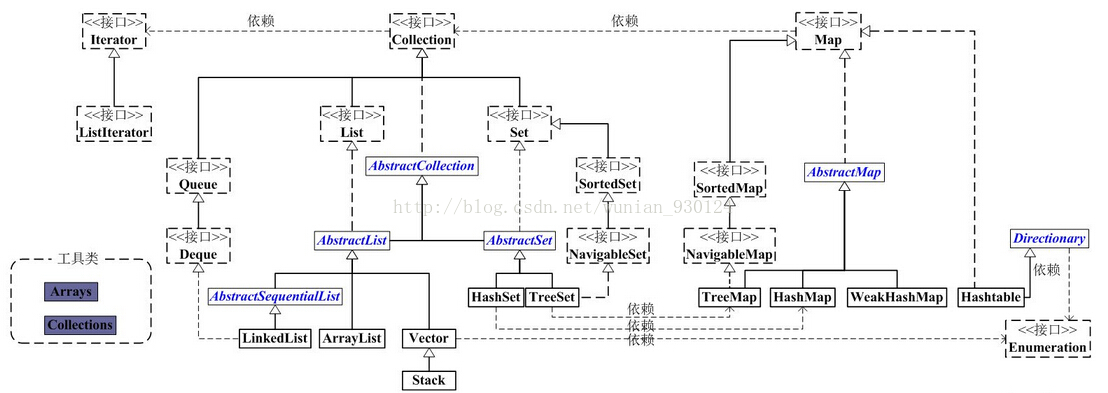
[Java工程师成神之路](http://www.hollischuang.com/archives/489)

# Java基础

## 集合



### Map

#### 类型

HashMap、HashTable、ConcurrentHashMap、LinkedHashMap、TreeMap

#### 特性

HashMap：初始大小16，阀值0.75，每次扩容增大到原来的2倍，底层以数组Entry<K,V>[]加链表的结构实现的，Entry里有指向下一个元素的Entry。

扩容将原数组中的Entry根据key重新计算hash值，并插入新链表的表头。

[HashMap死循环](http://pettyandydog.com/2016/08/28/HashMap_infinite_loop/#more)

线程不安全，原因就是：添加元素的方法不是同步的，两个线程同时调用addEntry()方法就会造成前一个线程写入的数据丢失（可能被覆盖），当数组满了要扩容的时候，复制数组有并发安全问题，支持null类型的key和value

HashTable：初始大小创建对象的时候指定，阀值0.75，底层基于旧对象Dictionary<K,V>实现，线程安全的，原因是put（）增加了代码同步锁synchronized，不支持null类型的key和value

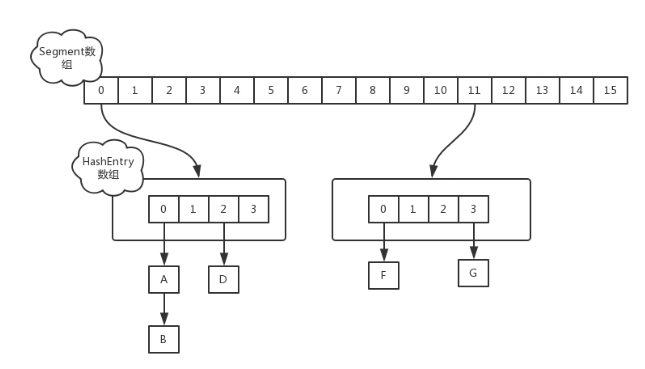
ConcurrentHashMap：初始大小16阀值0.75，采用的是hash分割方式，将对象分割成Segment对象，Segment（内部类）继承Lock锁类似于HashTable，Segment内部是HashEntry，用volatile关键字修饰的，不支持null类型的key和value

[深入并发包 ConcurrentHashMap](http://www.importnew.com/26049.html)

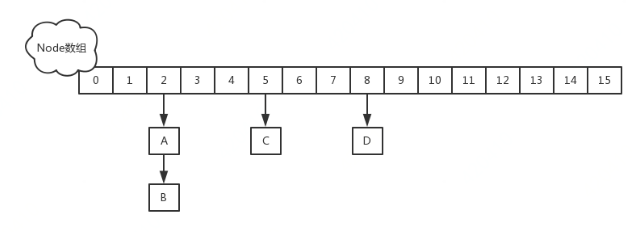
static final int *MAX\_SEGMENTS* = 1 << 16;

1的二进制左移16位，是2的16次方，<<与<<=，前者移位后临时变量，后者以为加赋值。 >>>将一个数的各二进制位无符号右移若干位,与运算符>>相同,移出的低位被舍弃,但不同的是最高位补0

static final class Segment<K,V> extends ReentrantLock implements Serializable {



**注意**：JDK1.8 相较于 JDK1.7 作了很大改动



**改进一：取消segments字段，直接采用transient volatile HashEntry<K,V>[] table保存数据，采用table数组元素作为锁，从而实现了对每一行数据进行加锁，进一步减少并发冲突的概率。**

**改进二：将原先table数组＋单向链表的数据结构，变更为table数组＋单向链表＋红黑树的结构。对于hash表来说，最核心的能力在于将key hash之后能均匀的分布在数组中。如果hash之后散列的很均匀，那么table数组中的每个队列长度主要为0或者1。但实际情况并非总是如此理想，虽然ConcurrentHashMap类默认的加载因子为0.75，但是在数据量过大或者运气不佳的情况下，还是会存在一些队列长度过长的情况，如果还是采用单向列表方式，那么查询某个节点的时间复杂度为O(n)；因此，对于个数超过8(默认值)的列表，jdk1.8中采用了红黑树的结构，那么查询的时间复杂度可以降低到O(logN)，可以改进性能。**

##### 红黑树：

1、每个节点都只能是红色或者黑色

2、根节点是黑色

3、每个叶节点（NIL节点，空节点）是黑色的。

4、如果一个结点是红的，则它两个子节点都是黑的。也就是说在一条路径上不能出现相邻的两个红色结点。

5、从任一节点到其每个叶子的所有路径都包含相同数目的黑色节点。

以上的Map都是无序的，有序的Map有：

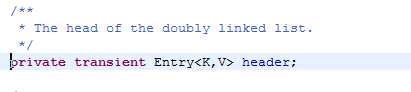
TreeMap和LinkedHashMap

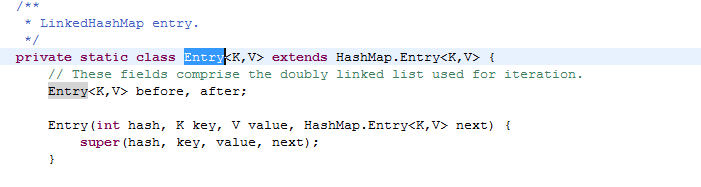
TreeMap底层实现了排序接口Comparator

LinkedHashMap继承自HashMap

原理：

1、底层维护了一个双向链表





2、重写了HashMap的部分方法，使得在每次添加或者删除元素时，都能维护这个双向链表

#### 性能

1、HashMap 与 HashTable

put方法，单线程1000万级的，速度都是10s左右（百万级的0.7以内），差不多，多线程肯定是HashMap快，因为HashTable加锁了。

遍历：性能也一样

2、HashMap与ConcurrentHashMap

put方法，千万级的，感觉比HashTable还要慢一点15s左右

### Connection

实现Iterable接口

ArrayList：底层是数组方式实现，初始容量是10，容量不够就扩容至原来的1.5倍，，有序的，查找速度快，插入删除慢，线程不安全，原因是数组扩容时，有并发安全问题。支持null元素

Vector：线程安全的，add方法有同步锁synchronized

LinkedList：底层是两个Node节点实现的，能指向前一个数据也可以指向后一个数据，没有大小限制，线程不安全，查找慢，需要从第一个元素开始索引，遍历快。

#### 性能比较

##### 遍历

ArrayList 遍历10000次的时间是：106毫秒

LinkedList 遍历10000次的时间是：38毫秒

ArrayList 遍历100000次的时间是：314

LinkedList 遍历100000次的时间是：289

ArrayList 遍历1000000次的时间是：5723

LinkedList 遍历1000000次的时间是：2460

##### 查询

100万级ArrayList 查询一个数的时间是：7698纳秒

100万级LinkedList 查询一个数的时间是：10730174纳秒

##### 删除

100万级ArrayList 删除一个数的时间是：314750纳秒

100万级LinkedList 删除一个数的时间是：12761086纳秒

##### 添加

100万级ArrayList 添加一个数的时间是：855纳秒

100万级LinkedList 添加一个数的时间是：428纳秒

### ArrayList的迭代器

[ArrayList之ConcurrentModificationException异常源码分析](https://blog.csdn.net/DeveloperLeaf/article/details/78031118)

ArrayList的remove（modCount++）和Iterator的remode方法(modCount重新赋值给expectedModCount)，因此前者会抛出异常，后者不会（iterator的next方法中比较这两个值）。

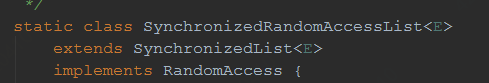
### CopyOnWriteArrayList

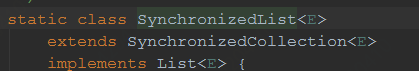
线程安全的类，内部有ReentrantLock锁，写入（add、remove、set）时先lock，然后新建数组，再深度copy（Arrays.copyOf(T[] original, int newLength)方法，若newLength< original的长度，则补值0）.最后unLock

### Collections.synchronizedList()

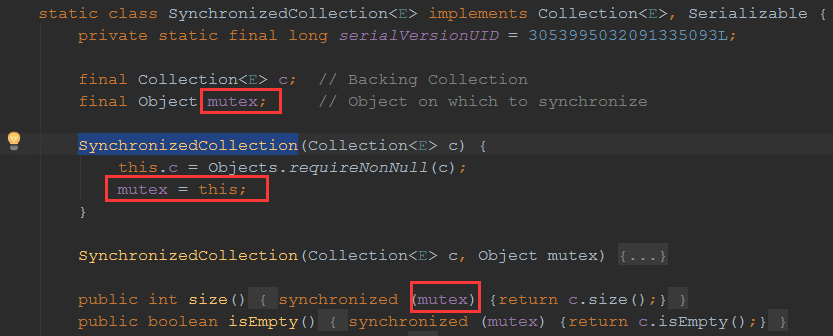


ArrayList extends RandomAccess，所以是new SynchronizedRandomAccessList。





在SynchronizedCollection内部使用synchronize内部锁，锁对象是当前对象。使用了装饰者模式，装饰了Collection



## 字符串

### String

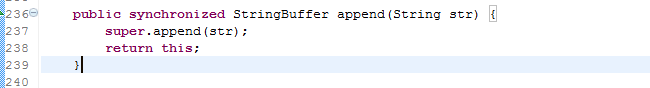
String对象是不可变的，原因是它底层是char数组，被定义成final类型的，所以在string对象上操作的其他方法，如replace、substring等，都会产生新的对象

### StringBuffer与StringBuilder

都是继承自AbstractStringBuilder

它们的操作效率都比String高，原因是它们不需要重新创建对象

StringBuffer是线程安全的，StringBuilder线程不安全，原因看下它们的append方法就知道了





上面是StringBuffer的，加了synchronized同步关键字

## 关键字

### static

1.修饰成员变量和成员方法，也可以形成静态的static代码块

2.被static修饰的成员变量和成员方法独立于该类的任何对象。也就是说，它不依赖类特定的实例，被类的所有实例共享

3. 用static修饰的代码块表示静态代码块，当Java虚拟机（JVM）加载类时，就会执行该代码块

4.静态变量在内存中只分配一次内存

5. 静态方法可以直接通过类名调用，任何的实例也都可以调用，

因此静态方法中不能用this和super关键字，不能直接访问所属类的实例变量和实例方法(就是不带static的成员变量和成员成员方法)，只能访问所属类的静态成员变量和成员方法。

因为实例成员与特定的对象关联！这个需要去理解，想明白其中的道理，不是记忆！！！

因为static方法独立于任何实例，因此static方法必须被实现，而不能是抽象的abstract

## 序列化与反序列化

1. 在Java中，只要一个类实现了java.io.Serializable接口，那么它就可以被序列化。
2. 通过ObjectOutputStream和ObjectInputStream对对象进行序列化及反序列化。
3. 虚拟机是否允许反序列化，不仅取决于类路径和功能代码是否一致，一个非常重要的一点是两个类的序列化 ID 是否一致（就是 private static final long serialVersionUID）。
4. 序列化并不保存静态变量。
5. 要想将父类对象也序列化，就需要让父类也实现Serializable 接口。
6. Transient 关键字的作用是控制变量的序列化，在变量声明前加上该关键字，可以阻止该变量被序列化到文件中，在被反序列化后，transient 变量的值被设为初始值，如 int 型的是 0，对象型的是 null。
7. 成员变量如果是个对象，也要实现java.io.Serializable接口。

### ArrayList的序列化

transient Object[] elementData;

这个变量用transient修饰了，为什么还可以被序列化呢？

因为Arraylist重写了writeObject和readObject方法

*在序列化过程中，如果被序列化的类中定义了writeObject 和 readObject 方法，虚拟机会试图调用对象类里的 writeObject 和 readObject 方法，进行用户自定义的序列化和反序列化。*

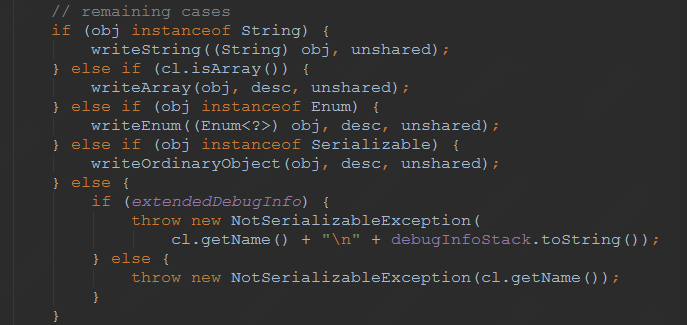
*如果没有这样的方法，则默认调用是 ObjectOutputStream 的 defaultWriteObject 方法以及 ObjectInputStream 的 defaultReadObject 方法。*

*用户自定义的 writeObject 和 readObject 方法可以允许用户控制序列化的过程，比如可以在序列化的过程中动态改变序列化的数值。*

ObjectOutputStream在序列化的时候，有如下代码：

调用链：

writeObject ---> writeObject0 --->writeOrdinaryObject--->writeSerialData--->invokeWriteObject



invokeWriteObject方法里，通过反射调用Arraylist的writeObject方法

#### why transient

ArrayList实际上是动态数组，每次在放满以后自动增长设定的长度值，如果数组自动增长长度设为100，而实际只放了一个元素，那就会序列化99个null元素。为了保证在序列化的时候不会将这么多null同时进行序列化，ArrayList把元素数组设置为transient。

### 总结

1、如果一个类想被序列化，需要实现Serializable接口。否则将抛出NotSerializableException异常，这是因为，在序列化操作过程中会对类型进行检查，要求被序列化的类必须属于Enum、Array和Serializable类型其中的任何一种。

2、在变量声明前加上该关键字，可以阻止该变量被序列化到文件中。

3、在类中增加writeObject 和 readObject 方法可以实现自定义序列化策略

## Date

java中Date类中的getTime()是获取时间戳的，java中生成的时间戳精确到毫秒级别，而unix中精确到秒级别，所以通过java生成的时间戳需要除以1000。

Mybatis中当前日期的前一天的时间戳：10位

unix\_timestamp(date\_sub(CURRENT\_DATE ,interval 1 day))

## 泛型

/\*\*

\* 泛型方法的基本介绍

\* @param tClass 传入的泛型实参

\* @return T 返回值为T类型

\* 说明：

\* 1）public 与 返回值中间<T>非常重要，可以理解为声明此方法为泛型方法。

\* 2）只有声明了<T>的方法才是泛型方法，泛型类中的使用了泛型的成员方法并不是泛型方法。

\* 3）<T>表明该方法将使用泛型类型T，此时才可以在方法中使用泛型类型T。

\* 4）与泛型类的定义一样，此处T可以随便写为任意标识，常见的如T、E、K、V等形式的参数常用于表示泛型。

\*/

public <T> T genericMethod(Class<T> tClass)throws InstantiationException ,

IllegalAccessException{

T instance = tClass.newInstance();

return instance;

}

[java 泛型详解-绝对是对泛型方法讲解最详细的，没有之一](https://blog.csdn.net/s10461/article/details/53941091)

# 、web容器



**web.xml 的加载顺序是：context-param -> listener -> filter -> servlet**

## 1.过滤器

过滤器Filter是实现了javax.servlet.Filter接口，主要有一下三个方法：

init()、doFilter()、destroy()

的服务器端程序，主要的用途是过滤字符编码、做一些业务逻辑判断等。

 (1)、启动服务器时加载过滤器的实例，并调用init()方法来初始化实例；   
 (2)、每一次请求时都只调用方法doFilter()进行处理；   
 (3)、停止服务器时调用destroy()方法，销毁实例。

在web.xml中配置：

<filter>

<filter-name>characterEncodingFilter</filter-name>

<filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>

<init-param>

<param-name>encoding</param-name>

<param-value>UTF-8</param-value>

</init-param>

<init-param>

<param-name>forceEncoding</param-name>

<param-value>true</param-value>

</init-param>

</filter>

<filter-mapping>

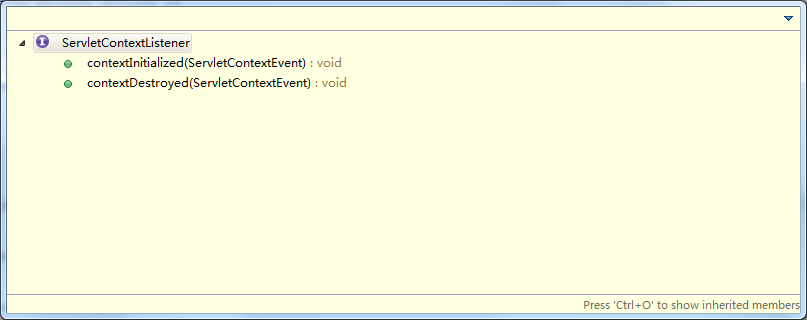
<filter-name>characterEncodingFilter</filter-name>

