# 多线程

## 基础多线程

<https://www.cnblogs.com/GarfieldEr007/p/5746362.html>

### Thread类和Runnable接口

实现Runnable接口比继承Thread类所具有的优势：

1）：适合多个相同的程序代码的线程去处理同一个资源

2）：可以避免java中的单继承的限制

3）：增加程序的健壮性，代码可以被多个线程共享，代码和数据独立

4）：线程池只能放入实现Runable或callable类线程，不能直接放入继承Thread的类

在java程序中，每次程序运行至少启动2个线程。一个是main线程，一个是垃圾收集线程

### 线程的状态转换

#### 新建状态（New）

新创建了一个线程对象。

#### 就绪状态（Runnable）

线程对象创建后，其他线程调用了该对象的start()方法。该状态的线程位于可运行线程池中，变得可运行，等待获取CPU的使用权。

#### 运行状态（Running）

就绪状态的线程获取了CPU，执行程序代码。

#### 阻塞状态（Blocked）

阻塞状态是线程因为某种原因放弃CPU使用权，暂时停止运行。直到线程进入就绪状态，才有机会转到运行状态。

##### 等待阻塞

运行的线程执行wait()方法，JVM会把该线程放入等待池中。(wait会释放持有的锁)

##### 同步阻塞

运行的线程在获取对象的同步锁时，若该同步锁被别的线程占用，则JVM会把该线程放入锁池中。

##### 其他阻塞

运行的线程执行sleep()或join()方法，或者发出了I/O请求时，JVM会把该线程置为阻塞状态。当sleep()状态超时、join()等待线程终止或者超时、或者I/O处理完毕时，线程重新转入就绪状态。（注意,sleep是不会释放持有的锁）

#### 死亡状态（Dead）

线程执行完了或者因异常退出了run()方法，该线程结束生命周期。

**线程生命周期结束，但是线程对象不一定被回收，既线程池中可以重复使用这样的对象执行任务（一个实现了Runnable接口的类的对象）避免不断新建和回收线程对象，提高效率**

### 线程调度

#### 线程优先级

线程优先级为1~10，Thread类有三个静态常量MAX\_PRIORITY、MIN\_PRIORITY、NORM\_PRIORITY默认应该使用此三个常量优先级，因为java线程的10个优先级和操作系统中线程优先级不一一对应。线程优先级设置只是增大了高优先级线程的执行机会，并不能完全保证线程的执行顺序。

#### 线程睡眠

Thread.sleep(long millis)方法，使线程转到阻塞状态。millis参数设定睡眠的时间，以毫秒为单位。当睡眠结束后，就转为就绪（Runnable）状态。sleep()平台移植性好

#### 线程等待

Object类中的wait()方法，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() 唤醒方法。这个两个唤醒方法也是Object类中的方法，行为等价于调用 wait(0)

#### 线程让步

Thread.yield() 方法，暂停当前正在执行的线程对象，把执行机会让给相同或者更高优先级的线程。如果所有线程的优先级都小于当前线程，则当前线程不会让步。既较低优先级的线程只能等待所有高优先级的线程执行完成后才能执行

**让步后线程的执行将和相同或更高优先级的线程进行重新调度，所以让步线程在让步后很可能继续执行**

#### 线程加入

join()方法，等待其他线程终止。在当前线程中调用另一个线程的join()方法，则当前线程转入阻塞状态，直到另一个进程运行结束，当前线程再由阻塞转为就绪状态

应用场景：当前线程的执行依赖其他线程的执行结果

#### 线程唤醒

Object类中的notify()方法，唤醒在此对象监视器上等待的单个线程。如果所有线程都在此对象上等待，则会选择唤醒其中一个线程。选择是任意性的。被唤醒的线程将以常规方式与在该对象上主动同步的其他所有线程进行竞争

