%A8%8B%E5%92%8C%E7%BA%BF%E7%A8%8B%E7%9A%84%E5%8C%BA%E5%88%AB)

[（5）系统开销： 由于在创建或撤销进程时，系统都要为之分配或回收资源，如内存空间、I/O设备等。因此，操作系统为此所付出的开销将显著地大于在创建或撤消线程时的开销。类似的，在进程切换时，涉及到整个当前进程CPU环境的保存环境的设置以及新被调度运行的CPU环境的设置，而线程切换只需保存和设置少量的寄存器的内容，并不涉及存储器管理方面的操作，可见，进程切换的开销也远大于线程切换的开销。此外，由于同一进程中的多个线程具有相同的地址空间，致使他们之间的同步和通信的实现也变得比较容易。在有的系统中，现成的切换、同步、和通信都无需操作系统内核的干预。](evernote-html-snippet://" \l "%E4%B8%89-%E8%BF%9B%E7%A8%8B%E5%92%8C%E7%BA%BF%E7%A8%8B%E7%9A%84%E5%8C%BA%E5%88%AB)

[（6）支持多处理机系统： 在多处理机系统中，对于传统的进程，即单线程进程，不管有多少处理机，该进程只能运行在一个处理机上。但对于多线程进程，就可以将一个进程中的多个线程分配到多个处理机上，使它们并行执行，这无疑将加速进程的完成。因此，现代处理机OS都无一例外地引入了多线程。](#%E4%B8%89-%E8%BF%9B%E7%A8%8B%E5%92%8C%E7%BA%BF%E7%A8%8B%E7%9A%84%E5%8C%BA%E5%88%AB)

## 线程的生命周期

<http://www.cnblogs.com/mengdd/archive/2013/02/16/2913649.html>

**会画出状态转换图**

**五种状态**

**1 新建（new Thread）**当创建Thread类的一个实例（对象）时，此线程进入新建状态（未被启动）。  
例如：Thread  t1=new Thread();

**2 就绪（runnable）**线程已经被启动，正在等待被分配给CPU时间片，也就是说此时线程正在就绪队列中排队等候得到CPU资源。例如：**t1.start();**

**3 运行（running）**线程获得CPU资源正在执行任务（run()方法），此时除非此线程自动放弃CPU资源或者有优先级更高的线程进入，线程将一直运行到结束。

**4 死亡（dead）**  
当线程执行完毕或被其它线程杀死，线程就进入死亡状态，这时线程不可能再进入就绪状态等待执行。

自然终止：正常运行run()方法后终止

异常终止：调用**stop()**方法让一个线程终止运行

**5 阻塞（blocked）**  
由于某种原因导致正在运行的线程让出CPU并暂停自己的执行，即进入阻塞状态。

正在睡眠：用sleep(long t) 方法可使线程进入睡眠方式。一个睡眠着的线程在指定的时间过去可进入就绪状态。

正在等待：调用wait()方法。（调用notify()方法回到就绪状态）

**常用的方法**

void run()   创建该类的子类时必须实现的方法

void start() 开启线程的方法

static void sleep(long t) 释放CPU的执行权，不释放锁

static void sleep(long millis,int nanos)

final void wait()释放CPU的执行权，释放锁

final void notify()

static void yied()可以对当前线程进行临时暂停（让线程将资源释放出来）

## 死锁

<http://c.biancheng.net/cpp/html/2605.html>

<http://blog.csdn.net/bxyill/article/details/8237339>

死锁：多个进程或线程竞争某一共享资源，而出现的一种互相等待的现象

产生死锁的主要原因：1 系统资源不够 2 进程或者线程运行推进的顺序不合适

3 资源分配不当

死锁的四个必要条件：

1 互斥条件：一个资源每次只能被一个进程使用

2 请求与保持条件：一个进程因请求资源而阻塞时，对已获得的资源保持不变

3 不剥夺条件：进程已获得的资源，在未使用完之前，不能强行剥夺。

4 循环等待条件：若干进程之间形成的一种头尾相接的循环等待资源关系。

只要上述条件之一不满足，就不会产生死锁。

### 死锁的处理策略

为使系统不发生死锁，必须设法破坏产生死锁的四个必要条件之一，或者允许死锁产生， 但当死锁发生时能检测出死锁，并有能力实现恢复。

#### 预防死锁

设置某些限制条件，破坏产生死锁的四个必要条件中的一个或几个，以防止发生死锁。

#### 避免死锁

在资源的动态分配过程中，用某种方法防止系统进入不安全状态，从而避免死锁。

#### 死锁的检测及解除

无需釆取任何限制性措施，允许进程在运行过程中发生死锁。通过系统的检测机构及时 地检测出死锁的发生，然后釆取某种措施解除死锁。  
  
预防死锁和避免死锁都属于事先预防策略，但预防死锁的限制条件比较严格，实现起来 较为简单，但往往导致系统的效率低，资源利用率低；避免死锁的限制条件相对宽松，资源 分配后需要通过算法来判断是否进入不安全状态，实现起来较为复杂。



### 死锁预防

防止死锁的发生只需破坏死锁产生的四个必要条件之一即可。

#### 1) 破坏互斥条件

如果允许系统资源都能共享使用，则系统不会进入死锁状态。但有些资源根本不能同时访问，如打印机等临界资源只能互斥使用。所以，破坏互斥条件而预防死锁的方法不太可行，而且在有的场合应该保护这种互斥性。

#### 2) 破坏不剥夺条件

当一个已保持了某些不可剥夺资源的进程，请求新的资源而得不到满足时，它必须释放已经保持的所有资源，待以后需要时再重新申请。这意味着，一个进程已占有的资源会被暂时释放，或者说是被剥夺了，或从而破坏了不可剥夺条件。  
  
该策略实现起来比较复杂，释放已获得的资源可能造成前一阶段工作的失效，反复地申请和释放资源会增加系统开销，降低系统吞吐量。这种方法常用于状态易于保存和恢复的资源，如CPU的寄存器及内存资源，一般不能用于打印机之类的资源。

#### 3) 破坏请求和保持条件

釆用预先静态分配方法，即进程在运行前一次申请完它所需要的全部资源，在它的资源未满足前，不把它投入运行。一旦投入运行后，这些资源就一直归它所有，也不再提出其他资源请求，这样就可以保证系统不会发生死锁。  
  
这种方式实现简单，但缺点也显而易见，系统资源被严重浪费，其中有些资源可能仅在运行初期或运行快结束时才使用，甚至根本不使用。而且还会导致“饥饿”现象，当由于个别资源长期被其他进程占用时，将致使等待该资源的进程迟迟不能开始运行。

#### 4) 破坏循环等待条件

为了破坏循环等待条件，可釆用顺序资源分配法。首先给系统中的资源编号，规定每个进程，必须按编号递增的顺序请求资源，同类资源一次申请完。也就是说，只要进程提出申请分配资源Ri，则该进程在以后的资源申请中，只能申请编号大于Ri的资源。  
  
这种方法存在的问题是，编号必须相对稳定，这就限制了新类型设备的增加；尽管在为资源编号时已考虑到大多数作业实际使用这些资源的顺序，但也经常会发生作业使甩资源的顺序与系统规定顺序不同的情况，造成资源的浪费；此外，这种按规定次序申请资源的方法，也必然会给用户的编程带来麻烦。

### 死锁避免

避免死锁同样是属于事先预防的策略，但并不是事先釆取某种限制措施破坏死锁的必要条件，而是在资源动态分配过程中，防止系统进入不安全状态，以避免发生死锁。这种方法所施加的限制条件较弱，可以获得较好的系统性能。

#### 1. 系统安全状态

避免死锁的方法中，允许进程动态地申请资源，但系统在进行资源分配之前，应先计算此次资源分配的安全性。若此次分配不会导致系统进入不安全状态，则将资源分配给进程； 否则，让进程等待。  
  
所谓安全状态，是指系统能按某种进程推进顺序( P1, P2, ..., Pn)，为每个进程Pi分配其所需资源，直至满足每个进程对资源的最大需求，使每个进程都可顺序地完成。此时称 P1, P2, ..., Pn 为安全序列。如果系统无法找到一个安全序列，则称系统处于不安全状态。  
  
假设系统中有三个进程P1、P2和P3,共有12 台磁带机。进程P1总共需要10台磁带机，P2和P3 分别需要4台和9台。假设在T0时刻，进程P1、P2 和P3已分别获得5合、2台和2台，尚有3台未分配，见表2-15。

|  |  |  |  |
| --- | --- | --- | --- |
| 表2-15 资源分配 | | | |
| **进程** | **最大需求** | **已分配** | **可用** |
| P1 | 10 | 5 | 3 |
| P2 | 4 | 2 |  |
| P3 | 9 | 2 |  |

则在T0时刻是安全的，因为存在一个安全序列P2、Pl、P3，即只要系统按此进程序列分配资源，则每个进程都能顺利完成。若在T0时刻后，系统分配1台磁带机给P3，则此时系统便进入不安全状态，因为此时已无法再找到一个安全序列。  
  
并非所有的不安全状态都是死锁状态，但当系统进入不安全状态后，便可能进入死锁状态；反之，只要系统处于安全状态，系统便可以避免进入死锁状态。

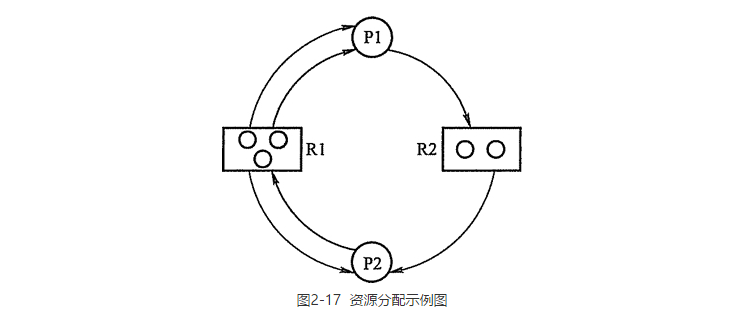
#### 2. 银行家算法

银行家算法是最著名的死锁避免算法。它提出的思想是：把操作系统看做是银行家，操作系统管理的资源相当于银行家管理的资金，进程向操作系统请求分配资源相当于用户向银行家贷款。操作系统按照银行家制定的规则为进程分配资源，当进程首次申请资源时，要测试该进程对资源的最大需求量，如果系统现存的资源可以满足它的最大需求量则按当前的申请量分配资源，否则就推迟分配。当进程在执行中继续申请资源时，先测试该进程已占用的资源数与本次申请的资源数之和是否超过了该进程对资源的最大需求量。若超过则拒绝分配资源，若没有超过则再测试系统现存的资源能否满足该进程尚需的最大资源量，若能满足则按当前的申请量分配资源，否则也要推迟分配。**可参考上述链接**

### 死锁的检测和解除

前面绍的死锁预防和避免算法，都是在为进程分配资源时施加限制条件或进行检测，若系统为进程分配资源时不釆取任何措施，则应该提供死锁检测和解除的手段。

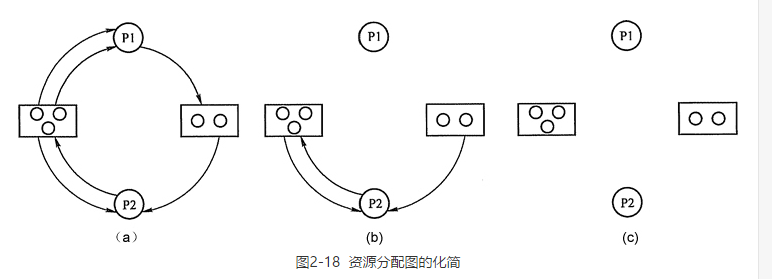
#### 资源分配图

系统死锁，可利用资源分配图来描述。如图2-17所示，用圆圈代表一个进程，用框代表一类资源。由于一种类型的资源可能有多个，用框中的一个点代表一类资源中的一个资源。从进程到资源的有向边叫请求边，表示该进程申请一个单位的该类资源；从资源到进程的边叫分配边，表示该类资源已经有一个资源被分配给了该进程。  
  
在图2-17所示的资源分配图中，进程P1已经分得了两个R1资源，并又请求一个R2 资源；进程P2分得了一个R1和一个R2资源，并又请求一个R1资源。

#### 死锁定理

可以通过将资源分配图简化的方法来检测系统状态S是否为死锁状态。简化方法如下：  
  
1) 在资源分配图中，找出既不阻塞又不是孤点的进程Pi（即找出一条有向边与它相连，且该有向边对应资源的申请数量小于等于系统中已有空闲资源数量。若所有的连接该进程的边均满足上述条件，则这个进程能继续运行直至完成，然后释放它所占有的所有资源）。消去它所有的请求边和分配边，使之成为孤立的结点。在图2-18(a)中，P1是满足这一条件的进程结点，将P1的所有边消去，便得到图248(b)所示的情况。  
  
2) 进程Pi所释放的资源，可以唤醒某些因等待这些资源而阻塞的进程，原来的阻塞进程可能变为非阻塞进程。在图2-17中，进程P2就满足这样的条件。根据第1) 条中的方法进行一系列简化后,若能消去图中所有的边，则称该图是可完全简化的，如图2-18(c)所示。  
  
S为死锁的条件是当且仅当S状态的资源分配图是不可完全简化的,该条件为死锁定理。

#### 死锁的解除

一旦检测出死锁，就应立即釆取相应的措施，以解除死锁。死锁解除的主要方法有：  
  
1) 资源剥夺法。挂起某些死锁进程，并抢占它的资源，将这些资源分配给其他的死锁进程。但应防止被挂起的进程长时间得不到资源，而处于资源匮乏的状态。  
  
2) 撤销进程法。强制撤销部分、甚至全部死锁进程并剥夺这些进程的资源。撤销的原则可以按进程优先级和撤销进程代价的高低进行。  
  
3) 进程回退法。让一（多）个进程回退到足以回避死锁的地步，进程回退时自愿释放资源而不是被剥夺。要求系统保持进程的历史信息，设置还原点。

## 进程同步与线程同步

**互斥锁和条件变量**是同步的基本组成部分。互斥锁和条件变量出自POSIX.1线程标准，它们总是可用来**同步一个进程内的各个线程**的。如果一个互斥锁或条件变量存放在多个进程间共享的某个内存中，那么POSIX还允许它用于这些**进程间的同步**。互斥锁、条件变量、读写锁、信号量均可用于进程、线程的同步。

**多线程同步方法：**

**1）互斥量（互斥锁）**

参考资料：互斥量：APUE第二版299页；  条件变量：第309页；  同步属性：318页。

        互斥锁是最基本的同步形式，用于保护临界区，确保**同一时间只有一个线程或进程**访问数据或执行其中的代码。互斥量（mutex）本质上是一把锁，访问共享资源前要对互斥量加锁，访问完后要释放锁。

**互斥量用于上锁，而条件变量用于等待**。

        允许多个进程将同一个内存区域映射到它们各自独立的地址空间，就像多线程一样共享数据，那么多个进程访问共享数据通常也需要同步。如果进程共享互斥量属性设置为PTHREAD\_PROCESS\_SHARED，**从多个进程共享的内存区域中分配的互斥量就可以用于这些进程的同步**了。进程共享互斥量属性设为PTHREAD\_PROCESS\_PRIVATE时，允许pthread线程库提供更加有效的互斥量实现，这在多线程应用程序中是默认的情况。

       用于给存放在共享内存区中供多个进程使用的条件变量设置PTHREAD\_PROCESS\_SHARED属性的一组语句跟用于互斥锁的语句几乎相同，只需要将mutex替换为cond。

**2）条件变量**

         条件变量与互斥量一起使用时，允许线程以无竞争的方式等待特定条件发送。**条件变量本身是由互斥量保护的**。线程在改变条件状态前必须首先锁住互斥量，必须锁住互斥量以后才能计算条件。如果在规定的时间内条件满足就通知线程，否则生成一个代表出错码的返回变量。

        可以用于进程的同步，设置PTHREAD\_PROCESS\_SHARED属性。

**3）读写锁（共享-独占锁）**

        和互斥量类似，不过读写锁允许更高的并行性。互斥量要么是锁住状态要么是不加锁状态，而且一次仅允许一个线程对互斥量加锁。

        读写锁有三种状态：a. 读模式下加锁状态（所有试图以读模式对其进行加锁的线程都可以获得访问权，但如果试图以写模式加锁，必须阻塞直到所有线程释放读锁）；b. 写模式下加锁状态（在这个锁被解锁之前，所有试图对这个锁加锁的线程都会被阻塞）；c. 不加锁状态。

**可用于进程、线程之间的同步**。

**4）信号量**

信号量可用于线程或进程间同步。

        信号量是一个计数器，用于多进程对共享数据对象的访问。若此信号量的值为正，则进程可以使用该资源；进程将信号量减1，表示它使用了一个资源单元。若此信号量的值为0，则进程进入休眠状态，直至信号量位于0。若一个进程不再使用由一个信号量控制的共享资源时，该信号量值增1。如果有进程正在休眠等待此信号量，则唤醒它们。

**POSIX信号量：**

参考：UNP卷2第10章

        POSIX信号量可以是有名的，也可以是基于内存的。有名信号量总是能够在不同进程间共享，基于内存的信号量则必须在创建时指定成是否在进程间共享。这两类信号量的持续性也有差别：有名信号量至少是随内核持续的，基于内存的信号量则是具有随进程持续的。

**System V信号量：**

如同进程一样，线程也可以通过信号量来实现通信，虽然是轻量级的。信号量函数的名字都以"sem\_"打头。线程使用的基本信号量函数有四个。

1. 信号量初始化。

int sem\_init (sem\_t \*sem , int pshared, unsigned int value);

这是对由sem指定的信号量进行初始化，设置好它的共享选项(linux 只支持为0，即表示它是当前进程的局部信号量)，然后给它一个初始值VALUE。

1. 等待信号量。给信号量减1，然后等待直到信号量的值大于0。

int sem\_wait(sem\_t \*sem);

1. 释放信号量。信号量值加1。并通知其他等待线程。

int sem\_post(sem\_t \*sem);

1. 销毁信号量。我们用完信号量后都它进行清理。归还占有的一切资源。

int sem\_destroy(sem\_t \*sem);

**进程间同步方法**

上述提到的**互斥量、条件变量、读写锁、信号量都可用于进程间同步**。

**互斥量**

**条件变量**

**读写锁**

**信号量**

**记录锁**

        Linux内核没有文件内的记录这一概念。任何关于记录的接收都是由读写文件的应用来进行的。然而linux内核提供的上锁特性却用**记录上锁**（record locking）这一术语描述。不过应用会指定文件中待上锁或解锁部分的字节范围，因为记录锁锁定的只是文件中的一个区域。

       记录锁是读写锁的一种拓展，可用于亲缘或非亲缘关系的进程间共享某个文件的读写。被锁住的文件通过其描述符访问，**执行上锁操作的函数是fcntl**。这类锁通常维护在内核中，其属主是由属主的进程ID来标识的。这意味着这些锁**用于不同进程间的上锁，不适用于同一进程不同线程间上锁**。

      使用fcntl记录上锁时，**等待着的读出者优先还是等待着的写入者优先没有保证**；如果这对于某个应用很重要，就需要开发测试程序（UNP卷2，9.6节），或者给该应用提供满足所需优先关系的专用读写锁实现。

## 进程间通信的几种方式

**1）管道（Pipe）及有名管道（named pipe，FIFO）**

**管道是进程之间进行单向通信的一种机制。**

**匿名管道**是Linux支持的最初Unix IPC形式之一，具有以下特点：

* 管道是半双工的，数据只能向一个方向流动；需要双方通信时，需要建立起两个管道；STREAMS管道是一个双向（全双工）管道，单个STREAMS管道就能向父、子进程提供双向数据流。Solaris支持STREAMS管道，Linux的可选附加包也提供了STREAMS管道。
* 只能用于父子进程或者兄弟进程之间（具有亲缘关系的进程）；
* 单独构成一种独立的文件系统：管道对于管道两端的进程而言，就是一个文件，但它不是普通的文件，它不属于某种文件系统，而是自立门户，单独构成一种文件系统，并且只存在与内存中。
* 数据的读出和写入：一个进程向管道中写的内容被管道另一端的进程读出。写入的内容每次都添加在管道缓冲区的末尾，并且每次都是从缓冲区的头部读出数据。

两者的不同点：

（1）匿名管道它没有名字，只能用于具有亲缘关系进程间的通信。有名管道FIFO克服了管道没有名字的限制，因此，除具有管道所具有的功能外，它还允许无亲缘关系进程间的通信。匿名管道只能用于本地进程之间的通信，有名管道还可以用于跨网络的不同计算机不同进程之间的通信。

（2）匿名管道对于管道两端的进程而言是一个文件，但不是Linux某种类型的文件、不属于某个文件系统，而且仅存在于内存中；pipe()函数打开了两个文件描述符，分别用于读写。有名管道提供一个路径名与之关联，是linux文件类型的一种，因FIFO文件形式存在于文件系统中。

相同点：

（1）管道和FIFO的数据是字节流，应用程序之间必须事先确定特定的传输"协议"，采用传播具有特定意义的消息。

（2）单向（半双工）数据流。

（3）系统对管道和FIFO的两个限制OPEN\_MAX（一个进程任意时刻打开的最大描述符数）、PIPI\_BUF（可原子地写往一个管道或FIFO的最[**大数据**](http://lib.csdn.net/base/20)量，posix要求至少512）。

4）**都是随进程持续的IPC（IPC对象一直存在到打开该对象的最后一个进程关闭该对象为止。）**

管道常用于两个方面：

（1）在shell中时常会用到管道（作为输入输入的重定向），在这种应用方式下，管道的创建对于用户来说是透明的；

（2）用于具有亲缘关系的进程间通信，用户自己创建管道，并完成读写操作。

**2)Unix域协议**

 Unix域协议**并不是**一个实际的协议族，而**是在单个主机上执行客户/服务器通信的一种方法**，所使用的API就是在不同主机上执行客户/服务器通信所用的API（套接字API）。

    Unix域套接字仅仅复制数据，并不执行协议处理，不需要添加或删除网络报头，无需计算校验和，不要产生顺序号，无需发送确认报文。Unix域套接字提供**流和数据报**两种接口。Unix域数据报服务是可靠的，既不会丢失消息也不会传递出错。它**是套接字和管道之间的混合物**。

**使用Unix域套接字的理由**：

（1）Unix域套接字往往比通信两端位于同一主机的TCP套接字快一倍（TCPv3）。Unix域套接字仅仅复制数据，并不执行协议处理，不需要添加或删除网络报头，无需计算校验和，不要产生顺序号，无需发送确认报文。

（2）可用于在同一台主机的不同进程之间传递描述符。

（3）Unix域套接字较新的实现把客户的凭证（用户ID和组ID）提供给服务器，从而提供了额外的安全检查措施。

为了创建一对非命名的、相互连接的UNIX域套接字，用户可以使用它们面向网络的域套接字接口，也可以使用socketpair函数。

**3）信号（Signal）**

     信号是比较复杂的通信方式，用于通知接受进程有某种事件发生，除了用于进程间通信外，进程还可以发送信号给进程本身；linux除了支持Unix早期信号语义函数signal外，还支持语义符合Posix.1标准的信号函数sigaction（实际上，该函数是基于BSD的，BSD为了实现可靠信号机制，又能够统一对外接口，用sigaction函数重新实现了signal函数）。sigaction包含了信号产生的相关信息。

**4）消息队列**

   消息队列是消息的链接表，包括Posix消息队列和system V消息队列。有足够权限的进程可以向队列中添加消息，被赋予读权限的进程则可以读走队列中的消息。消息队列克服了信号承载信息量少，管道只能承载无格式字节流以及缓冲区大小受限等缺点。而且消息队列是**随内核持续的**（IPC对象会一直存在，直到内核重启或显示删除该对象为止）。

