## 2．线程安全

线程安全定义：当多个线程访问某个类时，不管运行时环境采用【何种调度方式】或者这些进行如何交替执行，并且在主调代码中不需要【任何额外当同步或协同】，这个类都能表现出【正确都行为】，那么这个类时线程安全都。

线程安全特点：

* 原子性：提供类互斥访问，同一时刻只能有一个线程对它进行访问；
* 可加性：一个线程对住内存对修改可以及时对被其他线程观察到；
* 有序性：指令重排序对结果没有影响。

### 2.1 AtomicXXX

AtomicXXX:CAS乐观锁，全名Compare And Swap 比较交换。主要集中表在Unsafe.java类中的CompareAndSwapInt()／CompareAndSwapLong()等等等方法上。

|  |
| --- |
| 例如：执行【2+1】操作；  public final native boolean getAndAddInt(Object var1, long var2, Int var4){  //native修改对方式时jni调用  int var5;  do{  var5 = this.getIntVolatile(var1, var2);  //获取【底层/期望】值；  }while(!this.compareAndSwapInt(var1, var2, var5, var5+ var4));  //循环比较【当前】值是否与【底层/期望】值相同，如果相同择执行var5+var4【2+1】操作；如果不相同则继续循环获取底层对值。  } |
| 缺点：   1. 因为这种循环比较多情况，可见这是一种乐观锁。但是带来影响是在高并发的情况下，会导致某一个线程拿到的当前值与期望值永远不相等，进入死循环。 2. CAS引起ABA的问题。比如：当前t1线程的当前值是A，获取到的底层值也是A，但是事实上底层的值是t2线程奖B改成了A。这个就与AtomicXXX对设计不符。这个时，可以通过给修改的值添加版本号的方式解决。例如：AtomicStampReference类采用这中方式。   优点：   1. AtomicBoolean类可以来解决一段代码只执行一次的问题。 |

AtomicInteger代码如下：

|  |
| --- |
| **publicclass** AtomicXXXTest {  //请求总数  **publicstaticint***clientTotal* = 5000;  //同步并发执行对线程数  **publicstaticint***threadPool* = 200;  **publicstatic** AtomicInteger *count* = **new** AtomicInteger();  **publicstaticvoid** main(String[] args) **throws** Exception{  ExecutorService executorService = Executors.*newCachedThreadPool*();  **final** Semaphore semaphore = **new** Semaphore(*threadPool*);  **final** CountDownLatch countDownLatch = **new** CountDownLatch(*clientTotal*);  **for**(**int**i = 0; i<*clientTotal*; i++){  executorService.execute(()->{  **try** {semaphore.acquire();  *add*();  semaphore.release();  } **catch** (Exception e) {  e.printStackTrace();  }  countDownLatch.countDown();  });  }  countDownLatch.await();  executorService.shutdown();  System.***out***.println(*count*.get());  }  **privatestaticvoid** add(){  *count*.getAndIncrement();//先获取值再加1；  // count.incrementAndGet();//先加1再获取  }  } |

AtomicBoolean测试代码如下：【多线程情况下代码只执行一次】

|  |
| --- |
| **publicclass** AtomicBooleanTest {  //请求总数  **publicstaticint***clientTotal* = 5000;  //同步并发执行对线程数  **publicstaticint***threadPool* = 200;  **publicstatic** AtomicBoolean *isHappened* = **new** AtomicBoolean(**false**);    **publicstaticvoid** main(String[] args) **throws** Exception{  ExecutorService executorService = Executors.*newCachedThreadPool*();  **final** Semaphore semaphore = **new** Semaphore(*threadPool*);  **final** CountDownLatch countDownLatch = **new** CountDownLatch(*clientTotal*);  **for**(**int**i = 0; i<*clientTotal*; i++){  executorService.execute(()->{  **try** {semaphore.acquire();  *add*();  semaphore.release();  } **catch** (Exception e) {  e.printStackTrace();  }  countDownLatch.countDown();  });  }  countDownLatch.await();  executorService.shutdown();  System.***out***.println(*isHappened*.get());  }  **privatestaticvoid** add(){  **if**(*isHappened*.compareAndSet(**false**, **true**)){  System.***out***.println("yes");  }  }  } |