notifyAll()唤醒在此对象监视器上等待的所有线程

#### 线程调度应用和方法对比

##### 三线程打印ABC

**要求：**

建立三个线程，A线程打印10次A，B线程打印10次B,C线程打印10次C，要求线程同时运行，交替打印10次ABC。

**打印过程：**1、先获取上一个对象的锁：保证上一个字母不会被打印；2、 再获取当前对象锁：保证下一个字母不会被打印；3、唤醒当前对象上阻塞的线程：下一个字母可以被打印；4、阻塞上一个对象锁监听的线程：保证当前字母和上一个字母不会打印。

**代码：**

**public** **class** PrintABC **implements** Runnable {

**private** String name;

**private** Object prev;

**private** Object self;

**private** PrintABC(String name, Object prev, Object self) {

**this**.name = name;

**this**.prev = prev;

**this**.self = self;

}

@Override

**public** **void** run() {

**int** count = 10;

**while** (count > 0) {

// 先获取上一个对象的锁：保证上一个字母不会被打印

**synchronized** (prev) {

// 再获取当前对象锁：保证下一个字母不会被打印

**synchronized** (self) {

System.***out***.print(name);

count--;

// 唤醒当前对象上阻塞的线程：下一个字母可以被打印

self.notify();

}

**try** {

// 阻塞上一个对象锁监听的线程：保证当前字母和上一个字母不会打印

prev.wait();

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

}

}

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

Object a = **new** Object();

Object b = **new** Object();

Object c = **new** Object();

PrintABC pa = **new** PrintABC("A", c, a);

PrintABC pb = **new** PrintABC("B", a, b);

PrintABC pc = **new** PrintABC("C", b, c);

**new** Thread(pa).start();

Thread.*sleep*(100); // 确保按顺序A、B、C执行

**new** Thread(pb).start();

Thread.*sleep*(100);

**new** Thread(pc).start();

Thread.*sleep*(100);

}

}

##### 方法比较

###### sleep()

Thread类的静态方法

不释放对象锁：因为他是一个静态方法，不能操作对象锁

需要抛出异常

###### wiat()

Object的对象方法

释放对象锁

不需要抛出异常，但是nodify()方法要抛出异常

要在synchronized块儿中执行

### 线程数据传递

#### 构造函数传数据

创建线程对象的时候传递参数，所以当线程执行时数据已经到位。

参数很多时不方便

#### 变量和方法传递数据

构建完成线程对象后，通过public方法或者变量传递数据，数据赋值完成后再调用start方法

#### 回调函数传递数据

通过回调函数使线程内的数据在线程外被引用，既“传出”数据

### Synchronized关键字

#### 应用方式

修饰实例方法，作用于当前实例加锁，进入同步代码前要获得当前实例的锁

修饰静态方法，作用于当前类对象加锁，进入同步代码前要获得当前类对象的锁

修饰代码块，指定加锁对象，对给定对象加锁，进入同步代码库前要获得给定对象的锁

#### 底层实现

每个对象有一个监视器锁（monitor），存放在对象头（对象在内存中的分布：对象头、实例数据、填充数据）。当monitor被占用时就会处于锁定状态，线程执行monitorenter指令时尝试获取monitor的所有权，过程如下：

1、如果monitor的进入数为0，则该线程进入monitor，然后将进入数设置为1，该线程即为monitor的所有者。

2、如果线程已经占有该monitor，只是重新进入，则进入monitor的进入数加1.

3、如果其他线程已经占用了monitor，则该线程进入阻塞状态，直到monitor的进入数为0，再重新尝试获取monitor的所有权。

## 线程池

解决线程很多时，频繁的创建和关闭线程过量耗时的问题。

将一个任务（实现了Runnable接口的对象）传递到ThreadPoolExecutor类的execute()方法中，任务里的内容就会在一个线程里执行，不需要新建线程。