<url-pattern>/\*</url-pattern>

</filter-mapping>

## 2.监听器

现在来说说Servlet的监听器Listener，它是实现了javax.servlet.ServletContextListener



接口的服务器端程序，它也是随web应用的启动而启动，只初始化一次，随web应用的停止而销毁。主要作用是： 做一些初始化的内容添加工作(初始化spring容器)、设置一些基本的内容，打印容器启动信息，如spring中配置的bean加载信息，springMVC配置信息启动日志。

Web.xml中配置：

<listener>

<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>

</listener>

<listener>

<listener-class>com.yghsh.bxb.base.MyServletContextListener

</listener-class>

</listener>

### 监听启动spring IOC容器过程：

1. 监听到容器启动，调用监听器的



方法，即ContextLoader的

initWebApplicationContext(ServletContext servletContext)，

a．创建一个spring容器上下文context

context=org.springframework.web.context.WebApplicationContext

其实现类就是：

org.springframework.web.context.support.XmlWebApplicationContext

这个是在下面的配置文件里面配置好的

/org/springframework/web/context/ContextLoader.properties

配置如下：

org.springframework.web.context.WebApplicationContext=org.springframework.web.context.support.XmlWebApplicationContext

b．configureAndRefreshWebApplicationContext(cwac, servletContext);

将spring配置文件中配置的bean加载到spring ioc容器的上下文

WebApplicationContext中

## 3.拦截器

是配置在springMVC的配置文件中，随servlet的加载而加载

配置格式如下：

<bean id=*"handlerMapping"* class=*"org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping"*>

<property name=*"interceptors"*>

<list>

<bean class=*"com.yghsh.bxb.base.SpringMVCFliterInterceptor"*/>

</list>

</property>

</bean>

## 4.servlet

配置略过，作用略过

原理：几个重要的接口和类

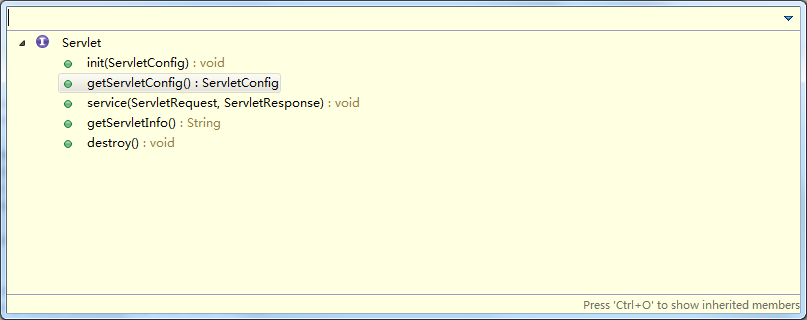
HttpServlet

GenericServlet

Servlet

ServletConfig

### Servlet接口



web.xml中配置的servlet

load-on-startup设置为负数或者不设置时，servlet第一次被用到才会加载

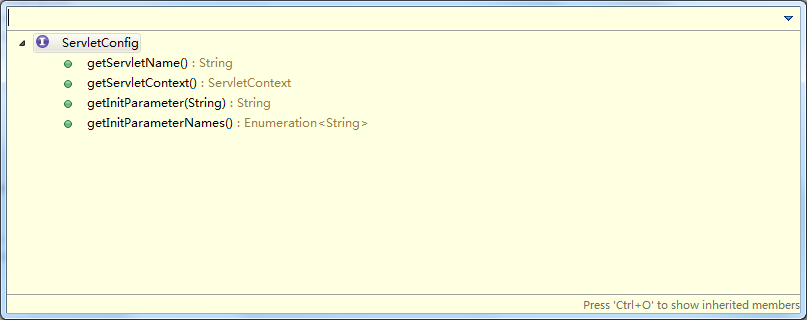
load-on-startup设置为1时，容器启动即加载servlet

servlet加载时，调用init（）初始化方法，只调用一次

service()，具体处理一个请求

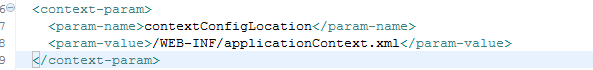
destroy()，servlet销毁时调用，即关闭服务器时，只调用一次

### ServletConfig接口



getServletName()，用于获取Servlet名字

getServletContext()，非常重要，返回的ServletContext，就是容器的上下文，所有的servlet共享，这里有很多配置信息，如web.xml中



加载的spring配置信息都在这里面

getInitParameter(String name)，获取servlet的init-param配置参数信息

### GenericServlet抽象类

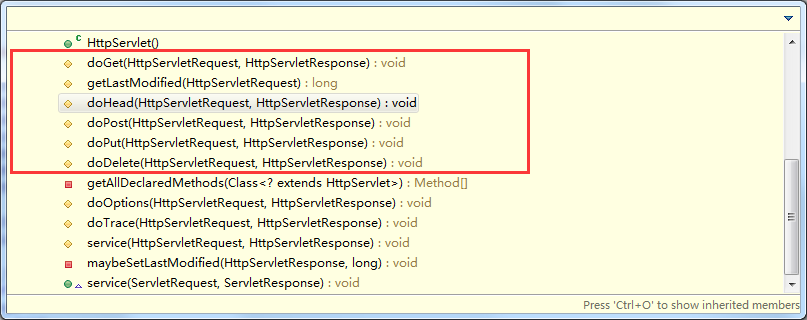
是上面两个接口的实现类，主要做了三件事：

1. 实现了ServletConfig接口，我们可以直接调用ServletConfig里面的方法
2. 提供了Servlet无参的init()方法
3. 提供了log方法

### HttpServlet抽象类

是用HTTP协议实现的Servlet类，springMVC中的DispatcherServlet就是继承自该类。

主要重写了service()方法，根据http请求类型的不同，将请求路由到不同的处理方法

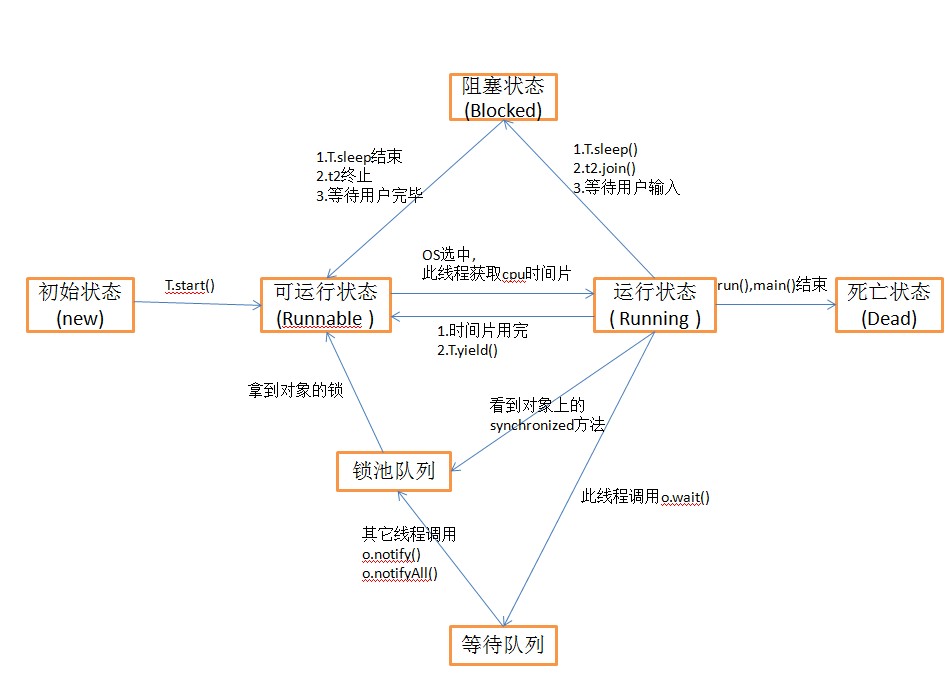


## Request

## Response

# 多线程

## 线程状态

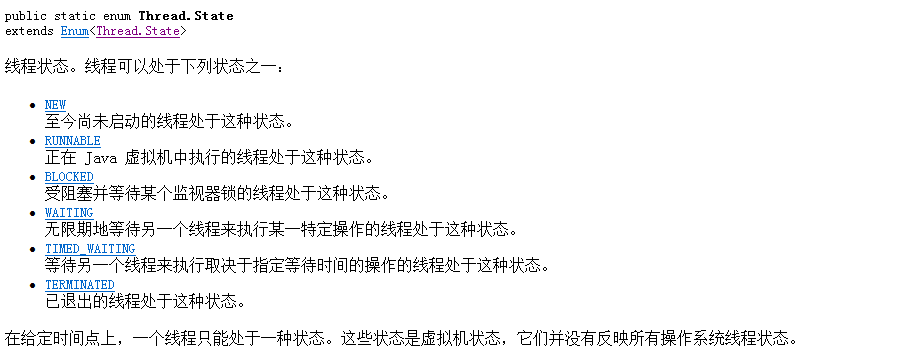


概念：

Object.wait();

在其他线程调用此对象的 notify() 方法或 notifyAll() 方法前，导致当前线程等待,并且释放锁

Thread.join();//等待当前线程执行完，不释放锁



### 新建（NEW）

新创建一个线程对象

### 可运行(RUNNABLE)

线程对象创建后，其他线程(比如main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，等待被线程调度选中，获取cpu 的使用权（可运行和运行中）

### 阻塞（blocked）

这个状态下, 是在多个线程有同步操作的场景, 比如正在等待另一个线程的synchronized 块的执行释放，或者在同步代码块里，别的线程调用了当前锁对象的wait方法。等待监视器锁进入同步代码块/方法

### WAITING

以下三种情况会进入此状态：

*\* <li>{****@link*** *Object#wait() Object.wait} with no timeout</li>  
\* <li>{****@link*** *#join() Thread.join} with no timeout</li>  
\* <li>{****@link*** *LockSupport#park() LockSupport.park}</li>*

这个状态下是指线程拥有了某个锁之后, 调用了锁对象的wait方法, 等待其他线程/锁拥有者调用 notify / notifyAll 以便该线程可以继续下一步操作, 这里要区分 BLOCKED 和 WATING 的区别, 一个是在临界点外面等待进入, 一个是在临界点里面wait等待别人notify, 线程调用了join方法 join了另外的线程的时候, 也会进入WAITING状态, 等待被他join的线程执行结束

### TIMED\_WAITING

以下5中情况进入此状态

*\* <li>{****@link*** *#sleep Thread.sleep}</li>  
\* <li>{****@link*** *Object#wait(long) Object.wait} with timeout</li>  
\* <li>{****@link*** *#join(long) Thread.join} with timeout</li>  
\* <li>{****@link*** *LockSupport#parkNanos LockSupport.parkNanos}</li>  
\* <li>{****@link*** *LockSupport#parkUntil LockSupport.parkUntil}</li>*

这个状态就是有限的(时间限制)的WAITING, 一般出现在调用wait(long), join(long)等情况下, 另外一个线程sleep后, 也会进入TIMED\_WAITING状态

### TERMINATED

这个状态下表示 该线程的run方法已经执行完毕了, 基本上就等于死亡了(当时如果线程被持久持有, 可能不会被回收)。

## 线程方法

### Join()/Join(long mills)

让主线程等待子线程结束之后才能继续运行**。**主线程进入 WAITING/MED\_WAITING 状态，但不释放对象锁,等待 join() 所属线程运行结束后或者millis时间到进入可运行状态。主线程调用方法不断去检查子线程的状态， 这是个主动的动作，**而不是子线程去唤醒主线程**。

join方法必须在线程start方法调用之后调用才有意义，

源码：isAlive()是native方法

下面代码中isAlive()是子线程，而wait()是主线程

public final synchronized void join(long millis)  
throws InterruptedException {  
 long base = System.*currentTimeMillis*();  
 long now = 0;  
 if (millis < 0) {  
 throw new IllegalArgumentException("timeout value is negative");  
 }  
 if (millis == 0) {  
 while (isAlive()) {  
 wait(0);  
 }  
 } else {  
 while (isAlive()) {  
 long delay = millis - now;  
 if (delay <= 0) {  
 break;  
 }  
 wait(delay);  
 now = System.*currentTimeMillis*() - base;  
 }  
 }  
}

### wait()

该方法应该只能被拥有该对象监视器的线程调用：

线程获得对象监视器的三种方法：

1. 执行对象的synchronize实例方法
2. 执行对象（作为锁）的同步代码块
3. 对象是Class类型，则通过执行这个类的同步静态方法

*This method should only be called by a thread that is the owner  
\* of this object's monitor*

当前线程调用对象的wait()方法(前提是必须拥有当前对象的监视器)，当前线程释放对象锁，进入等待队列。当前线程被线程调度禁用处于休眠状态，直到下面四种情况发生：

1. 其他线程中调用了对象notify()方法(唤醒任意线程进入锁池)，且当前线程恰好被选中为被唤醒的线程。notify方法应该被拥有该对象监视器的线程调用
2. 其他线程中调用了对象notifyAll()方法
3. 其他线程中interrupt()方法，则会抛出异常，并恢复到原状。
4. 或者wait(long timeout)timeout时间到自动唤醒。

当前线程从对象的等待队列中移除，重新参与线程调度，和其他线程以通常的方式竞争获得该对象的同步锁。一旦获得对象的控制权，该对象上所有的同步声明将恢复到原状(对象的wait方法被调用的时候)。

除了以上四种情况还有虚假唤醒，实践中很少发生，程序也必须防止。所以，等待应该始终发生在循环中，如下：

<pre>  
\* synchronized (obj) {  
\* while (&lt;condition does not hold&gt;)  
\* obj.wait(timeout);  
\* ... // Perform action appropriate to condition  
\* }  
\* </pre>

wait()、notify()/notifyAll()必须在synchronize同步代码块中调用，是Object的方法

### sleep(long millis)

使当前运行的线程休眠指定时间，受制于系统定时器和调度器的精确性和准确性。线程不会失去任何监视器的拥有权，不释放锁

### Interrupt()

正常活动线程：发出一个中断请求，把标志位设定为中断状态，不会终止线程运行。其他线程试图调用该方法，会检测是否有权限中断该线程（正常情况 下不会存在权限问题，这里可以忽略）

阻塞状态线程：Thread.sleep()、Thread.join()、object.wait()这些方法，会检测线程中断标志位，如果发现中断标志位为true则抛出异常并且将中断标志位设置为false。

Synchronize中不能中断

 public void interrupt()   
将调用者线程的中断状态设为true。

 public boolean isInterrupted()   
判断调用者线程的中断状态。

 public static boolean interrupted   
只能通过Thread.interrupted()调用。

 当你的捕获到一个InterruptedException异常后，亦可以处理它，或者向上抛出。

 抛出时要注意？？？：当你捕获到InterruptedException异常后，当前线程的中断状态已经被修改为false(表示线程未被中断)；此时你若能够处理中断，则不用理会该值；但如果你继续向上抛InterruptedException异常，你需要再次调用interrupt方法，将当前线程的中断状态设为true。

 **注意**：绝对不能“吞掉中断”！即捕获了InterruptedException而不作任何处理。这样违背了中断机制的规则，别人想让你线程中断，然而你自己不处理，也不将中断请求告诉调用者，调用者一直以为没有中断请求。

[Java里一个线程调用了Thread.interrupt()到底意味着什么？](https://www.zhihu.com/question/41048032?sort=created)

### yield()

使线程从运行状态🡪可运行状态

释放cup资源，不会释放锁

### LockSupport.park()

[LockSupport解析与使用](https://blog.csdn.net/secsf/article/details/78560013)

先unpart()后part()，输出b

Thread thread = Thread.*currentThread*();  
LockSupport.*unpark*(thread);//释放许可  
LockSupport.*park*();// 获取许可  
  
System.*out*.println("b");

先part()后unpart()，下面代码主线程被阻塞：

Thread thread = Thread.*currentThread*();  
LockSupport.*park*();// 获取许可  
LockSupport.*unpark*(thread);//释放许可  
  
System.*out*.println("b");

## 锁Lock

[深入理解AbstractQueuedSynchronizer（一）](http://www.ideabuffer.cn/2017/03/15/%E6%B7%B1%E5%85%A5%E7%90%86%E8%A7%A3AbstractQueuedSynchronizer%EF%BC%88%E4%B8%80%EF%BC%89/)

**实现类**

ReentrantLock、ReadWriteLock（实现类ReentrantReadWriteLock）

实现都依赖java.util.concurrent.AbstractQueuedSynchronizer类

以**ReentrantLock**实现类为例

**属性**：

**final** Sync sync; //Sync extends AbstractQueuedSynchronizer

**静态内部类（公平锁处理）**

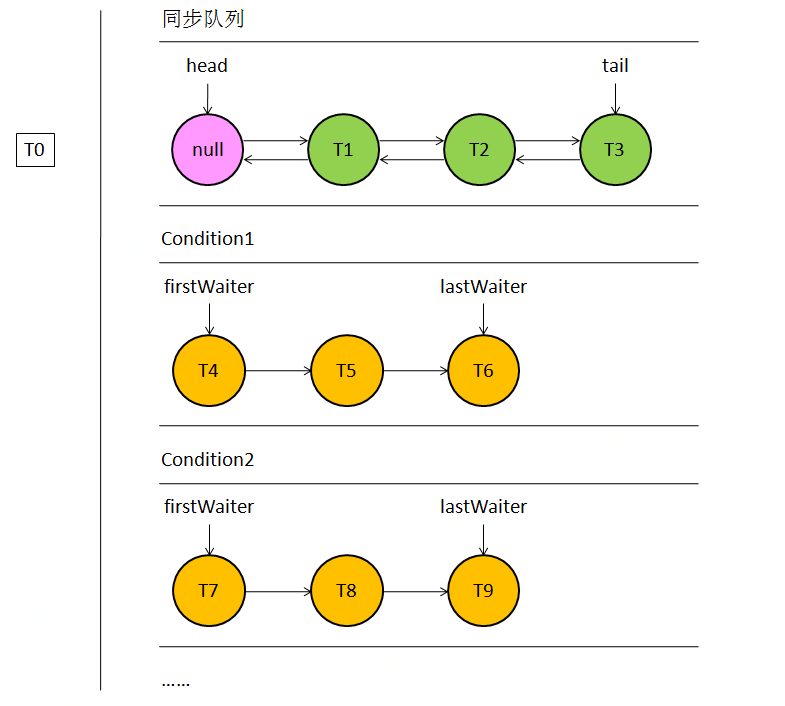
static final class FairSync extends Sync{}