* 1. 原子性

Synchronized锁；

作用域：

* 修饰代码块:根据写法不同进行区分，例如:Synchronized（this）{}，或着Synchronized（XXXClass.class）{}；
* 修改方法:所用于调用的对象；
* 修饰静态方法:作用于所有类;

Lock锁，常见于ReenTrantLock锁,需要自己进行锁定／解锁代码快；

Atomic

* 1. 可见性

可见性，就是一个线程对主内存的修改，能够及时的被其他的线程观察到。导致共享变量在线程间不可见的原因，就是变量更新后的值没有在工作内存与主村间及时更新。

JMM关于synchronized的两条规定：

* 线程解锁前，必须把共享变量的最新值刷新到主内存；
* 线程加锁时，将清空工作内存中共享变量的值，从而使用共享变量时需要从主内存中重新读去最新的值。（注意：加锁解锁时同一把锁）

volatile通过加入内存屏障和禁止重排序优化来实现：

* 对volatile变量写操作时，会在写操作后加入一条store屏障指令，将本地内存中的共享变量值刷新到主内存；
* 对volatile变量读操作时，会在读操作前加入一条load屏障指令，从主内存中读取共享变量。
* volatile修饰的对象，读时一定是最新的值。不具有原子性。

Volatile对应用场景，必须满足两个条件：第一，对变量的写操作不依赖当前值；第二该变量没出现在其他变量的式子中。所以volatile特别适合用作状态标记量。例如：

|  |
| --- |
| //线程1初始化  context1 = loadContext();  inited = **true**;  //线程2判断是否初始化完成  **while**(!inited){  sleep();  }  doSomethingWithConfi(context); |

* 1. 有序性

JMM中，允许编译器和处理器对指令进行冲排序，但重排序不会影响单线程对执行，却会影响到多线程并发执行的正确性。

Java中可以通过volatile／synchronized／lock来保证有序性。

* 有序性-happens-before原则

程序次序规则：一个线程内，安装代码顺序，书写在前面的操作先行发生于书写在后面的操作；

* 锁定规则：先释放锁，再获取锁原则。
* volatile变量规则：对一个变量的写操作先行发生于后面对这个变量的读操作；
* 传递规则：如果操作A先行发生于操作B，而操作B又先行发生于操作C，那么可以得出操作A先行发生于操作C。
* 线程启动规则：Thread对象的start()方法先行发生于此线程的每一个动作；
* 线程中断规则：对现场interrupt()方法的调用先行发生于被中断线程等代码检测到中断事件的发生；
* 线程终结规则：线程中所有的操作都先行发生于线程都终止检测，我们可以通过thread.join()方法结束，thread.isAlive()的返回值手段检测到线程已经终止执行；
* 对象终结规则：一个对象的初始化完成先行发生于它都finalize()方法都开始。

## AQS【Abstract Queue Synchronized】

### 3.1 CountDownLatch 计数器

应用场景：计数器、线程执行超时

注意点：

* 指定一个CountDownLatch的计数值；
* 每一个线程的执行的注意指定countDown()方法；
* await()等待执行完成。

### Semaphore 信号量

1. 应用场景：指定有限大小的链表；提供仅能有限使用的资源（比如数据库的最大连接数，数据库提供的最大连接数是20，而客户端的访问数远远大于当前的访问数，可以通过semaphore来控制并发数）；
2. 实现并发访问的控制