### 常用变量

#### corePoolSize

核心池的大小（即线程池中的线程数目大于这个参数时，提交的任务会被放进任务缓存队列）

#### maximumPoolSize

任务量过大时在corePoolSize的数量基础上线程的最大数量，是线程池的一种补救措施

#### largestPoolSize

记录线程池中曾经有过的最大线程数量

### 线程池状态

#### RUNNING状态

当创建线程池后，初始时，线程池处于RUNNING状态；

#### SHUTDOWN状态

调用了ThreadPoolExecutor 类的shutdown()方法后；

此时线程池不能够接受新的任务，它会等待所有任务执行完毕

#### STOP状态

调用了ThreadPoolExecutor 类的shutdownNow()方法；

此时线程池不能接受新的任务，并且会去尝试终止正在执行的任务

#### TERMINATED状态

当线程池处于SHUTDOWN或STOP状态，并且所有工作线程已经销毁，任务缓存队列已经清空或执行结束后；

### 常用的4类线程池

#### newCachedThreadPool()

缓存型池子，先查看池中有没有以前建立的线程，如果有，就 reuse 如果没有，就建一个新的线程加入池中，线程数量不受控制

通常用于执行一些生存期很短的异步型任务

能 reuse 的线程，必须是 timeout IDLE 内的池中线程，缺省 timeout 是 60s,超过这个 IDLE 时长，线程实例将被终止及移出池

#### newFixedThreadPool(int)

newFixedThreadPool 与 cacheThreadPool 差不多，也是能 reuse 就用，但不能随时建新的线程。

其独特之处:任意时间点，最多只能有固定数目的活动线程存在，此时如果有新的线程要建立，只能放在另外的队列中等待，直到当前的线程中某个线程终止直接被移出池子。

从方法的源代码看，cache池和fixed 池调用的是同一个底层 池，只不过参数不同:

* + fixed 池线程数固定，并且是0秒IDLE（无IDLE）。
  + cache 池线程数支持 0-Integer.MAX\_VALUE(显然完全没考虑主机的资源承受能力），60 秒 IDLE 。

#### newScheduledThreadPool(int)

调度型线程池

这个池子里的线程可以按 schedule 依次延时执行，或周期执行

#### SingleThreadExecutor()

单例线程，任意时间池中只能有一个线程

### 任务的执行

#### Runnable

实现void run()方法，没有返回值

#### Callable

实现V call()方法，有返回值，返回值通过Future的get()方法获取

### 任务队列满时的处理机制

1. Abort策略：默认策略，新任务提交时直接抛出未检查的异常RejectedExecutionException，该异常可由调用者捕获
2. CallerRuns策略：为调节机制，既不抛弃任务也不抛出异常，而是将某些任务回退到调用者。不会在线程池的线程中执行新的任务，而是在调用exector的线程中运行新的任务
3. Discard策略：新提交的任务被抛弃
4. DiscardOldest策略：抛弃队列前段的任务，然后尝试提交新的任务

## 显示锁

相对于synchronized的隐形锁，jdk1.5之后新增了显示锁ReentrantLock。

可重入：一个线程获取锁之后，可以再次获取锁。每次获取锁时，保持计数加1

公平锁和非公平锁：

当一个线程锁被其他线程拥有（保持计数大于0）时，无论公平锁或者非公平锁该线程都会进入队列等待被唤醒（lock方法最终会调用thread的interrupt方法中断线程）；但是当锁恰好被释放（保持计数等于0）时：

公平锁：该线程依然会进入队列，按入队顺序等待被唤醒执行；

非公平锁：该线程会直接获取锁，不入队排列。

#### 条件对象Condition

避免死锁的发生

通过ReentrantLock的newCondition获取，调用Condition的await()方法会阻塞当前线程，并释放锁，调用signal()或者signalAll()方法重新激活因这一条件而等待的线程。

#### 锁测试与超时

在lock()之前通过显示锁的tryLock()方法判断申请锁时是否有其他线程占有锁，避免阻塞。尝试获得锁时可以设置线程的阻塞时间

#### 读写锁

ReentrantReadWriteLock可以抽取出读锁和写锁，读锁线程可共享、写锁互斥

#### 线程局部变量

通过Threadlocal类可以为每个线程单独构造变量

#### Volatile域

避免虚拟机的重排序

保证线程可见性

## 线程安全集合

### ConcurrentHashMap

分段锁机制

内部采用了一个叫做Segment的结构，其内部维护了一个链表数组。

查找过程：两次hash定位，先定位到Segment，再在Segment内部的链表数组中定位到链表节点

### CopyOnWriteArrayList和CopyOnWriteArraySet

读写分离，读可以并发，写同步。可以用于读多写少的场景

#### CopyOnWrite容器

写时复制容器：

既写数据时不直接添加到当前容器，而是复制当前容器到一个新容器中，写完后再将原容器的引用指向新的容器。这样就读写分离，读数据时就可以并发的去读取，写时做同步处理。

#### 缺点

内存占用、数据有效性

## 阻塞队列

是一个支持两个附加操作的队列take(),put(e)。这两个附加的操作是：在队列为空时，获取take()元素的线程会等待队列变为非空，当队列满时，存储put(e)元素的线程会等待队列可用。这两个操作的阻塞是通过条件锁实现的。