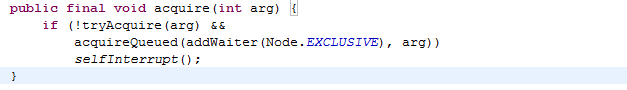
**静态内部类（非公平锁处理，默认是非公平的）**

static final class NonfairSync extends Sync {}

**AbstractQueuedSynchronizer详解**

[**Java并发系列[1]----AbstractQueuedSynchronizer源码分析之概要分析**](http://www.cnblogs.com/liuyun1995/p/8400663.html)



****

**属性：**

int state // ReentrantLock中 0代表没有线程正竞争该锁，不等于0说明已经有线程拥有的该锁

Node head // 队列头部

Node tail //队尾，未获取到锁的线程添加到队尾

Thread exclusiveOwnerThread //从父类继承来的属性，代表独占此锁的线程

### CAS(Compare and Swap):

[**Java CAS 和ABA问题**](https://www.cnblogs.com/549294286/p/3766717.html)

CAS操作需要输入两个数值，一个旧值（期望操作前的值）和一个新值，在操作期间先比较下旧值有没有发生变化，如果没有发生变化，才交换成新值，发生了变化则不交换。

CAS主要通过compareAndSwapXXX()方法来实现，而这个方法的实现需要涉及底层的unsafe类

CAS可能存在ABA问题：

解决：AtomicStampedReference和AtomicMarkableReference

**unsafe类：**java不能直接访问操作系统底层，而是通过本地方法来访问。Unsafe类提供了硬件级别的原子操作，底层是排它锁，无java锁 [浅析CompareAndSet(CAS)](https://blog.csdn.net/u013404471/article/details/47297123)

**可重入锁**

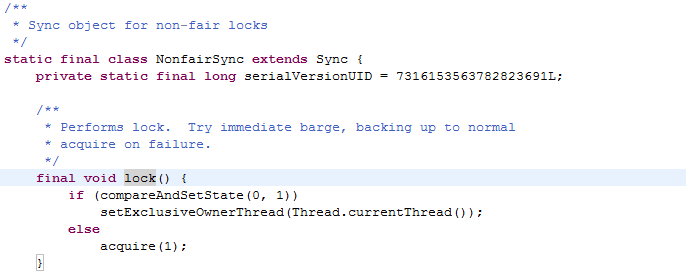
线程可以进入任何一个它已经拥有的锁所同步着的代码块。如synchronized，java.util.concurrent.locks.ReentrantLock

**不可重入锁**

如自旋锁 [可重入锁和不可重入锁](https://www.cnblogs.com/dj3839/p/6580765.html)

### 加锁过程

默认使用的是非公平锁的lock方法

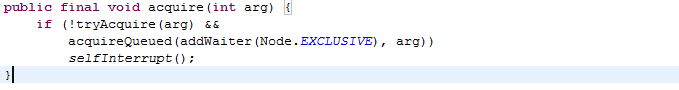


compareAndSetState(0, 1)方法：如果state==0则将state设置为1

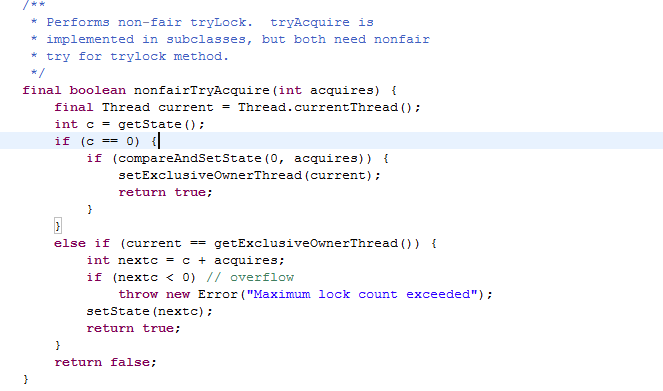
setExclusiveOwnerThread(Thread.currentThread());//将独占此锁的线程设置为当前线程

如果，state不等于0，则表示锁已被其他线程占用，则调用acquire(1)方法

acquire(1)，调用的是**AbstractQueuedSynchronizer**的方法



**tryAcquire(arg)**此方法由子类实现，调用的Sync. nonfairTryAcquire(int acquires)



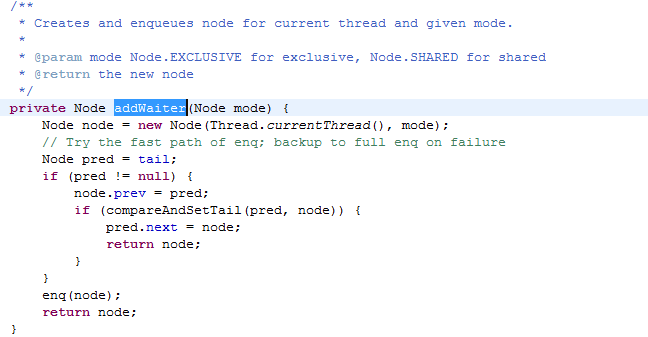
此方法完成两件事，

1.先试图获取锁，取到设置state=1，设置独占当前锁线程为此线程。

2.如果是锁重入，则将state+1

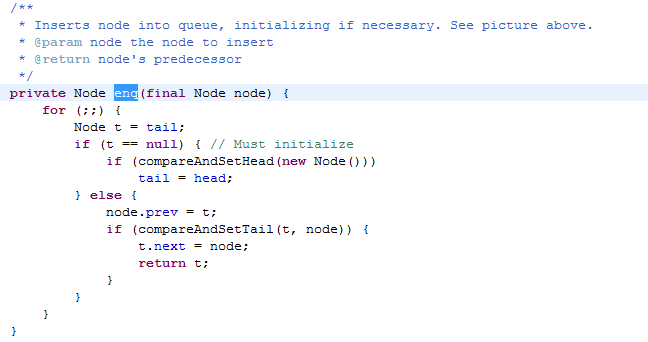
**acquireQueued(addWaiter(Node.EXCLUSIVE), arg)**

1、AbstractQueuedSynchronizer.addWaiter



负责把当前未获得锁的线程包装成一个Node节点，添加到队尾

其中参数mode是独占锁还是共享锁，默认为null，独占锁。追加到队尾的动作分两步：   
如果当前队尾已经存在(tail!=null)，则使用CAS把当前线程更新为Tail   
如果当前Tail为null或者线程调用CAS设置队尾失败，则通过enq方法继续设置Tail   
下面是enq方法：



该方法就是循环调用CAS，即使有高并发的场景，无限循环将会最终成功把当前线程追加到队尾（或设置队头）。总而言之，addWaiter的目的就是通过CAS把当前线程追加到队尾，并返回包装后的Node实例。

把线程要包装为Node对象的主要原因，除了用Node构造供虚拟队列外，还用Node包装了各种线程状态，这些状态被精心设计为一些数字值：（节点相当于座位，线程相当于客人，状态相当于座位上挂的牌子）

SIGNAL(-1) ：线程的后继线程正/已被阻塞，当该线程release或cancel时要重新这个后继线程(unpark)。当此线程需要睡一会，在前面的作为挂牌SIGNAL，前面的线程走后唤醒此线程

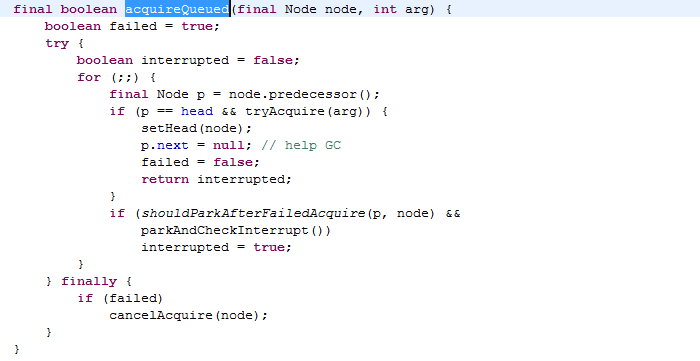
CANCELLED(1)：因为超时或中断，该线程已经被取消

CONDITION(-2)：表明该线程被处于条件队列，就是因为调用了Condition.await而被阻塞

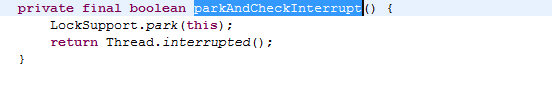
PROPAGATE(-3)：传播共享锁，表示后继节点可以直接获取锁

0：0代表无状态

2、AbstractQueuedSynchronizer.acquireQueued



acquireQueued的主要作用是把已经追加到队列的线程节点（addWaiter方法返回值）进行阻塞，但阻塞前又通过tryAccquire重试是否能获得锁（节点是head后的第一个节点），如果重试成功能则无需阻塞，直接返回，parkAndCheckInterrupt就是用来阻塞线程的（阻塞动作交给linux系统内核）



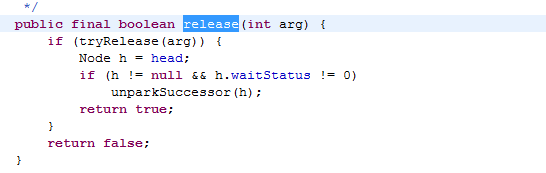
### 解锁

请求锁不成功的线程会被挂起在acquireQueued方法的第12行，12行以后的代码必须等线程被解锁锁才能执行，假如被阻塞的线程得到解锁，则执行第13行，即设置interrupted = true，之后又进入无限循环。

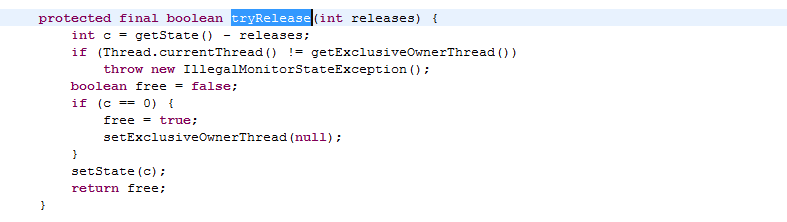
从无限循环的代码可以看出，并不是得到解锁的线程一定能获得锁，必须在第6行中调用tryAccquire重新竞争，因为锁是非公平的，有可能被新加入的线程获得，从而导致刚被唤醒的线程再次被阻塞，这个细节充分体现了“非公平”的精髓。通过之后将要介绍的解锁机制会看到，第一个被解锁的线程就是Head，因此p == head的判断基本都会成功。

至此可以看到，把tryAcquire方法延迟到子类中实现的做法非常精妙并具有极强的可扩展性，令人叹为观止！当然精妙的不是这个Template设计模式，而是Doug Lea对锁结构的精心布局。

解锁代码相对简单，主要体现在AbstractQueuedSynchronizer.release和Sync.tryRelease方法中：



Sync.tryRelease()



tryRelease与tryAcquire语义相同，把如何释放的逻辑延迟到子类中。

tryRelease语义很明确：如果线程多次锁定，则进行多次释放，直至status==0则真正释放锁，所谓释放锁即设置status为0，因为无竞争所以没有使用CAS。   
release的语义在于：如果可以释放锁，则唤醒队列第一个线程（Head）

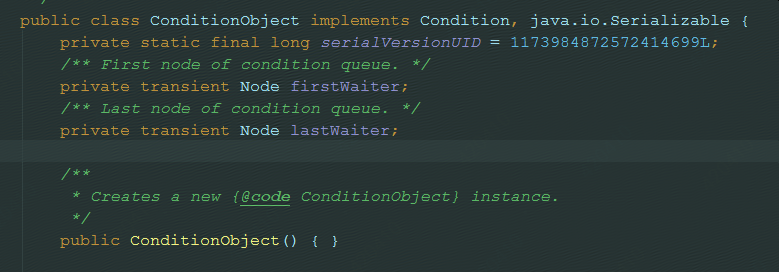
### Condition

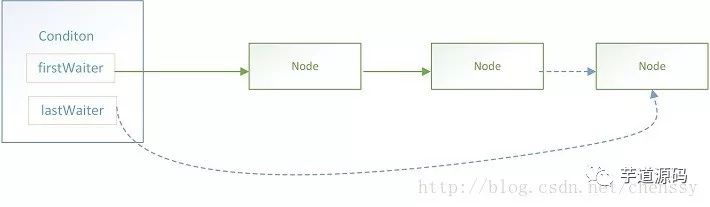
[【死磕Java并发】-----J.U.C之Condition](https://mp.weixin.qq.com/s/i-CnMqWvw_31YZ3111davA)

AQS的条件队列有多个。

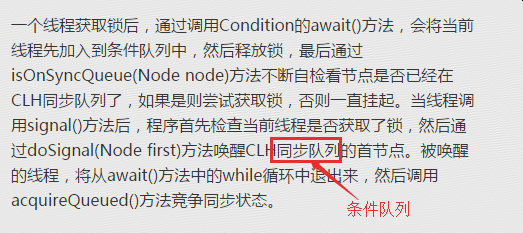


获取一个Condition必须要通过Lock的newCondition()方法。该方法定义在接口Lock下面，返回的结果是绑定到此 Lock 实例的新 Condition 实例。Condition为一个接口，其下仅有一个实现类ConditionObject，由于Condition的操作需要获取相关的锁，而AQS则是同步锁的实现基础，所以ConditionObject则定义为AQS的内部类。定义如下：





获取锁是await和signal的前置条件



## 栅栏CyclicBarrier

用于阻塞一组线程直到某个事件发生，然后再一起往下执行。

属性：

parties //构造器里给定的栅栏数

count //还有多少个线程未到达栅栏，初始化时等于parties

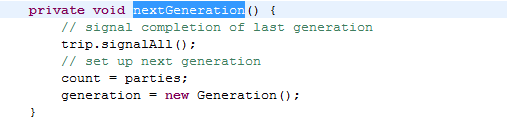
Generation generation = **new** Generation();//分代概念，每一批线程属于同一代，从而实现栅栏的复用

Condition trip = lock.newCondition();//条件状态

Runnable barrierCommand;//count==0时执行的command，由最后一个到达的线程执行

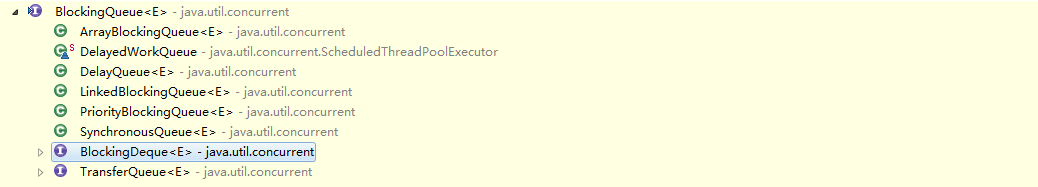
原理：当调用await方法是，count--，如果count大于0，则当前线程阻塞在条件trip上，

如果count==0，执行barrierCommand，重置generation，即调用这个方法



## 阻塞队列BlockingQueue

子类



有容量限制的队列，可能在存入数据和取出数据时在两端都可能产生阻塞。如果队列已满还要放数据则阻塞。队列已空还要获取数据也会阻塞。

协调生产者和消费者之间的生产消费过程。

抛出异常 特殊值 阻塞 超时

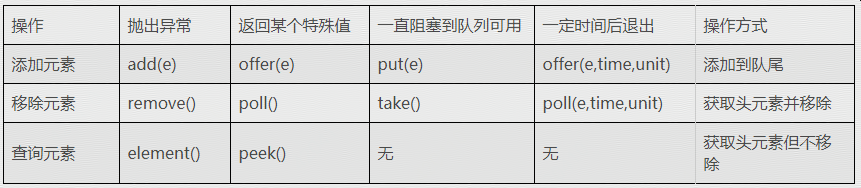
插入 add(e) offer(e) put(e) offer(e, time, unit)

移除 remove() poll() take() poll(time, unit)

检查 element() peek() 不可用 不可用

**ArrayBlockingQueue**底层是数组，长度有限，且长度在创建时指定后无法修改。

1. //队列满了之后会直接抛出异常
2. //arrayBlockingQueue.add(i);
3. //队列满了之后会等待队列腾出空间
4. //arrayBlockingQueue.put(i);
5. //将指定的元素插入到此队列的尾部（如果立即可行且不会超过该队列的容量），在成功时返回 true，如果此队列已满，则返回 false。
6. arrayBlockingQueue.offer(i);
7. //获取并移除此队列的头部，在指定的等待时间前等待可用的元素。如果已经没有可用的元素，则没10s返回一个null
8. // System.out.println(arrayBlockingQueue.poll(10000, TimeUnit.MILLISECONDS));
9. //获取并移除此队列的头部，在元素变得可用之前一直等待
10. System.out.println(arrayBlockingQueue.take());
11. //获取但不移除此队列的头；如果此队列为空，则返回 null
12. //System.out.println(arrayBlockingQueue.peek());



**LinkedBlockingQueue**底层是链表，长度可以指定也可以不指定如果不指定默认值为Integer.MaxValue,且底层的链表长度（所占用的空间）是动态变化的。

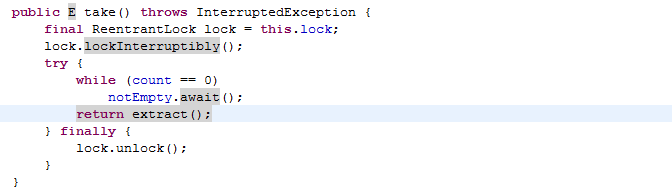
**DelayQueue**延迟队列 在阻塞式队列的基础上，增加了过期时间的机制，当元素过期时会在下一次获取元素之前被移除。