acquire()获取许可 acquire(n)获取n个许可

release()释放许可 release(n)释放n个许可

tryAcquire()尝试获取许可

tryAcquire(n, timeout, TimeUtil.SECOND) 尝试获取n个许可，获取不到等timeout时间。

### CyclicBarrier

CountDownLatch只能使用一次。

CyclicBarrier可以通过reset重复使用。

### ReentrantLock

ReentrantLock与Synchronized的区别：

* 可重入性：
* 锁的实现：

Synchronized是通过JVM来实现的，很难看得到源代码的。相当于操作系统来实现的；

ReentrantLock是通过JDK来实现的，是用户自己来实现的。

* 性能的区别：

在Synchronized引入了偏向锁和自旋锁以后，在性能上与ReetrantLock差不多，其实Synchronized的优化感觉上跟CAS很类似，都是试图在用户态把加锁状态解决，避免了内核态的线程阻塞。在相同的情况下，官网更推荐Synchronized，因为写法更加容易，不用luck，unlock。

* 功能区别：
* 便利性：Synchronize使用方便，是通过编译器锁的加锁和释放的；ReentrantLock需要用户进行Lock与unLock，并且在代码写法上为了避免死锁状态，通常采用try{}finally{}，在finally中释放锁。
* 锁的颗粒度和灵活度：ReentrantLock优于Synchronized。

ReentrantLock独有的功能：

1. ReentrantLock可指定是公平锁还是非公平锁；而Synchronized是非公平锁。所谓公平锁就是先等待的先获得锁。
2. 提供了一个Condition类，可以分组唤醒需要唤醒的线程。而Synchronized要么随机唤醒一个线程，要么唤醒全部线程。
3. 提供能够中断等待锁的线程的机制，lock.lockInterruptibly()

除非要用到ReentrantLock中以上3种功能之一的情况，都建议Synchronized。总之，ReentrantLock包含了Synchronized所有功能。

ReentrantReadWriteLock

里面包含了读锁和写锁，没有读写锁的时候才能写。可以实现悲观读写。当读操作很多的时候，写操作会一直等待。出现写饥饿。

|  |
| --- |
| **package** com.zuk.test.aqs.lock;  **import** java.util.Map;  **import** java.util.Set;  **import** java.util.TreeMap;  **import** java.util.concurrent.locks.Lock;  **import** java.util.concurrent.locks.ReentrantReadWriteLock;  **public** **class** ReentrantLockTest {  **private** **final** Map<String, Data> map = **new** TreeMap<String, ReentrantLockTest.Data>();  **private** **final** ReentrantReadWriteLock lock = **new** ReentrantReadWriteLock();  **private** **final** Lock readLock = lock.readLock();  **private** **final** Lock writeLock = lock.writeLock();  **public** Data get(String key){  readLock.lock();  **try**{  **return** map.get(lock);  } **finally**{  readLock.unlock();  }  }  **public** Set<String> getAllKeys(){  readLock.lock();  **try** {  **return** map.keySet();  } **finally**{  readLock.unlock();  }  }  **public** Data put(String key, Data data){  writeLock.lock();  **try** {  **return** map.put(key, data);  } **finally**{  writeLock.unlock();  }  }  **public** **static** **void** main(String[] args) {}  **class** Data{ }  } |

StampledLock

锁的总结：Synchronized是通过jvm层实现，不但可以通过一些工具监控Synchronized的锁定，而且当在程序中运行出现异常，jvm也是自动释放锁定。ReentrantLock是对象层面的锁定，一般通过try…finally的方式，保证锁一定释放。

### Condition

|  |
| --- |
| **package** com.zuk.test.aqs.lock;  **import** java.util.concurrent.locks.Condition;  **import** java.util.concurrent.locks.Lock;  **import** java.util.concurrent.locks.ReentrantLock;  **public** **class** ConditionTest {  **static** Lock *reentrantLock* = **new** ReentrantLock();  **static** Condition *condition* = *reentrantLock*.newCondition();  **public** **static** **void** main(String[] args) {  **new** Thread(()->{  *reentrantLock*.lock();  **try**{  System.***out***.println("wait signal.");  *condition*.await();  } **catch**( Exception e ){    } **finally**{  System.***out***.print("get signal.");  *reentrantLock*.unlock();  }  }).start();    **new** Thread(() -> {  *reentrantLock*.lock();  System.***out***.println("get lock");  **try**{  Thread.*sleep*(5000);  }**catch**(Exception e){    }**finally**{  *condition*.signalAll();  System.***out***.println(" send signal. ");  *reentrantLock*.unlock();  }  }).start();  }  } |