**5）共享内存**

参考文章：[进程通信方式：共享内存区](http://blog.csdn.net/u013074465/article/details/46118121)

        mmap：[Linux环境进程间通信（五）: 共享内存（上）](http://www.ibm.com/developerworks/cn/linux/l-ipc/part5/index1.html)

        System V共享内存：[Linux环境进程间通信（五）: 共享内存（下）](http://www.ibm.com/developerworks/cn/linux/l-ipc/part5/index2.html)

    使得多个进程可以访问同一块内存空间，**是最快的可用IPC形式**。是针对其他通信机制运行效率较低而设计的。往往与其它通信机制，如**与信号量结合使用，来达到进程间的同步及互斥**。

**6）信号量（semaphore）**

信号量是一个计数器，可以用来控制多个进程或者线程之间对共享资源的访问。常作为一种锁机制，防止某进程正在访问共享资源时，其他进程也访问该资源。   主要作为进程间以及同一进程不同线程之间的同步手段。

信号量和互斥锁区别：

<http://blog.csdn.net/tietao/article/details/7367827>

信号量用于不同进程之间或者线程之间。互斥锁用于线程之间。

在上锁时，只要信号量的value大于0，其他线程就可以sem\_wait成功，成功后信号量的value减一。若value的值不大于0，则sem\_wait使得线程阻塞，直到sem\_post释放后value值加1，但是sem\_wait返回之前还是将此value值减一。互斥锁，只要被锁住，其他线程都不可以访问被保护的资源。

**各种同步方式**：

线程同步的几种方式：参考文章：进程同步和线程同步

**7）套接口（Socket）**

    更为一般的进程间通信机制，可用于不同机器之间的进程间通信。起初是由Unix系统的BSD分支开发出来的，但现在一般可以移植到其它类Unix系统上：Linux和System V的变种都支持套接字。

## 共享内存

Inter-Process Communication

**定义：**

共享内存是最快的可用IPC（进程间通信）形式。它允许多个不相关的进程去访问同一部分逻辑内存。共享内存是由IPC为一个进程创建的一个特殊的地址范围，它将出现在进程的地址空间中。其他进程可以把同一段共享内存段“连接到”它们自己的地址空间里去。所有进程都可以访问共享内存中的地址。如果一个进程向这段共享内存写了数据，所做的改动会立刻被有访问同一段共享内存的其他进程看到。因此共享内存对于数据的传输是非常高效的。

**原理：**

共享内存是最有用的进程间通信方式之一，也是最快的IPC形式。两个不同进程A、B共享内存的意思是，同一块物理内存被映射到进程A、B各自的进程地址空间。进程A可以即时看到进程B对共享内存中数据的更新，反之亦然。

# 数据库

## 三大范式

### 第一范式1NF

数据库表的每一列都是不可分割的基本数据项，同一列中不能有多个值，即实体中的某个属性不能有多个值或者不能有重复的属性。

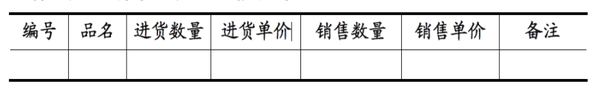
如果出现重复的属性，就需要定义一个新的实体，新的实体由重复的属性构成，新的实体和原实体之间是一对多关系。

在任何一个关系型数据库中，第一范式是基本的要求，必须满足。

下面看例子：



上面的数据表是不符合第一范式的，因为进货下面又可以再次划分为数量和单价，销售同理。满足第一范式的数据表应设计如下：



仅仅满足第一范式的数据库表还会存在很多问题，如数据冗余过大，插入异常，删除异常，修改异常。看下表；



存在的问题：

1 每一名学生的学号 姓名 系名 系主任这些数据重复过多，每个系与系主任的数据也重复多次——数据冗余

2 假设学校新开了一个系，但是这个系还没有开始招生，那么现在是无法将系主任和系名的数据添加到上述的表中——插入异常

3 假设将某个系中的所有学生的相关记录全部删除，那么系和系主任的数据也随之消失了，但是学生记录不存在，并不意味着这个系就没有了——删除异常

4 假设小明转到哲学系，那么为了保证数据库的一致性，需要修改三条记录中系与系主人的数据——修改异常

因此需要提高设计标准，使其符合更高一级的范式2NF，即“规范化”。

### 第二范式2NF

第二范式是在第一范式基础上建立起来的，即必须先满足第一范式。第二范式要求数据库表中的每个实例或者行必须可以被唯一的区分。为实现区分，通常需要为每个表加上一个列，以存储各个实例的唯一标识。这个唯一的属性列称为主键字或主键 主码。

第二范式要求实体的属性完全依赖于主关键字。所谓完全依赖是指不能存在仅依赖主关键字一部分属性，如果存在，那么这个属性和主关键字的这一部分应该分离出来形成一个新的实体，新实体和原实体之间是一对多的关系。简而言之，就是非主属性非部分依赖于主关键字。

如下表：

学生 课程 教师 教师职称 教材 教室 上课时间  
李四 Spring 张老师 java讲师 《Spring深入浅出》 301 08:00  
张三 Struts 杨老师 java讲师 《Struts in Action》 302 13:30

这里通过（学生 课程）可以确定一条记录。故将其作为联合主键，但是教材并不完全依赖于（学生 课程），只拿出部分课程就可以确定教材。这是部分依赖。

正确的表应设计如下：

选课表

学生 课程 教师 教师职称 教室 上课时间  
李四 Spring 张老师 java讲师 301 08:00  
张三 Struts 杨老师 java讲师 302 13:30

课程包：

课程 教材   
Spring 《Spring深入浅出》   
Struts 《Struts in Action》

第二范式消除了部分依赖，减少了插入异常 删除异常和修改异常。

如果继续以第一范式中那张表为例，（学号 课名）作为主键才能唯一标识分数，但是其他的属性如姓名 系名和系主任只需要学号就可以唯一确定了，这也是部分依赖了。

将表修改如下：



是否还存在之前的问题？

1 小明转到法律系，只需要修改一次对应的系的值即可。

2 姓名 系名 系主任没有重复那么多次，数据冗余减少了。

3 删除某个系中学生记录，信息仍然全部丢失。

4 插入一个尚无学生的新系信息，仍然存在插入异常。

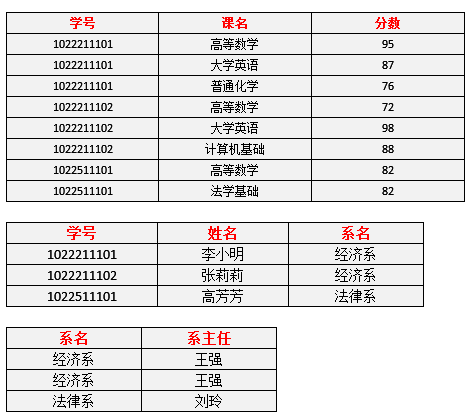
### 第三范式3NF

满足第三范式必须先满足第二范式。第三范式要求一个数据表中不包含已在其他表中包含的非主关键字信息。例如一个部门信息表，每个部门有个部门编号，部门名称等，那么在员工信息表中列出部门编号后，就不能再将部门名称等与部门相关的信息再加入到员工信息表中。如果不存在部门信息表，那根据第三范式也应该构建他，否则就会出现大量的冗余。简而言之，第三范式就是属性不依赖于其他非主属性。（消除传递依赖，消除冗余）。

仍以上述表为例，对于学生表，系名决定了系主任，存在了非主属性系主任对学号的传递函数依赖，不符合3NF。

故对表进行进一步分解：

选课（学号，课名，分数）  
学生（学号，姓名，系名）  
系（系名，系主任）



上述的问题也都得到了解决。

由此可见，3NF基本上解决了数据冗余过大，插入异常，修改异常，删除异常的问题。

总结：1NF就是原子性，字段不可再分割

2NF就是完全依赖，非主属性完全依赖于主属性，不存在部分依赖

3NF就是非主属性中没有传递依赖。每列中都与主键有直接关系。

## 数据库优化思路

### 1.SQL语句优化

1）应尽量避免在 where 子句中使用!=或<>操作符，否则将引擎放弃使用索引而进行全表扫描。  
2）应尽量避免在 where 子句中对字段进行 null 值判断，否则将导致引擎放弃使用索引而进行全表扫描，如：  
select id from t where num is null  
**可以在num上设置默认值0，确保表中num列没有null值**，然后这样查询：  
select id from t where num=0  
3）很多时候用 exists 代替 in 是一个好的选择  
4）用Where子句替换HAVING 子句 因为HAVING 只会在检索出所有记录之后才对结果集进行过滤

### 2.索引优化

看上文索引

### 3.数据库结构优化

1）范式优化： 比如消除冗余（节省空间。。） 2）反范式优化：比如适当加冗余等（减少join） 3）拆分表： 分区将数据在物理上分隔开，不同分区的数据可以制定保存在处于不同磁盘上的数据文件里。这样，当对这个表进行查询时，只需要在表分区中进行扫描，而不必进行全表扫描，明显缩短了查询时间，另外处于不同磁盘的分区也将对这个表的数据传输分散在不同的磁盘I/O，一个精心设置的分区可以将数据传输对磁盘I/O竞争均匀地分散开。对数据量大的时时表可采取此方法。可按月自动建表分区。  
4）拆分其实又分垂直拆分和水平拆分： 案例： 简单购物系统暂设涉及如下表： 1.产品表（数据量10w，稳定） 2.订单表（数据量200w，且有增长趋势） 3.用户表 （数据量100w，且有增长趋势） 以[mysql](http://www.2cto.com/database/MySQL/)为例讲述下水平拆分和垂直拆分，mysql能容忍的数量级在百万静态数据可以到千万 **垂直拆分：**解决问题：表与表之间的io竞争 不解决问题：单表中数据量增长出现的压力 方案： 把产品表和用户表放到一个server上 订单表单独放到一个server上 **水平拆分：** 解决问题：单表中数据量增长出现的压力 不解决问题：表与表之间的io争夺  
方案： 用户表通过性别拆分为男用户表和女用户表 订单表通过已完成和完成中拆分为已完成订单和未完成订单 产品表 未完成订单放一个server上 已完成订单表盒男用户表放一个server上 女用户表放一个server上(女的爱购物 哈哈)

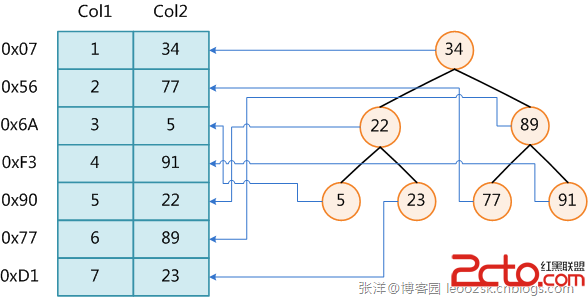
### 4.服务器硬件优化

## 索引的工作原理及其种类

**数据库索引**，是数据库管理系统中一个排序的数据结构，以协助快速查询、更新数据库表中数据。**索引的实现通常使用B树及其变种B+树**。

在数据之外，数据库系统还维护着满足特定查找算法的数据结构，这些数据结构以某种方式引用（指向）数据，这样就可以在这些数据结构上实现高级查找算法。这种数据结构，就是索引。

为表设置索引要付出代价的：一是增加了数据库的存储空间，二是在插入和修改数据时要花费较多的时间(因为索引也要随之变动)。



图展示了一种可能的索引方式。左边是数据表，一共有两列七条记录，最左边的是数据记录的物理地址（注意逻辑上相邻的记录在磁盘上也并不是一定物理相邻的）。为了加快Col2的查找，可以维护一个右边所示的二叉查找树，每个节点分别包含索引键值和一个指向对应数据记录物理地址的指针，这样就可以运用二叉查找在O(log2n)的复杂度内获取到相应数据。

创建索引可以大大提高系统的性能。

第一，通过创建唯一性索引，可以保证数据库表中每一行数据的唯一性。

第二，可以大大加快数据的检索速度，这也是创建索引的最主要的原因。

第三，可以加速表和表之间的连接，特别是在实现数据的参考完整性方面特别有意义。

第四，在使用分组和排序子句进行数据检索时，同样可以显著减少查询中分组和排序的时间。

第五，通过使用索引，可以在查询的过程中，使用优化隐藏器，提高系统的性能。

也许会有人要问：增加索引有如此多的优点，为什么不对表中的每一个列创建一个索引呢？因为，增加索引也有许多不利的方面。

第一，创建索引和维护索引要耗费时间，这种时间随着数据量的增加而增加。

第二，索引需要占物理空间，除了数据表占数据空间之外，每一个索引还要占一定的物理空间，如果要建立聚簇索引，那么需要的空间就会更大。

第三，当对表中的数据进行增加、删除和修改的时候，索引也要动态的维护，这样就降低了数据的维护速度。

索引是建立在数据库表中的某些列的上面。在创建索引的时候，应该考虑在哪些列上可以创建索引，在哪些列上不能创建索引。**一般来说，应该在这些列上创建索引：**在经常需要搜索的列上，可以加快搜索的速度；在作为主键的列上，强制该列的唯一性和组织表中数据的排列结构；在经常用在连接的列上，这些列主要是一些外键，可以加快连接的速度；在经常需要根据范围进行搜索的列上创建索引，因为索引已经排序，其指定的范围是连续的；在经常需要排序的列上创建索引，因为索引已经排序，这样查询可以利用索引的排序，加快排序查询时间；在经常使用在WHERE子句中的列上面创建索引，加快条件的判断速度。

同样，对于有些列不应该创建索引。**一般来说，不应该创建索引的的这些列具有下列特点：**

第一，对于那些在查询中很少使用或者参考的列不应该创建索引。这是因为，既然这些列很少使用到，因此有索引或者无索引，并不能提高查询速度。相反，由于增加了索引，反而降低了系统的维护速度和增大了空间需求。

第二，对于那些只有很少数据值的列也不应该增加索引。这是因为，由于这些列的取值很少，例如人事表的性别列，在查询的结果中，结果集的数据行占了表中数据行的很大比例，即需要在表中搜索的数据行的比例很大。增加索引，并不能明显加快检索速度。

第三，对于那些定义为text, image和bit数据类型的列不应该增加索引。这是因为，这些列的数据量要么相当大，要么取值很少。

第四，当修改性能远远大于检索性能时，不应该创建索引。这是因为，**修改性能和检索性能是互相矛盾的**。当增加索引时，会提高检索性能，但是会降低修改性能。当减少索引时，会提高修改性能，降低检索性能。因此，当修改性能远远大于检索性能时，不应该创建索引。

根据数据库的功能，可以在[数据库设计](http://www.2cto.com/database/)器中创建三种索引：**唯一索引、主键索引和聚集索引**。

**唯一索引**

唯一索引是不允许其中任何两行具有相同索引值的索引。

当现有数据中存在重复的键值时，大多数数据库不允许将新创建的唯一索引与表一起保存。数据库还可能防止添加将在表中创建重复键值的新数据。例如，如果在employee表中职员的姓(lname)上创建了唯一索引，则任何两个员工都不能同姓。 **主键索引** 数据库表经常有一列或列组合，其值唯一标识表中的每一行。该列称为表的主键。 在数据库关系图中为表定义主键将自动创建主键索引，主键索引是唯一索引的特定类型。该索引要求主键中的每个值都唯一。当在查询中使用主键索引时，它还允许对数据的快速访问。 **聚集索引** 在聚集索引中，表中行的物理顺序与键值的逻辑（索引）顺序相同。**一个表只能包含一个聚集索引。**

如果某索引不是聚集索引，则表中行的物理顺序与键值的逻辑顺序不匹配。**与非聚集索引相比，聚集索引通常提供更快的数据访问速度。**

### 局部性原理与磁盘预读

由于存储介质的特性，磁盘本身存取就比主存慢很多，再加上机械运动耗费，磁盘的存取速度往往是主存的几百分分之一，因此为了提高效率，要尽量减少磁盘I/O。为了达到这个目的，磁盘往往不是严格按需读取，而是每次都会预读，即使只需要一个字节，磁盘也会从这个位置开始，顺序向后读取一定长度的数据放入内存。这样做的理论依据是计算机科学中著名的**局部性原理**：**当一个数据被用到时，其附近的数据也通常会马上被使用。程序运行期间所需要的数据通常比较集中。**

由于磁盘顺序读取的效率很高（不需要寻道时间，只需很少的旋转时间），因此对于具有局部性的程序来说，预读可以提高I/O效率。

预读的长度一般为页（page）的整倍数。页是计算机管理存储器的逻辑块，硬件及操作系统往往将主存和磁盘存储区分割为连续的大小相等的块，每个存储块称为一页（在许多操作系统中，页得大小通常为4k），主存和磁盘以页为单位交换数据。当程序要读取的数据不在主存中时，会触发一个缺页异常，此时系统会向磁盘发出读盘信号，磁盘会找到数据的起始位置并向后连续读取一页或几页载入内存中，然后异常返回，程序继续运行。

### B-/+Tree索引的性能分析

到这里终于可以分析B-/+Tree索引的性能了。

上文说过一般使用磁盘I/O次数评价索引结构的优劣。先从B-Tree分析，根据B-Tree的定义，可知检索一次最多需要访问h个节点。数据库系统的设计者巧妙利用了磁盘预读原理，将一个节点的大小设为等于一个页，这样每个节点只需要一次I/O就可以完全载入。为了达到这个目的，在实际实现B-Tree还需要使用如下技巧：

每次新建节点时，直接申请一个页的空间，这样就保证一个节点物理上也存储在一个页里，加之计算机存储分配都是按页对齐的，就实现了一个node只需一次I/O。

**B-Tree中一次检索最多需要h-1次I/O（根节点常驻内存），渐进复杂度为O(h)=O(logdN)。**一般实际应用中，出度d是非常大的数字，通常超过100，因此h非常小（通常不超过3）。

而红黑树这种结构，h明显要深的多。由于逻辑上很近的节点（父子）物理上可能很远，无法利用局部性，所以红黑树的I/O渐进复杂度也为O(h)，效率明显比B-Tree差很多。

**综上所述，用B-Tree作为索引结构效率是非常高的。**

## 数据库事务的四个特性及含义

数据库事务transanction正确执行的四个基本要素。ACID,原子性(Atomicity)、一致性(Correspondence)、隔离性(Isolation)、持久性(Durability)。  
**原子性**:整个事务中的所有操作，要么全部完成，要么全部不完成，不可能停滞在中间某个环节。事务在执行过程中发生错误，会被回滚（Rollback）到事务开始前的状态，就像这个事务从来没有执行过一样。  
**一致性**:在事务开始之前和事务结束以后，数据库的完整性约束没有被破坏。  
**隔离性**:隔离状态执行事务，使它们好像是[系统](http://www.2cto.com/os/)在给定时间内执行的唯一操作。如果有两个事务，运行在相同的时间内，执行 相同的功能，事务的隔离性将确保每一事务在系统中认为只有该事务在使用系统。这种属性有时称为串行化，为了防止事务操作间的混淆，必须串行化或序列化请 求，使得在同一时间仅有一个请求用于同一数据。  
**持久性**:在事务完成以后，该事务所对数据库所作的更改便持久的保存在数据库之中，并不会被回滚。

## 视图的作用，视图可以更改么？

视图是虚拟的表，与包含数据的表不一样，视图只包含使用时动态检索数据的查询；不包含任何列或数据。使用视图可以简化复杂的sql操作，隐藏具体的细节，保护数据；视图创建后，可以使用与表相同的方式利用它们。  
视图不能被索引，也不能有关联的触发器或默认值，如果视图本身内有order by 则对视图再次order by将被覆盖。  
创建视图：create view XXX as XXXXXXXXXXXXXX;  
对于某些视图比如未使用联结子查询分组聚集函数Distinct Union等，是可以对其更新的，对视图的更新将对基表进行更新；但是视图主要用于简化检索，保护数据，并不用于更新，而且大部分视图都不可以更新。

## drop,delete与truncate的区别

drop直接删掉表 truncate删除表中数据，再插入时自增长id又从1开始 delete删除表中数据，可以加where字句。

（1） DELETE语句执行删除的过程是每次从表中删除一行，并且同时将该行的删除操作作为事务记录在日志中保存以便进行进行回滚操作。TRUNCATE TABLE 则一次性地从表中删除所有的数据并不把单独的删除操作记录记入日志保存，删除行是不能恢复的。并且在删除的过程中不会激活与表有关的删除触发器。执行速度快。

（2） 表和索引所占空间。当表被TRUNCATE 后，这个表和索引所占用的空间会恢复到初始大小，而DELETE操作不会减少表或索引所占用的空间。drop语句将表所占用的空间全释放掉。

（3） 一般而言，drop > truncate > delete

（4） 应用范围。TRUNCATE 只能对TABLE；DELETE可以是table和view

（5） TRUNCATE 和DELETE只删除数据，而DROP则删除整个表（结构和数据）。

（6） truncate与不带where的delete ：只删除数据，而不删除表的结构（定义）drop语句将删除表的结构被依赖的约束（constrain),触发器（trigger)索引（index);依赖于该表的存储过程/函数将被保留，但其状态会变为：invalid。

（7） delete语句为DML（data maintain Language),这个操作会被放到 rollback segment中,事务提交后才生效。如果有相应的 tigger,执行的时候将被触发。

（8） truncate、drop是DLL（data define language),操作立即生效，原数据不放到 rollback segment中，不能回滚

（9） 在没有备份情况下，谨慎使用 drop 与 truncate。要删除部分数据行采用delete且注意结合where来约束影响范围。回滚段要足够大。要删除表用drop;若想保留表而将表中数据删除，如果于事务无关，用truncate即可实现。如果和事务有关，或老师想触发trigger,还是用delete。

（10） Truncate table 表名 速度快,而且效率高,因为:  
truncate table 在功能上与不带 WHERE 子句的 DELETE 语句相同：二者均删除表中的全部行。但 TRUNCATE TABLE 比 DELETE 速度快，且使用的系统和事务日志资源少。DELETE 语句每次删除一行，并在事务日志中为所删除的每行记录一项。TRUNCATE TABLE 通过释放存储表数据所用的数据页来删除数据，并且只在事务日志中记录页的释放。

（11） TRUNCATE TABLE 删除表中的所有行，但表结构及其列、约束、索引等保持不变。新行标识所用的计数值重置为该列的种子。如果想保留标识计数值，请改用 DELETE。如果要删除表定义及其数据，请使用 DROP TABLE 语句。

（12） 对于由 FOREIGN KEY 约束引用的表，不能使用 TRUNCATE TABLE，而应使用不带 WHERE 子句的 DELETE 语句。由于 TRUNCATE TABLE 不记录在日志中，所以它不能激活触发器。

## 存储过程与触发器的区别

触发器与存储过程非常相似，触发器也是SQL语句集，**两者唯一的区别是触发器不能用EXECUTE语句调用，而是在用户执行Transact-SQL语句时自动触发（激活）执行。触发器是在一个修改了指定表中的数据时执行的存储过程。**通**常通过创建触发器来强制实现不同表中的逻辑相关数据的引用完整性和一致性。**由于用户不能绕过触发器，所以可以用它来强制实施复杂的业务规则，以确保数据的完整性。触发器不同于存储过程，**触发器主要是通过事件执行触发而被执行的**，而**存储过程可以通过存储过程名称名字而直接调用**。当对某一表进行诸如UPDATE、INSERT、DELETE这些操作时，SQLSERVER就会自动执行触发器所定义的SQL语句，从而确保对数据的处理必须符合这些SQL语句所定义的规则。