常用于生产者和消费者的场景，生产者是往队列里添加元素的线程，消费者是从队列里拿元素的线程。阻塞队列就是生产者存放元素的容器，而消费者也只从容器里拿元素。

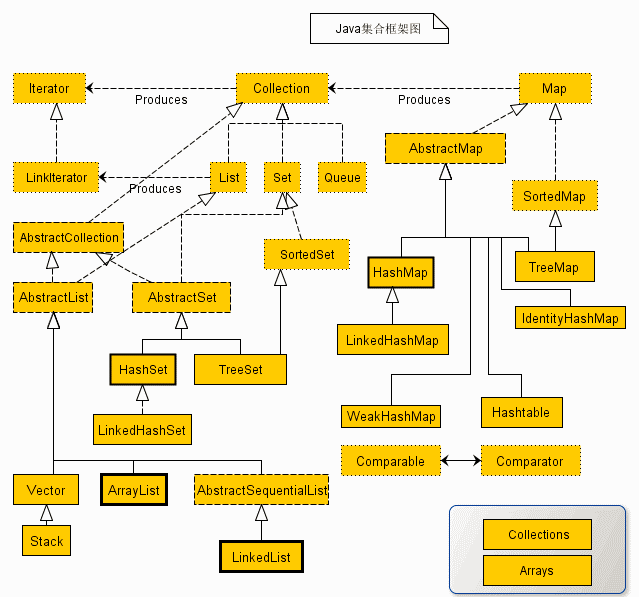
# 集合框架

<http://www.importnew.com/16658.html>

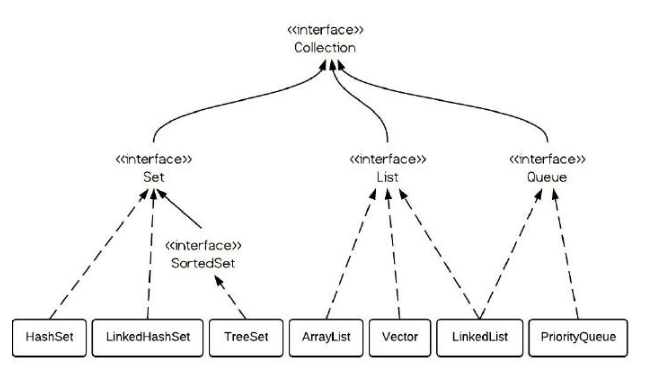
<https://www.cnblogs.com/xiaoxi/p/6089984.html>