这个例子的协助过程是节点在AQS等待队列和Condition等待队列来回移动来实现的，Condition作为一个条件类，很好维护了一个等待信号的队列，并在适时的时候将节点加入到AQS等待队列中，实现唤醒操作。

### 3.6 FutureTask

创建线程有两种方式，继承thread和实现Runnable接口，但是这两种方式都不能获取线程执行的结果。从java1.5后就提供了callable和future，通过他们可以再任务执行完毕之后得到任务的执行结果。

Callable和Runnable接口的对比：

Callable的方法是call();

Future接口

FutrueTask类

|  |
| --- |
| **package** com.zuk.test.aqs.future;  **import** java.util.concurrent.Callable;  **import** java.util.concurrent.ExecutorService;  **import** java.util.concurrent.Executors;  **import** java.util.concurrent.Future;  **public** **class** FutureTest {  **static** **class** MyCallable **implements** Callable<String>{  @Override  **public** String call() **throws** Exception {  System.***out***.println("do something.");  Thread.*sleep*(3000);  **return** "helloWorld.";  }  }  **public** **static** **void** main(String[] args) **throws** Exception {  ExecutorService exec = Executors.*newCachedThreadPool*();  Future<String> future = exec.submit(**new** MyCallable());  System.***out***.println("do something in main.");  Thread.*sleep*(1000);  String res = future.get();  System.***out***.println("res:" + res);  }  } |

FutrueTask的使用

|  |
| --- |
| **package** com.zuk.test.aqs.future;  **import** java.util.concurrent.Callable;  **import** java.util.concurrent.ExecutionException;  **import** java.util.concurrent.FutureTask;  **public** **class** FutrueTaskTest {  **public** **static** **void** main(String[] args) **throws** Exception {  FutureTask<String> futureTask  = **new** FutureTask<String>(**new** Callable<String>() {  @Override  **public** String call() **throws** Exception {  System.***out***.println("do something.");  Thread.*sleep*(3000);  **return** "helloWorld.";  }  });  **new** Thread(futureTask).start();  System.***out***.println("do something in mian.");  String res = futureTask.get();  System.***out***.println("res:" + res);  }  } |

### 3.7 BlockQueue阻塞队列

### 3.8 线程池

1. new thread弊端

* 每次 new thread新建对象，性能差；
* 线程缺乏统一管理，可能无限制的新建线程，相互竞争，有可能占用过多系统资源导致死机或者OOM；
* 缺少更多功能，如更多执行，定期执行，线程中断。

1. 线程池的好

* 重用存在的线程，减少对象创建，消亡的开销，性能佳
* 可有效控制最大并发线程数，提高系统资源利用率，同时可以避免过多资源竞争，避免阻塞。
* 提供定时执行，定期执行，单线程，并发数控制等功能。

1. 核心参数

* corePoolSize：核心线程数；
* maxumumPoolSize：最大线程数
* blockqueue：阻塞队列，存储等待执行的任务，很重要，会对线程池运行过程产生重大影响。
* 指定队列满了之后的拒绝策略。
* 以上参数的关系：如果运行的线程数小于核心线程数，直接创建新线程来完成任务 ；当运行的线程数大于等于核心线程数且小于最大线程数，同时阻塞队列满的时，才创建新的任务，如果列队没有满，则放入队列。如果运行线程数据大于最大线程数，且队列已经满了，则根据拒绝策略处理任务。总之，判断顺序就是核心线程数、阻塞队列、最大线程数。

1. 其他参数

* KeepAliveTime：线程没有任务执行时最多保持多久时间中止；
* Unit: keepAliveTime的时间单位；
* ThreadFactory：线程工厂，用来创建线程；
* RejectHandler：当拒绝处理任务时的策略：直接抛出异常（默认），丢弃队列中最靠前的任务；直接丢弃当前任务。