## 主键 超键 候选键 外键

**主 键：**

数据库表中对储存数据对象予以唯一和完整标识的数据列或属性的组合。**一个数据列只能有一个主键**，且主键的取值不能缺失，即不能为空值（Null）。

**超 键：**

在关系中能唯一标识元组的属性集称为关系模式的超键。一个属性可以为作为一个超键，多个属性组合在一起也可以作为一个超键。**超键包含候选键和主键。**

**候选键：**

是**最小超键**，即没有冗余元素的超键。

**外 键：**

在一个表中存在的**另一个表的主键**称此表的外键。

## 乱码问题

set names utf8/gbk;可以解决乱码问题。

character\_set\_client，character\_set\_connection，character\_set\_results均会变为相应的编码。

信息输入路径：client→connection→server；信息输出路径：server→connection→results。每个路径都需要经过三次改变字符集的编码，但是如果字符集编码不兼容，转化过程不可逆，就会出现乱码。如latin1和utf8不兼容。

## 存储引擎

存储引擎也叫表类型，是指的是数据表的存储机制，索引方案等配套相关功能。

默认的存储引擎通过my.ini文件配置Default-storage-engine=INNODB

在创建表或者编辑表时，可以指定存储引擎。

alter table tbl\_name engine myisam;

create table tbl\_name (

) engine myisam character set utf-8;

Innodb和myisam区别：

保存文件时，myisam，一个表，三个文件。

Tbl\_name.frm 结构

Tbl\_name.myd数据

Tbl\_name.myi索引

Innodb：一个表一个文件，tb\_name.frm结构。所有的innodb表都使用相同的innodb存储表空间保存数据和索引。（ibd文件）

数据和索引的保存的方式不同：myisam 是分开保存的，innodb是保存到表空间。

Myisam支持索引压缩，innodb索引和数据是绑定保存不压缩，体积大。

Innodb很多时候是行级锁，而myisam是表级锁，innodb并发高。

Innodb支持事务，外键，数据完整性约束要强。而myisam不支持。

Innodb中不保存表中的具体行数，也就是说select count(\*) from table时，innodb要扫描一遍表来计算有多少行，但是myisam只要简单的读出保存好的行数即可。

对于auto\_increment类型的字段，innodb中必须包含只有该字段的索引，但是在myisam表中，可以和其他字段一起建立联合索引。

## 高可用

http://gitbook.cn/books/583c1335c7f2666319396f7f/index.html

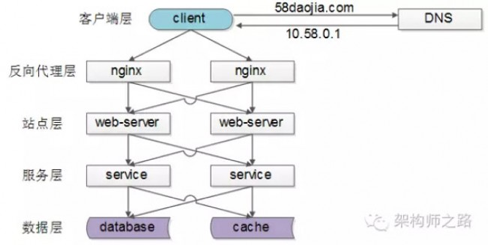
## 负载均衡

负载均衡（Load Balance）是分布式系统架构设计中必须考虑的因素之一，它通常是指，将请求/数据【均匀】分摊到多个操作单元上执行，**负载均衡的关键在于【均匀】**。常见互联网分布式架构如上，分为客户端层、反向代理nginx层、站点层、服务层、数据层。

**什么是负载均衡**

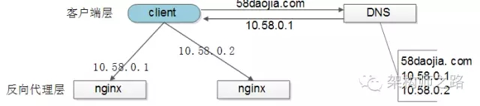
负载均衡(Load Balance)是分布式系统架构设计中必须考虑的因素之一，它通常是指，将请求/数据【均匀】分摊到多个操作单元上执行，负载均衡的关键在于【均匀】。

**常见的负载均衡方案**



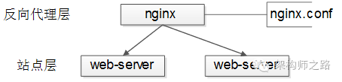
常见互联网分布式架构如上，分为客户端层、反向代理nginx层、站点层、服务层、数据层。可以看到，每一个下游都有多个上游调用，只需要做到，**每一个上游都均匀访问每一个下游，就能实现“将请求/数据【均匀】分摊到多个操作单元上执行”。**

**【客户端层->反向代理层】的负载均衡**



【客户端层】到【反向代理层】的负载均衡，是**通过“DNS轮询”实现**的：DNS-server对于一个域名配置了多个解析ip，每次DNS解析请求来访问DNS-server，会轮询返回这些ip，保证每个ip的解析概率是相同的。这些ip就是nginx的外网ip，以做到每台nginx的请求分配也是均衡的。

**【反向代理层->站点层】的负载均衡**



【反向代理层】到【站点层】的负载均衡，是**通过“nginx”实现**的。通过修改nginx.conf，可以实现多种负载均衡策略：

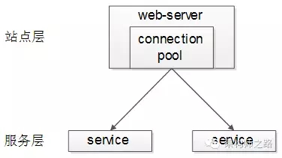
1)请求轮询：和DNS轮询类似，请求依次路由到各个web-server

2)最少连接路由：哪个web-server的连接少，路由到哪个web-server

3)ip哈希：按照访问用户的ip哈希值来路由web-server，只要用户的ip分布是均匀的，请求理论上也是均匀的，ip哈希均衡方法可以做到，同一个用户的请求固定落到同一台web-server上，此策略适合有状态服务，例如session(58沈剑备注：可以这么做，但强烈不建议这么做，站点层无状态是分布式架构设计的基本原则之一，session最好放到数据层存储)

4)…

**【站点层->服务层】的负载均衡**



【站点层】到【服务层】的负载均衡，是**通过“服务连接池”实现**的。

上游连接池会建立与下游服务多个连接，每次请求会“随机”选取连接来访问下游服务。

上一篇文章《RPC-client实现细节》中有详细的负载均衡、故障转移、超时处理的细节描述，欢迎点击link查阅，此处不再展开。

**【数据层】的负载均衡**

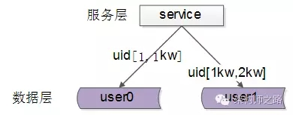
在数据量很大的情况下，由于数据层(db，cache)涉及数据的水平切分，所以数据层的负载均衡更为复杂一些，它分为“数据的均衡”，与“请求的均衡”。

数据的均衡是指：水平切分后的每个服务(db，cache)，数据量是差不多的。

请求的均衡是指：水平切分后的每个服务(db，cache)，请求量是差不多的。

业内常见的水平切分方式有这么几种：

**一、按照range水平切分**



每一个数据服务，存储一定范围的数据，上图为例：

user0服务，存储uid范围1-1kw

user1服务，存储uid范围1kw-2kw

这个方案的好处是：

(1)规则简单，service只需判断一下uid范围就能路由到对应的存储服务

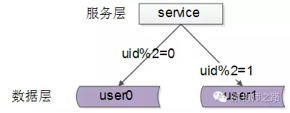
(2)数据均衡性较好

(3)比较容易扩展，可以随时加一个uid[2kw,3kw]的数据服务

不足是：

(1)请求的负载不一定均衡，一般来说，新注册的用户会比老用户更活跃，大range的服务请求压力会更大

**二、按照id哈希水平切分**



每一个数据服务，存储某个key值hash后的部分数据，上图为例：

user0服务，存储偶数uid数据

user1服务，存储奇数uid数据

这个方案的好处是：

(1)规则简单，service只需对uid进行hash能路由到对应的存储服务

(2)数据均衡性较好

(3)请求均匀性较好

不足是：

(1)不容易扩展，扩展一个数据服务，hash方法改变时候，可能需要进行数据迁移

**总结**

负载均衡(Load Balance)是分布式系统架构设计中必须考虑的因素之一，它通常是指，将请求/数据【均匀】分摊到多个操作单元上执行，负载均衡的关键在于【均匀】。

(1)【客户端层】到【反向代理层】的负载均衡，是通过“DNS轮询”实现的

(2)【反向代理层】到【站点层】的负载均衡，是通过“nginx”实现的

(3)【站点层】到【服务层】的负载均衡，是通过“服务连接池”实现的

(4)【数据层】的负载均衡，要考虑“数据的均衡”与“请求的均衡”两个点，常见的方式有“按照范围水平切分”与“hash水平切分”

**Nginx/LVS/HAProxy负载均衡软件**

一般对负载均衡的使用是随着网站规模的提升根据不同的阶段来使用不同的技术。具体的应用需求还得具体分析，如果是中小型的Web应用，比如日PV小于1000万，用Nginx就完全可以了；如果机器不少，可以用DNS轮询，LVS所耗费的机器还是比较多的；大型网站或重要的服务，且服务器比较多时，可以考虑用LVS。

一种是通过硬件来进行进行，常见的硬件有比较昂贵的F5和Array等商用的负载均衡器，它的优点就是有专业的维护团队来对这些服务进行维护、缺点就是花销太大，所以对于规模较小的网络服务来说暂时还没有需要使用；另外一种就是类似于Nginx/LVS/HAProxy的基于[Linux](http://www.ha97.com/category/linux)的开源免费的负载均衡软件，这些都是通过软件级别来实现，所以费用非常低廉。

目前关于网站架构一般比较合理流行的架构方案：**Web前端采用Nginx/HAProxy+Keepalived作负载均衡器；后端采用**[**MySQL**](http://www.ha97.com/tag/mysql)**数据库一主多从和读写分离，采用LVS+Keepalived的架构。**当然要根据项目具体需求制定方案。  
下面说说各自的特点和适用场合。

一、Nginx

**Nginx的优点是：**

1、工作在网络的7层之上，可以针对http应用做一些分流的策略，比如针对域名、目录结构，它的**正则规则**比HAProxy更为强大和灵活，这也是它目前广泛流行的主要原因之一，Nginx单凭这点可利用的场合就远多于LVS了。  
2、**Nginx对网络稳定性的依赖非常小**，理论上能ping通就就能进行负载功能，这个也是它的优势之一；相反LVS对网络稳定性依赖比较大，这点本人深有体会；  
3、Nginx安装和配置比较简单，测试起来比较方便，它基本能把错误用日志打印出来。LVS的配置、测试就要花比较长的时间了，LVS对网络依赖比较大。  
3、可以承担高负载压力且稳定，在硬件不差的情况下一般能支撑几万次的并发量，负载度比LVS相对小些。  
4、Nginx可以通过端口检测到服务器内部的故障，比如根据服务器处理网页返回的状态码、超时等等，并且会把返回错误的请求重新提交到另一个节点，不过其中缺点就是不支持url来检测。比如用户正在上传一个文件，而处理该上传的节点刚好在上传过程中出现故障，Nginx会把上传切到另一台服务器重新处理，而LVS就直接断掉了，如果是上传一个很大的文件或者很重要的文件的话，用户可能会因此而不满。  
5、Nginx不仅仅是一款优秀的负载均衡器/反向代理软件，它同时也是功能强大的Web应用服务器。LNMP也是近几年非常流行的web架构，在高流量的环境中稳定性也很好。  
6、Nginx现在作为Web反向加速缓存越来越成熟了，速度比传统的[Squid](http://www.ha97.com/tag/squid)服务器更快，可以考虑用其作为反向代理加速器。  
7、Nginx可作为中层反向代理使用，这一层面Nginx基本上无对手，唯一可以对比Nginx的就只有lighttpd了，不过lighttpd目前还没有做到Nginx完全的功能，配置也不那么清晰易读，社区资料也远远没Nginx活跃。  
8、Nginx也可作为静态网页和图片服务器，这方面的性能也无对手。还有Nginx社区非常活跃，第三方模块也很多。

淘宝的前端使用的Tengine就是基于nginx做的二次开发定制版。

Nginx常规的HTTP请求和响应流程图：

**Nginx的缺点是：**  
1、Nginx仅能支持http、[https](http://www.ha97.com/tag/https)和Email协议，这样就在适用范围上面小些，这个是它的缺点。  
2、**对后端服务器的健康检查，只支持通过端口来检测，不支持通过url来检测。**不支持Session的直接保持，但能通过ip\_hash来解决。

**二、LVS**

LVS：使用[Linux](http://www.ha97.com/tag/linux)内核集群实现一个高性能、高可用的负载均衡服务器，它具有很好的可伸缩性（Scalability)、可靠性（Reliability)和可管理性（Manageability)。

**LVS的优点是：**  
1、抗负载能力强、是工作在网络4层之上仅作分发之用，**没有流量的产生**，这个特点也决定了它在负载均衡软件里的性能最强的，对内存和cpu资源消耗比较低。  
2、配置性比较低，这是一个缺点也是一个优点，因为没有可太多配置的东西，所以并不需要太多接触，大大减少了人为出错的几率。  
3、工作稳定，因为其本身抗负载能力很强，自身有完整的双机热备方案，如LVS+Keepalived，不过我们在项目实施中用得最多的还是LVS/DR+Keepalived。  
4、无流量，LVS只分发请求，而流量并不从它本身出去，这点保证了均衡器IO的性能不会收到大流量的影响。  
5、应用范围比较广，因为LVS工作在4层，所以它几乎可以对所有应用做负载均衡，包括http、数据库、在线聊天室等等。

LVS DR(Direct Routing)模式的网络流程图：

**LVS的缺点是：**  
1、软件本身**不支持正则表达式处理，不能做动静分离**；而现在许多网站在这方面都有较强的需求，这个是Nginx/HAProxy+Keepalived的优势所在。  
2、如果是网站应用比较庞大的话，LVS/DR+Keepalived实施起来就比较复杂了，特别后面有Windows [Server](http://www.ha97.com/tag/server)的机器的话，如果实施及配置还有维护过程就比较复杂了，相对而言，Nginx/HAProxy+Keepalived就简单多了。

**三、HAProxy**

**HAProxy的特点是：**  
1、HAProxy也是支持虚拟主机的。  
2、HAProxy的优点能够补充Nginx的一些缺点，比如**支持Session的保持，Cookie的引导；同时支持通过获取指定的url来检测后端服务器的状态。**  
3、HAProxy跟LVS类似，本身就只是一款负载均衡软件；单纯从效率上来讲HAProxy会比Nginx有更出色的负载均衡速度，在并发处理上也是优于Nginx的。  
4、HAProxy**支持TCP协议的负载均衡转发**，可以对MySQL读进行负载均衡，对后端的MySQL节点进行检测和负载均衡，大家可以用LVS+Keepalived对MySQL主从做负载均衡。  
5、HAProxy负载均衡策略非常多，HAProxy的负载均衡算法现在具体有如下8种：  
① roundrobin，表示简单的轮询，这个不多说，这个是负载均衡基本都具备的；  
② static-rr，表示根据权重，建议关注；  
③ leastconn，表示最少连接者先处理，建议关注；  
④ source，表示根据请求源IP，这个跟Nginx的IP\_hash机制类似，我们用其作为解决session问题的一种方法，建议关注；  
⑤ ri，表示根据请求的URI；  
⑥ rl\_param，表示根据请求的URl参数’balance url\_param’ requires an URL parameter name；  
⑦ hdr(name)，表示根据HTTP请求头来锁定每一次HTTP请求；  
⑧ rdp-cookie(name)，表示根据据cookie(name)来锁定并哈希每一次TCP请求。

**四、总结**

**Nginx和LVS对比的总结：**  
1、Nginx工作在网络的7层，所以它可以针对http应用本身来做分流策略，比如针对域名、目录结构等，相比之下LVS并不具备这样的功能，所以Nginx单凭这点可利用的场合就远多于LVS了；但Nginx有用的这些功能使其可调整度要高于LVS，所以经常要去触碰触碰，触碰多了，人为出问题的几率也就会大。  
2、Nginx对网络稳定性的依赖较小，理论上只要ping得通，网页访问正常，Nginx就能连得通，这是Nginx的一大优势！Nginx同时还能区分内外网，如果是同时拥有内外网的节点，就相当于单机拥有了备份线路；LVS就比较依赖于网络环境，目前来看服务器在同一网段内并且LVS使用direct方式分流，效果较能得到保证。另外注意，LVS需要向托管商至少申请多一个ip来做Visual IP，貌似是不能用本身的IP来做VIP的。要做好LVS管理员，确实得跟进学习很多有关网络通信方面的知识，就不再是一个HTTP那么简单了。  
3、Nginx安装和配置比较简单，测试起来也很方便，因为它基本能把错误用日志打印出来。LVS的安装和配置、测试就要花比较长的时间了；LVS对网络依赖比较大，很多时候不能配置成功都是因为网络问题而不是配置问题，出了问题要解决也相应的会麻烦得多。  
4、Nginx也同样能承受很高负载且稳定，但负载度和稳定度差LVS还有几个等级：Nginx处理所有流量所以受限于机器IO和配置；本身的bug也还是难以避免的。  
5、Nginx可以检测到服务器内部的故障，比如根据服务器处理网页返回的状态码、超时等等，并且会把返回错误的请求重新提交到另一个节点。目前LVS中 ldirectd也能支持针对服务器内部的情况来监控，但LVS的原理使其不能重发请求。比如用户正在上传一个文件，而处理该上传的节点刚好在上传过程中出现故障，Nginx会把上传切到另一台服务器重新处理，而LVS就直接断掉了，如果是上传一个很大的文件或者很重要的文件的话，用户可能会因此而恼火。  
6、Nginx对请求的异步处理可以帮助节点服务器减轻负载，假如使用apache直接对外服务，那么出现很多的窄带链接时apache服务器将会占用大 量内存而不能释放，使用多一个Nginx做apache代理的话，这些窄带链接会被Nginx挡住，apache上就不会堆积过多的请求，这样就减少了相当多的资源占用。这点使用squid也有相同的作用，即使squid本身配置为不缓存，对apache还是有很大帮助的。  
7、Nginx能支持http、https和email（email的功能比较少用），LVS所支持的应用在这点上会比Nginx更多。在使用上，一般最前端所采取的策略应是LVS，也就是DNS的指向应为LVS均衡器，LVS的优点令它非常适合做这个任务。重要的ip地址，最好交由LVS托管，比如数据库的 ip、webservice服务器的ip等等，这些ip地址随着时间推移，使用面会越来越大，如果更换ip则故障会接踵而至。所以将这些重要ip交给 LVS托管是最为稳妥的，这样做的唯一缺点是需要的VIP数量会比较多。Nginx可作为LVS节点机器使用，一是可以利用Nginx的功能，二是可以利用Nginx的性能。当然这一层面也可以直接使用squid，squid的功能方面就比Nginx弱不少了，性能上也有所逊色于Nginx。Nginx也可作为中层代理使用，这一层面Nginx基本上无对手，唯一可以撼动Nginx的就只有lighttpd了，不过lighttpd目前还没有能做到 Nginx完全的功能，配置也不那么清晰易读。另外，中层代理的IP也是重要的，所以中层代理也拥有一个VIP和LVS是最完美的方案了。具体的应用还得具体分析，如果是比较小的网站（日PV小于1000万），用Nginx就完全可以了，如果机器也不少，可以用DNS轮询，LVS所耗费的机器还是比较多的；大型网站或者重要的服务，机器不发愁的时候，要多多考虑利用LVS。

**现在对网络负载均衡的使用是随着网站规模的提升根据不同的阶段来使用不同的技术：**

第一阶段：利用Nginx或HAProxy进行单点的负载均衡，这一阶段服务器规模刚脱离开单服务器、单数据库的模式，需要一定的负载均衡，但是仍然规模较小没有专业的维护团队来进行维护，也没有需要进行大规模的网站部署。这样利用Nginx或HAproxy就是第一选择，此时这些东西上手快， 配置容易，在七层之上利用HTTP协议就可以。这时是第一选择。

第二阶段：随着网络服务进一步扩大，这时单点的Nginx已经不能满足，这时使用LVS或者商用Array就是首要选择，Nginx此时就作为LVS或者Array的节点来使用，具体LVS或Array的是选择是根据公司规模和预算来选择，**Array的应用交付功能非常强大，本人在某项目中使用过，性价比也远高于F5，商用首选！**但是一般来说这阶段相关人才跟不上业务的提升，所以购买商业负载均衡已经成为了必经之路。

第三阶段：这时网络服务已经成为主流产品，此时随着公司知名度也进一步扩展，相关人才的能力以及数量也随之提升，这时无论从开发适合自身产品的定制，以及降低成本来讲开源的LVS，已经成为首选，这时LVS会成为主流。  
最终形成比较理想的基本架构为：**Array/LVS — Nginx/Haproxy — Squid/Varnish — AppServer**

## 数据库分库分表

<http://blog.csdn.net/lihuayong/article/details/42044235>

<http://www.cnblogs.com/miketwais/articles/mysql_partition.html>

<http://www.2cto.com/database/201503/380348.html>

<http://mp.weixin.qq.com/s/ezD0CWHAr0RteC9yrwqyZA>

http://blog.csdn.net/heirenheiren/article/details/7896546

### 基本思想

Sharding（分片分表）的基本思想就要把一个[**数据库**](http://lib.csdn.net/base/mysql)切分成多个部分放到不同的数据库(server)上，从而缓解单一数据库的性能问题。对于海量数据的数据库，如果是因为表多而数据多，这时候适合使用垂直切分，即把关系紧密（比如同一模块）的表切分出来放在一个服务器上。如果表并不多，但每张表的数据非常多，这时候适合水平切分，即把表的数据按某种规则（比如按ID散列）切分到多个数据库(server)上。根据实际情况做出选择，也可能会综合使用垂直与水平切分。

## Innodb myisam merge存储引擎

http://blog.csdn.net/qmhball/article/details/7499432?reload

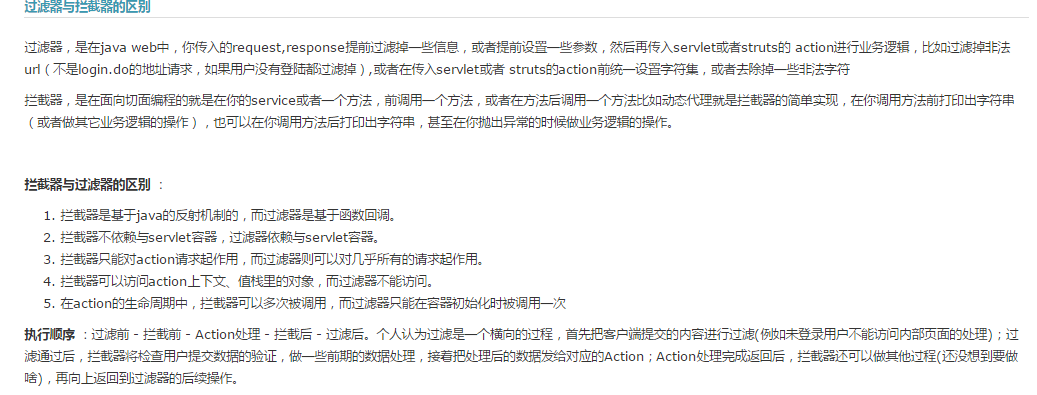
http://www.cnblogs.com/qiuqiuqiu/p/6476449.html