\*\*存入该队列的元素必须实现Delayed接口。

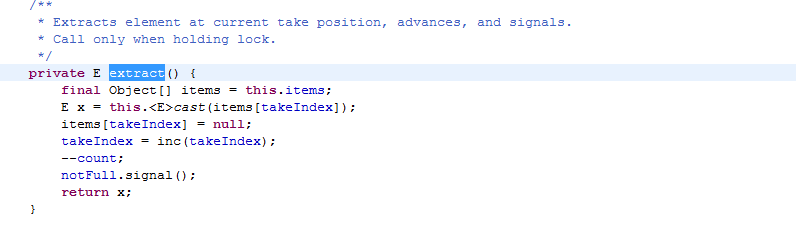
**SynchronousQueue**长度为1的阻塞式队列，与其说他是一个队列，更像一个线程之间的汇合点。

**源码底层实现**

以**ArrayBlockingQueue**为例，take方法是如何实现线程阻塞的



当count==0时，通过Condition的await方法阻塞



notFull.signal()唤醒一个往队列里放元素阻塞的线程

## 并发的ConcurrentHashMap

初始容量16，阈值0.75

不支持null类型的key也不支持null类型的value

线程安全：在添加元素或者扩容时，使用了Lock锁

效率高，因为将容器分割成了一个个的Segment对象，使用了分段锁技术

## 线程递减锁 CountDownLatch

可以在构造时指定要监测的数量

await()方法将会阻塞直到监测数量变为0

countDown()可以递减监测数量

基于AQS（AbstarctQueuedSynchronizer）实现的，原理可参考Lock锁的实现

## 信号量 Semaphore

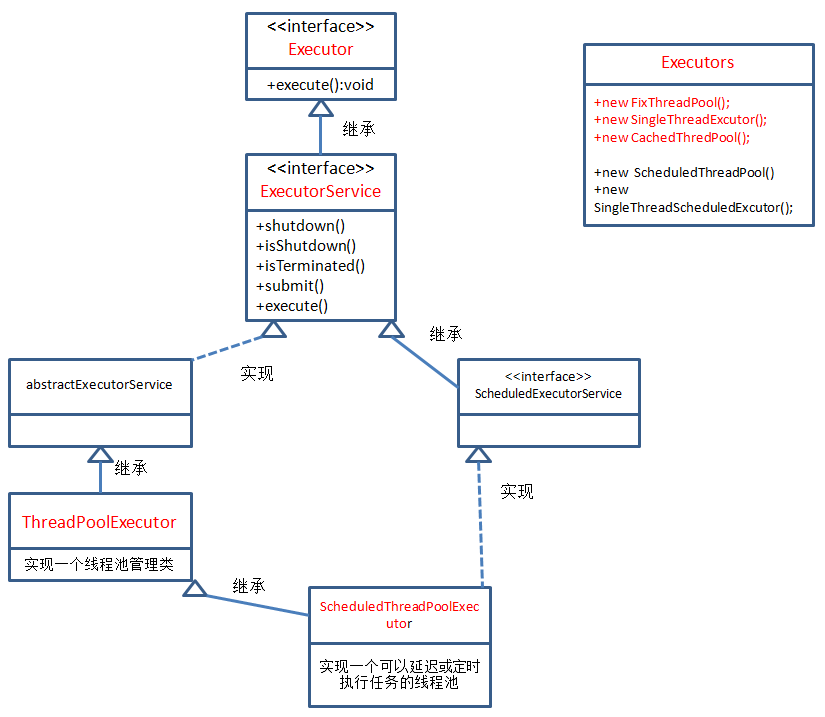
用来控制同时访问某个特定资源的操作数量或者同时执行某个指定操作的数量，还可以用来实现某种资源池，或者对容器施加边界。

基于AQS（AbstarctQueuedSynchronizer）实现的，原理可参考Lock锁的实现

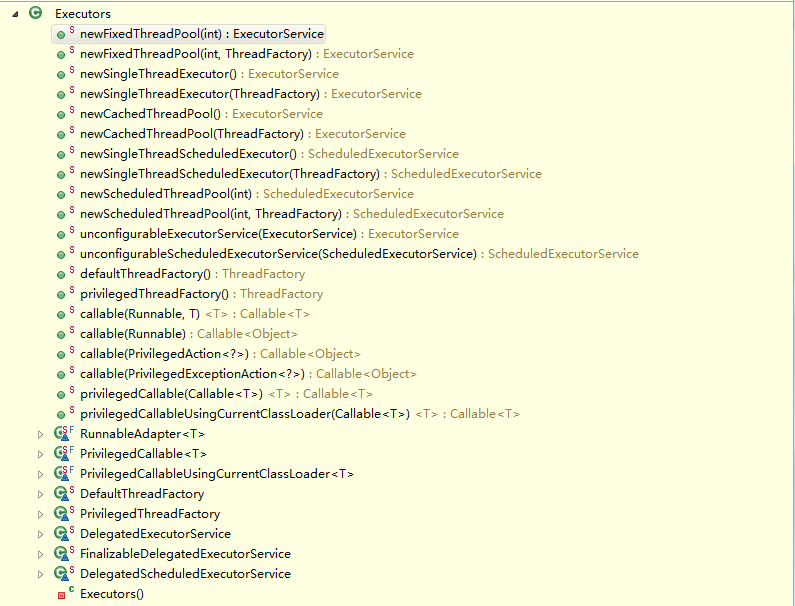
应用的话：可以用来控制往阻塞队列里放元素或者取出元素的并发个数，想要取元素，先得获得一个信号量

## 线程池 – ExecutorService

[**Java线程池带图详解**](https://blog.csdn.net/sd09044901guic/article/details/80131877)



### 线程池的工具类Executors



### 创建线程池，两种方式

第一种用工具类**Executors（内部也是new** ThreadPoolExecutor**）**

**Executors.newSingleThreadScheduledExecutor()**

创建一个使用单个 worker 线程的 Executor，以无界队列方式来运行该线程。

**Executors.newFixedThreadPool(int nThreads)**

创建一个可重用固定线程数的线程池，以共享的无界队列方式来运行这些线程。 核心线程数=最大线程数

**Executors.newScheduledThreadPool(int corePoolSize)**

创建一个线程池，它可安排在给定延迟后运行命令或者定期地执行。

**Executors.newCachedThreadPool()**

创建一个可根据需要创建新线程的线程池，但是在以前构造的线程可用时将重用它们。对于执行很多短期异步任务的程序而言，这些线程池通常可提高程序性能。

### 第二种手动自己创建线程池

ExecutorService es = **new** ThreadPoolExecutor(2, 4, 5, TimeUnit.*SECONDS*, **new** ArrayBlockingQueue<Runnable>(2)

, **new** RejectedExecutionHandler() {

**public** **void** rejectedExecution(Runnable r, ThreadPoolExecutor executor) {

executor.remove(r);

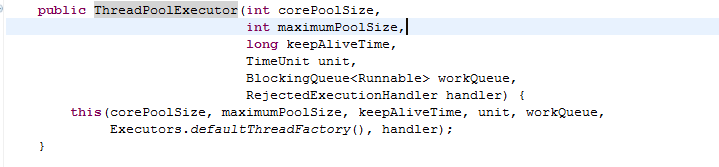
System.*out*.println("实在放不下了。。。。撂挑子不干了。。。");

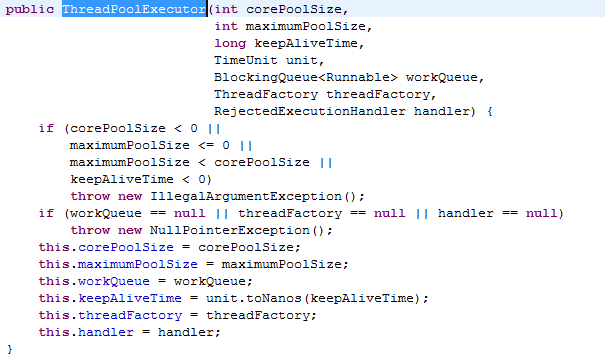
}

});

参数说明：

构造器





/\*\*

\* 第一个参数 corePoolSize - 核心线程数，如果核心线程数没满，只要有新任务就会创建新线程，核心线程默认不会退出，

\*核心线程在allowCoreThreadTimeout被设置为true时会超时退出

\* 第二个参数 maximumPoolSize - 池中允许的最大线程数

\* 第三个参数 keepAliveTime -当线程池允许线程超时且运行中的线程数量超过corePoolSize时，超过这个时间，线程会退出

\* 第四个参数 unit - keepAliveTime 参数的时间单位

\* 第五个参数 workQueue - 执行前用于保持任务的队列。此队列仅由保持 execute 方法提交的 Runnable 任务

\* 第六个参数 handler - 由于超出线程范围和队列容量而使执行被阻塞时所使用的处理程序

\* 关系：核心线程数没满，有新的任务就创建新的线程，核心线程数满了，新的任务就放到队列里，队列满了就根据最大线程数

\* 再创建新的线程，超过最大线程数就调用handler方法，

\*/

#### 拒绝策略RejectedExecutionHandler

RejectedExecutionHandler接口，内部只有一个方法

void rejectedExecution(Runnable r, ThreadPoolExecutor executor);

参数r是要被拒绝的任务command，executor是当前线程池

java提供了4种丢弃处理的方法，都是ThreadPoolExcutor的公有静态内部类，实现接口RejectedExecutionHandler。当然你也可以自己实现接口方法定义想要拒绝策略

1. AbortPolicy（默认）

直接抛出异常RejectedExecutionException来拒绝一个任务

1. DiscardPolicy

默默的丢弃任务，其他什么操作也不做

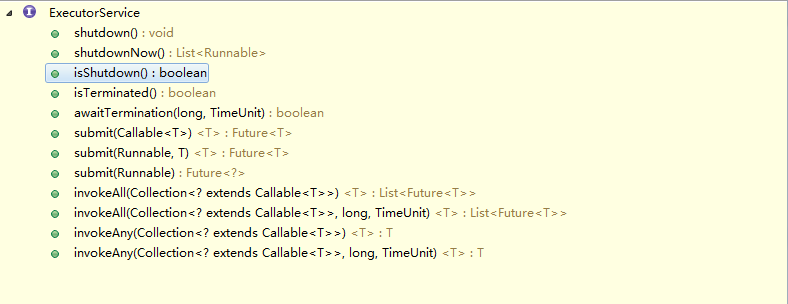
1. DiscardOldestPolicy

抛出等待队列的队头，执行当前任务。除非线程池已经shutdown，则丢弃任务

1. CallerRunsPolicy

用调用者所在线程直接执行任务，除非线程池已经shutdown，则丢弃任务

### 接口提供的方法



关闭：

ExcutorService在使用完成后一定要关闭，否则其中的线程一直不死，虚拟机无法正常退出。

shutdown();//不会立即关闭线程池，而是不再接收新的任务，直到已有的所有任务都执行完，才真正关闭。

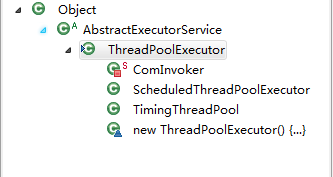
shutdownNow();//不管其中有没有线程再执行，都立即关闭。

ScheduledExecutorService//可以执行延时任务或定时任务的线程池执行器

### 底层源码实现

[教你彻底弄明白Executor框架线程池任务执行的全过程](https://blog.csdn.net/fngy123/article/details/48738409)

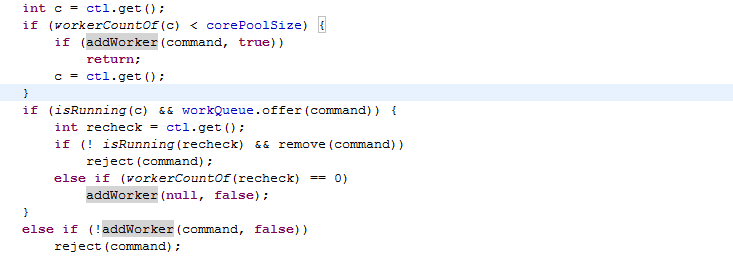
以ThreadPoolExecutor为例介绍



**AbstarctExcutorService implements ExecutorService**

**ExecutorService extends Executor**

excute()方法，它的执行实际上分了三步



1、如果当前线程池中运行的线程数小于核心线程数，那么新开启一个线程执行任务task，

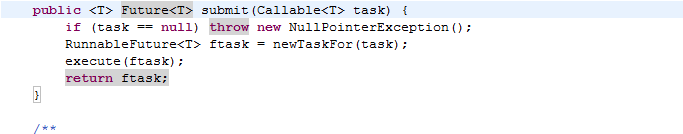
将task 包装成一个Worker，具体可见addWorker()方法

2、如果当前运行的worker数量已经超过核心线程数了，那么将task任务添加到workQueue队列里。

3、如果队列也添加不进去，按最大线程数创建线程，满了调用handle方法

在addWorker方法中，会调用Thread.start，使用线程池中的工作者线程执行任务

submit方法，此方法是在**AbstarctExcutorService** 中实现的



可以看出是借助于RunnableFuture<T>实现的

### 核心线程数和队列长度设值

**核心线程数**：

最佳核心线程数=（线程等待时间与线程CPU时间之比 + 1）\* CPU数目

比如平均每个线程CPU运行时间为0.5s，而线程等待时间（非CPU运行时间，比如IO）为1.5s，CPU核心数为8，那么根据上面这个公式估算得到：((0.5+1.5)/0.5)\*8=32

线程等待时间：调用JSF接口、IO等

线程cpu时间：例如内部的for循环，只会占用cpu时间，不会有线程等待时间

**队列长度**：

8核服务器，开启10个核心线程

比如一个任务最大等待时间是3s，每个task执行时间0.5s，那么每秒线程池能处理20个任务，那么阻塞队列长度应设置在60个以内。

## Runnable与Callable<V>

1、Runnable是**package** java.lang;包下的，JDK1.0版本就提供的，Callable<V>是JDK1.5版本下提供的，在**package** java.util.concurrent;下

2、Callable<V>提供泛型支持

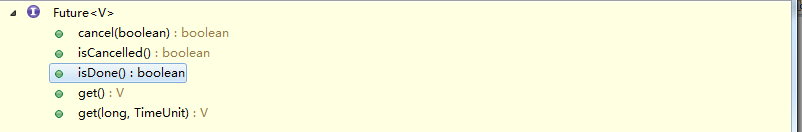
3、Callable<V>的入口方法是call()，Runnable的入口方法是run()

4、call()有返回值，run()没有void

5、call()方法可以throws异常，但是异常不会影响线程池中其他任务的执行，异常需要使用Future.get()结果才能收集到异常信息。

## Future

Future.get()方法是一个阻塞方法，直到有结果。不过取结果的线程和执行任务的线程是解耦的。



## FutureTask

[深入学习futureTask](http://www.importnew.com/25286.html)

使用的是Treiber stack：无锁并发栈[Treiber Stack介绍](http://www.cnblogs.com/micrari/p/7719408.html)

内部任务的执行状态：7种

private volatile int state;

private static final int NEW      = 0;

private static final int COMPLETING   = 1;

private static final int NORMAL     = 2;

private static final int EXCEPTIONAL  = 3;

private static final int CANCELLED   = 4;

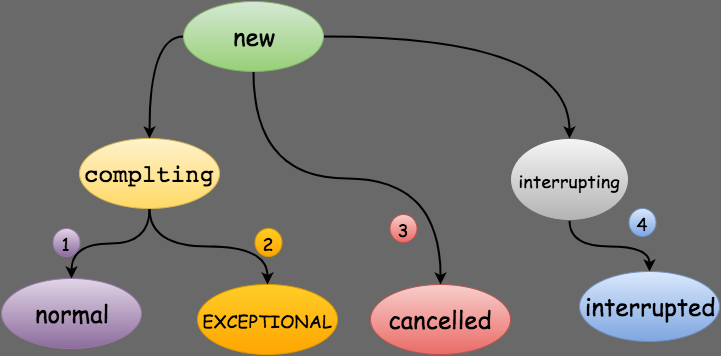
private static final int INTERRUPTING = 5;

private static final int INTERRUPTED  = 6;

* NEW:表示是个新的任务或者还没被执行完的任务。这是初始状态。
* COMPLETING:任务已经执行完成或者执行任务的时候发生异常，但是任务执行结果或者异常原因还没有保存到outcome字段(outcome字段用来保存任务执行结果，如果发生异常，则用来保存异常原因)的时候，状态会从NEW变更到COMPLETING。但是这个状态会时间会比较短，属于中间状态。
* NORMAL:任务已经执行完成并且任务执行结果已经保存到outcome字段，状态会从COMPLETING转换到NORMAL。这是一个最终态。
* EXCEPTIONAL:任务执行发生异常并且异常原因已经保存到outcome字段中后，状态会从COMPLETING转换到EXCEPTIONAL。这是一个最终态。
* CANCELLED:任务还没开始执行或者已经开始执行但是还没有执行完成的时候，用户调用了cancel(false)方法取消任务且不中断任务执行线程，这个时候状态会从NEW转化为CANCELLED状态。这是一个最终态。
* INTERRUPTING: 任务还没开始执行或者已经执行但是还没有执行完成的时候，用户调用了cancel(true)方法取消任务并且要中断任务执行线程但是还没有中断任务执行线程之前，状态会从NEW转化为INTERRUPTING。这是一个中间状态。
* INTERRUPTED:调用interrupt()中断任务执行线程之后状态会从INTERRUPTING转换到INTERRUPTED。这是一个最终态。