1. 线程池的状态：
2. 主要方法：

execute():提交任务，交给线程池执行；

submit()；提交任务，能够返回结果，execute+future

shutdown():关闭线程池，等待任务执行完成，不再接受新的任务；

shutdownNow()；关闭线程池，不等待任务执行完成

监控方法：

getTaskCount()；线程池已执行和未执行的任务总数；

getCompleteTaskCount();已完成的任务数量；

getPoolSize()；线程池当前的线程数量；

getActiveCount();当前线程池中正在执行任务的线程数量

每隔一分钟取出线程池中的状态。

1. 常用的线程池：

Executors.newCachedThreadPool();

Executors.newFixedThreadPool();

Executors.newScheduledThreadPool();

Executors.newSingleThreadPool();

1. 线程池-合理配置

CPU密集型任务，就需要尽量压榨CPU，参考值可以设为NCPU+1；

IO密集型任务，参考值可以设置为2\*NCPU;

1. a
2. a
3. a

## 多线程并发最佳实践

使用本地变量

使用不可变类

最小化锁的作用域范围：S=1/(1-a + a/n)

使用线程池的executor而不是new thread

宁可使用同步也不要使用线程的wait和notify

使用Blockqueue实现生产、消费模式

使用并发集合而不是加了锁的同步集合

使用semaphore创建有界访问：对于文件系统，数据库，socket的访问，要设置有界访问

宁可使用同步代码块，也不使用同步方法。

## HashMap和ConcurrentHashMap

### 5.1 初始参数

* hashMap创建时的初始容量默认是16
* 加载因子0.75
* 当hashMap中的容量与加载因子的乘积到达12时就进行扩容，调用resize方法进行扩容，将容量扩展1倍。
* JDK1.8中红黑树的转换是链长度是8时进行转换。

### 5.2 寻址

Key的哈希与数组长度进行取模操作。在计算机中取模的代码远远超过位操作的代价，所以hashmap要求数组的长度必须是2的n次方。2的n次方减1的与运行来代码取模计算。

## 提高并发性

### 6.1 服务器扩展

### 6.2 数据库扩展根据读操作、写操作

### 6.3 缓存

为了提高命中率，缓存主要适用于读多写少的操作。清楚策略，FIFO，LFU（最少使用策略），LRU（最近使用策略），过期时间，随机。

### 6.4 缓存的粒子要小

### 6.5 Memcache

不是一个分布式集群，而是单个的缓存服务器，具体调用那台服务器的分布式路由具体在客户端进行计算。

Memache在内存中的存储方式是将内存划分为一个固定大小的chunk。

### 6.6 Redis

数据类型：String/list/set/hash/sorted set

### 6.7 高并发场景下缓存常见问题

* 缓存一致性；

与数据库中的数据保持一直，这就比价依赖缓存的过期，更新策略。

* 缓存并发问题；

解决方案：上并发锁。

* 缓存穿透问题；

解决方案：缓存一个空对象，避免请求穿透到数据上。

* 缓存雪崩问题

解决方案：避免缓存同时超时释放

常见提问：

1. Spring主要包含了IOC（控制反转），DI（依赖注入）、AOP（面向切面编程）。
2. 如何理解控制反转？

举例：对象a依赖对象b，当对象a需要使用b对象的时候，就必须主动去创建b对象。但是当引入了IOC容器后，a不在需要主动去创建b，而是通过IOC容器去创建b对象，并注入到a中。a对b对象的创建由主动行为变成了被动行为，因此叫做控制反转。

1. 如何理解DI

DI（依赖注入）是实现控制反转的一种设计模式，依赖注入就是将实例变量传入到一个对象中去。

1. Spring中用到的设计模式

* 工厂类设计模式：spring使用beanFactory或者applicationContext创建bean类；
* 单例模式：scope的作用域，单例模式 singleton，多例模式prototype
* 代理模式：AOP模式
* 观察者模式：spring的事件驱动模型；spring的事件流程总结：