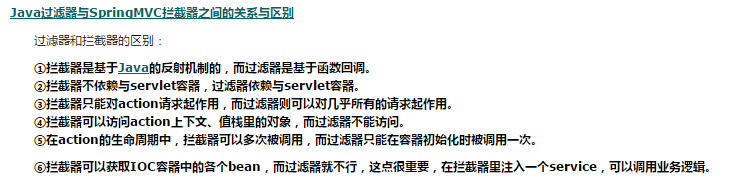
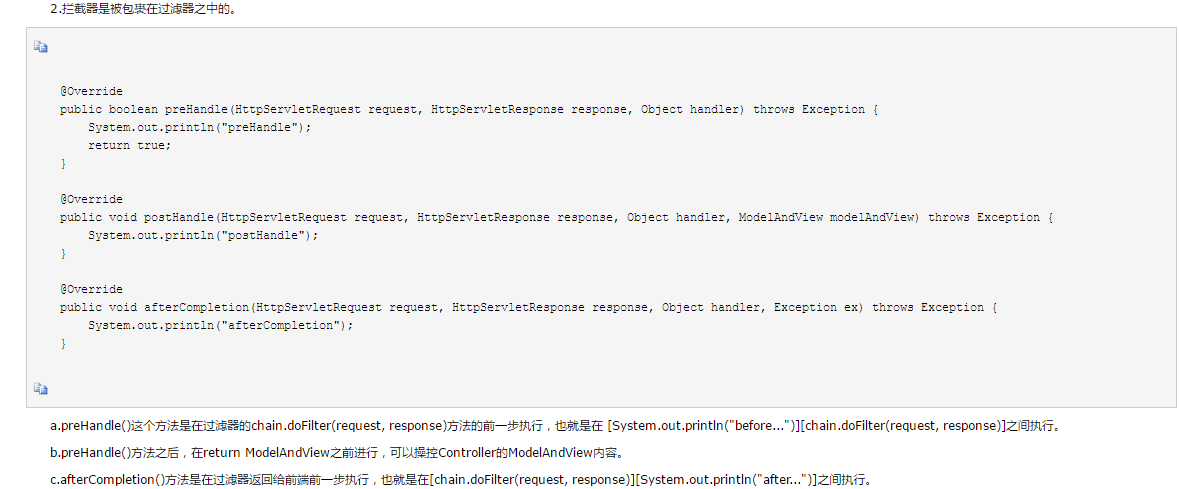
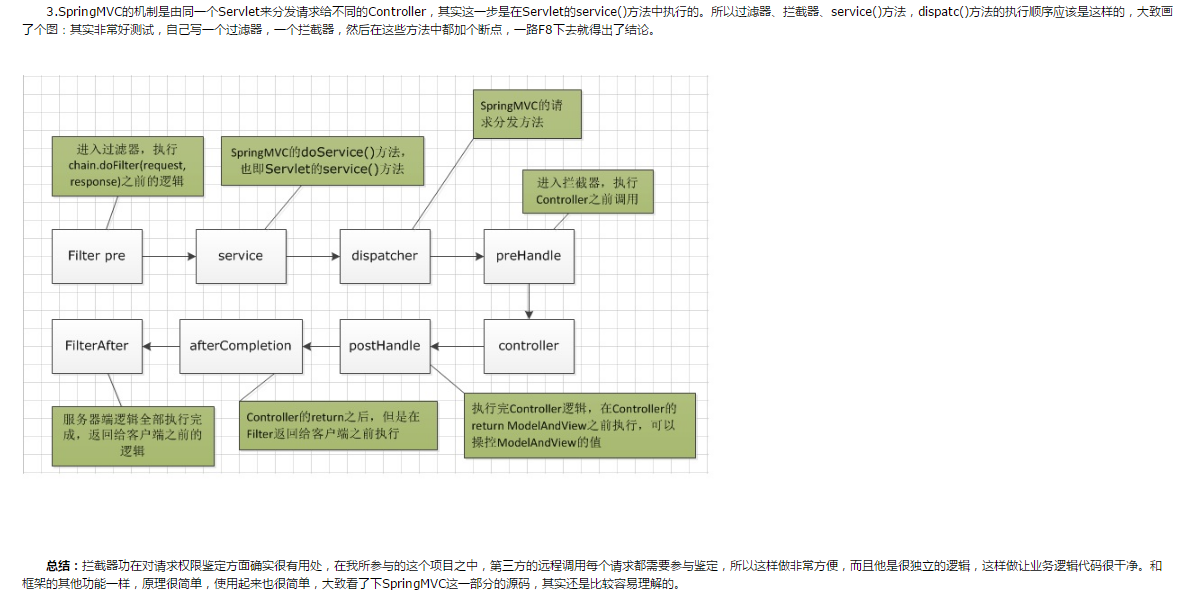
# Servlet

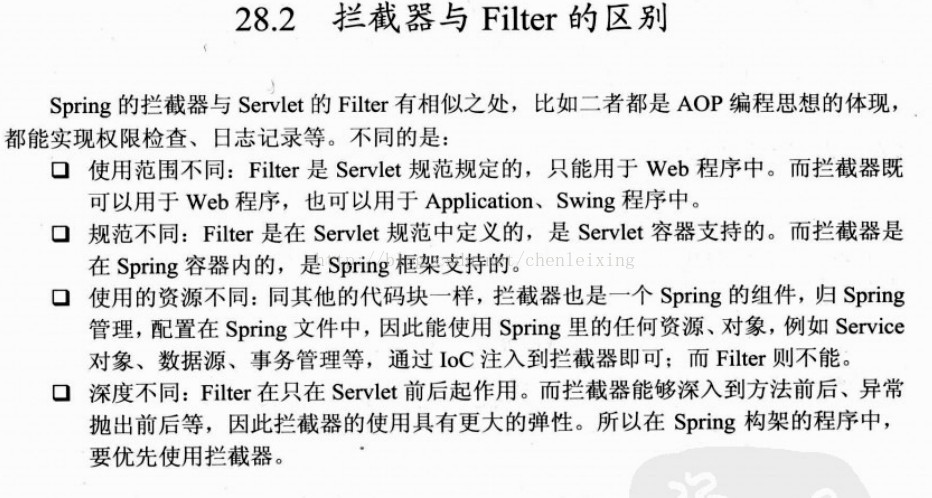
## filter servlet listener interceptor区别

<http://www.cnblogs.com/doit8791/p/4209442.html>

<http://www.cnblogs.com/zhangshitong/p/5995552.html>





## ServletConfig

在servlet配置文件中，可以使用<init-param>标签为servlet配置一些初始化参数。

当servlet配置了初始化参数后，web容器在创建servlet实例对象时，会自动将这些初始化参数封装到servletConfig对象中，并调用servlet的init方法时候，将servletConfig对象传递给Servlet，从而通过ServletConfig对象可以得到当前Servlet的初始化参数的信息。

String getInitParameter(String name)       得到参数

Enumeration getInitParameterNames()      得到所有参数

ServletContext getServletContext()           得到上下文对象

String getServletName()                          得到Servlet名称

可以获得配置文件中配置编码信息（防止硬编码） 数据库连接信息和配置文件信息

## ServletContext

ServletContext是一个Servlet上下文对象，该对象表示当前的web应用环境的信息，一个web应用只会创建一个ServletContext对象。Web容器启动时，ServletContext被创建，web容器关闭时，ServletContext被销毁。一个用户可以有多个request，只有一个session。但是对于ServletContext而言，所有用户共用一个。

在一个web应用中，所有servlet共享一个ServletContext对象，故多个Servlet可以通过ServletContext对象实现数据共享。

ServletContext对象是作为ServletConfig对象成员变量传入到servlet中。通过ServletConfig的getServletContext()方法就可以得到ServletContext对象。ServletConfig对象是在调用init方法之前创建的，在ServletContext对象创建之前。

换言之，运行在java虚拟机中的每一个web应用程序都有一个与之相关的Servlet上下文。ServletContext对象是Web服务器中的一个已知路径的根，Servlet上下文被定位于http://localhost:8080/项目名.以 /项目名 请求路径（称为上下文路径）开始的所有请求被发送到与此ServletContext关联的Web应用程序。一个ServletContext对象表示了一个Web应用程序的上下文。

Servlet上下文：Servlet上下文提供对应用程序中所有Servlet所共有的各种资源和功能的访问。Servlet上下文API用于设置应用程序中所有Servlet共有的信息。Servlet可能需要共享他们之间的共有信息。运行于同一服务器的Servlet有时会共享资源，如JSP页面、文件和其他Servlet。

例如：做一个购物类的网站，要从数据库中提取物品信息，如果用session保存这些物品信息，每个用户都访问一便数据库，效率就太低了；所以要用来Servlet上下文来保存，在服务器开始时，就访问数据库，将物品信息存入Servlet上下文中，这样，每个用户只用从上下文中读入物品信息就行了。

this.getServletContext(); 直接获取

this.getServletConfig.getServleyContext(); 间接获取

ServletContext的作用：

1.获取web的上下文路径

　　String getContextPath();

2.获取全局的参数

　　String getInitParameter(String name);

　　Enumeration getInitParameterNames();

3.和域对象相关的

　　void setAttribute(String name,Onject object);

　　Object getAttribute(String name);

　　void removeAttribute(String name);

　　域对象（域对象就是在不同资源之前来共享数据，保存数据，获取数据）

　　ServletContext是我们学习的第一个域对象(Servlet共有三个域对象ServletContext、HttpServletRequest、HttpSession)

4. 请求转发的

　　RequestDispatcher getRequestDispatcher(String path);

　　在Servlet跳转页面:

　　4.1请求重定向（你找我借钱，我没有，你自己去找他借钱）

　　　　1.地址栏**会改变**，变成重定向到的地址

　　　　2.**可以**跳转到项目内的资源，**也可以**跳转项目外的资源

　　　　3.浏览器向服务器发出**两次请求**，那么不能使用请求来作为域对象来共享数据。

　　4.2请求转发（你找我借钱，我没有，我帮你去向他借钱）

　　　　1.地址栏**不会改变**

　　　　2.**只能**跳转到项目内的资源，**不能**跳转项目外的资源。

　　　　3.浏览器向服务器发出**一次请求**，那么可以使用请求作为域对象共享数据。

5.读取web项目的资源文件

　　String getRealPath(String path);

　　InputStream getResourceAsStream(String path);

　　URL getResource(String path);

## servlet的生命周期

<http://blog.csdn.net/javaloveiphone/article/details/8154791>

**servlet的四个生命周期方法**

1 构造方法：在创建servlet对象的时候调用，只会调用一次。单例

2 init方法：创建完对象后会调用该方法进行初始化操作，只调用一次。

3 service方法：每次请求servlet时，都会调用。

4 destroy方法：在servlet对象销毁后，调用该方法，只调用一次。（服务器重启或者web应用重启）

servlet服务器是由web服务器调用，web服务器收到客户端的请求之后，web服务器首先检查是否已经装载并创建了该servlet实例对象。如果没有，则装载并创建该servlet的实例对象，调用servlet实例对象的init方法对其进行初始化。然后创建一个用于封装http请求消息的httpservletrequest对象和一个代表http响应的httpservletresponse对象，然后调用service()方法并将请求和响应作为参数传递进去。最后当tomcat重启或者应用重启时，会销毁该servlet对象（destroy方法）。

**servlet何时被创建？**

1 在第一次访问某个servlet时，web容器会创建该servlet实例对象。

2 在web.xml中，如果<servlet></servlet>元素中配置了一个<load-on-start>子元素，则容器在启动的时候就会自动加载这些servlet并调用init方法，完成一些全局性的初始化工作。

## servlet的单例多线程

struts2的action是多实例的，就是对于每个请求都会产生一个action对象。因为页面上的数据会存放在action中的成员变量中，如果是单例，那么这些数据在多线程的环境下就会影响。但是用spring管理的struts2的action时，默认是单例的。

在servlet中，采取的是单例模式。servlet容器维护一个线程池，里面存放着工作者线程来响应相应的请求，同时会有一个调度者线程来管理工作者线程。

当容器收到一个servlet请求后，调度线程就会从线程池中取出一个工作者线程，让工作者线程处理这个请求即调用service方法。当这个线程执行的时候，收到另外一个请求，调度线程就会从线程池中取出另外一个工作者线程来响应新的请求。容器不关心请求的是否访问同一个servlet，当多个请求同时访问同一个servlet时，这个servlet的service方法将在多线程中并发执行。并发执行会出现线程安全问题。

**如何同时处理多个请求？**

工作者线程Work Thread:执行代码的一组线程   
调度线程Dispatcher Thread：每个线程都具有分配给它的线程优先级,线程是根据优先级调度执行的   
Servlet采用多线程来处理多个请求同时访问。servlet依赖于一个线程池来服务请求。线程池实际上是一系列的工作者线程集合。Servlet使用一个调度线程来管理工作者线程.   
当容器收到一个Servlet请求，调度线程从线程池中选出一个工作者线程,将请求传递给该工作者线程，然后由该线程来执行Servlet的service方法。当这个线程正在执行的时候,容器收到另外一个请求,调度线程同样从线程池中选出另一个工作者线程来服务新的请求,容器并不关心这个请求是否访问的是同一个Servlet.当容器同时收到对同一个Servlet的多个请求的时候，那么这个Servlet的service()方法将在多线程中并发执行。   
Servlet容器默认采用单实例多线程的方式来处理请求，这样减少产生Servlet实例的开销，提升了对请求的响应时间，对于Tomcat可以在server.xml中通过元素设置线程池中线程的数目。   
就实现来说：   
调度者线程类所担负的责任是线程的调度，只需要利用自己的属性完成自己的责任。所以该类是承担了责任的，并且该类的责任又集中到唯一的单体对象中。   
而其他对象又依赖于该特定对象所承担的责任，我们就需要得到该特定对象。那该类就是一个单例模式的实现了。工作者线程Work Thread:执行代码的一组线程   
调度线程Dispatcher Thread：每个线程都具有分配给它的线程优先级,线程是根据优先级调度执行的   
Servlet采用多线程来处理多个请求同时访问。servlet依赖于一个线程池来服务请求。线程池实际上是一系列的工作者线程集合。Servlet使用一个调度线程来管理工作者线程.   
当容器收到一个Servlet请求，调度线程从线程池中选出一个工作者线程,将请求传递给该工作者线程，然后由该线程来执行Servlet的service方法。当这个线程正在执行的时候,容器收到另外一个请求,调度线程同样从线程池中选出另一个工作者线程来服务新的请求,容器并不关心这个请求是否访问的是同一个Servlet.当容器同时收到对同一个Servlet的多个请求的时候，那么这个Servlet的service()方法将在多线程中并发执行。   
Servlet容器默认采用单实例多线程的方式来处理请求，这样减少产生Servlet实例的开销，提升了对请求的响应时间，对于Tomcat可以在server.xml中通过元素设置线程池中线程的数目。   
就实现来说：   
调度者线程类所担负的责任是线程的调度，只需要利用自己的属性完成自己的责任。所以该类是承担了责任的，并且该类的责任又集中到唯一的单体对象中。   
而其他对象又依赖于该特定对象所承担的责任，我们就需要得到该特定对象。那该类就是一个单例模式的实现了。

**如何做到线程安全？**

1变量的线程安全：这里的变量指字段和共享数据(如表单参数值)。

* 1. 将 参数变量本地化。多线程并不共享局部变量.所以我们要尽可能的在servlet中使用局部变量。 例如：String user =   
     “”;   
     user = request.getParameter(“user”);   
     b，使用同步块Synchronized，防止可能异步调用的代码块。这意味着线程需要排队处理。 在使用同步块的时候要尽可能的缩小同步代码的范围，不要直接在sevice方法和响应方法上使用同步，这样会严重影响性能。

2,属性的线程安全：ServletContext，HttpSession，ServletRequest对象中属性   
ServletContext：（线程是不安全的）   
ServletContext是可以多线程同时读/写属性的，线程是不安全的。要对属性的读写进行同步处理或者进行深度Clone()。   
所以在Servlet上下文中尽可能少量保存会被修改（写）的数据，可以采取其他方式在多个Servlet中共享，比方我们可以使用单例模式来处理共享数据。   
HttpSession：（线程是不安全的）   
HttpSession对象在用户会话期间存在，只能在处理属于同一个Session的请求的线程中被访问，因此Session对象的属性访问理论上是线程安全的。   
当用户打开多个同属于一个进程的浏览器窗口，在这些窗口的访问属于同一个Session，会出现多次请求，需要多个工作线程来处理请求，可能造成同时多线程读写属性。   
这时我们需要对属性的读写进行同步处理：使用同步块Synchronized和使用读/写器来解决。   
ServletRequest：（线程是安全的）   
对于每一个请求，由一个工作线程来执行，都会创建有一个新的ServletRequest对象，所以ServletRequest对象只能在一个线程中被访问。ServletRequest是线程安全的。   
注意：ServletRequest对象在service方法的范围内是有效的，不要试图在service方法结束后仍然保存请求对象的引用。

3 使用同步的集合类，如Hash Table等的。

4不要在Servlet中创建自己的线程来完成某个功能。

Servlet本身就是多线程的，在Servlet中再创建线程，将导致执行情况复杂化，出现多线程安全问题。

5在多个servlet中对外部对象(比方文件)进行修改操作一定要加锁，做到互斥的访问。

6 javax.servlet.SingleThreadModel接口是一个标识接口，如果一个Servlet实现了这个接口，那Servlet容器将保证在一个时刻仅有一个线程可以在给定的servlet实例的service方法中执行。将其他所有请求进行排队。如果一个Servlet实现了SingleThreadModel接口，Servlet引擎将为每个新的请求创建一个单独的Servlet实例，这将引起大量的系统开销。SingleThreadModel在Servlet2.4中已不再提倡使用

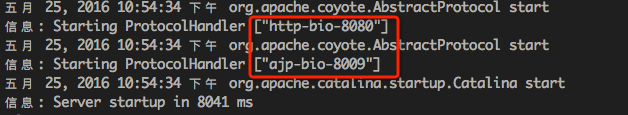
注意：服务器可以使用多个实例来处理请求，代替单个实例的请求排队带来的效益问题。服务器创建一个Servlet类的多个Servlet实例组成的实例池，对于每个请求分配Servlet实例进行响应处理，之后放回到实例池中等待下此请求。这样就造成并发访问的问题。   
此时,局部变量(字段)也是安全的，但对于全局变量和共享数据是不安全的，需要进行同步处理。而对于这样多实例的情况SingleThreadModel接口并不能解决并发访问问题。 SingleThreadModel接口在servlet规范中已经被废弃了。

# Tomcat

## 三种工作模式

1 BIO模式

阻塞式I/O操作，表示Tomcat使用的是传统**Java**I/O操作(即**Java**.io包及其子包)。Tomcat7及其以下版本默认情况下是以bio模式运行的，由于每个请求都要创建一个线程来处理，线程开销较大，不能处理高并发的场景，在三种模式中性能也最低。启动tomcat看到如下日志，表示使用的是BIO模式：



2 NIO模式

是**Java SE** 1.4及后续版本提供的一种新的I/O操作方式(即java.nio包及其子包)。是一个基于缓冲区、并能提供非阻塞I/O操作的Java API，它拥有比传统I/O操作(bio)更好的并发运行性能。要让Tomcat以nio模式来运行比较简单，只需要在Tomcat安装目录/conf/server.xml文件中将如下配置：

<Connector port="8080" protocol="HTTP/1.1"

connectionTimeout="20000"

redirectPort="8443" />

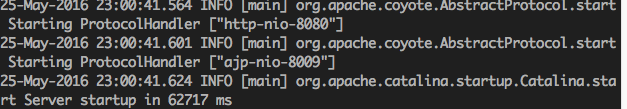
**修改为**

<Connector port="8080" protocol="org.apache.coyote.http11.Http11NioProtocol"

connectionTimeout="20000"

redirectPort="8443" />

注意：Tomcat8以上版本，默认使用的就是NIO模式，不需要额外修改



3 apr模式

从操作系统级别解决异步IO问题，大幅度的提高服务器的处理和响应性能，也是Tomcat运行高并发应用的首选模式。

APR(Apache Portable Runtime/Apache可移植运行时)，是Apache HTTP服务器的支持库。你可以简单地理解为:Tomcat将以JNI的形式调用 Apache HTTP服务器的核心动态链接库来处理文件读取或网络传输操作，从而大大地提高 Tomcat对静态文件的处理性能。

1 安装依赖库

因为**apr模式本质是使用JNI技术调用操作系统IO接口**，需要用到相关API的头文件

yum install apr-devel

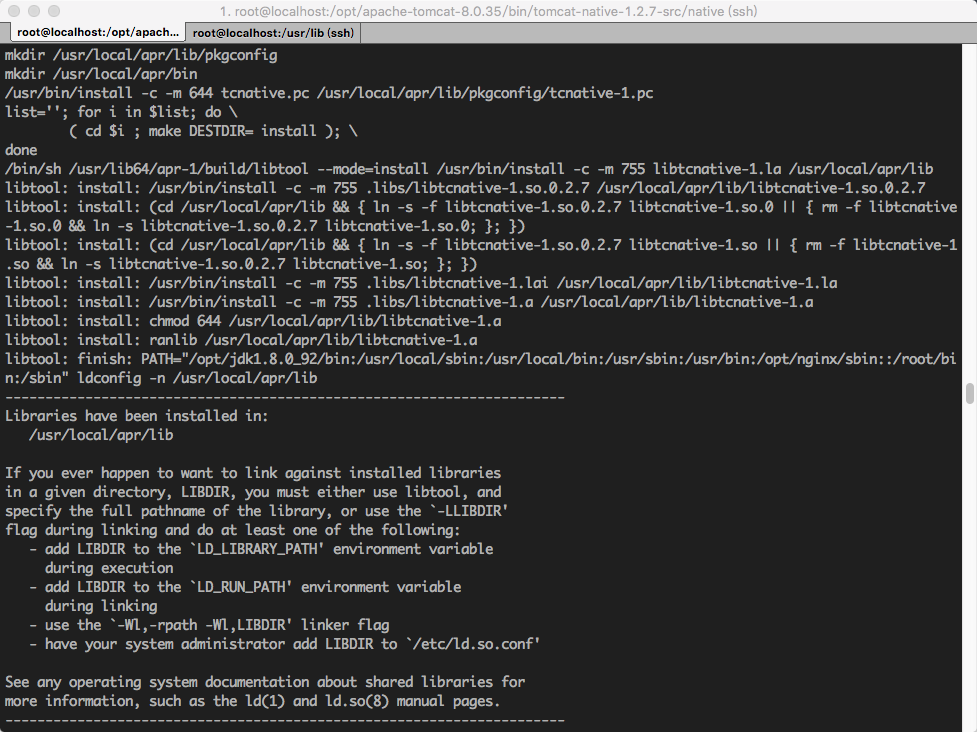
yum install openssl-devel

yum install gcc

yum install make

2 安装apr动态库

进入tomcat的bin目录，解压tomcat-native.tar.gz文件，并进入tomcat-native-1.2.7-src/native目录，执行./configure && make && make install 命令，动态库默认安装在/usr/local/apr/lib目录下，如下图所示：



3 配置APR本地库到系统共享库搜索路径中

方式1 设置LD\_LIBRARY\_PATH和LD\_RUN\_PATH环境变量，指向/usr/local/apr/lib目录，可配置到$HOME/.profile文件中

export LD\_LIBRARY\_PATH=$LD\_LIBRARY\_PATH:/usr/local/apr/lib

export LD\_RUN\_PATH=$LD\_RUN\_PATH:/usr/local/apr/lib

方式2 拷贝/usr/local/apr/lib目录下所有动态库到/usr/lib或/lib系统共享库搜索目录下即可。

copy /usr/local/apr/lib/libtcnative\* /usr/lib/

方式3（推荐）

编辑$TOMCAT\_HOME/bin/catalina.sh文件，在虚拟机启动参数JAVA\_OPTS中添加java.library.path参数，指定apr库的路径

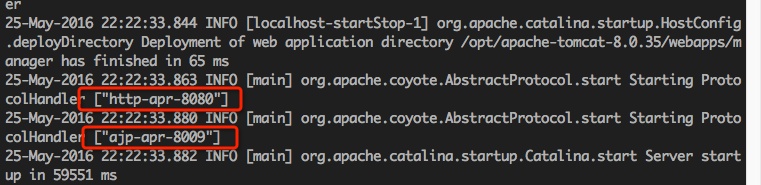
JAVA\_OPTS="$JAVA\_OPTS -Djava.library.path=/usr/local/apr/lib"

Tomcat8以下版本，需要指定运行模式，将protocol从HTTP/1.1改成org.apache.coyote.http11.Http11AprProtocol