有一点需要注意的是，所有值大于COMPLETING的状态都表示任务已经执行完成(任务正常执行完成，任务执行异常或者任务被取消)。

各个状态之间的可能转换关系如下图所示:

[](http://www.importnew.com/?attachment_id=25288)

基于AQS实现的

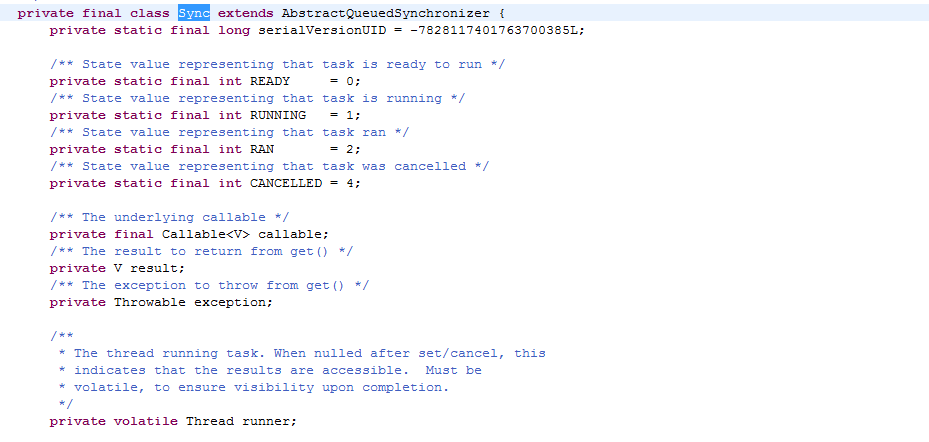
继承关系：

FutureTask<V> **implements** RunnableFuture<V>**extends** Runnable, Future<V>

属性：

**Jdk1.8中，没找到下面代码**

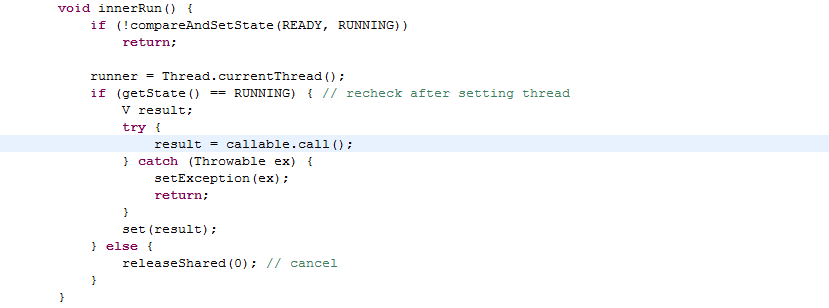
**private** **final** Sync sync;



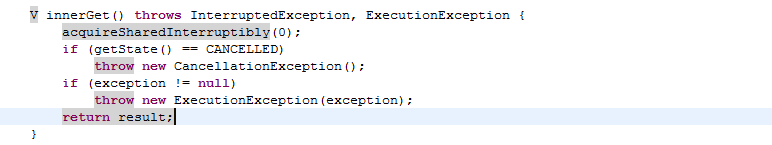
**从这可以看出底层还是基于AQS实现的**

其中result就是返回值

执行run方法



此方法用于获取返回值



## ThreadLocal<T>线程本地变量副本

Thread有个属性，ThreadLocal.ThreadLocalMap threadLocals = null;

ThreadLocalMap是ThreadLocal的静态内部类

static class ThreadLocalMap {

Entry 是ThreadLocalMap的静态内部类，继承弱引用

static class Entry extends WeakReference<ThreadLocal<?>> {  
 */\*\* The value associated with this ThreadLocal. \*/* Object value;  
 Entry(ThreadLocal<?> k, Object v) {  
 super(k);  
 value = v;  
 }  
}

void set(Object value)方法:先获取到当前线程，然后取出当前线程中的map，再将ThreadLocal作为key，要放入的值作为value存入到map中。

public T get()方法：先获取到当前线程，然后取出当前线程中的map，若map不为空，则return出map中的value。若map为空，则调用setInitialValue()方法。

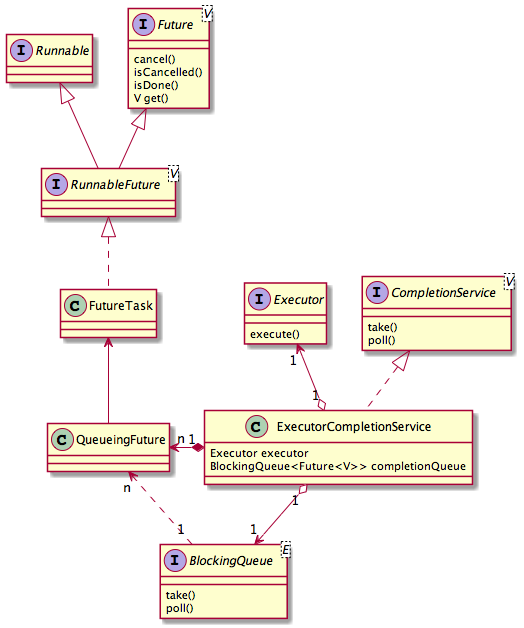
private T setInitialValue()方法：先获取初始值，再获取到当前线程，然后取出当前线程中的map，若map不为空则将ThreadLocal作为key，要放入的值作为value存入到map中，为空则创建map。最后返回value值。

public void remove()：将当前线程局部变量的值删除，目的是为了减少内存的占用，该方法是JDK 5.0新增的方法。需要指出的是，当线程结束后，对应该线程的局部变量将自动被垃圾回收，所以显式调用该方法清除线程的局部变量并不是必须的操作，但它可以加快内存回收的速度。

protected Object initialValue() : 返回该线程局部变量的初始值，该方法是一个protected的方法，显然是为了让子类覆盖而设计的。这个方法是一个延迟调用方法，在线程第1次调用get()或set(Object)时才执行，并且仅执行1次。ThreadLocal中的缺省实现直接返回一个null。

在tomcat中使用，要注意tomcat线程池的问题，线程复用会导致数据错乱，最好是在一个线程结束后调用remove方法。

## CompletionService



**[java并发编程之CompletionService](https://blog.csdn.net/u010185262/article/details/56017175)**

当向Executor提交多个任务并且希望获得它们在完成之后的结果，如果用FutureTask，可以循环获取task，并调用get方法去获取task执行结果，但是如果task还未完成，获取结果的线程将阻塞直到task完成，由于不知道哪个task优先执行完毕，使用这种方式效率不会很高。在jdk5时候提出接口CompletionService，它整合了Executor和BlockingQueue的功能，可以更加方便在多个任务执行时获取到任务执行结果。

可以获得任务结果与任务执行完成的顺序一致，不会出现因为上一个任务没有执行完成，二一致阻塞在future.get()方法上。

## [Java并发编程——this引用逸出("this"Escape)](https://blog.csdn.net/flysqrlboy/article/details/10607295)

# Spring

http://jinnianshilongnian.iteye.com/category/231099

## IOC

1. 定义好Spring的配置文件。  
2. 通过Resource对象将Spring配置文件进行抽象，抽象成一个Resource对象。  
3. 定义好Bean工厂（各种BeanFactory）。  
4. 定义好XmlBeanDefinitionReader对象，并将工厂作为参数传递进去供后续回调使用。  
5. 通过XmlBeanDefinitionReader对象读取之前抽象出的Resource对象（包含了XML文件的解析过程）。  
6. IoC容器创建完毕，用户可以通过容器获取到所需的对象信息。

### 实现原理：

#### 三种启动方式

##### 监听器启动：

ContextLoaderListener与RequestContextListener区别：

ContextLoaderListener(或ContextLoaderServlet)将Web容器与spring容器整合。RequestContextListener将Spring容器与Web容器结合的更加密切。 前者为**必选配置**，后者为**可选配置**，并且后者与**scope=”request”**搭配使用。

在web.xml中配置spring的监听：

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>/WEB-INF/classes/applicationContext-\*.xml</param-value>

</context-param>

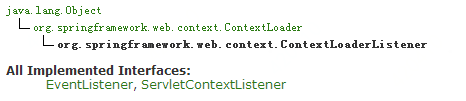
<listener>

<listener-class>org.springframework.web.context.ContextLoaderListener

</listener-class>

</listener>

ContextLoaderListener的继承关系：



当容器启动触发监听实现，调用初始化WebApplicationContext的方法

public void contextInitialized(ServletContextEvent event) {  
 this.initWebApplicationContext(event.getServletContext());  
}

接下来介绍spring容器初始化过程中非常关键的类或接口

**XmlWebApplicationContext：应用的容器，上下文，BeanFactory就是放在这里的**

**DefaultListableBeanFactory：spring IOC的容器**

**BeanDefinitionRegistry：将BeanDefinition注册到beanFactory里**

**XmlBeanDefinitionReader：spring xml文件的阅读器**

**：存放spring配置文件xml解析的资源**

**DefaultBeanDefinitionDecumentReader：xml解析成Decument之后的阅读器**

**BeanDefinitionParserDelegate：BeanDefinition解析授权**

**BeanDefinitionHolder：BeanDefinition的持有者**

**GenericBeanDefinition：里面存放了beanClass全路径名**

其中**DefaultListableBeanFactory是XmlWebApplicationContext的属性，GenericBeanDefinition是DefaultListableBeanFactory的属性，**

**ResourceLoader ApplicationContext**

**Implements extends**

**DefaultResourceLoader ConfiguableApplicationContext**

**extends implemetns**

**AbstractApplicationContext**

**extends**

**AbstractRefreshConfigApplicationContext**

**extends**

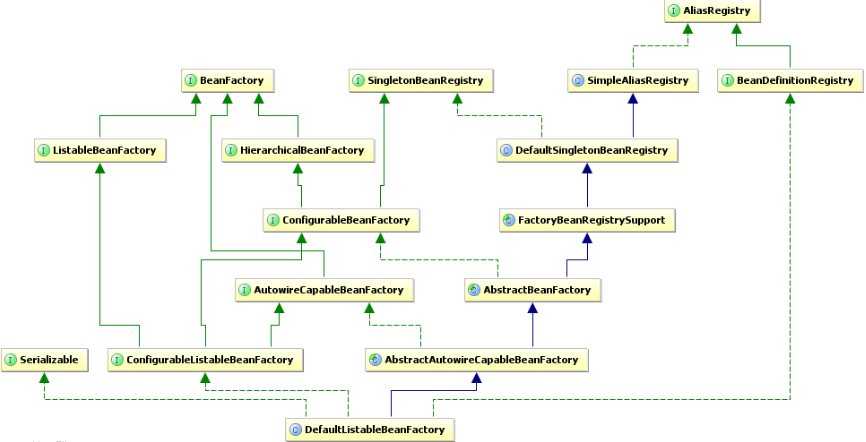
**AbstractRefreshConfigApplicationContext**

**extends**

**AbstractRefreshableWebApplicationContext**

**extends**

**XmlWebApplicationContext**

****

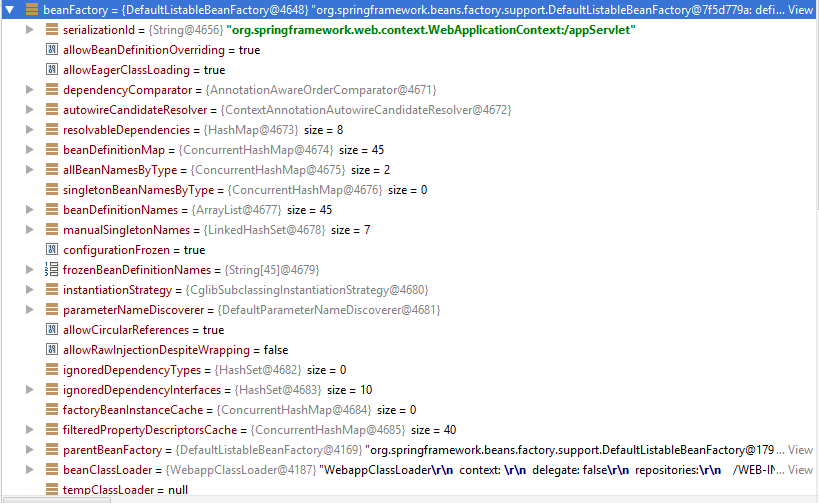
**XmlWebApplicationContext详解**

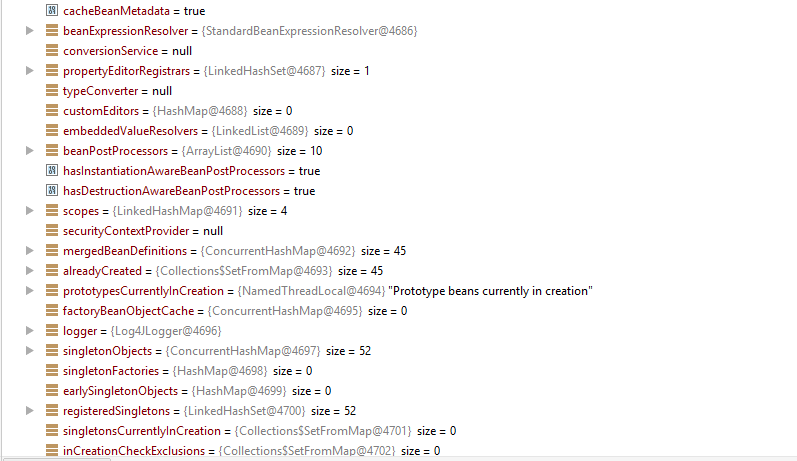
应用的容器，beanFactory作为他的属性，即为spring容器的宿主，并且它继承了ResourceLoader接口，所以也可以作为Resource对象的加载器

bean 实例的初始化是在**AbstractApplicationContext**的

finishBeanFactoryInitialization( beanFactory )方法里进行的，这个方法主要做了以下几件事：

**DefaultListableBeanFactory详解**





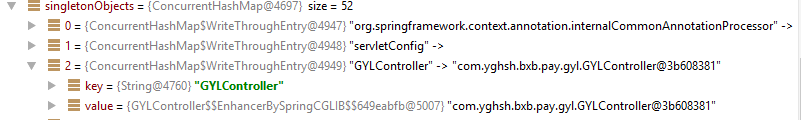


属性：

Map<String, BeanDefinition> beanDefinitionMap

singletonObjects=new ConcurrentHashMap<String, Object>(256);

这个属性是从**DefaultSingletonBeanRegistry**类继承来的



**GenericBeanDefinition详解**

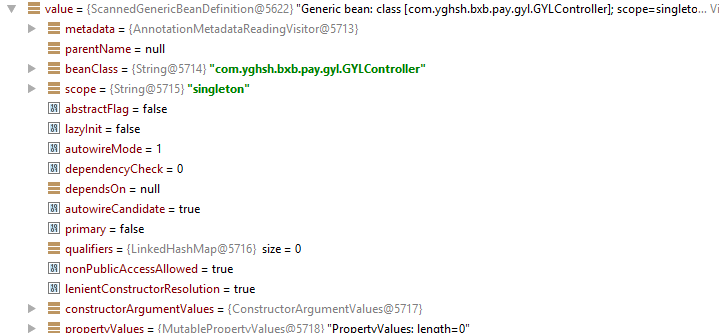
属性：

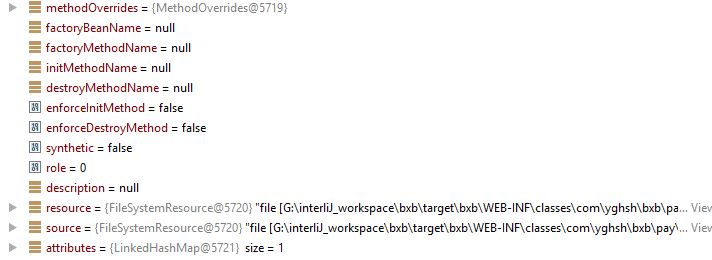
Object beanClass：bean类的全路径名

String scope：范围

boolean lazyInit ：是不是懒加载

int autowireMode





##### servlet方式

spring3.0已经不再支持

<servlet>

<servlet-name>context</servlet-name>

<servlet-class>org.springframework.web.context.ContextLoaderServlet</servlet-class>

<load-on-startup>1</load-on-startup>

</servlet>

Spring3.0以后不再支持

##### plugin配置方式

<plug-in className="org.springframework.web.struts.ContextLoaderPlugIn">

<set-property property="contextConfigLocation" value="/WEB-INF/applicationContext.xml,/WEB-INF/action-servlet.xml" />

</plug-in>

用于和struts整合

## DI

## AOP

[Spring AOP中pointcut expression表达式解析](https://www.cnblogs.com/panie2015/p/5889729.html)

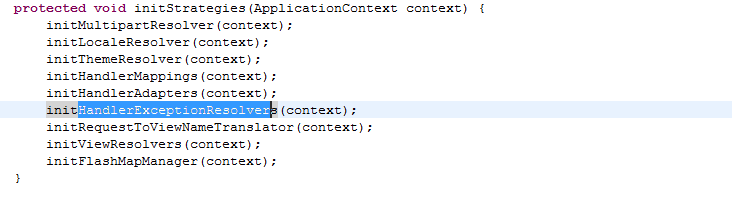
**见spring下的aop原理与使用**

## SpringMVC

使用springMVC在web.xml中配置如下：

<servlet>  
 <servlet-name>mis</servlet-name>  
 <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>  
 <init-param>  
 <param-name>contextConfigLocation</param-name>  
 <param-value>classpath:conf/base-mis-servlet.xml</param-value>  
 </init-param>  
 <load-on-startup>1</load-on-startup>  
</servlet>  
<servlet-mapping>  
 <servlet-name>mis</servlet-name>  
 <url-pattern>/\*</url-pattern>  
</servlet-mapping>