1. 定义一个事件：事件一个继承applicationEvent；
2. 定义一个事件监听者：实现applicationListener接口；
3. 使用事件发布者发布消息：可以通过applicationEventPublisher的publishEvent()方法发布消息。

* 适配器模式：

适配器模式将一个接口转换成客户希望的另一个接口。

* Spring AOP中的适配器模式；
* Spring MVC中的适配器模式

在spring mvc中，dispatchServlet（前端控制器）根据请求的url调用HandlerMapping（处理器映射器），映射器根据请求找到对应的controller控制器后，开始由HandlerAdapter适配器处理。HandlerAdapter作为期望接口，具体的适配器实现类用于对目标类进行适配，controller作为需要适配的类。

为什么要在springmvc中使用适配器模式？因为springmvc中的controller种类众多，不同类型的controller通过不同的方法来对请求进行处理。如果不利用适配器模式的话，DispatchServlet直接获取对应类型的Controller，需要自行来判断，像下面这段代码一样：

|  |
| --- |
| if(mappedHandler.getHandler() instance of MultiActionController){  } else if(….){  } |

## HashMap分析

### 7.1 hashMap扩容

hashMap的默认大小是16，扩展因子是0.75，换言之当存储容量达到12时出现扩容。为什么hashmap的初始大小和扩容的大小都是2的幂次，因为在hash在求模运算的效率比移位运算的效率低。

扩容步骤：

* 当hashmap中的元素个数超过数组大小与加载因子的乘积时就触发扩容机制。第一次扩容是因为数组的默认大小是16，加载因子是0.75，那么当hashmap的大小达到12的时候就会进行扩容，扩大一倍。
* 将原来的数据重新计算写入到新的hashmap中
* JDK1.7、JDK1.8中hashmap的差别，在jdk7采用的是链表，jdk8链在到达8时会转成红黑树。

### 7.2 hashmap的安全性分析

Hashmap的不安全表现在put出入元素时超出了容量范的时会出现扩容，将原数组的数组hash到新的数组中，在多线程情景下，存在其他线程在put操作，如果hash值相同，可能出现在同一个链表中出现闭环，那么在get的时候就会出现死循环。

### 7.3 hashTable的安全性分析

Hashtable是线程安全，这是因为hashtable的所有操作方法都加上了Syncronize修饰，如此线程是安全，效率是最低的。

### 7.4 concurrentHashMap分析

ConcurrentHashMap的数据结构是由一个segement数组与多个hashEntry组成。

## 分库分表：

### 8.1 分库分表类型：

垂直分库：根据业务不同，将不同业务的表创建在不同的库中。

垂直分表：将表字段创建在不同的表中。

|  |
| --- |
| 垂直分库垂直分表一般在项目开始时就进行区分。垂直分表一般根据常用字段、非常用字段，text、blob字段类型进行拆分。带来的好处是降低了表中行锁的概率（主要表现在update操作）。 |

水平分库：主要是将行数据分布到不同的库中，从而可以降低一个数据库（服务器）的访问压力。

水平分表：主要是将行数据分布到同一个数据库中不同的表中。能够小幅度提升性能，仅作为水平分库的一个补充

|  |
| --- |
| 分库分表带来的好处主要表示在，表锁，行锁，查询效率；服务器的CPU、内存、IO上。 |

一般在系统设计之初，就应该根据业务来设计垂直分库，垂直分表方案，在访问压力不是特别大的情况下，首先考虑缓存、读写分离，索引等技术。如果数据量极大，且数据持续增长，再考虑水平分库分表方案。