<Connector port="8080" protocol="org.apache.coyote.http11.Http11AprProtocol"

connectionTimeout="20000"

redirectPort="8443" />



### APR详解

APR(Apache Portable Runtime)是一个高可移植库，它是Apache HTTP Server 2.x的核心。APR有很多用途，包括访问高级 IO功能(例如sendfile,epoll和OpenSSL)，OS级别功能(随机数生成，系统状态等等)，本地进程管理(共享内存，NT管道和UNIX sockets)。这些功能可以使Tomcat作为一个通常的前台WEB服务器，能更好地和其它本地web技术集成，总体上让Java更有效率作为一个高性能web服务器平台而不是简单作为后台容器。   
        APR的目的如其名称一样，主要为上层的应用程序提供一个可以跨越多操作系统平台使用的底层支持接口库。在早期的Apache版本中，应用程序本身必须能够处理各种具体操作系统平台的细节，并针对不同的平台调用不同的处理函数。随着Apache的进一步开发，Apache组织决定将这些通用的函数独立出来并发展成为一个新的项目。这样，APR的开发就从Apache中独立出来，Apache仅仅是使用APR而已。目前APR主要还是由Apache使用，不过由于APR的较好的移植性，因此一些需要进行移植的C程序也开始使用APR。   
        APR使得平台细节的处理进行下移。对于应用程序而言，它们根本就不需要考虑具体的平台，不管是Unix、Linux还是Window，应用程序执行的接口基本都是统一一致的。因此对于APR而言，可移植性和统一的上层接口是其考虑的一个重点。而APR最早的目的并不是如此，它最早只是希望将Apache中用到的所有代码合并为一个通用的代码库，然而这不是一个正确的策略，因此后来APR改变了其目标。有的时候使用公共代码并不是一件好事，比如如何将一个请求映射到线程或者进程是平台相关的，因此仅仅一个公共的代码库并不能完成这种区分。APR的目标则是希望安全合并所有的能够合并的代码而不需要牺牲性能。

## 优化

如果想要Tomcat达到最优的效果，首先要争取使得[**操作系统**](http://lib.csdn.net/base/operatingsystem)以及网络资源达到最优，并且最好使用高版本的JDK。对于有大量静态页面的系统，采用Apache集成Tomcat的方式，把静态页面交由Apache处理，动态部分交由Tomcat处理，能极大解放Tomcat的处理能力。使用ARP库也能极大的提高Tomcat对静态文件的处理能力。对于并发要求较高的系统，采用Apache加Tomcat集群的方式，将负载分别分担到多个Tomcat上，能很大的提高系统的性能，充分利用硬件资源。同时需要对Tomcat自身进行优化，包括增大内存、调节并发线程数等。

Windows 每个进程中线程数不超过2000

Linux 每个进程中线程数不超过1000

在java中每开启一个线程需要消耗1MB的JVM内存空间用于线程栈

Tomcat的最大并发数与硬件性能和CPU有关。Tomcat默认（7之前）的http是阻塞式的，每个请求都需要创建一个线程去处理。这种模式下并发量受到了线程数的限制。

### 提高tomcat并发能力的方法

1、Apache + Tomcat 结合起来用Apache 负责静态页面，Tomcat负责动态页面，同时减少connectionTimeout的时间，以应对并发量大线程回收来不及的情况。  
2、压力过大的问题，可以做负载均衡，一个TOMCAT无论如何也不可能担当如此多的线程负载，而且JVM过大，其内存管理成本将显著加大。2G的内存，做3-4个TOMCAT实例（512RAM\*4），更为科学合理。  
3、数据库连接池，不少人，都推荐使用C3P0，能提高访问数据库的并发性能好几倍。（有博文称使用tomcat自带的jdbc-pool更好，还没试过）  
4、采用Tomcat集群可以最大程度的发挥服务器的性能，可以在配置较高的服务器上部署多个Tomcat，也可以在多台服务器上分别部署 Tomcat，Apache和Tomcat整合的方式还是JK方式。经过验证，系统对大用户量使用的响应方面，Apache+3Tomccat集群> Apache+2Tomcat集群> Apache集成Tomcat >单个Tomcat。并且采用Apache+多Tomcat集群的部署方式时，如果一个Tomcat出现宕机，系统可以继续使用，所以在硬件系统性能足够优越的情况下，需要尽量发挥软件的性能，可以采用增加Tomcat集群的方式。  
5. 打开KeepAlive支持  
KeepAlive on, KeepAliveTimeout 15 MaxKeepAliveRequests 1000  
根据实际经验，通过Apache和Tomcat集群的方式提高系统性能的效果十分明显，这种方式可以最大化的利用硬件资源，通过多个Tomcat的处理来分担单Tomcat时的压力。  
web server允许的最大连接数还受制于操作系统的内核参数设置，通常Windows是2000个左右，Linux是1000个左右。

## Tomcat优化

<http://itindex.net/detail/53789-tomcats>

http://itindex.net/detail/53998-tomcat-%E4%BC%98%E5%8C%96

### 内存优化

内存优化主要是针对tomcat启动参数优化。

在 tomcat 的启动脚本 catalina.sh 中设置 java\_OPTS 参数。   
　　JAVA\_OPTS参数说明   
　　-server 启用jdk 的 server 版；   
　　-Xms java虚拟机初始化时的最小堆内存；   
　　-Xmx java虚拟机可使用的最大堆内存；   
　　-XX: PermSize 内存永久保留区域   
　　-XX:MaxPermSize 内存最大永久保留区域

Tomcat默认使用的内存为128M

linux修改TOMCAT\_HOME/bin/catalina.sh，在前面加入

JAVA\_OPTS="-XX:PermSize=64M -XX:MaxPermSize=128m -Xms512m -Xmx1024m

-Duser.timezone=Asia/Shanghai"

windows修改TOMCAT\_HOME/bin/catalina.bat，在前面加入

set JAVA\_OPTS=-XX:PermSize=64M -XX:MaxPermSize=128m -Xms512m -Xmx1024m

初始化堆内存为512M，可使用的最大内存1024M

另外，虚拟机堆的大小决定了虚拟机花费在收集垃圾上的时间和频率。如果堆的空间很大，那么一次full gc所消耗的时间会很长，但是频率比较低。调整堆的大小目的是最小化垃圾收集的时间，以在特定时间内最大化处理客户请求。将最小堆内存和最大堆内存设置为同一值，这样可以避免浪费用于调整堆大小所需的vm资源。

### 并发优化

**1 tomcat连接相关参数**

server.xml中：

<Connector port="9027"

　　protocol="HTTP/1.1"

　　maxHttpHeaderSize="8192"

　　minProcessors="100"

　　maxProcessors="1000"

　　acceptCount="1000"

　　redirectPort="8443"

　　disableUploadTimeout="true"/>

**2 调整连接器connector的并发处理能力**

参数说明

maxThreads tomcat可以创建的最大线程数（也是可以处理的同时请求的最大数目，并发数） tomcat使用线程来处理接收的每个请求 maxThreads并不是配置越大越好，这个最大值是受操作系统及相关硬件制约的，并且最大值不一定是最优值。

 QPS（Query Per Second）：每秒查询率QPS是对一个特定的查询服务器在规定时间内所处理流量多少的衡量标准。我们常常使用 QPS值来衡量一个服务器的性能。   
        QPS = 并发数 / 平均响应时间   
        或者   
        并发数 = QPS \* 平均响应时间

一个系统吞吐量通常由QPS、并发数两个因素决定，每套系统的这两个值都有一个相对极限值，在应用场景访问压力下，只要某一项达到系统最高值，系统的吞吐量就上不去了，如果压力继续增大，系统的吞吐量反而会下降，原因是系统超负荷工作，上下文切换、内存等等其它消耗导致系统性能下降。所谓吞吐量这里可以理解为每秒能处理请求的次数。   
        所以选择一个合理的 maxThreads值，其实并不是那么容易的事。因为过多的线程只会造成，更多的内存开销，更多的CPU开销，但是对提升QPS确毫无帮助；找到最佳线程数后通过简单的设置，可以让web系统更加稳定，得到最高，最稳定的QPS输出。

我们可以通过以下几种方式来获取 maxThreads的最佳值：   
        （1）通过线上系统不断使用和用户的不断增长来进行性能测试，观察QPS，响应时间，这种方式会在爆发式增长时系统崩溃，如双12等。   
        （2）根据公式计算，服务器端最佳线程数量=((线程等待时间+线程cpu时间)/线程cpu时间) \* cpu数量，这种方式有时会被误导，因为某些系统处理环节可能会耗时比较长，从而影响公式的结果。   
        （3）单、多用户压力测试，查看CPU的消耗，然后直接乘以百分比，再进行压测，一般这个值的附近应该就是最佳线程数量，这种方式理想场景比较适用，实际情况会比这个复杂的多。   
        （4）根据系统的自身情况调整，如硬件限制，系统限制，程序处理能力限制等。   
        （5）定期修改为不同的 maxThreads值，看服务器响应结果及用户反应。

 QPS和线程数的关系   
        （1）在最佳线程数量之前，QPS和线程是互相递增的关系，线程数量到了最佳线程之后，QPS持平，不在上升，甚至略有下降，同时相应时间持续上升。   
        （2）同一个系统而言，支持的线程数越多（最佳线程数越多而不是配置的线程数越多），QPS越高。

QPS和响应时间的关系   
        （1）对于一般的web系统，响应时间一般有CPU执行时间+IO等待时间组成。   
        （2）CPU的执行时间减少，对QPS有实质的提升，IO时间的减少，对QPS提升不明显。如果要想明显提升QPS，优化系统的时候要着重优化CPU消耗大户。   
        所以想要找出 maxThreads的最优值可并不容易，没有最好只有更好，更好的值只能通过时间来显现，如果你不想考虑那么多，一般情况下设置成1000即可。   
　　minSpareThreads Tomcat初始化时创建的 socket 线程数 。也是线程最小运行数目。这些线程始终保持运行。即使没有人使用，也开这么多空线程等待。  
　　maxSpareThreads Tomcat连接器的最大空闲 socket 线程数 （一旦创建的线程数超过这个值，tomcat就会关闭不再需要的socket线程）  
　　enableLookups 若设为true, 则支持域名解析，可把 ip 地址解析为主机名 设为false  
　　redirectPort 在需要基于安全通道的场合，把客户请求转发到基于SSL 的

redirectPort 端口   
　　acceptAccount 监听端口队列最大数，满了之后客户请求会被拒绝（当线程数达到maxThreads后，后续的请求会被放入到一个等待队列中，这个acceptCount是这个队列的大小，如果这个队列也满了，直接拒绝连接）

　　connectionTimeout 连接超时   
　　minProcessors 服务器创建时的最小处理线程数   
　　maxProcessors 服务器同时最大处理线程数   
　　URIEncoding URL统一编码

配置示例

<Connector port="9027"

　　protocol="HTTP/1.1"

　　maxHttpHeaderSize="8192"

　　maxThreads="1000"

　　minSpareThreads="100"

　　maxSpareThreads="1000"

　　minProcessors="100"

　　maxProcessors="1000"

　　enableLookups="false"

　　URIEncoding="utf-8"

　　acceptCount="1000"

　　redirectPort="8443"

　　disableUploadTimeout="true"/>

**3 tomcat缓存优化**

参数说明

compression 打开压缩功能   
　　compressionMinSize 启用压缩的输出内容大小，这里面默认为2KB   
　　compressableMimeType 压缩类型   
　　connectionTimeout 定义建立客户连接超时的时间. 如果为 -1, 表示不限制建立客户连接的时间

配置示例

<Connector port="9027"

　　protocol="HTTP/1.1"

　　maxHttpHeaderSize="8192"

　　maxThreads="1000"

　　minSpareThreads="100"

　　maxSpareThreads="1000"

　　minProcessors="100"

　　maxProcessors="1000"

　　enableLookups="false"

　　compression="on"

　　compressionMinSize="2048"

　　compressableMimeType="text/html,text/xml,text/javascript,text/css,text/plain"

　　connectionTimeout="20000"

　　URIEncoding="utf-8"

　　acceptCount="1000"

　　redirectPort="8443"

　　disableUploadTimeout="true"/>

HTTP 压缩可以大大提高浏览网站的速度，它的原理是，在客户端请求网页后，从服务器端将网页文件压缩，再下载到客户端，由客户端的浏览器负责解压缩并浏览。相对于普通的浏览过程HTML,CSS,Javascript , Text ，它可以节省40%左右的流量。更为重要的是，它可以对动态生成的，包括CGI、PHP , JSP , ASP , Servlet,SHTML等输出的网页也能进行压缩，压缩效率惊人。

### 线程池

Executor代表了一个线程池，可以在Tomcat组件之间共享。使用线程池的好处在于减少了创建销毁线程的相关消耗，而且可以提高线程的使用效率。   
        要想使用线程池，首先需要在 Service标签中配置 Executor，如下：

<Executor name="tomcatThreadPool"     
         namePrefix="catalina-exec-"     
         maxThreads="1000"     
         minSpareThreads="100"    
         maxIdleTime="60000"    
         maxQueueSize="Integer.MAX\_VALUE"    
         prestartminSpareThreads="false"    
         threadPriority="5"    
         className="org.apache.catalina.core.StandardThreadExecutor"/>    
  ....

name：线程池名称，用于 Connector中指定。   
        namePrefix：所创建的每个线程的名称前缀，一个单独的线程名称为 namePrefix+threadNumber。   
        maxThreads：池中最大线程数。   
        minSpareThreads：活跃线程数，也就是核心池线程数，这些线程不会被销毁，会一直存在。   
        maxIdleTime：线程空闲时间，超过该时间后，空闲线程会被销毁，默认值为6000（1分钟），单位毫秒。   
        maxQueueSize：在被执行前最大线程排队数目，默认为Int的最大值，也就是广义的无限。除非特殊情况，这个值不需要更改，否则会有请求不会被处理的情况发生。   
        prestartminSpareThreads：启动线程池时是否启动 minSpareThreads部分线程。默认值为false，即不启动。   
        threadPriority：线程池中线程优先级，默认值为5，值从1到10。   
        className：线程池实现类，未指定情况下，默认实现类为org.apache.catalina.core.StandardThreadExecutor。如果想使用自定义线程池首先需要实现 org.apache.catalina.Executor接口。   
        线程池配置完成后需要在 Connector中指定：

<Connector executor="tomcatThreadPool"    
...

### Listener

另一个影响Tomcat 性能的因素是内存泄露。Server标签中可以配置多个Listener，其中 JreMemoryLeakPreventionListener是用来预防JRE内存泄漏。此Listener只需在Server标签中配置即可，默认情况下无需配置，已经添加在 Server中。

<Listener className="org.apache.catalina.core.JreMemoryLeakPreventionListener" />

### 禁用DNS查询

当web应用程序向要记录客户端的信息时，它也会记录客户端的IP地址或者通过域名服务器查找机器名 转换为IP地址。

DNS查询需要占用网络，并且包括可能从很多很远的服务器或者不起作用的服务器上去获取对应的IP的过程，这样会消耗一定的时间。

修改server.xml文件中的Connector元素，修改属性enableLookups参数值: enableLookups="false"

如果为true，则可以通过调用request.getRemoteHost()进行DNS查询来得到远程客户端的实际主机名，若为false则不进行DNS查询，而是返回其ip地址

### 设置session过期时间

conf\web.xml中通过参数指定：

<session-config>

<session-timeout>180</session-timeout>

</session-config>

单位为分钟。

### Apr插件提高tomcat性能

Tomcat可以使用APR来提供超强的可伸缩性和性能，更好地集成本地服务器技术.

  APR(Apache Portable Runtime)是一个高可移植库，它是Apache HTTP Server 2.x的核心。APR有很多用途，包括访问高级IO功能(例如sendfile,epoll和OpenSSL)，OS级别功能(随机数生成，系统状态等 等)，本地进程管理(共享内存，NT管道和UNIX sockets)。这些功能可以使Tomcat作为一个通常的前台WEB服务器，能更好地和其它本地web技术集成，总体上让Java更有效率作为一个高 性能web服务器平台而不是简单作为后台容器。

  在产品环境中，特别是直接使用Tomcat做WEB服务器的时候，应该使用Tomcat Native来提高其性能

  要测APR给tomcat带来的好处最好的方法是在慢速网络上（模拟Internet），将Tomcat线程数开到300以上的水平，然后模拟一大堆并发请求。   
  如果不配APR，基本上300个线程狠快就会用满，以后的请求就只好等待。但是配上APR之后，并发的线程数量明显下降，从原来的300可能会马上下降到只有几十，新的请求会毫无阻塞的进来。   
  在 局域网环境测，就算是400个并发，也是一瞬间就处理/传输完毕，但是在真实的Internet环境下，页面处理时间只占0.1%都不到，绝大部分时间都 用来页面传输。如果不用APR，一个线程同一时间只能处理一个用户，势必会造成阻塞。所以生产环境下用apr是非常必要的。

## Tomcat与apache的连接方式

1、JK方式

这是最常见的方式。JK 是通过 AJP 协议与 Tomcat 服务器进行通讯的，Tomcat 默认的 AJP Connector 的端口是 8009。JK 本身提供了一个监控以及管理的页面 jkstatus，通过 jkstatus 可以监控 JK 目前的工作状态以及对到 tomcat 的连接进行设置

2、http\_proxy 方式

这是利用 Apache 自带的 mod\_proxy 模块使用代理技术来连接 Tomcat。在配置之前请确保是否使用的是 2.2.x 版本的 Apache 服务器。因为 2.2.x 版本对这个模块进行了重写，大大的增强了其功能和稳定性。

http\_proxy 模式是基于 HTTP 协议的代理，因此它要求 Tomcat 必须提供 HTTP 服务，也就是说必须启用 Tomcat 的 HTTP Connector。  
3、ajp\_proxy 方式

ajp\_proxy 连接方式其实跟 http\_proxy 方式一样，都是由 mod\_proxy 所提供的功能。配置也是一样，只需要把 http:// 换成 ajp:// ，同时连接的是 Tomcat 的 AJP Connector 所在的端口。

三者比较

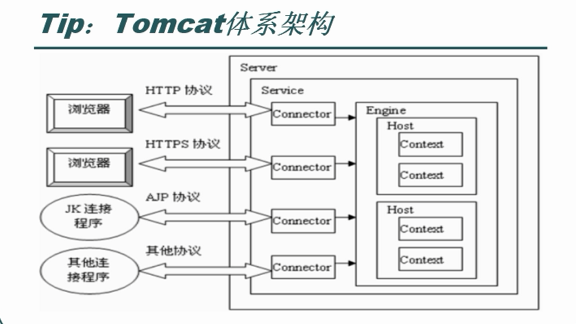
相对于 JK 的连接方式，后两种在配置上是比较简单的，灵活性方面也一点都不逊色。但就稳定性而言就不像 JK 这样久经考验，毕竟 Apache 2.2.3 推出的时间并不长，采用这种连接方式的网站还不多，因此，如果是应用于关键的互联网网站，还是建议采用 JK 的连接方式。  
配置比 JK 简单多了，而且它也可以通过一个页面来监控集群运行的状态，并做一些简单的维护设置。

**tomcat与apache为什么要整合起来，有什么意义？**

对于静态页面的数据，Apache的处理速度比Tomcat要快很多。  
如果客户端请求的是静态页面，则只需要Apache服务器响应请求，  
如果客户端请求动态页面，则是Tomcat服务器响应请求，  
因为jsp是服务器端解释代码的，这样整合就可以减少Tomcat的服务开销，提升对静态文件的处理性能，  
可以利用 Web 服务器来做负载均衡以及容错，无缝的升级应用程序。

## Tomcat完整的http请求

<http://www.cnblogs.com/hggen/p/6264475.html>



1) 请求被发送到本机端口8080，被在那里侦听的Coyote HTTP/1.1 Connector获得   
2) Connector把该请求交给它所在的Service的Engine来处理，并等待来自Engine的回应   
3) Engine获得请求localhost/yy/index.jsp，匹配它所拥有的所有虚拟主机Host   
4) Engine匹配到名为localhost的Host（即使匹配不到也把请求交给该Host处理，因为该Host被定义为该Engine的默认主机）   
5) localhost Host获得请求/yy/index.jsp，匹配它所拥有的所有Context   
6) Host匹配到路径为/yy的Context（如果匹配不到就把该请求交给路径名为”“的Context去处理）   
7) path=”/yy”的Context获得请求/index.jsp，在它的mapping table中寻找对应的servlet   
8) Context匹配到URL PATTERN为\*.jsp的servlet，对应于JspServlet类   
9) 构造HttpServletRequest对象和HttpServletResponse对象，作为参数调用JspServlet的doGet或doPost方法   
10)Context把执行完了之后的HttpServletResponse对象返回给Host   
11)Host把HttpServletResponse对象返回给Engine   
12)Engine把HttpServletResponse对象返回给Connector   
13)Connector把HttpServletResponse对象返回给客户browser

# Spring

## 注解的实现机制？

<http://www.cnblogs.com/Johness/archive/2013/04/17/3026689.html>

Spring的配置文件为applicationContext.xml文件，在项目启动的时候加载这个配置文件。

Spring需要在配置文件中开启注解

<!-- 开启自动扫描 -->

<context:annotation-config/>

<context:component-scan base-package="com.itxxz" use-default-filters="false">

<context:include-filter type="annotation" expression="org.springframework.stereotype.Service" />

<context:include-filter type="annotation" expression="org.springframework.stereotype.Repository" />

<context:include-filter type="annotation" expression="org.springframework.stereotype.Component" />

</context:component-scan>

需要制定要扫描的包路径

## 什么是spring

Spring的核心是一个轻量级（Lightweight）的容器（Container），它是实现IoC（Inversion of Control）容器和非入侵性（No intrusive）的框架，并提供AOP（Aspect-oriented programming）概念的实现方式；提供对持久层（Persistence）、事物（Transcation）的支持；提供MVC Web框架的实现，并对一些常用的企业服务API（Application Interface）提供一致的模型封装，是一个全方位的应用程序框架（Application Framework），除此之外，对现存的各种框架（Structs、JSF、Hibernate、Ibatis、Webwork等），Spring也提供了与他们相整合的方案。

**轻量级**：sring框架在系统初始化时不用加载所有的服务，只需要加载我们使用的核心服务，这就是轻量级。

**重量级**：对于EJB来说，在系统初始化时需要加载所有的服务。

**非入侵**：允许应用系统自由选择和组装spring框架的各个功能模块，并不强制要求应用系统的类必须继承框架提供的类或者实现接口。

**入侵**：强制必须去继承框架提供的类，这样以后如果想更换框架，那么之前写的代码几乎无法重用。（strust1是入侵的）

**如何实现非入侵？**

1 应用反射机制，通过动态调用的方式来提供各方面的功能，建立核心组件BeanFactory

2 配合使用Spring框架中的BeanWrapper和BeanFactory组件类最终达到对象的实例创建和属性注入

**优点**：允许所开发出来的应用系统能够在不用的环境中自由移植，不需要修改应用系统中的核心功能实现的代码

## Spring框架中的主要模块



## 使用spring框架带来的好处

Spring是一个轻量级框架，可以自己选择里面的组件模块。

Spring通过控制反转（IOC）实现了松散耦合，对各个层之间进行了解耦。

Spring支持了面向编程（AOP），将业务逻辑和系统服务分开。

Spring中的容器存储并管理了应用中对象的生命周期和配置。

Spring提供的web框架springmvc。

Spring提供了一个持续的事物管理接口，可以扩展到上至本地事务下至全局事务。

Spring提供了单元测试功能，可以很方便的使用依赖注入来写入测试数据。

## 什么是控制反转和依赖注入

控制反转：以前的做法是由程序代码去创建需要使用的对象，现在由容器进行创建和管理对象。

依赖注入：对象的属性关系由spring管理。

依赖注入是控制反转的基础。

依赖注入和控制反转是对同一件事情的不同描述，从某个方面讲，就是它们描述的角度不同。依赖注入是从应用程序的角度在描述，可以把依赖注入描述完整点：应用程序依赖容器创建并注入它所需要的外部资源；而控制反转是从容器的角度在描述，描述完整点：容器控制应用程序，由容器反向的向应用程序注入应用程序所需要的外部资源。