**重要的组件说明：**

****

**MultipartResolver：上传文件处理组件**

**LocaleResolver：国际化处理组件（zh-cn之类）**

**ThemeResolver：主题处理组件**

**HandlerMapping：根据request找到相应的HandlerMethod（处理器）**

**HandlerAdapter：使用处理器处理的组件**

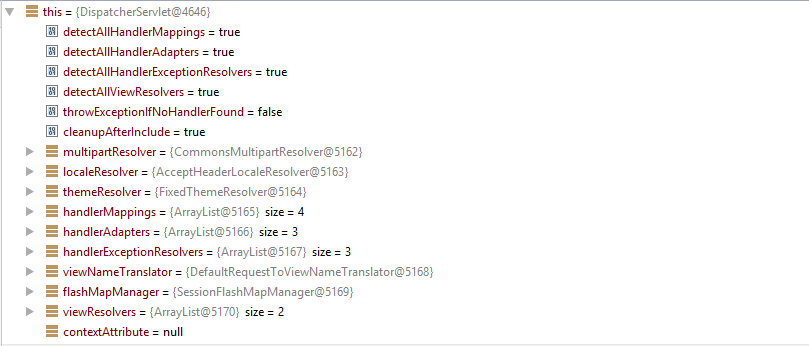
**HandlerExceptionResolver：异常处理组件**

**RequestToViewNameTranslator：**

**ViewResolvers：视图解析器**

**FlashMapManager：重定向传递参数组件**

**DispatcherServlet**

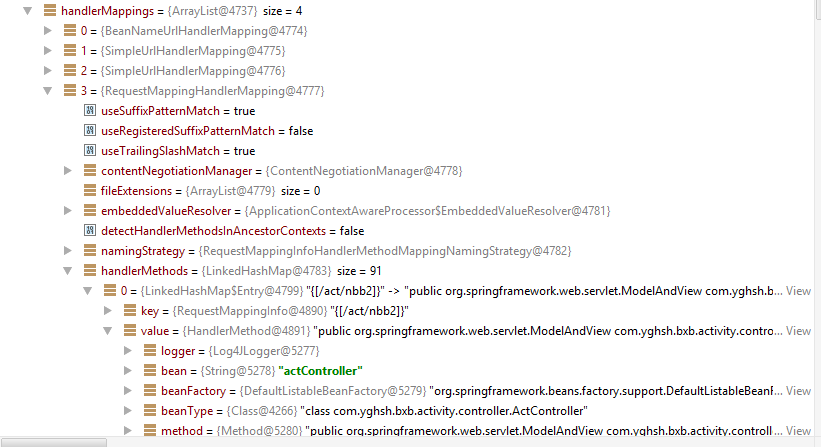




### Init()初始化

#### handlerMapping详解

RequestMappingHandlerMapping **extends** RequestMappingInfoHandlerMapping **extends** AbstractHandlerMethodMapping<RequestMappingInfo> **extends** AbstractHandlerMapping **implements** InitializingBean



当spring初始化handlerMapping这个bean的时候，AbstractHandlerMethodMapping<T>属性加载完会调用afterPropertiesSet()方法，这个方法里调用initHandlerMethods

整个初始化工作由AbstractHandlerMethodMapping的initHandlerMethods主导.

　　1. 取出容器中所有的beanNames

　　2. 迭代类,分别判断isHandler判断目标类是否Handler

　　2.1 RequestMappingHandlerMapping.isHandler根据@Controller或@RequestMapping注解判断(有任意一个)

　　3. 对handler解析出所有需要分发的方法detectHandlerMethods

　　　　3.1 获取原始的Class<?>

　　　　3.2 使用HandlerMethodSelector.selectMethods过滤具体handler method,预留getMappingForMethod模板方法给子类

　　　　　　RequestMappingHandlerMapping.getMappingForMethod根据类,方法上的RequestMapping注解生成匹配条件RequestMappingInfo

　　　　3.3 对过滤到的每个method进行注册registerHandlerMethod

　　　　　　a, 使用createHandlerMethod封装处理器为HandlerMethod

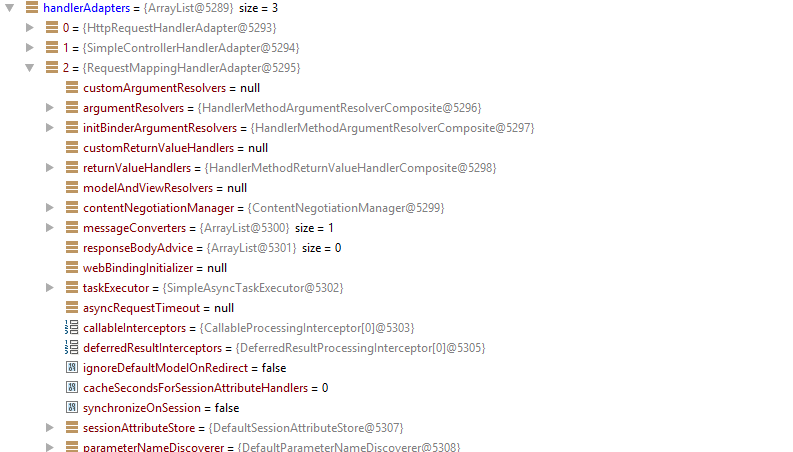
　　　　　　b, 判断之前是否已经匹配条件对应的处理器是否冲突(相同的匹配条件只能有一个对应的处理器)

　　　　　　c, 设置匹配条件到handler method的映射关系

　　　　　　d, 从匹配条件中解析出url,并注册到urlMap(url到匹配条件的映射),这边由RequestMappingInfoHandlerMapping.getMappingPathPatterns

#### handlerAdapter详解

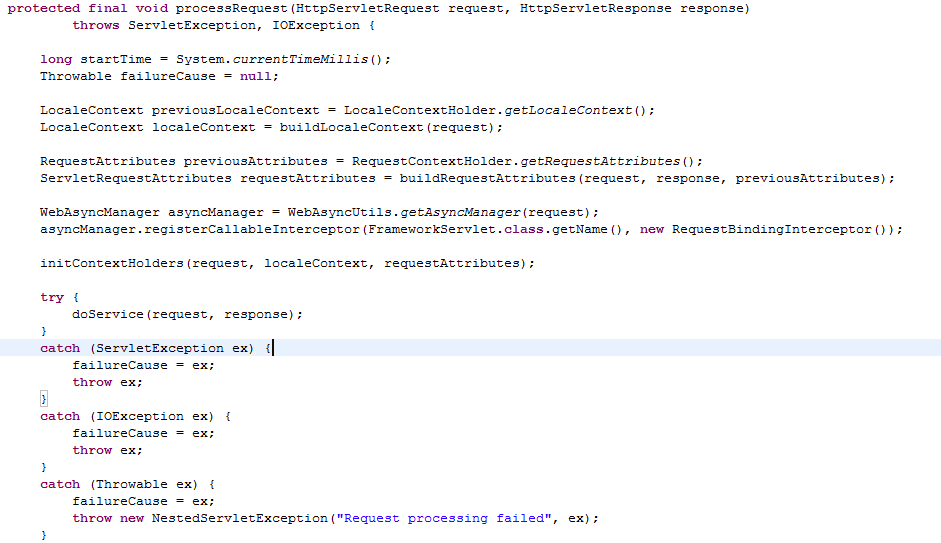
RequestMappingHandlerAdapter **extends** AbstractHandlerMethodAdapter  
 **implements** BeanFactoryAware, InitializingBean

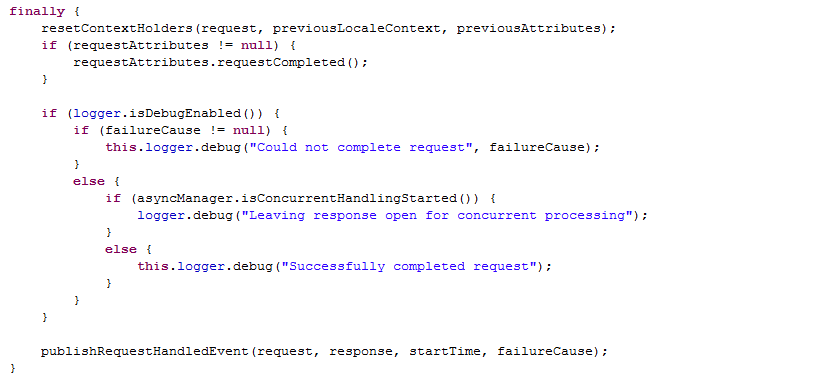


### service()方法

#### FrameworkServlet类

重写了doGet、doPost等方法，然后全部调用processRequest（）方法





最核心的是doService(request, response);方法，此为模板方法，在DispatcherServlet中实现

此方法主要做了两件事：

1.对LocaleContext和RequestAttributes的设置和恢复

LocaleContext：本地化信息，如zh-cn

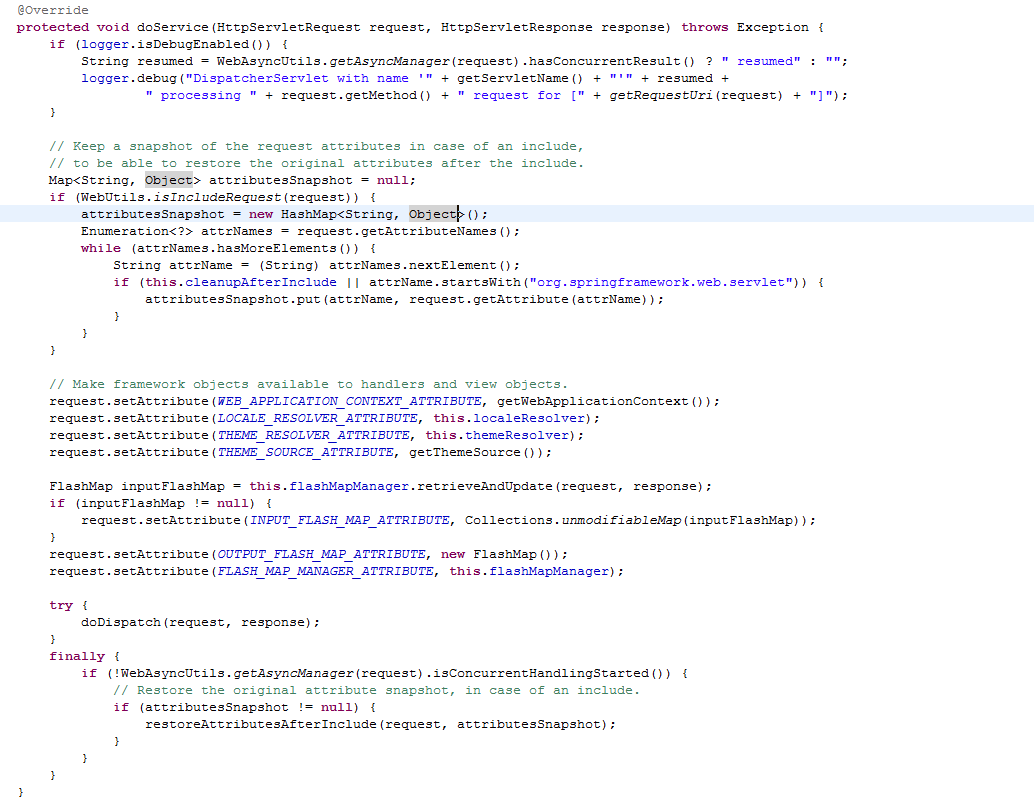
RequestAttributes：可以通过它操作request或者session中参数

2.处理完发布了一个ServletRequestHandledEvent事件消息

#### DispatcherServlet

springMVC最核心的类

##### doService（）方法



1.检查是不是include方法，是就对request的Attribute做一个快照备份

2.对request设置一些属性

FlashMap：主要用于redirect转发时的参数传递

redirect传递参数的三种方法：

Controller的handle方法，引入参数RedirectAttributes attr

然后attr.addFlashAttribute(“orderId”,”123”);

3.调用**doDispatch(request, response);**

##### doDispatch





1.根据request找到handler

2.根据handler找到对应的HandlerAdapter

预处理（拦截器实现）

3.用HandlerAdapter处理Handler

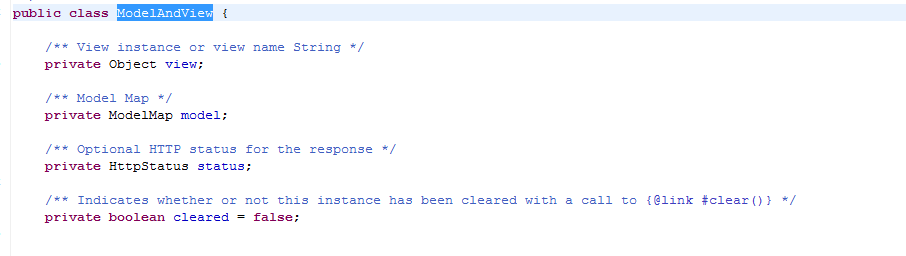
后处理（拦截器实现）

4.调用processDispatchResult处理上面结果（包含找到view并渲染输出给用户）

multipartRequestParsed 是不是上传请求标示

这里面牵涉到几个类：

ModelAndView



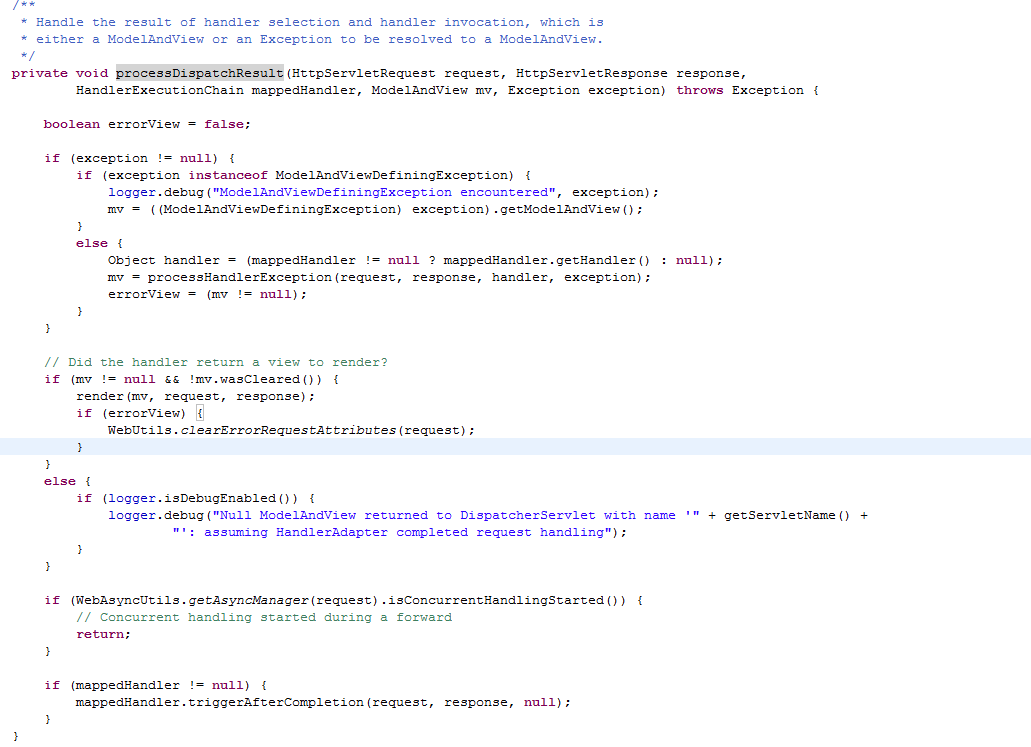
HandlerExecutionChain

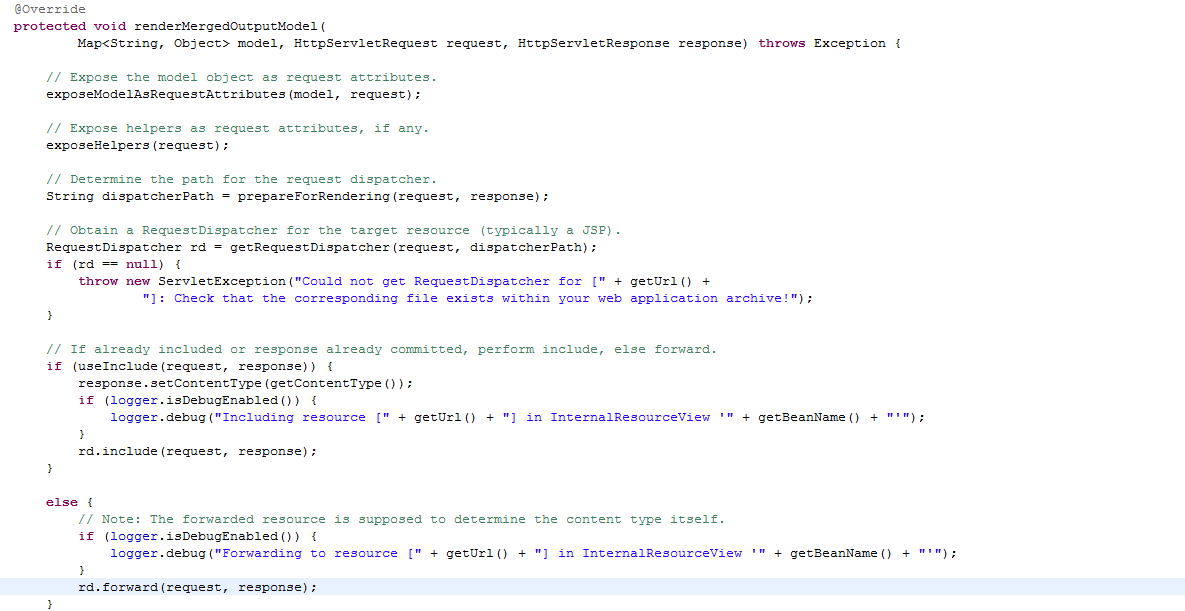
处理请求的处理器链（包含处理器和对应的interceptor拦截器）

执行的顺序是，请求的时候先执行拦截器的preHandle，再执行Handler，然后再执行postHandler方法

Controller的方法就是在Handler里面执行的

##### 渲染页面





## springMVC实战问题

405错误：[Spring MVC Request method 'POST' not supported错误](https://blog.csdn.net/ruiruiafafafa/article/details/51507150)