线程安全的集合：hashtable、vevtor

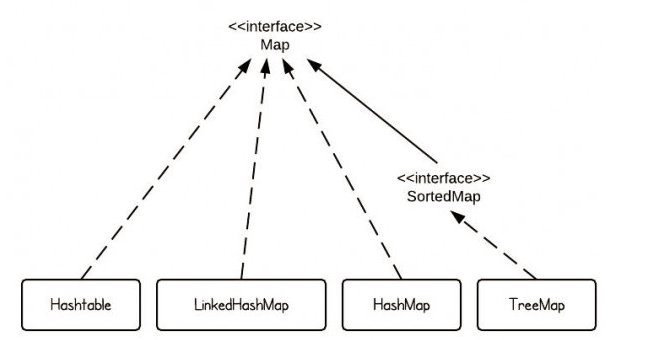
集合总体框架图：



Collection接口：



Map接口：



## Collection接口

### Set接口

表示不允许有重复元素的集合

#### Hashset

内部数据结构为一个为HashMap，迭代无序，线程不安全

#### LinkedHashSet

Hashset的子类，内部数据结构为LinkedHashMap，迭代有序（插入顺序），线程不安全

#### TreeSet

内部数据结构为TreeMap，所以默认支持排序（同样也是按照自然顺序或者构造函数传进来的比较器机型比较排序），线程不安全

### List接口

表示允许有重复元素的集合

#### ArrayList

动态数组，默认长度为10

线程不安全

#### Vector

动态数组，默认长度为10

线程安全：直接对方法进行同步（被synchronized修饰），保证了线程安全，但是效率低

##### Stack

继承自Vector，实现一个线程安全的栈

#### Linkedlist

基于双向链表实现，所以其删除和插入的效率高，但是查询的效率低。可以被当作堆栈、队列或双端队列进行操作。线程不安全。

### Queue接口

#### LinkedList

LinkedList同时实现了List和Queue接口，可以当做队列处理

#### PriorityQueue

<https://www.cnblogs.com/yangming1996/p/6944400.html>

逻辑结构为完全二叉树，实际储存在数组里（完全二叉树的特性使其可以通过计算获得其父子以及兄弟节点的位置，因而不需要额外储存其和其他节点的连接关系，储存在数组即可节省开销）

优先：优先队列的入队和出队操作后都会调整数据位置，使其满足堆结构（小根堆）。所以每次出队的一定是最小元素，既最小最优先出队

## Map接口

### HashMap

<https://www.cnblogs.com/chengxiao/p/6059914.html>

http://www.importnew.com/7099.html

#### 数据结构

采用哈希表+链表结构实现，链表解决哈希冲突问题。

哈希表：本质为数组，通过哈希函数将当前元素关键字映射到数组的某个位置（将key哈希为数组下标）

Hashmap数组长度为2的次幂，可以减少哈希冲突

#### 储存、读取数据

储存：Key值hash得到哈希表中的位置将元素储存到该位置，如果发生哈希冲突则存到链表

读取：根据key值的hash值获取哈希表上的值hashcode&(length-1)，如果存在链表（哈希冲突）则通过key的equal方法逐一对比链表上的节点

#### 数组扩容

当实际键值对个数达到阈值或发生哈希冲突时，进行扩容：新建一个长度为之前数组2倍的新数组，然后将当前的Entry数组中的元素全部传输过去，过程比较耗时；扩容后数组长度翻倍，因此也比较消耗空间；

阈值=数组容量（默认16）\*负载因子（默认0.75）；

#### LinkedHashMap

LinkedHashMap是HashMap的子类，线程不安全

##### 和HashMap区别

HashMap是无序的，既迭代HashMap的顺序和插入的顺序不一致

LinkedHashMap是有序的

##### LinkedHashMap的有序实现

<https://www.cnblogs.com/xiaoxi/p/6170590.html>

通过额外维护一个双向链表保证元素迭代的顺序，该迭代孙旭可以是插入顺序或访问顺序

访问顺序：访问（get或put更新value）LenkedHashMap时，将访问的节点存到双向链表的末端。可以实现基于LRU算法的缓存，既最近最少使用算法缓存，当缓存满时最先清除最不常访问（位于双向链表前端）的数据

### HashTable

HashTabel的内部实现和HashMap相同。

#### 和HashMap的区别

##### 线程安全

HashTabel线程安全：内部直接对方法进行同步处理，方法被synchronize修饰

HashMap线程不安全

##### 父类不同

Hashtable继承自Dictionary类，而HashMap继承自AbstractMap类。但二者都实现了Map接口

##### Hash计算

hashtable的hash值直接使用对象的hashcode，hashmap重新计算hash值

##### 扩容

hashtable扩容为2倍加1，hashmap为2倍

##### 其他

hashtable保留了continue方法；hashtable的key和value都不允许为null

### TreeMap

TreeMap是一个排序的key-value集合、是通过红黑树实现的。

TreeMap是通过key值进行排序的，构造TreeMap时如果提供了比较器则用该比较器进行比较，否则根据key值的自然顺序进行排序；

红黑树：是特殊的二叉查找树，其每个节点都有属性表示他的颜色，既红和黑。通过其红黑属性的一些特性（具体参见http://www.cnblogs.com/skywang12345/p/3245399.html）是其通过不断的调整（旋转：左旋；右旋）达到接近平衡的状态，已保证其查找效率。

# 流

## 字节流和字符流

抽象类InputStream和OutputStream构成了基于一个字节的输入、输出字节流

抽象类Reader和Writer构成了基于两个字节的输入、输出Unicode字符流

## Nio和io

IO是面向流的，NIO是面向缓冲区的；

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

NIO的选择器允许一个单独的线程来监视多个输入通道，你可以注册多个通道使用一个选择器，然后使用一个单独的线程来“选择”通道。这种选择机制，使得一个单独的线程很容易来管理多个通道。