依赖注入的四种实现方式：

1 通过setter方法注入

2 构造器注入

3 接口注入：要求组件必须与特定的接口相关联，具有入侵性。

## IOC

Spring中的 org.springframework.beans 包和 org.springframework.context包构成了Spring框架IoC容器的基础。

BeanFactory 接口提供了一个先进的配置机制，使得任何类型的对象的配置成为可能。ApplicationContex接口对BeanFactory（是一个子接口）进行了扩展，在BeanFactory的基础上添加了其他功能，比如与Spring的AOP更容易集成，也提供了处理message resource的机制（用于国际化）、事件传播以及应用层的特别配置，比如针对Web应用的WebApplicationContext。

**org.springframework.beans.factory.BeanFactory** 是Spring IoC容器的具体实现，用来包装和管理前面提到的各种bean。BeanFactory接口是Spring IoC 容器的核心接口。

## BeanFactory和ApplicationContext的区别

**BeanFactory：**是Spring里面最低层的接口，提供了最简单的容器的功能，只提供了实例化对象和拿对象的功能；

**ApplicationContext：**应用上下文，继承BeanFactory接口，它是Spring的一种更高级的容器，提供了更多的有用的功能；

1) 提供了支持国际化的文本消息

BeanFactory是不支持国际化功能的，因为BeanFactory没有扩展Spring中MessageResource接口。

2) 强大的事件机制。

基本上牵涉到事件(Event)方面的设计，就离不开观察者模式。  
ApplicationContext的事件机制主要通过ApplicationEvent和ApplicationListener这两个接口来提供的，和java swing中的事件机制一样。即当ApplicationContext中发布一个事件的时，所有扩展了ApplicationListener的Bean都将会接受到这个事件，并进行相应的处理。   
  
Spring提供了部分内置事件，主要有以下几种：    
ContextRefreshedEvent ：ApplicationContext发送该事件时，表示该容器中所有的Bean都已经被装载完成，此ApplicationContext已就绪可用   
ContextStartedEvent：生命周期 beans的启动信号    
ContextStoppedEvent: 生命周期 beans的停止信号    
ContextClosedEvent：ApplicationContext关闭事件，则context不能刷新和重启，从而所有的singleton bean全部销毁(因为singleton bean是存在容器缓存中的)   
  
  虽然，spring提供了许多内置事件，但用户也可根据自己需要来扩展spriong中的事物。注意，要扩展的事件都要实现ApplicationEvent接口。

3) 对资源的访问，如URL和文件（ResourceLoader）

http://snkcxy.iteye.com/blog/1913796

 ApplicationContext扩展了ResourceLoader(资源加载器)接口，从而可以用来加载多个Resource，而BeanFactory是没有扩展ResourceLoader

**ClassPathXmlApplicationContext**：从classpath的XML配置文件中读取上下文，并生成上下文定义。应用程序上下文从程序环境变量中取得。

|  |  |
| --- | --- |
|  | ApplicationContext context = new ClassPathXmlApplicationContext(“bean.xml”); |

**FileSystemXmlApplicationContext** ：由文件系统中的XML配置文件读取上下文。

|  |  |
| --- | --- |
|  | ApplicationContext context = new FileSystemXmlApplicationContext(“bean.xml”); |

**XmlWebApplicationContext**：由Web应用的XML文件读取上下文。

3) 载入多个（有继承关系）上下文 ，使得每一个上下文都专注于一个特定的层次，比如应用的web层

4) 消息发送、响应机制（ApplicationEventPublisher）

5) AOP（拦截器）

一般拦截器都是实现HandlerInterceptor接口，其中有三个方法**preHandle、postHandle、afterCompletion**

1. preHandle：执行controller之前执行

2. postHandle：执行完controller，return modelAndView之前执行，主要操作modelAndView的值

3. afterCompletion：controller返回后执行

装载bean的区别：

BeanFactory在启动的时候不会去实例化Bean，只有从容器中拿Bean的时候才会去实例化；

ApplicationContext在启动的时候就把所有的Bean全部实例化了。它还可以为Bean配置lazy-init=true来让Bean延迟实例化；好处是节省了在使用时才去对bean实例化的那部分时间，而且可以在启动时发现spring配置有没有错误。

## 基于XML的配置

Spring的XML配置方式是使用被Spring命名空间的所支持的一系列的XML标签来实现的。Spring有以下主要的命名空间：context、beans、jdbc、tx、aop、mvc和aso。

在web.xml中配置DispatcherServlet

## 基于java的配置

Spring对Java配置的支持是由@Configuration注解和@Bean注解来实现的。由@Bean注解的方法将会实例化、配置和初始化一个新对象，这个对象将由Spring的IoC容器来管理。@Bean声明所起到的作用与<bean/> 元素类似。被@Configuration所注解的类则表示这个类的主要目的是作为bean定义的资源。被@Configuration声明的类可以通过在同一个类的内部调用@bean方法来设置嵌入bean的依赖关系。



在上面的例子中，com.acme包首先会被扫到，然后再容器内查找被@Component 声明的类，找到后将这些类按照Sring bean定义进行注册。

如果你要在你的web应用开发中选用上述的配置的方式的话，需要用AnnotationConfigWebApplicationContext 类来读取配置文件，可以用来配置Spring的Servlet监听器ContrextLoaderListener或者Spring MVC的DispatcherServlet。



## 基于注解的配置



## BeanWrapper

<http://www.cnblogs.com/mesopotamia/p/5002122.html>

在org.springframework.beans包中，还有两个非常重要的类：BeanWrapper接口及它的实现BeanWrapperImpl。BeanWrapper封装了一个Bean的行为，提供了设置和获得属性值的功能。通过BeanWrapper可以获得Bean的属性和描述、查询只读或者可写属性。

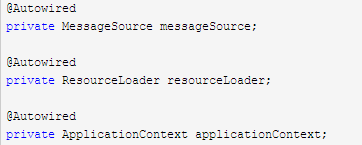
## Spring中常用的接口和类

### ApplicationContextAware接口

     当一个类需要获取ApplicationContext实例时，可以让该类实现ApplicationContextAware接口。



通过@Autowired注解可以自动装配一些常用对象实例：



### ApplicationEvent抽象类

当需要创建自定义事件时，可以新建一个继承自ApplicationEvent抽象类的类。

### ApplicationListener接口

当需要监听自定义事件时，可以新建一个实现ApplicationListener接口的类，并将该类配置到Spring容器中。重写onApplicationEvent(ApplicationEvent event)方法。

### BeanNameAware接口

     当bean需要获取自身在容器中的id/name时，可以实现BeanNameAware接口。

实现BeanNameAware接口，接口中就一个方法setBeanName()

### BeanFactoryAware

让Bean获取配置他们的BeanFactory的引用。

实现BeanFactoryAware接口，其中只有一个方法即setFactory(BeanFactory factory)。使用上与BeanNameAware接口无异，只不过BeanFactoryAware注入的是个工厂，BeanNameAware注入的是个Bean的名字。

### InitializingBean接口

      当需要在bean的全部属性设置成功后做些特殊的处理，可以让该bean实现InitializingBean接口。效果等同于bean的init-method属性的使用或者@PostContsuct注解的使用。三种方式的执行顺序：先注解，然后执行InitializingBean接口中定义的方法，最后执行init-method属性指定的方法。

### DisposableBean接口

当需要在bean销毁之前做些特殊的处理，可以让该bean实现DisposableBean接口。效果等同于bean的destroy-method属性的使用或者@PreDestory注解的使用。三种方式的执行顺序：先注解，然后执行DisposableBean接口中定义的方法，最后执行destroy-method属性指定的方法

### BeanPostProcessor接口

当需要对受管bean进行预处理时，可以新建一个实现BeanPostProcessor接口的类，并将该类配置到Spring容器中。

实现BeanPostProcessor接口时，需要实现以下两个方法“

postProcessBeforeInitialization在受管bean的初始化动作之前调用

postProcessAfterInitialization在受管bean的初始化动作之后调用

容器中的每个Bean在创建时都会恰当地调用它们。

### BeanFactoryPostProcessor接口

当需要对Bean工厂进行预处理时，可以新建一个实现BeanFactoryPostProcessor接口的类，并将该类配置到Spring容器中。

## Spring bean的生命周期

<http://www.cnblogs.com/zrtqsk/p/3735273.html>

<http://blog.csdn.net/qq_33642117/article/details/51924653>

Spring bean factory 负责管理在spring容器中被创建的bean的生命周期。Bean的生命周期由两组回调（call back）方法组成。

1. 初始化之后调用的回调方法。
2. 销毁之前调用的回调方法。

Spring框架提供了以下四种方式来管理bean的生命周期事件：

* InitializingBean和DisposableBean回调接口
* 针对特殊行为的其他Aware接口
* Bean配置文件中的Custom init()方法和destroy()方法
* @PostConstruct和@PreDestroy注解方式

简单的说就是：

对于单例，spring在启动时会去读取xml文件，创建实例对象。在创建对象的时候会先调用构造器，然后调用init-method属性值中的方法。在对象被销毁的时候，会调用destroy-method属性值中所指定的方法。Lazy-init=true可以让这个对象在第一次被访问时被创建。

对于非单例，spring在读取xml文件时，不会去创建对象，在每一次访问这个对象时，spring容器才会去创建并调用init-method属性值中的方法，在对象销毁时，spring不会帮我们调用任何方法，因为非单例，spring在创建完毕之后就不会管理这个对象了。

## Spring bean的作用域

1. singleton：这种bean范围是默认的，这种范围确保不管接受到多少个请求，每个容器中只有一个bean的实例，单例的模式由bean factory自身来维护。
2. prototype：原形范围与单例范围相反，为每一个bean请求提供一个实例。
3. request：在请求bean范围内会每一个来自客户端的网络请求创建一个实例，在请求完成以后，bean会失效并被垃圾回收器回收。
4. Session：与请求范围类似，确保每个session中有一个bean的实例，在session过期后，bean会随之失效。
5. global-session：global-session和Portlet应用相关。当你的应用部署在Portlet容器中工作时，它包含很多portlet。如果你想要声明让所有的portlet共用全局的存储变量的话，那么这全局变量需要存储在global-session中。

全局作用域与Servlet中的session作用域效果相同

## Spring内部bean

就是在一个bean的内部再装配一个bean。那么此时内部bean是不能被其他bean使用的。可以通过setter注入属性和构造方法注入构造参数的方式实现。



## Spring中单例bean是线程安全的么

Spring框架并没有对单例bean进行任何多线程的封装处理。关于单例bean的线程安全和并发问题需要开发者自行去搞定。但实际上，大部分的Spring bean并没有可变的状态(比如Serview类和DAO类)，所以在某种程度上说Spring的单例bean是线程安全的。如果你的bean有多种状态的话（比如 View Model 对象），就需要自行保证线程安全。

最浅显的解决办法就是将多态bean的作用域由**“singleton**”变更为“**prototype**”。

## Spring中注入集合

Spring提供了以下四种集合类的配置元素：

* **<list>** :   该标签用来装配可重复的list值。
* **<set>** :    该标签用来装配没有重复的set值。
* **<map>**:   该标签可用来注入键和值可以为任何类型的键值对。
* **<props>** : 该标签支持注入键和值都是字符串类型的键值对。

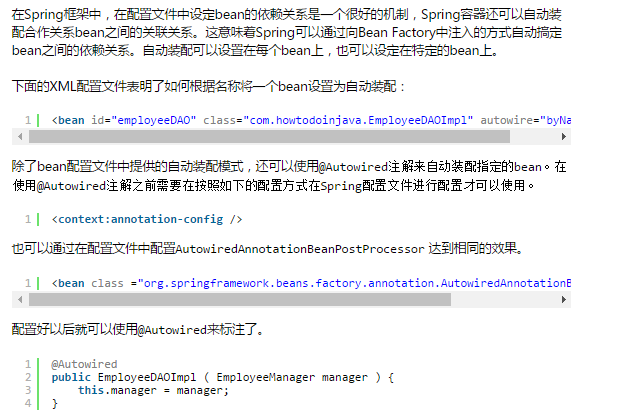


## Spring中注入一个java.util.Properties



## Spring的自动装配及分类

装配:spring容器把bean组装到一起,前提是容器需要知道bean的依赖关系,通过依赖注入将他们装配到一起.



少写几个ref

1. **no：**这是Spring框架的默认设置，在该设置下自动装配是关闭的，开发者需要自行在bean定义中用标签明确的设置依赖关系。
2. **byName：**该选项可以根据bean名称设置依赖关系。当向一个bean中自动装配一个属性时，容器将根据bean的名称自动在在配置文件中查询一个匹配的bean。如果找到的话，就装配这个属性，如果没找到的话就报错。
3. **byType：**该选项可以根据bean类型设置依赖关系。当向一个bean中自动装配一个属性时，容器将根据bean的类型自动在在配置文件中查询一个匹配的bean。如果找到的话，就装配这个属性，如果没找到的话就报错。
4. **constructor：**造器的自动装配和byType模式类似，但是仅仅适用于与有构造器相同参数的bean，如果在容器中没有找到与构造器参数类型一致的bean，那么将会抛出异常。
5. **autodetect：**该模式自动探测使用构造器自动装配或者byType自动装配。首先，首先会尝试找合适的带参数的构造器，如果找到的话就是用构造器自动装配，如果在bean内部没有找到相应的构造器或者是无参构造器，容器就会自动选择byTpe的自动装配方式。

## 如何开启基于注解的自动装配

要使用 @Autowired，需要注册 AutowiredAnnotationBeanPostProcessor，可以有以下两种方式来实现：

1 引入配置文件中的<bean>下引入 <context:annotation-config>

<beans>

    <context:annotation-config />

</beans>

2 在bean配置文件中直接引入AutowiredAnnotationBeanPostProcessor

<beans>

    <bean class="org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor"/>

</beans>

## @Required注解

在产品级别的应用中，IoC容器可能声明了数十万了bean，bean与bean之间有着复杂的依赖关系。设值注解方法的短板之一就是验证所有的属性是否被注解是一项十分困难的操作。可以通过在<bean>中设置“**dependency-check**”来解决这个问题。

在应用程序的生命周期中，你可能不大愿意花时间在验证所有bean的属性是否按照上下文文件正确配置。或者你宁可验证某个bean的特定属性是否被正确的设置。即使是用“**dependency-check**”属性也不能很好的解决这个问题，在这种情况下，你需要使用@Required 注解。

@ Required 注解应用于bean属性的setter方法上，只会检查属性是否已经设置而不会测试属性是否为空。



检查bean的属性是否被严格的设置。dependency-check”（依赖检查）属性将会非常有用。它默认为“none”，不进行依赖检查。“simple”会核对所有的原始类型和String类型的属性。“objects”只做对象间的关联检查（包括集合）。“all”会检查所有的属性，包括“simple”和“objects”。

## @Autowired注解

@Autowired注解可以像@Required注解、构造器一样被用于在bean的设值方法上自动装配bean的属性，一个参数或者带有任意名称或带有多个参数的方法。

比如，可以在设值方法上使用@Autowired注解来替代配置文件中的 <property>元素。当Spring容器在setter方法上找到@Autowired注解时，会尝试用**byType** 自动装配。如果有多个，则会再按照byName方式比对，如果还有多个，则报异常。

当然我们也可以在构造方法上使用@Autowired 注解。带有@Autowired 注解的构造方法意味着在创建一个bean时将会被自动装配，即便在配置文件中使用<constructor-arg> 元素。

@Autowired也可以手动指定按照byName方式注入，使用@Qualifier标签，例如：

@Autowired ()

@Qualifier ( "beanName" )

@Resource标签是按照byName方式注入的

## @Resource注解

@Resource的作用相当于@Autowired，只不过@Autowired按byType自动注入，而@Resource默认按 byName自动注入罢了。@Resource有两个属性是比较重要的，分是name和type，Spring将@Resource注解的name属性解析为bean的名字，而type属性则解析为bean的类型。所以如果使用name属性，则使用byName的自动注入策略，而使用type属性时则使用byType自动注入策略。如果既不指定name也不指定type属性，这时将通过反射机制使用byName自动注入策略。

@Resource装配顺序   
　　1. 如果同时指定了name和type，则从Spring上下文中找到唯一匹配的bean进行装配，找不到则抛出异常   
　　2. 如果指定了name，则从上下文中查找名称（id）匹配的bean进行装配，找不到则抛出异常   
　　3. 如果指定了type，则从上下文中找到类型匹配的唯一bean进行装配，找不到或者找到多个，都会抛出异常   
　　4. 如果既没有指定name，又没有指定type，则自动按照byName方式进行装配；如果没有匹配，则回退为一个原始类型进行匹配，如果匹配则自动装配；

## @Qualifier注解

当使用@Autowired注解时，如果此时匹配的类型有多个，则会按照byName去查找。但是如果还会有多个，就会报出异常。使用@Qualifier注解，可以可以按照指定的名称去匹配。



## 构造方法注入和sette方法注入区别

设置注入：

* 设值注入需要该Bean包含这些属性的setter方法
* 与传统的JavaBean的写法更相似，程序开发人员更容易理解、接收。
* 对于复杂的依赖关系，如果采用构造注入，会导致构造器臃肿，难以阅读。Spring在创建Bean实例时，需要同时实例化器依赖的全部实例，因而导致性能下降。而使用设值注入，则能避免这些问题
* 尤其是在某些属性可选的情况况下，多参数的构造器显得更加笨重

构造注入：

* 构造注入需要该Bean包含带有这些属性的构造器
* 构造注入可以在构造器中决定依赖关系的注入顺序，优先依赖的优先注入。例如，组件中其他依赖关系的注入，常常要依赖于DataSrouce的注入。采用构造注入，可以在代码中清晰的决定注入顺序。
* 对于依赖关系无需变化的Bean，构造注入更有用处。因为没有Setter方法，所有的依赖关系全部在构造器内设定。因此，无需担心后续的代码对依赖关系产生破坏。
* 依赖关系只能在构造器中设定，则只有组件的创建者才能改变组件的依赖关系。对组件的调用者而言，组件内部的依赖关系完全透明，更符合高内聚的原则

更重要的是对于循环依赖的解决。

## Spring中循环依赖问题及解决

**spring**容器循环依赖包括构造器循环依赖和setter循环依赖。

Spring容器能够顺利的实例化以构造函数注入方式配置bean的前提：bean构造函数参数引用的对象必须已经准备就绪。如果两个bean都采用构造注入，并通过构造函数参数引用了对方，就会发生类似线程死锁的循环依赖问题。

1 构造器循环依赖问题

通过构造器注入构成的循环依赖，这种依赖是无法解决的。只能抛出BeanCurrentlyIn CreationException异常表示循环依赖。

比如，在创建A类时，构造器需要B类，将去创建B类，在创建B类时，发现又需要A类，就去创建A类…从而形成了循环依赖，没办法创建。

Spring容器将每一个正在创建的bean标识符放在一个“当前创建bean池”中，bean标识符在创建过程中会一直保持在这个池中，因为如果在创建bean过程中发现自己已经在“当前创建bean池”中，会抛出BeanCurrentlyIn CreationException异常。对于创建完毕的bean，将从当前创建bean池中清除掉。

2 setter循环依赖

通过setter注入方式构成的循环依赖，是通过spring容器提前暴露刚完成构造器注入但未完成其他步骤（setter注入）的bean来完成的。但是只能解决单例作用域的bean循环依赖。通过提前暴露一个单例工厂方法，从而使其他bean能引用到该bean，如下代码所示：

1. addSingletonFactory(beanName, new ObjectFactory() {
2. public Object getObject() throws BeansException {
3. return getEarlyBeanReference(beanName, mbd, bean);
4. }
5. });

Spring容器创建A时，首先根据无参构造器创建bean，并暴露一个“ObjectFactory”用于返回一个提前暴露一个创建中的bean，并将A的标识符放到当前创建bean中，然后进行setter注入B类。

Spring容器创建B类时，首先根据无参构造器创建bean，并暴露一个“ObjectFactory”用于返回一个提前暴露一个创建中的bean，并将B类标识符放到当前创建bean池中，然后进行setter注入A，由于提前暴露了“ObjectFactory”工厂，从而使用它返回提前暴露一个创建中的bean。

完成了依赖注入。

对于非单例（prototype）作用域的bean，spring容器无法完成依赖注入。因为spring容器不会进行缓存prototype作用域的bean，因此无法提前暴露一个创建中的bean。

对于"singleton"作用域bean，可以通过"setAllowCircularReferences(false）；"来禁用循环引用。

## Spring框架中的事件机制

<http://blog.csdn.net/blueboz/article/details/49949573>

<http://xls9577087.iteye.com/blog/2121752>

事件驱动模型采用的是观察者模式或者发布-订阅模式：

1 是一种对象间的一对多的关系，如信号灯是目标（1），行人注视着信号灯（多方）

2 当目标发送改变（发布）时，观察者就可以接收到改变

3 观察者如何处理，目标无需干涉，所以就松散耦合了它们之间的关系。

我们可以自己把普通的java事件操作定义为bean集成到bean容器中，也可以使用spring提供的事件支持。

Spring的ApplicationContext 提供了支持事件和代码中监听器的功能。

**ApplicationEvent：**抽象类，继承自EventObject，同时是spring的application中事件的父类，需要被自定义的事件继承。

**ApplicationListener：**继承自EventListener，spring的application中的监听器必须实现的接口，需要被自定义的监听器实现其onApplicationEvent方法

 ApplicationEventPublisherAware：在spring的context中希望能发布事件的类必须实现的接口，该接口中定义了设置ApplicationEventPublisher的方法，由ApplicationContext调用并设置。在自己实现的ApplicationEventPublisherAware子类中，需要有ApplicationEventPublisher属性的定义。

ApplicationEventPublisher：spring的事件发布者接口，定义了发布事件的接口方法publishEvent。**因为ApplicationContext实现了该接口，**因此spring的ApplicationContext实例具有发布事件的功能(publishEvent方法在AbstractApplicationContext中有实现)。在使用的时候，只需要把ApplicationEventPublisher的引用定义到ApplicationEventPublisherAware的实现中，spring容器会完成对ApplicationEventPublisher的注入。

在spring的bean配置中，因为事件是由事件源发出的，不需要注册为bean由spring容器管理。所以在spring的配置中，只需配置自定义的ApplicationEventListener和publisherAware(即实现了ApplicationEventPublisherAware接口的发布类)，而对于ApplicationEventPublisher的管理和注入都由容器来完成。

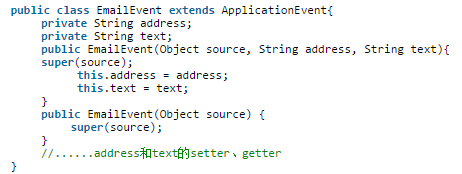
------------------------------------------------------------------------------------------------------------------------------

ApplicationContext事件机制是观察者设计模式的实现，通过ApplicationEvent类和ApplicationListener接口，可以实现ApplicationContext事件处理。如果容器中有一个ApplicationListener Bean，每当ApplicationContext发布ApplicationEvent时，ApplicationListener Bean将自动被触发。

Spring的事件框架有如下两个重要的成员：

* ApplicationEvent：容器事件，必须由ApplicationContext发布
* ApplicationListener：监听器，可由容器中的任何监听器Bean担任

实际上，Spring的事件机制与所有时间机制都基本相似，它们都需要事件源、事件和事件监听器组成。只是此处的事件源是ApplicationContext，且事件必须由Java程序显式触发。下面的程序将演示Spring容器的事件机制。程序先定义了一个ApplicationEvent类，其对象就是一个Spring容器事件。