# servlet

[注解@PostConstruct与@PreDestroy详解及实例](https://blog.csdn.net/wo541075754/article/details/52174900)

# RabbitMQ消息中间件

RabbitMQ是一个开源的，RabbitMQ是实现AMQP（高级消息队列协议）的消息中间件的一种，最初起源于金融系统，用于在分布式系统中存储转发消息，在易用性、扩展性、高可用性等方面表现不俗。消息中间件主要用于组件之间的解耦，消息的发送者无需知道消息使用者的存在，反之亦然。

支持主流的操作系统Linux、Windows、MacOX等。

多种开发语言支持java、Python、Ruby、.NET、PHP、C/C++、node.js等

## 安装

**Centos6.6环境**

rabbitMQ是用erlang语言开发的，所以安装前得安装erlang的环境

1、yum安装依赖包

yum -y install  gtk+\*

yum -y install ncurses-devel

yum -y install unixODBC-devel

yum -y install openssl-devel

2、安装wxWidgets

上传wxWidgets-3.1.0.zip到/application/hadoop/RabbitMQ

./configure --with-gtk --enable-unicode --disable-shared

make

安装wxWidgets:

root 用户执行

make install

3、安装erlang

上传otp\_src\_17.1.tar.gz到：/application/hadoop/RabbitMQ



1、解压

tar –zxvf otp\_src\_17.1.tar.gz

2、切换到otp\_src\_17.1目录

cd otp\_src\_17.1

执行命令：

第一步配置：

./configure --prefix=/usr/local/erlang --with-ssl -enable-threads -enable-smmp-support -enable-kernel-poll --enable-hipe --without-javac

上述参数说明：

--prefix  指定安装目录

--enable-smp-support启用对称多处理支持（Symmetric Multi-Processing对称多处理结构的简称）

--enable-threads启用异步线程支持

--enable-sctp启用流控制协议支持（Stream Control Transmission Protocol，流控制传输协议）

--enable-kernel-poll启用Linux内核poll

--enable-hipe启用高性能Erlang（High Performance Erlang）参考资料：http://www.cnblogs.com/me-sa/archive/2012/10/09/erlang\_hipe.html

--with-ssl使用SSL包

第二步编译：

make

第三步安装：

用root用户

make install

配置环境变量

export PATH=$PATH:/usr/local/erlang/bin;/usr/lib/rabbitmq/bin

source /etc/profile

4、rpm安装rabbitMQ

上传rabbitmq-server-3.6.5-1.noarch.rpm到/application/hadoop/RabbitMQ

root 用户执行

执行命令：rpm -ivh --nodeps rabbitmq-server-3.6.5-1.noarch.rpm

bin路径：/usr/lib/rabbitmq/bin

配置文件模板就在这：/usr/share/doc/rabbitmq-server-3.6.5

RabbitMQ的根目录：/var/lib/rabbitmq  
RabbitMQ的配置文件目录：/etc/rabbitmq/  
RabbitMQ的日志文件目录：/var/log/rabbitmq  
两个配置文件分别是：  
rabbitmq.config   
rabbitmq-env.conf



5. rabbitmq-server-generic-unix-3.6.6.zip安装

安装erlang环境

**yum安装依赖包**

yum -y install  gtk+\*

yum -y install ncurses-devel

yum -y install unixODBC-devel

yum -y install openssl-devel

上传otp\_src\_17.1.tar.gz到/application/hadoop/RabbitMQ/erlang

解压缩：tar -zxvf otp\_src\_17.1.tar.gz

切换到otp\_src\_17.1目录

**接下来用root账户操作**

配置安装：

两个命令

**./configure --prefix=/application/hadoop/RabbitMQ/erlang --with-ssl -enable-threads -enable-smmp-support -enable-kernel-poll --enable-hipe --without-javac**

上述参数说明：

--prefix  指定安装目录

--enable-smp-support启用对称多处理支持（Symmetric Multi-Processing对称多处理结构的简称）

--enable-threads启用异步线程支持

--enable-sctp启用流控制协议支持（Stream Control Transmission Protocol，流控制传输协议）

--enable-kernel-poll启用Linux内核poll

--enable-hipe启用高性能Erlang（High Performance Erlang）参考资料：http://www.cnblogs.com/me-sa/archive/2012/10/09/erlang\_hipe.html

--with-ssl使用SSL包

**make && make install**

解压：rabbitmq-server-generic-unix-3.6.6.zip

配置erlang的环境变量

vi /etc/profile

export PATH=$PATH:/application/hadoop/RabbitMQ/erlang/bin

export PATH=$PATH:/application/hadoop/RabbitMQ/rabbitmq\_server-3.6.6/sbin

source /etc/profile

给配置文件赋值权限：

chmod 755 rabbitmq-env rabbitmq-plugins rabbitmq-server rabbitmq-defaults rabbitmqctl

### 配置：

Rabbitmq的配置文件有三个，位于**/etc/rabbitmq/**

**rabbitmq.conf**,设置rabbitmq的运行参数。该配置文件中的每个参数为一个erlang的 tuple，结构为｛Key，Value｝, Key为atom类型， Value为一个term。其中几个关键参数为:

tcp\_listerners #设置rabbimq的监听端口，默认为[5672]。

disk\_free\_limit #磁盘低水位线，若磁盘容量低于指定值则停止接收数据，默认值为{mem\_relative, 1.0},即与内存相关联1：1，也可定制为多少byte.

vm\_memory\_high\_watermark #设置内存低水位线，若低于该水位线，则开启流控机制，默认值是0.4，即内存总量的40%。

hipe\_compile #将部分rabbimq代码用High Performance Erlang compiler编译，可提升性能，该参数是实验性，若出现erlang vm segfaults，应关掉。

force\_fine\_statistics #该参数属于rabbimq\_management，若为true则进行精细化的统计，但会影响性能。

frame\_max #包大小，若包小则低延迟，若包则高吞吐，默认是131072=128K。

heartbeat #客户端与服务端心跳间隔，设置为0则关闭心跳，默认是600秒。

**Rabbitmq-env.conf**

RABBITMQ\_NODE\_IP\_ADDRESS= //IP地址，空串bind所有地址，指定地址bind指定网络接口

RABBITMQ\_NODE\_PORT= //TCP端口号，默认是5672

RABBITMQ\_NODENAME= //节点名称。默认是rabbit

RABBITMQ\_CONFIG\_FILE= //配置文件路径 ，即rabbitmq.config文件路径

RABBITMQ\_MNESIA\_BASE= //mnesia所在路径

RABBITMQ\_LOG\_BASE= //日志所在路径

RABBITMQ\_PLUGINS\_DIR= //插件所在路径

### 启动：

cd /application/hadoop/RabbitMQ/rabbitmq\_server-3.6.6/sbin

安装web管理插件

./rabbitmq-plugins enable rabbitmq\_management

./rabbitmq-server start

### 创建一个用户：

./rabbitmqctl add\_user root root

./rabbitmqctl set\_permissions -p / root ".\*" ".\*" ".\*"

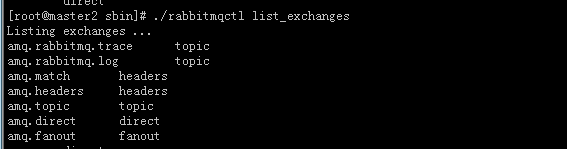
./rabbitmqctl set\_user\_tags root administrator

### rabbitmqctl命令详解

1、上面已经有了创建一个用户

2、列出当前所有的exchange

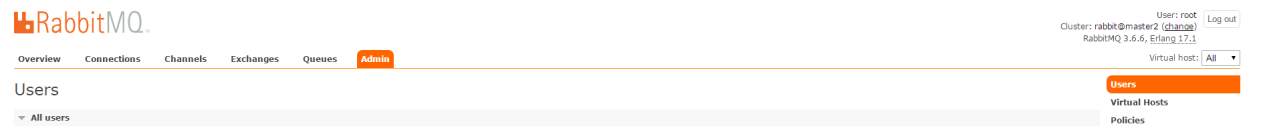
./rabbitmqctl list\_exchanges



### 管理界面

可以用新创建的root用户登录





在Users里可以添加用户

在Virtual Hosts中可以添加虚拟机（相当于mysql中的数据库），并与用户关联

## 原理

AMQP（Advanced Message Queue Protocol）

高级消息队列协议

AMQP当中有四个概念非常重要: 虚拟主机（virtual host），交换机（exchange），队列（queue）和绑定（binding）。一个虚拟主机持有一组交换机、队列和绑定。为什么需要多个虚拟主机呢？很简单，RabbitMQ当中，用户只能在虚拟主机的粒度进行权限控制。因此，如果需要禁止A组访问B组的交换机/队列/绑定，必须为A和B分别创 建一个虚拟主机。每一个RabbitMQ服务器都有一个默认的虚拟主机“/”。

Producer 要产生消息必须要创建一个 Exchange ，Exchange 用于转发消息，但是它不会做存储，如果没有 Queue bind 到 Exchange 的话，它会直接丢弃掉 Producer 发送过来的消息，当然如果消息总是发送过去就被直接丢弃那就没有什么意思了，一个 Consumer 想要接受消息的话，就要创建一个 Queue ，并把这个 Queue bind 到指定的 Exchange 上，然后 Exchange 会把消息转发到 Queue 那里，Queue 会负责存储消息，Consumer 可以通过主动 Pop 或者是 Subscribe 之后被动回调的方式来从 Queue 中取得消息。

queue对load balance的处理是完美的。对于多个Consumer来说，RabbitMQ 使用循环的方式（round-robin）的方式均衡的发送给不同的Consumer。

## 使用

rabbitMQ工作模式和使用场景

|  |  |  |
| --- | --- | --- |
| **序号** | **工作模式** | **说明** |
| 1 | 简单模式：单发送单接收  http://images.cnitblog.com/i/434101/201408/171507104838529.png | 简单的发送与接收，没有特别的处理。 |
| 2 | 工作模式：单发送多接收（广播）  http://images.cnitblog.com/i/434101/201408/171513043112193.png | 一个发送端，多个接收端，如分布式的任务派发。为了保证消息发送的可靠性，不丢失消息，使消息持久化了。同时为了防止接收端在处理消息时down掉，只有在消息处理完成后才发送ack消息。 |
| 3 | 发布订阅模式：Publish/Subscribe  http://images.cnitblog.com/i/434101/201408/171657207955618.png | 发布、订阅模式，发送端发送广播消息，多个接收端接收。 |
| \*4 | 路由模式：Routing (按路线发送接收)  http://images.cnitblog.com/i/434101/201408/171709256231158.png | 发送端按routing key发送消息，不同的接收端按不同的routing key接收消息。  在实际项目中普遍采用Direct路由方式 |
| 5 | 主题模式：Topics (按topic发送接收)  http://images.cnitblog.com/i/434101/201408/171739210926049.png | 发送端不只按固定的routing key发送消息，而是按字符串“匹配”发送，接收端同样如此。 |

### 与spring整合

依赖jar包

<!-- 消息队列 -->

<dependency>

<groupId>com.rabbitmq</groupId>

<artifactId>amqp-client</artifactId>

<version>3.5.1</version>

</dependency>

<dependency>

<groupId>org.springframework.amqp</groupId>

<artifactId>spring-rabbit</artifactId>

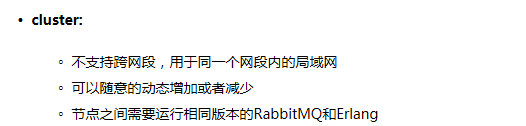
<version>1.4.0.RELEASE</version>

</dependency>

## 分布式集群架构和高可用性（HA）

### 1、集群配置方式

三种方式：cluster、federation（联盟）、shovel



**federation:**应用于广域网，允许单台服务器上的交换机或队列接收发布到另一台服务器上交换机或队列的消息，可以是单独机器或集群。federation队列类似于单向点对点连接，消息会在联盟队列之间转发任意次，直到被消费者接受。通常使用federation来连接internet上的中间服务器，用作订阅分发消息或工作队列。

**shovel:**连接方式与federation的连接方式类似，但它工作在更低层次。可以应用于广域网。

### 2、节点类型

**RAM node:**内存节点将所有的队列、交换机、绑定、用户、权限和vhost的***元数据***定义存储在内存中，好处是可以使得像交换机和队列声明等操作更加的快速。

**Disk node:**将元数据存储在磁盘中，单节点系统只允许磁盘类型的节点，防止重启RabbitMQ的时候，丢失系统的配置信息。

**问题说明：** RabbitMQ要求在集群中至少有一个磁盘节点，所有其他节点可以是内存节点，当节点加入或者离开集群时，必须要将该变更通知到至少一个磁盘节点。如果集群中唯一的一个磁盘节点崩溃的话，集群仍然可以保持运行，但是无法进行其他操作（增删改查），直到节点恢复。   
**解决方案：**设置两个磁盘节点，至少有一个是可用的，可以保存元数据的更改。

### 3 Erlang Cookie

[Erlang Cookie](http://www.rabbitmq.com/clustering.html)是保证不同节点可以相互通信的密钥，要保证集群中的不同节点相互通信必须共享相同的Erlang Cookie。具体的目录存放在/var/lib/rabbitmq/.erlang.cookie。

**说明：** 这就要从rabbitmqctl命令的工作原理说起，RabbitMQ底层是通过Erlang[**架构**](http://lib.csdn.net/base/16)来实现的，所以rabbitmqctl会启动Erlang节点，并基于Erlang节点来使用Erlang系统连接RabbitMQ节点，在连接过程中需要正确的Erlang Cookie和节点名称，Erlang节点通过交换Erlang Cookie以获得认证。

### 4 镜像队列

**功能和原理**   
RabbitMQ的Cluster集群模式一般分为两种，普通模式和镜像模式。

* **普通模式：**默认的集群模式，以两个节点（rabbit01、rabbit02）为例来进行说明。对于Queue来说，消息实体只存在于其中一个节点rabbit01（或者rabbit02），rabbit01和rabbit02两个节点仅有相同的元数据，即队列的结构。当消息进入rabbit01节点的Queue后，consumer从rabbit02节点消费时，RabbitMQ会临时在rabbit01、rabbit02间进行消息传输，把A中的消息实体取出并经过B发送给consumer。所以consumer应尽量连接每一个节点，从中取消息。即对于同一个逻辑队列，要在多个节点建立物理Queue。否则无论consumer连rabbit01或rabbit02，出口总在rabbit01，会产生瓶颈。当rabbit01节点故障后，rabbit02节点无法取到rabbit01节点中还未消费的消息实体。如果做了消息持久化，那么得等rabbit01节点恢复，然后才可被消费；如果没有持久化的话，就会产生消息丢失的现象。
* **镜像模式：**将需要消费的队列变为镜像队列，存在于多个节点，这样就可以实现RabbitMQ的HA高可用性。作用就是消息实体会主动在镜像节点之间实现同步，而不是像普通模式那样，在consumer消费数据时临时读取。缺点就是，集群内部的同步通讯会占用大量的网络带宽。

**实现机制**   
镜像队列实现了RabbitMQ的高可用性（HA），具体的实现策略如下所示：

| **ha-mode** | **ha-params** | **功能** |
| --- | --- | --- |
| all | 空 | 镜像队列将会在整个集群中复制。当一个新的节点加入后，也会在这 个节点上复制一份。 |
| exactly | count | 镜像队列将会在集群上复制count份。如果集群数量少于count时候，队列会复制到所有节点上。如果大于Count集群，有一个节点crash后，新进入节点也不会做新的镜像。 |
| nodes | node name | 镜像队列会在node name中复制。如果这个名称不是集群中的一个，这不会触发错误。如果在这个node list中没有一个节点在线，那么这个queue会被声明在client连接的节点。 |

### 5、设置hosts文件

vi /etc/hosts

192.168.1.28 master

192.168.1.22 master2

192.168.1.23 slave2

### 6、设置不同节点间同一认证的Erlang Cookie

cookie 一般在root目录下就有，如果用rpm安装的就在/var/lib/rabbitmq下

将192.168.1.22上的.erlang.cookie复制到其他节点上去/root

### 7、使用 -detached运行各节点

rabbitmqctl stop

rabbitmq-server --detached &

### 8、创建并部署集群

在192.168.1.23\192.168.1.28机器上，分别执行

rabbitmqctl stop\_app

rabbitmqctl join\_cluster rabbit@master2

rabbitmqctl start\_app

将其他两个节点加到集群里去

可以在添加节点到集群命令中添加参数：--ram

rabbitmqctl join\_cluster --ram rabbit@master2

指定节点类型是RAM的还是DISC的

### 9、RabbitMQ Cluster管理

1、查询集群状态

rabbitmqctl cluster\_status

2、更改节点类型（内存还是磁盘）

rabbitmqctl stop\_app  
rabbitmqctl change\_cluster\_node\_type disc  
或  
rabbitmqctl change\_cluster\_node\_type ram  
rabbitmqctl start\_app

3、镜像队列的设置

镜像队列的配置通过添加policy完成，policy添加的命令为：

rabbitmqctl  set\_policy  [-p Vhost]  Name  Pattern  Definition  [Priority]