### 8.2 分库分表的优缺点

优点：分库分表有效解决了单表单库带来的性能瓶颈和压力，突破网络IO，硬件资源，连接数的瓶颈。

缺点：

* **事务一致性**；
* **跨节点关联查询**：分库分表后的数据可能无法做关联查询，因为数据不在同一个库，甚至不在同一个服务器。就得将关联查询，拆分为多次查询，再拼装数据。
* **跨节点分页、排序函数**：跨节点多库进行查询时，limit分页、order by排序等问题会变得异常复杂。需要将各节点的数据进行排序并返回，然后将不同节点返回的结果集再进行汇算排序。
* **主键避重：**自分库分表环境中，数据自带的自增主键将无用武之地。需要单独设计全局主键，以避免跨库主键的重复问题。
* **公共表：**实际业务中参数表、字典表是数据量小，变化小，并且属于高频联合点查询的依赖表。可以将这类表在每个数据库中都保存一份，所有公共表的修改都分发的所有库中。

## Springboot-start开发【步骤】

### 9.1 添加start开发的依赖

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <project xmlns="http://maven.apache.org/POM/4.0.0"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">  <modelVersion>4.0.0</modelVersion>  <groupId>org.walker.planes</groupId>  <artifactId>date-format-spring-boot-starter</artifactId>  <version>1.0-SNAPSHOT</version>  <dependencies>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-autoconfigure</artifactId>  <version>2.3.0.RELEASE</version>  </dependency>  </dependencies>  </project> |

### 9.2 创建配置信息实体类

在开发springboot项目的时候，有时候需要在application.properties配置中配置项目的修改和开关。

配置方法：创建一个配置文件的映射类，类中的成员变量对应application.properties的配置项目。如下：

|  |
| --- |
| import org.springframework.boot.context.properties.ConfigurationProperties;  /\*\*\* 配置信息实体类 \*/  **@ConfigurationProperties("formatter")**  public class **DateFormatProperties** {  private String pattern = "yyyy-MM-dd HH:mm:ss";  // 忽略 getter setter  } |
| 其中**@ConfigurationProperties("formatter")**注解的作用是表示以formatter为前缀的配置项映射为实体类 |

### 9.3 配置类

配置类的作用是将核心功能类注入到IOC容器中，使得start被项目引入时，就能完成自动装配。

|  |
| --- |
| import org.springframework.boot.autoconfigure.condition.ConditionalOnProperty;  import org.springframework.boot.context.properties.EnableConfigurationProperties;  import org.springframework.context.annotation.Bean;  import org.springframework.context.annotation.Configuration;  ​  import java.text.SimpleDateFormat;  @Configuration  @EnableConfigurationProperties(DateFormatProperties.class)  @ConditionalOnProperty(prefix = "formatter", name = "enabled", havingValue = "true")  public class DateFormatConfiguration {  private **DateFormatProperties** dateFormatProperties;  public DateFormatConfiguration(**DateFormatProperties** dateFormatProperties) {  this.dateFormatProperties = dateFormatProperties;  }  ​  @Bean(name = "myDateFormatter")  public SimpleDateFormat myDateFormatter() {  System.out.println("start to initialize SimpleDateFormat with pattern: " + dateFormatProperties.getPattern());  return new SimpleDateFormat(dateFormatProperties.getPattern());  }  } |
| * @Configuration 注解的作用是将 DateFormatConfiguration 类作为配置类注入容器; * @EnableConfigurationProperties 注解的作用是开启资源实体类的加载，也就是说开启配置文件映射为资源实体类的功能; * @ConditionalOnProperty 注解是开启条件化配置，只有在配置文件中的 formatter.enabled 属性的值为 true 时， starter 才会生效； |

### **9.4 指定自动装配**

指定DateFormatConfiguration为自动化装配类。在resources包中新建一个META-INFO文件夹，并创建spring.factories的文件，在文件中写入：

|  |
| --- |
| org.springframework.boot.autoconfigure.EnableAutoConfiguration=org.walker.planes.DateFormatConfiguration |

这行代码的意思是将DateFormatConfiguration 设置为自动装配类，在Springboot工程启动时会去扫描META-INFO文件夹中的spring.factories文件，并记载这个文件中指定的类，启动自动装配功能。

### 发布引用

通过mvn clean install/deploy方式打包发布到本地或者上传到nexus仓库中。