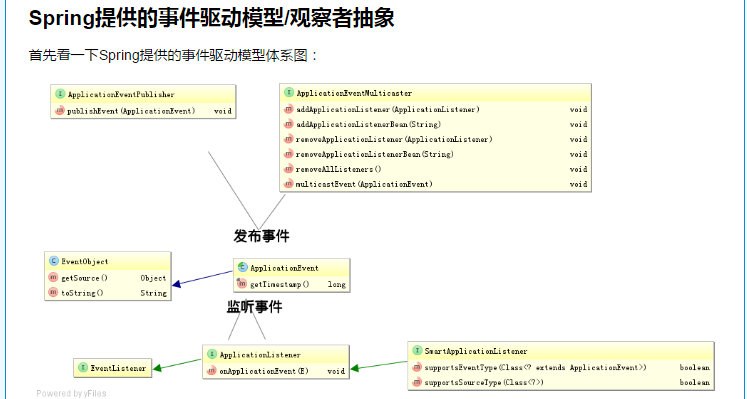


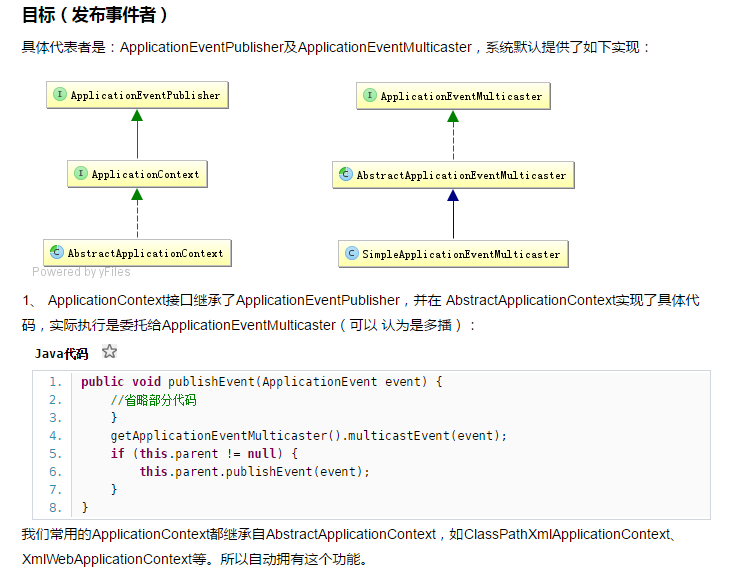
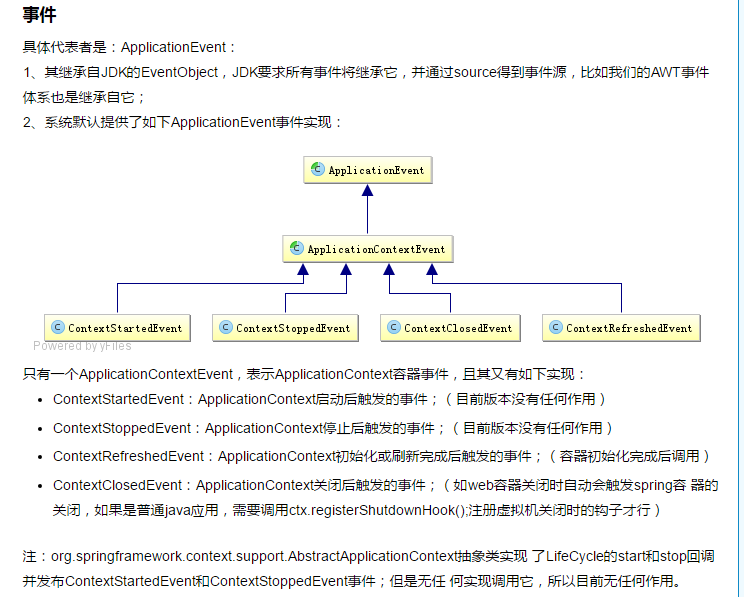


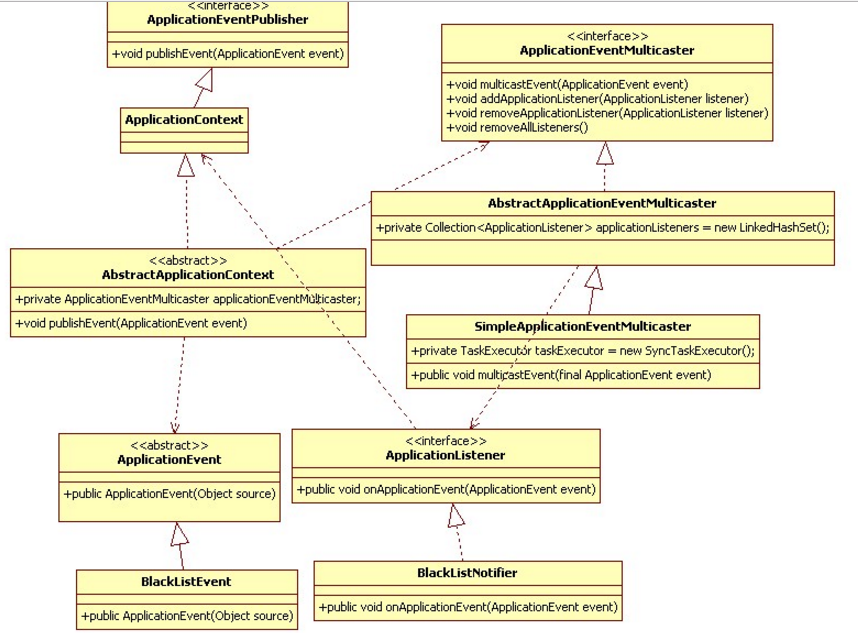
上述是使用applicationContext作为事件源，来发布某个事件的。也可以自己定义一个事件源用来发布事件，需要实现ApplicationEventPublisherAware接口。这样该类就可以有发布事件的功能了。

也可以









* **ApplicationListener 就是我们的 Observer，需要到容器中注册。他要关心他所关心的ApplicationEvent 。一般有如下代码：if (event instanceof BlackListEvent) {}**
* **ApplicationEventMulticaster是我们的SUBJECT一个代理。他会管理我们的 ApplicationListener 。**
* **ApplicationEvent 是事件，它就是媒介，充当介质的作用。**



Spring提供如下几个内置事件：

* ContextRefreshedEvent：上下文更新事件。ApplicationContext容器初始化或刷新时触发该事件。此处的初始化是指：所有的Bean被成功装载，后处理Bean被检测并激活，所有Singleton Bean 被预实例化，ApplicationContext容器已就绪可用
* ContextStartedEvent：上下文开始事件。当使用ConfigurableApplicationContext(ApplicationContext的子接口）接口的start()方法启动ApplicationContext容器时触发该事件。容器管理声明周期的Bean实例将获得一个指定的启动信号，这在经常需要停止后重新启动的场合比较常见
* ContextClosedEvent：上下文关闭事件。当使用ConfigurableApplicationContext接口的close()方法关闭ApplicationContext时触发该事件。容器关闭时，其管理的单例bean全部被销毁。
* ContextStoppedEvent：上下文停止事件。当使用ConfigurableApplicationContext接口的stop()方法使ApplicationContext容器停止时触发该事件。此处的停止，意味着容器管理生命周期的Bean实例将获得一个指定的停止信号，被停止的Spring容器可再次调用start()方法重新启动
* RequestHandledEvent：请求处理事件。请求处理结束后触发。Web相关事件，只能应用于使用DispatcherServlet的Web应用。在使用Spring作为前端的MVC控制器时，当Spring处理用户请求结束后，系统会自动触发该事件。

## FileSystemResource和ClassPathResource

在FileSystemResource 中需要给出spring-config.xml文件在你项目中的相对路径或者绝对路径。在ClassPathResource中spring会在ClassPath中自动搜寻配置文件，所以要把ClassPathResource 文件放在ClassPath下。

如果将spring-config.xml保存在了src文件夹下的话，只需给出配置文件的名称即可，因为src文件夹是默认。

**简而言之，ClassPathResource在环境变量中读取配置文件，FileSystemResource在配置文件中读取配置文件**

## Spring框架中用到了哪些设计模式

<http://www.cnblogs.com/yuefan/p/3763898.html>

<http://blog.csdn.net/adoocoke/article/details/8286902>

代理模式——在AOP和remoting中用的比较多

单例模式——在spring配置文件中定义的bean默认是单例

模板方法——解决代码重复问题，如果HibernateTemplate

前端控制器——spring提供了DispatcherServlet来对请求进行分发

工厂模式——BeanFactory用来创建对象的实例

依赖注入——贯穿于BeanFactory/ApplicationContext接口的核心理念

## Spring对hibernate的操作方式

继承HibernateDaoSupport类:提供一个AOP拦截器

或者组合HibernateTemplate：控制反转 callback

用spring的sessionfactory调用localsessionfactory。

配置hibernate sessionfactory

继承hibernatedaosupport

在aop支持的事务中装配

## spring支持的事务管理

<http://blog.csdn.net/z69183787/article/details/8199932>

<http://blog.csdn.net/trigl/article/details/50968079>

https://www.ibm.com/developerworks/cn/education/opensource/os-cn-spring-trans/

* **编程式事务管理**：这意味你通过编程的方式管理事务，给你带来极大的灵活性，但是难维护。
* **声明式事务管理：**这意味着你可以将业务代码和事务管理分离，你只需用注解和XML配置来管理事务。

**一般选择声明式事务，因为它对应用代码影响最小，更符合一个无侵入的轻量级容器的思想。**

## AOP

http://blog.csdn.net/moreevan/article/details/11977115/

AOP把软件系统分为两个部分：核心关注点和横切关注点。业务处理的主要流程是核心关注点，与之关系不大的部分是横切关注点。横切关注点的一个特点是，他们经常发生在核心关注点的多处，而各处都基本相似。比如权限认证、日志、事务处理。Aop 的作用在于分离系统中的各种关注点，将核心关注点和横切关注点分离开来。

实现AOP的技术，主要分为两大类：一是采用动态代理技术，利用截取消息的方式，对该消息进行装饰，以取代原有对象行为的执行；二是采用静态织入的方式，引入特定的语法创建“方面”，从而使得编译器可以在编译期间织入有关“方面”的代码。

## Aspect切面

AOP核心就是切面，它将多个类的通用行为封装成可重用的模块，该模块含有一组API提供横切功能。比如，一个日志模块可以被称作日志的AOP切面。根据需求的不同，一个应用程序可以有若干切面。在Spring AOP中，切面通过带有@Aspect注解的类实现。

## ****在Spring AOP 中，关注点和横切关注的区别****

关注点是应用中一个模块的行为，一个关注点可能会被定义成一个我们想实现的一个功能。

横切关注点是一个关注点，此关注点是整个应用都会使用的功能，并影响整个应用，比如日志，安全和数据传输，几乎应用的每个模块都需要的功能。因此这些都属于横切关注点。

对哪些方法进行拦截，拦截后怎么处理，这些关注点称之为横切关注点

## 连接点

连接点代表一个应用程序的某个位置，在这个位置我们可以插入一个AOP切面，它实际上是个应用程序执行Spring AOP的位置。

Spring中只支持方法类型的连接点。

## 切入点

是一个或者一组连接点，通知会在这些位置执行。对连接点进行拦截的定义

## 通知

通知是个在方法执行前或执行后要做的动作，实际上是程序执行时要通过SpringAOP框架触发的代码段。

Spring切面可以应用五种类型的通知：

* **before**：前置通知，在一个方法执行前被调用。
* **after:**在方法执行之后调用的通知，无论方法执行是否成功。
* **after-returning:**仅当方法成功完成后执行的通知。
* **after-throwing:**在方法抛出异常退出时执行的通知。
* **around:**在方法执行之前和之后调用的通知。

前置、后置、异常、最终、环绕通知五类

## 织入

将切面应用到目标对象并导致代理对象创建的过程。织入可以在编译 加载 或者 运行时完成。

## 引入

在不修改代码的前提下，引入可以在**运行期**为类动态地添加一些方法或字段

## 目标对象

被一个或者多个切面所通知的对象。它通常是一个代理对象。也指被通知（advised）对象。

## 代理

代理是通知目标对象后创建的对象。从客户端的角度看，代理对象和目标对象是一样的。

## ****有几种不同类型的自动代理****

BeanNameAutoProxyCreator

DefaultAdvisorAutoProxyCreator

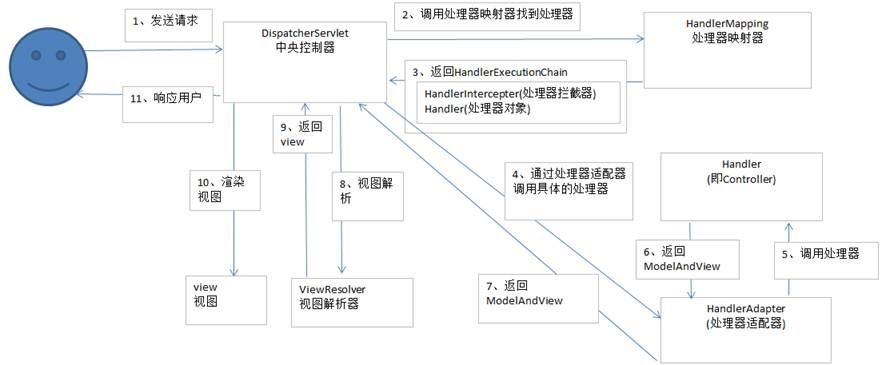
Metadata autoproxying

# Springmvc

## ****什么是Spring的MVC框架？****

Spring 配备构建Web 应用的全功能MVC框架。Spring可以很便捷地和其他MVC框架集成，如Struts，Spring 的MVC框架用控制反转把业务对象和控制逻辑清晰地隔离。它也允许以声明的方式把请求参数和业务对象绑定。

## 简述springmvc的工作流程



1 用户发送请求到达前端控制器DispatcherServlet

2 前端控制器收到请求后，会调用处理器映射器HandlerMapping

3 处理器映射器找到具体的处理器handler，生成处理器对象及处理器拦截器（如果有则生成），返回给前端控制器

4 前端控制器调用处理器适配器HandlerAdapter（也叫后端控制器 Controller）去执行handler

5 controller执行完毕之后，给适配器返回ModelAndView

6 处理器适配器将ModelAndView返回给前端 控制器

7 前端控制器将ModelAndView传给视图解析器（ViewReslover）进行解析（根据逻辑视图名解析成真正的视图jsp）

8 视图解析器解析完毕后向前端控制器返回具体的view

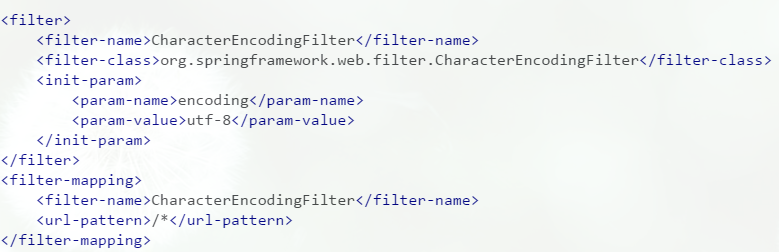
9 DispatcherServlet根据view进行视图渲染（即将模型数据填充到request域中）

10 前端控制器向用户返回结果

## 如何解决POST GET请求中文乱码问题

对于post请求：

在web.xml中，进行过滤器配置：CharacterEncodingFilter



对于get请求：

1 修改tomcat配置文件 编码与工程中编码一致



2 对参数进行重新编码



ISO8859-1是tomcat默认编码，需要将tomcat编码后的内容按utf-8编码

## Spingmvc和struts2的区别

1 springmvc的入口是一个servlet即前端控制器，而struts2的入口是一个filter过滤器

2 springmvc是基于方法开发的，是方法级别的拦截，一个方法对应于request上下文，方法跟url对应，传递参数是通过方法的形参，可以单例或多例。strust2是基于类开发，是类级别的拦截，一个类对应于request上下文，传递参数是通过类的属性（模型驱动和属性驱动），只能设计为多例。

所以从架构本身上springmvc更加容易实现restful api。因为struts2中的一个方法可以对应一个url，但是其类属性是被所有方法共享的。这也就无法使用注解或者其他方式标识其所属的方法了。

3 struts2采用值栈存储请求和响应的数据，通过OGNL存取数据，springmvc通过参数解析器将request对象的内容进行解析成方法形参，将响应数据和页面封装成ModelAndView对象，最后将模型数据通过request对象传输到页面上。Jsp页面上默认使用jstl。

4 springmvc中集成了AJAX，使用比较方便，只需使用注解@RequestBody和@ResponseBody。

而struts2拦截器集成了AJAX，在action中处理时一般必须安装插件或者自己写代码集成进去，不方便。

5 Struts2更加符合OOP的编程思想，springmvc是在servlet上扩展。

## ****DispatcherServlet****

Spring的MVC框架是围绕DispatcherServlet来设计的，它用来处理所有的HTTP请求和响应。

## ****WebApplicationContext****

WebApplicationContext 继承了ApplicationContext 并增加了一些WEB应用必备的特有功能，它不同于一般的ApplicationContext ，因为它能处理主题，并找到被关联的servlet。

## ****Spring MVC框架的控制器****

控制器提供一个访问应用程序的行为，此行为通常通过服务接口实现。控制器解析用户输入并将其转换为一个由视图呈现给用户的模型。Spring用一个非常抽象的方式实现了一个控制层，允许用户创建多种用途的控制器。

## ****@Controller 注解****

该注解表明该类扮演控制器的角色，Spring不需要你继承任何其他控制器基类或引用Servlet API。

## ****@RequestMapping 注解****

该注解是用来映射一个URL到一个类或一个特定的方处理法上。

# Hibernate

## Hibernate和mybatis区别

## Hibernate中的缓存机制以及为什么使用

## Hibernate中的session和httpsession区别

## Sql注入问题

## 如果不使用参数绑定，怎么防止sql注入

## OpenSessionInView

Spring解决了Hibernate中的Session的关闭和开启问题。

Hibernate允许对关联对象 属性进行延迟加载，但是必须保证延迟加载的操作限于同一个Hibernate Session范围之内。如果Service层返回了一个启用延迟加载功能的领域对象给Web层，当Web层访问到那些需要延迟加载的数据时，由于加载领域对象的session已经关闭，这时会报异常LazyInitializationException。

Spring提供了OpenSessionInViewFilter过滤器，可以把一个Hibernate Session和一次完整的请求过程对应的线程绑定。也就是在web层开启了session，请求结束关闭session。这样，就可以允许在事务提交之后延迟加载所需要显示的对象。

OpenSessionInViewFilter 过滤器将 Hibernate Session 绑定到请求线程中，它将自动被 Spring 的事务管理器探测到。所以 OpenSessionInViewFilter 适用于 Service 层使用HibernateTransactionManager 或 JtaTransactionManager 进行事务管理的环境，也可以用于非事务只读的数据操作中。

<filter>  
        <filter-name>Spring OpenSessionInViewFilter</filter-name>  
        <filter-class>org.springframework.orm.hibernate3.support.OpenSessionInViewFilter</filter-class>  
     <init-param>  
      
<!--   
指定org.springframework.orm.hibernate3.LocalSessionFactoryBean在spring配置文件中的名称,默认值为sessionFactory  
     如果LocalSessionFactoryBean在spring中的名称不是sessionFactory,该参数一定要指定,否则会出现找不到sessionFactory的例外  
-->  
     <param-name>sessionFactoryBean</param-name>  
   <param-value>sessionFactory</param-value>  
  </init-param>   
    </filter>  
    <filter-mapping>  
        <filter-name>Spring OpenSessionInViewFilter</filter-name>  
        <url-pattern>/\*</url-pattern>  
    </filter-mapping>

# Java web

## Memcache

**memcache**是一套**分布式的高速缓存系统**，由LiveJournal的Brad Fitzpatrick开发，但目前被许多网站使用以提升网站的访问速度，尤其对于一些大型的、需要频繁访问数据库的网站访问速度提升效果十分显著。这是一套开放源代码软件，以BSD license授权发布。

MemCache的工作流程如下：先检查客户端的请求数据是否在memcached中，如有，直接把请求数据返回，不再对数据库进行任何操作；如果请求的数据不在memcached中，就去查数据库，把从数据库中获取的数据返回给客户端，同时把数据缓存一份到memcached中（memcached客户端不负责，需要程序明确实现）；每次更新数据库的同时更新memcached中的数据，保证一致性；当分配给memcached内存空间用完之后，会使用**LRU（Least Recently Used，最近最少使用）策略加上到期失效策略，失效数据首先被替换，然后再替换掉最近未使用的数据。**

Memcache是一个**高性能的分布式的内存对象缓存系统**，通过**在内存里维护一个统一的巨大的hash表**，它能够用来存储各种格式的数据，包括图像、视频、文件以及数据库检索的结果等。简单的说就是**将数据调用到内存中，然后从内存中读取，从而大大提高读取速度。**

Memcache是danga的一个项目，最早是LiveJournal 服务的，最初为了加速 LiveJournal 访问速度而开发的，后来被很多大型的网站采用。

**Memcached是以守护程序(监听)方式运行于一个或多个服务器中，随时会接收客户端的连接和操作。**

在 Memcached中可以保存的item数据量是没有限制的，只要内存足够 。  
　　Memcached单进程在32位系统中**最大使用内存为2G**，若在64位系统则没有限制,这是由于**32位系统限制单进程最多可使用2G内存**,要使用更多内存，可以分多个端口开启多个Memcached进程 ,最大30天的数据过期时间,设置为永久的也会在这个时间过期，常量REALTIME\_MAXDELTA 60\*60\*24\*30控制 　　**最大键长为250字节，大于该长度无法存储，常量KEY\_MAX\_LENGTH 250控制**　　**单个item最大数据是1MB，超过1MB数据不予存储**，常量POWER\_BLOCK 1048576进行控制，它是默认的slab大小 　　最大同时连接数是200，通过 conn\_init()中的freetotal进行控制，最大软连接数是1024，通过settings.maxconns = 1024 进行控制 　　跟空间占用相关的参数：settings.factor=1.25, settings.chunk\_size=48, 影响slab的数据占用和步进方式

memcached是一种**无阻塞的socket通信方式服务**，基于libevent库，由于无阻塞通信，对内存读写速度非常之快。　　memcached分服务器端和客户端，可以配置多个服务器端和客户端，应用于分布式的服务非常广泛。　　memcached作为**小规模的数据分布式平台**是十分有效果的。

**memcached是键值一一对应，key默认最大不能超过128个字 节，value默认大小是1M**，也就是一个slabs，如果要存2M的值（连续的），不能用两个slabs，因为两个slabs不是连续的，无法在内存中 存储，故需要修改slabs的大小，多个key和value进行存储时，即使这个slabs没有利用完，那么也不会存放别的数据。

memcached已经可以支持C/C++、Perl、PHP、Python、Ruby、Java、C#、Postgres、Chicken Scheme、Lua、MySQL和Protocol等语言客户端。

memcache 可以应对任意多个连接，使用非阻塞的网络IO。由于它的工作机制是在内存中开辟一块空间，然后建立一个**HashTable**，Memcached自管理这些HashTable。

为什么会有Memcache和memcached两种名称？

其实Memcache是这个项目的名称，而memcached是它服务器端的主程序文件名，知道我的意思了吧~~~~。一个是项目名称，一个是主程序文件名，在网上看到了很多人不明白，于是混用了。

# 计算机网络

## socket 中的长连接和短连接

**短连接**

连接->传输数据->关闭连接

HTTP是无状态的，浏览器和服务器每进行一次HTTP操作，就建立一次连接，但任务结束就中断连接。

也可以这样说：短连接是指SOCKET连接后发送后接收完数据后马上断开连接。

**长连接**

连接->传输数据->保持连接 -> 传输数据-> 。。。 ->关闭连接。

长连接指建立SOCKET连接后不管是否使用都保持连接，但安全性较差。

**http的长连接**

HTTP也可以建立长连接的，使用Connection:keep-alive，HTTP 1.1默认进行持久连接。HTTP1.1和HTTP1.0相比较而言，最大的区别就是增加了持久连接支持(貌似最新的 http1.0 可以显示的指定 keep-alive),但还是无状态的，或者说是不可以信任的。