# 其他知识点

## 数据库

### 查询优化

那些可以过滤掉最大数量记录的条件必须写在WHERE子句的末尾。尤其是“主键ID=？”这样的条件

SELECT子句中避免使用 \*

索引

用union all 替换union

### 数据库事务隔离级别

1、幻想读：事务T1读取一条指定where条件的语句，返回结果集。此时事务T2插入一行新记录，恰好满足T1的where条件。然后T1使用相同的条件再次查询，结果集中可以看到T2插入的记录，这条新纪录就是幻想。**查询事务中插入**

2、不可重复读取：事务T1读取一行记录，紧接着事务T2修改了T1刚刚读取的记录，然后T1再次查询，发现与第一次读取的记录不同，这称为不可重复读。**查询事务中修改**

3、脏读：事务T1更新了一行记录，还未提交所做的修改，这个T2读取了更新后的数据，然后T1执行回滚操作，取消刚才的修改，所以T2所读取的行就无效，也就是脏数据。**修改后的事务被另一事务读取后回滚**

为了处理这些问题，SQL标准定义了以下几种事务隔离级别

READ UNCOMMITTED 幻想读、不可重复读和脏读都允许。

READ COMMITTED 允许幻想读、不可重复读，不允许脏读

REPEATABLE READ 允许幻想读，不允许不可重复读和脏读

SERIALIZABLE 幻想读、不可重复读和脏读都不允许

Oracle数据库支持READ COMMITTED 和 SERIALIZABLE这两种事务隔离级别。所以Oracle不支持脏读

SQL标准所定义的默认事务隔离级别是SERIALIZABLE，但是Oracle 默认使用的是READ COMMITTED

事务四大特性：原子性、隔离性、一致性、持久性

## Spring

### 注入DI

#### 传统依赖

每个对象负责管理它所依赖的对象的引用，导致高度耦合和难以测试的代码

耦合的两面性：

紧密的耦合代码难以测试、难以复用、难以理解，并且容易出现打地鼠的bug（报销就是啊）

耦合是必须的，完全不耦合的代码什么也做不了。所以符合耦合是必须的，但管理要谨慎。

#### DI

松耦合：

通过DI，对象的依赖关系由系统负责协调。对象无需自行创建和管理他们的依赖关系，依赖关系将自动注入到需要他们的对象中，实现了松耦合。

如果一个对象只通过接口（而不是具体实现或初始化过程）来表明依赖关系，那么这种依赖就能够在对象本身毫不知情的情况下用不同的具体实现进行替换。

控制反转IOC（个人理解，有待考证）：

传统依赖有对象控制，例如：对象a创建对象b则进行b的相关操作，创建c则进行c的相关操作。但通过DI，对象的操作由注入的对象决定，既注入的是b则进行b的操作，注入的是c则进行c的操作，既所谓控制反转

##### DI实现

装配：创建应用组件之间协作的行为

装配方式：xml文件配置、java描述（java注解）

通过应用上下文装载bean的定义并把他们组装起来，例如通过ClassPathXmlApplicationContext装配xml文件中配置的bean。

组装的方式可以有构造器或者setter（我们的程序主要是这种方式），底层都是通过反射实现。

### 切面编程AOP

使业务组件更高内聚，更加关注自身业务

在配置文件中将公共业务声明为界面<aop:config></aop:config>

切面中定义切点（需要调用该公共业务的地方，例如通过正则表达式定位到某个类的某个方法），还可声明切点前置通知或者后置通知（既在切点函数前或者后或者前后执行切面中的方法）。