**什么时候用长连接，短连接？**

 长连接多用于操作频繁，点对点的通讯，而且连接数不能太多情况，。每个TCP连接都需要三步握手，这需要时间，如果每个操作都是先连接，再操作的话那么处理速度会降低很多，所以每个操作完后都不断开，次处理时直接发送数据包就OK了，不用建立TCP连接。例如：数据库的连接用长连接， 如果用短连接频繁的通信会造成socket错误，而且频繁的socket 创建也是对资源的浪费。

而像WEB网站的http服务一般都用短链接，因为长连接对于服务端来说会耗费一定的资源，而像WEB网站这么频繁的成千上万甚至上亿客户端的连接用短连接会更省一些资源，如果用长连接，而且同时有成千上万的用户，如果每个用户都占用一个连接的话，那可想而知吧。所以并发量大，但每个用户无需频繁操作情况下需用短连好。

总之，长连接和短连接的选择要视情况而定。

## 七层与四层协议

**OSI分层 （7层）**：物理层、数据链路层、网络层、传输层、会话层、表示层、应用层。

**TCP/IP分层（4层）**：网络接口层、 网际层、运输层、 应用层。

**五层协议 （5层）**：物理层、数据链路层、网络层、运输层、 应用层。

**每一层的协议如下：**

物理层：RJ45、CLOCK、IEEE802.3 （中继器，集线器，网关）

数据链路：PPP、FR、HDLC、VLAN、MAC （网桥，交换机）

网络层：**IP、ICMP、ARP、RARP、OSPF、IPX、RIP、IGRP**（路由器）

传输层：TCP、UDP、SPX

会话层：NFS、SQL、NETBIOS、RPC

表示层：JPEG、MPEG、ASII

应用层：**FTP、DNS、Telnet、SMTP、HTTP、WWW、NFS**

**每一层的作用如下：**

**物理层：**通过媒介传输比特,确定机械及电气规范（比特Bit） 不可靠的

**数据链路层**：将比特组装成帧和点到点的传递（**帧Frame**） 可靠的

**网络层**：负责数据包从源到宿的传递和网际互连（**包PackeT**）

**传输层**：提供端到端的可靠报文传递和错误恢复（**段Segment**）

**会话层**：建立、管理和终止会话（会话协议数据单元SPDU）3

**表示层**：对数据进行翻译、加密和压缩（表示协议数据单元PPDU）

**应用层**：允许访问OSI环境的手段（应用协议数据单元APDU） 直接为用户的应用进程提供服务

为什么会有TCP/IP四层模型？

OSI七层模型是一种框架性的设计方法，建立七层模型的主要目的是解决异种网络互连时出现的兼容性问题。最主要的功能是使不同类型的主机实现数据传输。最大的优点是将服务 接口和协议三个概念区分开来。

服务：下层为上层提供的功能

接口：上层如何使用下层的服务

协议：如何实现本层的服务

便于研究和教学。

网络接口层：TCP/IP参考模型并没有真正描述这一层的实现。只是要求能够提供给其上层——网际层一个访问接口，以便在其上传递IP分组。

网际层：是整个TCP/IP协议栈的核心。将分组发到目标网络或主机。为了尽快发送分组，可能需要沿不同路径进行传递，这就需要上层进行排序。除了路由功能外，还能将不同类型的网络互连。

运输层：传输层的功能是使源端主机和目标端主机上的对等实体可以进行会话。在传输层定义了两种服务质量不同的协议。即：传输控制协议TCP（transmission control protocol）和用户数据报协议UDP（user datagram protocol）。　　  
　　TCP协议是一个面向连接的、可靠的协议。它将一台主机发出的字节流无差错地发往互联网上的其他主机。在发送端，它负责把上层传送下来的字节流分成报文段并传递给下层。在接收端，它负责把收到的报文进行重组后递交给上层。TCP协议还要处理端到端的流量控制，以避免缓慢接收的接收方没有足够的缓冲区接收发送方发送的大量数据。UDP协议是一个不可靠的、无连接协议，主要适用于不需要对报文进行排序和流量控制的场合。

应用层：TCP/IP模型将OSI参考模型中的会话层和表示层的功能合并到应用层实现。应用层面向不同的网络应用引入了不同的应用层协议。其中，有基于TCP协议的，如文件传输协议（File Transfer Protocol，FTP）、虚拟终端协议（TELNET）、超文本链接协议（Hyper Text Transfer Protocol，HTTP），也有基于UDP协议的。

**对于TCP/IP网络，更加关注的是IP和TCp层，故将其他层直接合并在一起了。Osi七层模型有点冗余。**

## TCP三次握手和四次挥手

<http://blog.csdn.net/whuslei/article/details/6667471/>

http://www.cnblogs.com/zmlctt/p/3690998.html

为什么需要三次握手和四次挥手

<http://blog.csdn.net/xifeijian/article/details/12777187>

**三次握手：**

第一次握手：客户端发送syn包(syn=x)到服务器，并进入SYN\_SEND状态，等待服务器确认；

第二次握手：服务器收到syn包，必须确认客户的SYN（ack=x+1），同时自己也发送一个SYN包（syn=y），即SYN+ACK包，此时服务器进入SYN\_RECV状态；

第三次握手：客户端收到服务器的SYN＋ACK包，向服务器发送确认包ACK(ack=y+1)，此包发送完毕，客户端和服务器进入ESTABLISHED状态，完成三次握手。

握手过程中传送的包里不包含数据，三次握手完毕后，客户端与服务器才正式开始传送数据。理想状态下，TCP连接一旦建立，在通信双方中的任何一方主动关闭连接之前，TCP 连接都将被一直保持下去。

**四次握手**

与建立连接的“三次握手”类似，断开一个TCP连接则需要“四次握手”。

第一次挥手：主动关闭方发送一个FIN，用来关闭主动方到被动关闭方的数据传送，也就是主动关闭方告诉被动关闭方：我已经不 会再给你发数据了(当然，在fin包之前发送出去的数据，如果没有收到对应的ack确认报文，主动关闭方依然会重发这些数据)，但是，此时主动关闭方还可 以接受数据。

第二次挥手：被动关闭方收到FIN包后，发送一个ACK给对方，确认序号为收到序号+1（与SYN相同，一个FIN占用一个序号）。

第三次挥手：被动关闭方发送一个FIN，用来关闭被动关闭方到主动关闭方的数据传送，也就是告诉主动关闭方，我的数据也发送完了，不会再给你发数据了。

第四次挥手：主动关闭方收到FIN后，发送一个ACK给被动关闭方，确认序号为收到序号+1，至此，完成四次挥手。

## ARP地址解析协议

是根据IP地址获取物理地址的一个TCP/IP协议。

1：首先，每个主机都会在自己的ARP缓冲区中建立一个ARP列表，以表示IP地址和MAC地址之间的对应关系。

2：当源主机要发送数据时，首先检查ARP列表中是否有对应IP地址的目的主机的MAC地址，如果有，则直接发送数据，如果没有，就向本网段的所有主机发送ARP数据包，该数据包包括的内容有：**源主机** **IP地址，源主机MAC地址，目的主机的IP 地址**。

3：当本网络的所有主机收到该ARP数据包时，首先检查数据包中的IP地址是否是自己的IP地址，如果不是，则忽略该数据包，如果是，则首先从数据包中取出源主机的IP和MAC地址写入到ARP列表中，如果已经存在，则覆盖，然后将自己的MAC地址写入ARP响应包中，告诉源主机自己是它想要找的MAC地址。

4：源主机收到ARP响应包后。将目的主机的IP和MAC地址写入ARP列表，并利用此信息发送数据。如果源主机一直没有收到ARP响应数据包，表示ARP查询失败。

**广播发送ARP请求，单播发送ARP响应**

RARP是逆地址解析协议，作用是完成硬件地址到IP地址的映射，主要用于无盘工作站，因为给无盘工作站配置的IP地址不能保存。工作流程：在网络中配置一台RARP服务器，里面保存着IP地址和MAC地址的映射关系，当无盘工作站启动后，就封装一个RARP数据包，里面有其MAC地址，然后广播到网络上去，当服务器收到请求包后，就查找对应的MAC地址的IP地址装入响应报文中发回给请求者。因为需要广播请求报文，因此RARP只能用于具有广播能力的网络。

## 在浏览器中输入www.baidu.com后执行的全部过程

<http://blog.csdn.net/qq_16681169/article/details/50866290>

https://www.zhihu.com/question/27199686

1、客户端浏览器通过DNS解析到www.baidu.com的IP地址220.181.27.48，通过这个IP地址找到客户端到服务器的路径。客户端浏览器发起一个HTTP会话到220.161.27.48，然后通过TCP进行封装数据包，输入到网络层。

2、在客户端的传输层，把HTTP会话请求分成报文段，添加源和目的端口，如服务器使用80端口监听客户端的请求，客户端由系统随机选择一个端口如5000，与服务器进行交换，服务器把相应的请求返回给客户端的5000端口。然后使用IP层的IP地址查找目的端。

3、客户端的网络层不用关系应用层或者传输层的东西，主要做的是通过查找路由表确定如何到达服务器，期间可能经过多个路由器，这些都是由路由器来完成的工作，我不作过多的描述，无非就是通过查找路由表决定通过那个路径到达服务器。

4、客户端的链路层，包通过链路层发送到路由器，通过邻居协议查找给定IP地址的MAC地址，然后发送ARP请求查找目的地址，如果得到回应后就可以使用ARP的请求应答交换的IP数据包现在就可以传输了，然后发送IP数据包到达服务器的地址。

## TCP和UDP的区别

TCP提供面向连接的、可靠的数据流传输，而UDP提供的是非面向连接的、不可靠的数据流传输。

TCP传输单位称为TCP报文段，UDP传输单位称为用户数据报。

TCP注重数据安全性，UDP数据传输快，因为不需要连接等待，少了许多操作，但是其安全性却一般。

**TCP对应的协议和UDP对应的协议**

**TCP对应的协议：**

（1） **FTP**：定义了文件传输协议，使用**21**端口。

（2） **Telnet**：一种用于远程登陆的端口，使用**23**端口，用户可以以自己的身份远程连接到计算机上，可提供基于DOS模式下的通信服务。

（3） **SMTP**：邮件传送协议，用于发送邮件。服务器开放的是**25**号端口。

（4） **POP3**：它是和SMTP对应，POP3用于接收邮件。POP3协议所用的是**110**端口。

（5）**HTTP**：是从Web服务器传输超文本到本地浏览器的传送协议。

**UDP对应的协议：**

（1） **DNS**：用于域名解析服务，将域名地址转换为IP地址。DNS用的是**53**号端口。

（2） **SNMP**：简单网络管理协议，使用**161**号端口，是用来管理网络设备的。由于网络设备很多，无连接的服务就体现出其优势。

（3）**TFTP**(Trival File Tran敏感词er Protocal)，简单文件传输协议，该协议在熟知端口**69**上使用UDP服务。

## 面向连接和非面向连接的服务的特点是什么

面向连接的服务，通信双方在进行通信之前，要先在双方建立起一个完整的可以彼此沟通的通道，在通信过程中，整个连接的情况一直可以被实时地监控和管理。

非面向连接的服务，不需要预先建立一个联络两个通信节点的连接，需要通信的时候，发送节点就可以往网络上发送信息，让信息自主地在网络上去传，一般在传输的过程中不再加以监控。

## 端口及对应的服务

|  |  |  |  |
| --- | --- | --- | --- |
| 服务 | 端口号 | 服务 | 端口号 |
| FTP | 21 | SSH | 22 |
| telnet | 23 | SMTP | 25 |
| Domain(域名服务器) | 53 | HTTP | 80 |
| POP3 | 110 | NTP（网络时间协议） | 123 |
| MySQL数据库服务 | 3306 | Shell或 cmd | 514 |
| POP-2 | 109 | SQL Server | 1433 |

## 交换机 路由器 网关

1）交换机

在计算机网络系统中，交换机是针对共享工作模式的弱点而推出的。交换机拥有一条高带宽的背部总线和内部交换矩阵。交换机的所有的端口都挂接在这条背 部总线上，当控制电路收到数据包以后，处理端口会查找内存中的地址对照表以确定目的MAC（网卡的硬件地址）的NIC（网卡）挂接在哪个端口上，通过内部 交换矩阵迅速将数据包传送到目的端口。目的MAC若不存在，交换机才广播到所有的端口，接收端口回应后交换机会“学习”新的地址，并把它添加入内部地址表 中。

交换机工作于OSI参考模型的第二层，即数据链路层。交换机内部的CPU会在每个端口成功连接时，通过ARP协议学习它的MAC地址，保存成一张 ARP表。在今后的通讯中，发往该MAC地址的数据包将仅送往其对应的端口，而不是所有的端口。因此，交换机可用于划分数据链路层广播，即冲突域；但它不 能划分网络层广播，即广播域。

交换机被广泛应用于二层网络交换，俗称“二层交换机”。

交换机的种类有：二层交换机、三层交换机、四层交换机、七层交换机分别工作在OSI七层模型中的第二层、第三层、第四层盒第七层，并因此而得名。

2）路由器

**路由器**（**Router**）是一种计算机网络设备，提供了路由与转送两种重要机制，可以决定数据包从来源端到目的端所经过 的路由路径（host到host之间的传输路径），这个过程称为路由；将路由器输入端的数据包移送至适当的路由器输出端(在路由器内部进行)，这称为转 送。路由工作在OSI模型的第三层——即网络层，例如网际协议。

**路由器的一个作用是连通不同的网络，另一个作用是选择信息传送的线路。** 路由器与交换器的差别，路由器是属于OSI第三层的产品，交换器是OSI第二层的产品(这里特指二层交换机)。

3）网关

**网关**（Gateway），**网关**顾名思义就是连接两个网络的设备，区别于路由器（由于历史的原因，许多有关TCP/IP 的文献曾经把网络层使用的路由器（Router）称为网关，在今天很多局域网采用都是路由来接入网络，因此现在通常指的网关就是路由器的IP），经常在家 庭中或者小型企业网络中使用，用于连接局域网和Internet。 网关也经常指把一种协议转成另一种协议的设备，比如语音网关。

在传统TCP/IP术语中，网络设备只分成两种，一种为网关（gateway），另一种为主机（host）。网关能在网络间转递数据包，但主机不能 转送数据包。在主机（又称终端系统，end system）中，数据包需经过TCP/IP四层协议处理，但是在网关（又称中介系 统，intermediate system）只需要到达网际层（Internet layer），决定路径之后就可以转送。在当时，网关 （gateway）与路由器（router）还没有区别。

在现代网络术语中，网关（gateway）与路由器（router）的定义不同。网关（gateway）能在不同协议间移动数据，而路由器（router）是在不同网络间移动数据，相当于传统所说的IP网关（IP gateway）。

**网关**是连接两个网络的设备，对于语音网关来说，他可以连接PSTN网络和以太网，这就相当于VOIP，把不同电话中的模拟信号通过网关而转换成数字信号，而且加入协议再去传输。在到了接收端的时候再通过**网关**还原成模拟的电话信号，最后才能在电话机上听到。

对于以太网中的**网关**只能转发三层以上数据包，这一点和路由是一样的。而不同的是**网关**中并没有路由表，他只能按照预先设定的不同网段来进行转发。网关最重要的一点就是端口映射，子网内用户在外网看来只是外网的IP地址对应着不同的端口，这样看来就会保护子网内的用户。

## 流量控制和拥塞控制

**流量控制**

所谓流量控制就是让发送方的发送速率不要太快，要让对方来得及接收。

利用滑动窗口机制可以很方便地在 TCP 连接上实现对发送方的流量控制。

发送方的发送窗口不能超过接收方给出的接收窗口的数值。

注意，TCP 的窗口单位是字节，不是报文段。

**拥塞控制**

在计算机网络中的链路容量（即带宽）、交换节点中的缓存和处理机等，都是网络的资源。在某段时间，若对网络中的某一资源的需求超过了该资源所能提供的可用部分，网络中的性能就要变坏。这种情况就叫做拥塞。

若网络中的许多资源同时出现供应不足，网络的性能就要明显变坏，整个网络德吞吐量将随输入负荷的增大而下降。

拥塞控制的四种算法：慢开始(slow start)、拥塞避免(congestion avoidance)、快重传(fast retransmit)、快恢复(fast recovery)。

**拥塞控制与流量控制**

所谓拥塞控制就是防止过多的数据注入到网络中，这样可以使网络中德路由器或链路不致过载。拥塞控制所要做的都有一个前提，就是网路能够承受现有的网络负荷。拥塞控制是一个全局性德过程，涉及到所有的主机、所有的路由器、以及与降低网路传输性能有关的所有因素。

相反，流量控制往往指点对点通信量德控制，是个端到端问题（接收端控制发送端）。流量控制所要做的就是抑制发送端发送数据的速率，以便使接收端来得及接收。

拥塞控制和流量控制之所以常常被搞混，是因为某些拥塞控制算法是向发送端发送控制报文，并告诉发送端，网络出现麻烦，必须放慢发送速率。这点又和流量控制很相似。

## 网络层协议简介

**ARP地址解析协议**

网络层使用的是IP地址，但在实际网络的链路上传送数据帧时，最终还是必须使用该网络德硬件地址。但IP地址和下面的网络的硬件地址之间由于格式不同而不存在简单的映射关系（例如，IP地址有32位，而局域网的硬件地址是48位）。此外，在一个网络中，经常会有新的主机加进来，或撤走一些主机。更换网络适配器也会使主机的硬件地址改变。地址解析协议ARP解决这个问题的方法是在主机ARP高速缓存中应存放一个从IP地址到硬件地址的映射表，并且这个映射表还经常动态更新（新增或超时删除）。

每一个主机都设有一个ARP告诉缓存，里面有本局域网上的各主机和路由器的IP地址到硬件地址的映射表，这些都是该主机目前知道的一些地址。

注意，ARP是解决同一个局域网上的主机或路由器的IP地址和硬件地址的映射问题。从IP地址到硬件地址的解析是自动进行的，主机的用户对这种地址解析过程是不知道的。

**ICMP网际控制报文协议**

ICMP报文的种类有两种，即ICMP差错报告报文和ICMP询问报文。

ICMP差错报告报文共有5种，即：

（1）终点不可达    当路由器或主机不能交付数据报时就向源点发送终点不可达报文。

（2）源点抑制    当路由器或主机由于拥塞而丢弃数据报时，就向源点发送源点抑制报文，使源点知道应当把数据报的发送速率放慢。

（3）时间超过    当路由器收到生存时间为零的数据报时，除丢弃该数据报外，还要向源点发送时间超过报文。当终点在预先规定时间内不能收到一个数据报的全部数据报片时，就把已收到的数据报片都丢弃，并向源点发送时间超过报文。

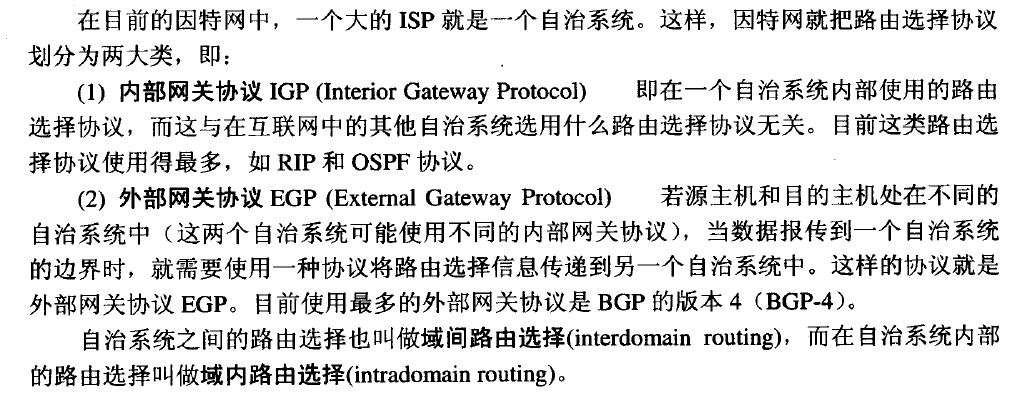
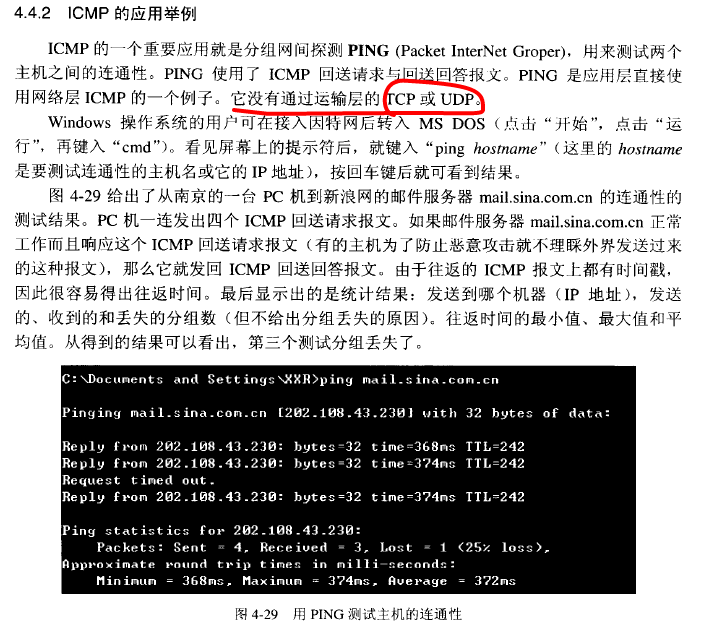
（4）参数问题    当路由器或目的主机收到的数据报的首部中有的字段值不正确时，就丢弃该数据报，并向源点发送参数问题报文。

（5）改变路由（重定向）    路由器把改变路由报文发送给主机，让主机知道下次应将数据报发送给另外的路由器（可通过更好的路由）。

ICMP询问报文有两种，即：

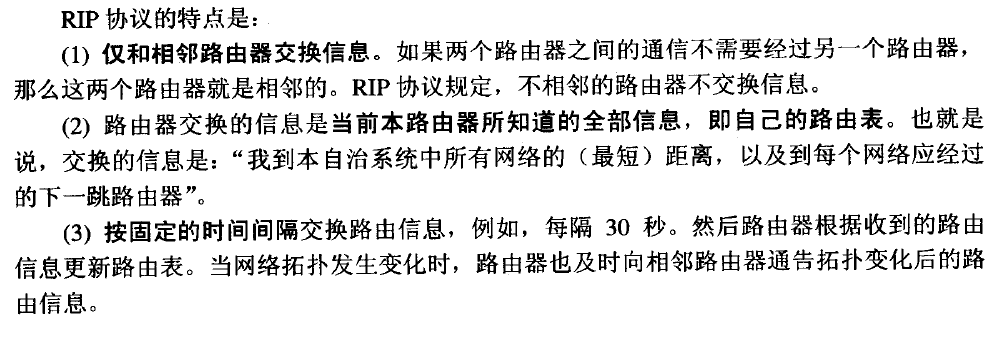
（1）回送请求和回答    ICMP回送请求报文是由主机和路由器向一个特定的目的主机发出的询问。收到此报文的主机必须给源主机或路由器发送ICMP回送回答报文。这种询问报文用来测试目的站是否可达以及了解其有关状态。

（2）时间戳请求和回答    ICMP时间戳请求报文是请某个主机或路由器回答当前的日期和时间。在ICMP时间戳回答报文中有一个32位的字段，其中写入的整数代表从1900年1月1日到当前时刻一共多少秒。时间戳请求与回答可用来进行时钟同步和测量时延。



**RIP路由信息协议**

RIP是一种分布式的基于距离向量的路由选择协议。



**OSPF最短路径优先协议**