切面本身也是个bean在<aop:config>元素中被引用

Spring AOP织入方式：运行时在切点出通过代理织入切面代码

### Bean作用域和生命周期

#### 作用域

1. singleton：单例模式，Spring IoC容器中只会存在一个共享的Bean实例，无论有多少个Bean引用它，始终指向同一对象。Singleton作用域是Spring中的缺省作用域，也可以显示的将Bean定义为singleton模式，配置为：
   * <bean id="userDao" class="com.ioc.UserDaoImpl" scope="singleton"/>
2. prototype:原型模式，每次通过Spring容器获取prototype定义的bean时，容器都将创建一个新的Bean实例，每个Bean实例都有自己的属性和状态，而singleton全局只有一个对象。根据经验，对有状态的bean使用prototype作用域，而对无状态的bean使用singleton作用域。
3. request：在一次Http请求中，容器会返回该Bean的同一实例。而对不同的Http请求则会产生新的Bean，而且该bean仅在当前Http Request内有效。
   * <bean id="loginAction" class="com.cnblogs.Login" scope="request"/>,针对每一次Http请求，Spring容器根据该bean的定义创建一个全新的实例，且该实例仅在当前Http请求内有效，而其它请求无法看到当前请求中状态的变化，当当前Http请求结束，该bean实例也将会被销毁。
4. session：在一次Http Session中，容器会返回该Bean的同一实例。而对不同的Session请求则会创建新的实例，该bean实例仅在当前Session内有效。
   * <bean id="userPreference" class="com.ioc.UserPreference" scope="session"/>,同Http请求相同，每一次session请求创建新的实例，而不同的实例之间不共享属性，且实例仅在自己的session请求内有效，请求结束，则实例将被销毁。
5. global Session：在一个全局的Http Session中，容器会返回该Bean的同一个实例，仅在使用portlet context时有效。

#### 生命周期

1. 实例化一个Bean，也就是我们通常说的new

2. 按照Spring上下文对实例化的Bean进行配置，也就是IOC注入

3. 如果这个Bean实现了BeanNameAware接口，会调用它实现的setBeanName(String beanId)方法，此处传递的是Spring配置文件中Bean的ID

4. 如果这个Bean实现了BeanFactoryAware接口，会调用它实现的setBeanFactory()，传递的是Spring工厂本身（可以用这个方法获取到其他Bean）

5. 如果这个Bean实现了ApplicationContextAware接口，会调用setApplicationContext(ApplicationContext)方法，传入Spring上下文，该方式同样可以实现步骤4，但比4更好，以为ApplicationContext是BeanFactory的子接口，有更多的实现方法

6. 如果这个Bean关联了BeanPostProcessor接口，将会调用postProcessBeforeInitialization(Object obj, String s)方法，BeanPostProcessor经常被用作是Bean内容的更改，并且由于这个是在Bean初始化结束时调用After方法，也可用于内存或缓存技术

7. 如果这个Bean在Spring配置文件中配置了init-method属性会自动调用其配置的初始化方法

8. 如果这个Bean关联了BeanPostProcessor接口，将会调用postAfterInitialization(Object obj, String s)方法

注意：以上工作完成以后就可以用这个Bean了，那这个Bean是一个single的，所以一般情况下我们调用同一个ID的Bean会是在内容地址相同的实例

9. 当Bean不再需要时，会经过清理阶段，如果Bean实现了DisposableBean接口，会调用其实现的destroy方法

10. 最后，如果这个Bean的Spring配置中配置了destroy-method属性，会自动调用其配置的销毁方法

## Redis

### 数据类型

**1.String（字符串）**

string类型是二进制安全的。意思是redis的string可以包含任何数据。比如jpg图片或者序列化的对象 。

**2.Hash（哈希）**

Redis hash 是一个键值对集合。

**3.List（列表）**

Redis 列表是简单的字符串列表，按照插入顺序排序。你可以添加一个元素导列表的头部（左边）或者尾部（右边）。  
它的底层实际是个链表

**4.Set（集合）**

Redis的Set是string类型的无序集合。它是通过HashTable实现实现的，

**5.zset(sorted set：有序集合)**

Redis zset 和 set 一样也是string类型元素的集合,且不允许重复的成员。  
不同的是每个元素都会关联一个double类型的分数。  
redis正是通过分数来为集合中的成员进行从小到大的排序。zset的成员是唯一的,但分数(score)却可以重复。

### Redis为什么快

纯内存数据库：相对于读写磁盘，读写内存的速度就不是几倍几十倍了

多路复用IO：“多路”指的是多个网络连接，“复用”指的是复用同一个线程。采用多路 I/O 复用技术可以让单个线程高效的处理多个连接请求（尽量减少网络IO的时间消耗）