-p Vhost:  可选参数，针对指定vhost下的queue进行设置

Name:  policy的名称

Pattern:  queue的匹配模式（正则表达式）

Definition:  镜像定义，包括三个部分 ha-mode，ha-params，ha-sync-mode

                 ha-mode:  指明镜像队列的模式，有效值为 all/exactly/nodes

                                all表示在集群所有的节点上进行镜像

                                exactly表示在指定个数的节点上进行镜像，节点的个数由ha-params指定

                                nodes表示在指定的节点上进行镜像，节点名称通过ha-params指定

                 ha-params: ha-mode模式需要用到的参数

                 ha-sync-mode:  镜像队列中消息的同步方式，有效值为automatic，manually

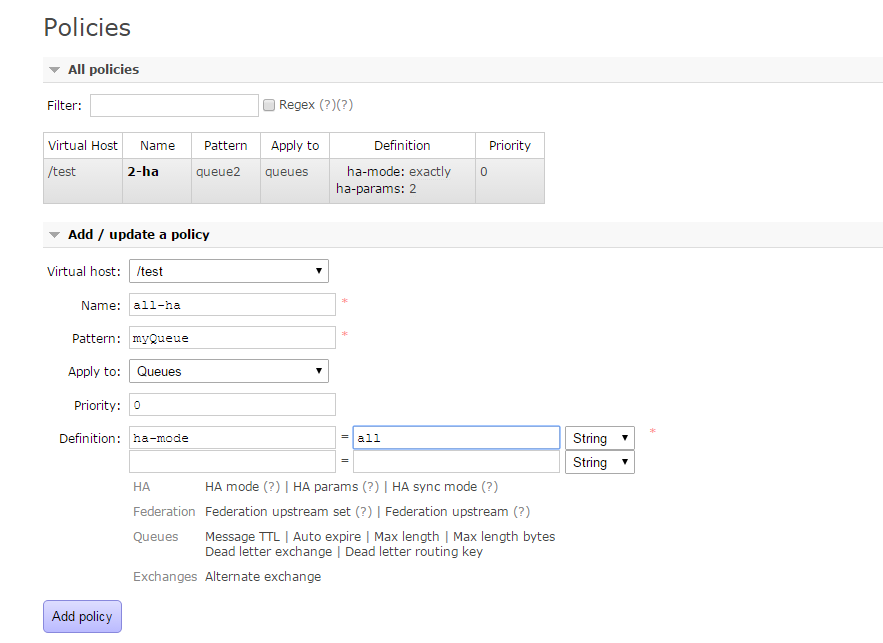
Priority:  可选参数， policy的优先级

例如，对队列名称以hello开头的所有**队列**进行镜像，并在集群的两个节点上完成镜像，policy的设置命令为：

rabbitmqctl set\_policy 2-ha "queue2" '{"ha-mode":"exactly","ha-params":2,"ha-sync-mode":"automatic"}'

也可以在web管理后台设置：

admin🡪policies



### 10、rabbitMQ负载均衡

#### 1、介绍HAProxy

HAProxy提供高可用性、负载均衡以及基于TCP和HTTP应用的代理

**特点**：

1、HAProxy是支持虚拟主机的，，并能支持上万级别的连接;  
  2、能够补充Nginx的一些缺点比如Session的保持，cookie的引导等工作;  
  3、支持url检测后端的服务器出问题的检测会有很好的帮助;  
  4、它跟LVS一样，本身仅仅就只是一款负载均衡软件；单纯从效率上来讲HAProxy更会比Nginx有更出色的负载均衡速度，在并发处理上也是优于Nginx的;  
  5、HAProxy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，不过在后端的MySQL slaves数量超过10台时性能不如LVS，所以我向大家推荐LVS+Keepalived;  
  6、能够提供4层，7层代理。HAProxy支持两种主要的代理模式："**tcp"**也即4层（大多用于邮件服务器、内部协议通信服务器等），和7层（**HTTP**）。在4层模式 下，HAProxy仅在客户端和服务器之间转发双向流量，7层模式下，HAProxy会分析协议，并且能通过允许、拒绝、交换、增加、修改或者删除请求 (request)或者回应(response)里指定内容来控制协议，这种操作要基于特定规则；  
  7、HAProxy的算法现在也越来越多了，具体有如下8种：  
     ①roundrobin，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；  
     ②static-rr，表示根据权重，建议关注；  
     ③leastconn，表示最少连接者先处理，建议关注；  
     ④source，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；  
     ⑤ri，表示根据请求的URI；  
     ⑥rl\_param，表示根据请求的URl参数'balance url\_param' requires an URL parameter name；  
     ⑦hdr(name)，表示根据HTTP请求头来锁定每一次HTTP请求；  
     ⑧rdp-cookie(name)，表示根据据cookie(name)来锁定并哈希每一次TCP请求。

#### 2、安装HAProxy1.6

版本：haproxy-1.6.3.tar.gz

上传安装包至：/application/hadoop

解压：tar -zxvf haproxy-1.6.3.tar.gz

编译安装：

make TARGET=linux26 ARCH=x86\_64 PREFIX=/application/hadoop/haproxy

make install PREFIX=/application/hadoop/haproxy

#**参数说明**

TARGET=linux26 #内核版本，使用uname -r查看内核，如：2.6.18-371.el5，此时该参数就为linux26；kernel 大于2.6.28的用：TARGET=linux2628

ARCH=x86\_64 #系统位数

PREFIX=/usr/local/haprpxy #/usr/local/haprpxy为haprpxy安装路径

#### 3、配置文件设置

路径：/application/hadoop/haproxy/sbin/haproxy.cfg

这个文件需要自己手动创建

启动在sbin路径下：./haproxy -f haproxy.cfg

##### 负载均衡算法：

一、roundrobin，表示简单的轮询，每个服务器根据权重轮流使用，在服务器的处理时间平均分配的情况下这是最流畅和公平的算法。该算法是动态的，对于实例启动慢的服务器权重会在运行中调整。

二、static-rr，表示根据权重，建议关注；每个服务器根据权重轮流使用，类似roundrobin，但它是静态的，意味着运行时修改权限是无效的。另外，它对服务器的数量没有限制。

三、leastconn，表示最少连接者先处理，建议关注；leastconn建议用于长会话服务，例如LDAP、SQL、TSE等，而不适合短会话协议。如HTTP.该算法是动态的，对于实例启动慢的服务器权重会在运行中调整。

四、source，表示根据请求源IP，建议关注；对请求源IP地址进行哈希，用可用服务器的权重总数除以哈希值，根据结果进行分配。

只要服务器正常，同一个客户端IP地址总是访问同一个服务器。如果哈希的结果随可用服务器数量而变化，那么客户端会定向到不同的服务器；

该算法一般用于不能插入cookie的Tcp模式。它还可以用于广域网上为拒绝使用会话cookie的客户端提供最有效的粘连；

该算法默认是静态的，所以运行时修改服务器的权重是无效的，但是算法会根据“hash-type”的变化做调整。

五、uri，表示根据请求的URI；表示根据请求的URI左端（问号之前）进行哈希，用可用服务器的权重总数除以哈希值，根据结果进行分配。

只要服务器正常，同一个URI地址总是访问同一个服务器。

一般用于代理缓存和反病毒代理，以最大限度的提高缓存的命中率。该算法只能用于HTTP后端；

该算法一般用于后端是缓存服务器；

该算法默认是静态的，所以运行时修改服务器的权重是无效的，但是算法会根据“hash-type”的变化做调整。

六、url\_param，表示根据请求的URl参数'balance url\_param' requires an URL parameter name

在HTTP GET请求的查询串中查找<param>中指定的URL参数，基本上可以锁定使用特制的URL到特定的负载均衡器节点的要求；

该算法一般用于将同一个用户的信息发送到同一个后端服务器；

该算法默认是静态的，所以运行时修改服务器的权重是无效的，但是算法会根据“hash-type”的变化做调整。

七、hdr(name)，表示根据HTTP请求头来锁定每一次HTTP请求；

在每个HTTP请求中查找HTTP头<name>，HTTP头<name>将被看作在每个HTTP请求，并针对特定的节点；

如果缺少头或者头没有任何值，则用roundrobin代替；

该算法默认是静态的，所以运行时修改服务器的权重是无效的，但是算法会根据“hash-type”的变化做调整。

八、rdp-cookie(name)，表示根据据cookie(name)来锁定并哈希每一次TCP请求。

为每个进来的TCP请求查询并哈希RDP cookie<name>；

该机制用于退化的持久模式，可以使同一个用户或者同一个会话ID总是发送给同一台服务器。

如果没有cookie，则使用roundrobin算法代替；

该算法默认是静态的，所以运行时修改服务器的权重是无效的，但是算法会根据“hash-type”的变化做调整。

#其实这些算法各有各的用法，我们平时应用得比较多的应该是roundrobin、source和lestconn。

haproxy负载均衡算法

# Zookeeper

[**ZooKeeper客户端curator组件介绍**](https://blog.csdn.net/zhu_tianwei/article/details/42047499)

# 设计模式

装饰模式

模板方法模式

工厂模式

代理模式

## 适配器模式

**概念**：

适配器模式把一个类的接口变换成客户端所期待的另一种接口，从而使原本因接口不匹配而无法在一起工作的两个类能够在一起工作

**描述**：

用电器做例子，笔记本电脑的插头一般都是三相的，即除了阳极、阴极外，还有一个地极。而有些地方的电源插座却只有两极，没有地极。电源插座与笔记本电脑的电源插头不匹配使得笔记本电脑无法使用。这时候一个三相到两相的转换器（适配器）就能解决此问题，而这正像是本模式所做的事情

1、类适配器模式

原对象：

适配对象：

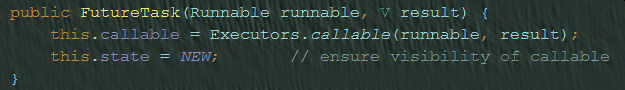
目标对象：

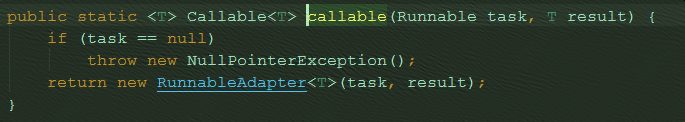
继承源类实现目标接口

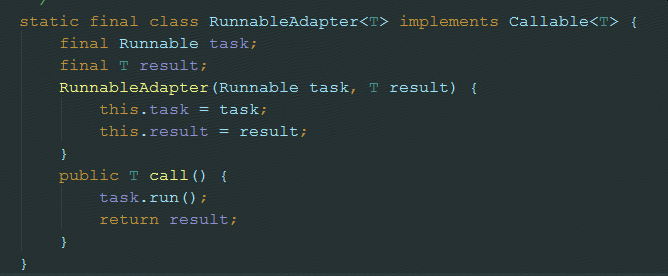
2、对象适配器模式

Executors类中使用了该模式，就是实现目标接口，持有原对象。

RunnableAdapter可以将Runnable转换成Callable，在FutureTask的构造函数中用







3、接口适配器模式

## 命令模式

[设计模式-命令模式](http://www.cnblogs.com/f-zhao/p/6203208.html)

命令模式有三个角色

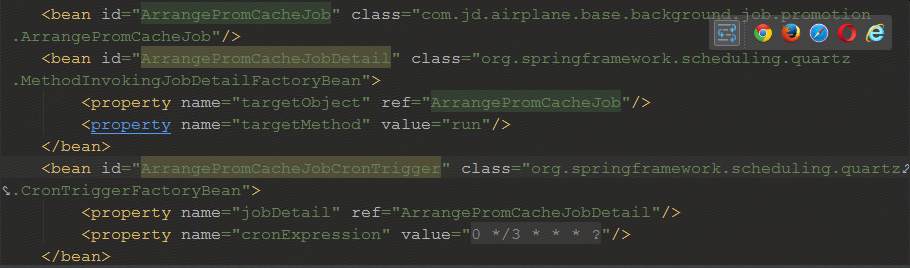
Command：要执行的命令，execute()方法

Invoker：命令的入口，会持有command

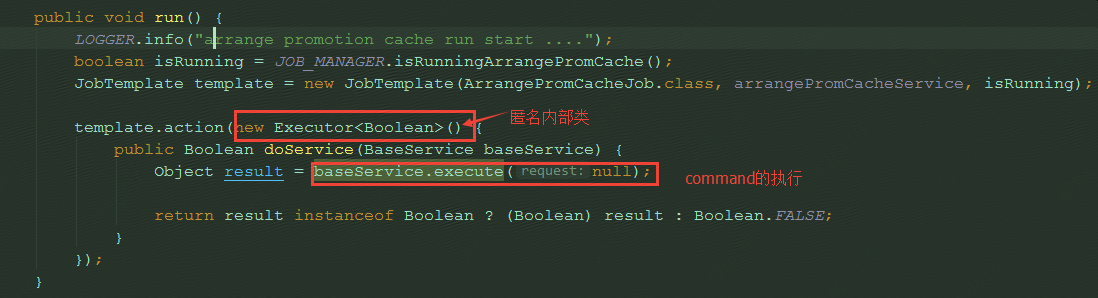
Receiver：命令的接受者

项目中的应用：

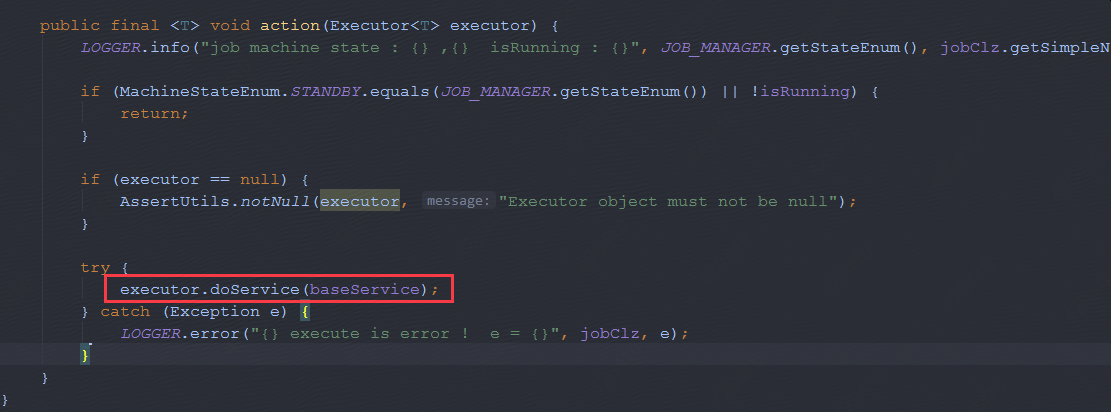
Airbase\_backgroud中的worker执行，程序入口是ArrangePromCacheJob类（spring的quartz任务配置定时，入口方法是run()）

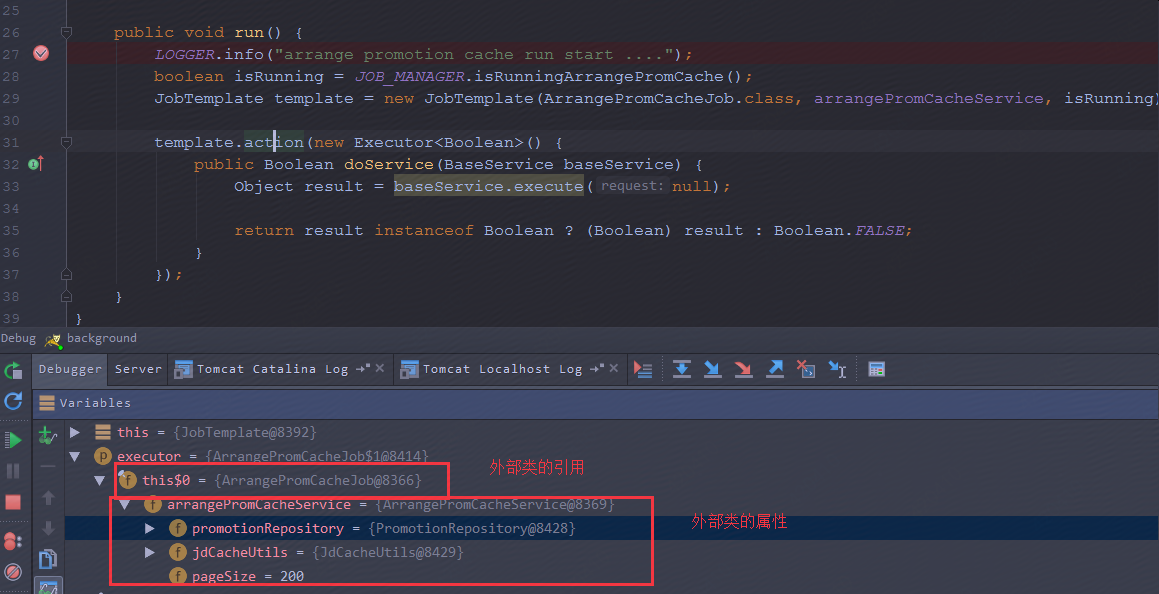


Run()中持有JonTemplate对象，调用JonTemplate.action(Executor)，使用匿名内部类



Jobtemplate的action方法，调用匿名内部类的doService方法。

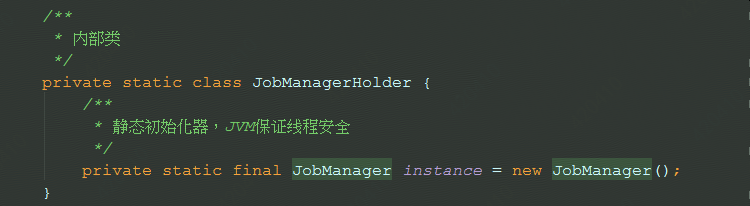




## 单例模式

在上面的ArrangePromCacheJob中持有JobManager对象。这是一个单例类，通过内部类的方式。

1. 线程安全：jvm对一个类的初始化会做限制，同一时间只会允许一个线程去初始化一个类
2. 懒加载：加载一个类时，其内部类不会同时被加载。一个类被加载，当且仅当其某个静态成员（静态域、构造器、静态方法等）被调用时发生。



由于是需要和外部类创建时一起创建，使用static修饰，只限于本类中使用，private修饰

提供给外部使用



## Jdk动态代理与cglib代理

1、原理区别：

[Java](http://lib.csdn.net/base/17)动态代理是利用反射机制生成一个实现代理接口的匿名类，在调用具体方法前调用InvokeHandler来处理。

而cglib动态代理是利用asm开源包，对代理对象类的class文件加载进来，通过修改其字节码生成子类来处理。

1、如果目标对象实现了接口，默认情况下会采用JDK的动态代理实现AOP   
2、如果目标对象实现了接口，可以强制使用CGLIB实现AOP

3、如果目标对象没有实现了接口，必须采用CGLIB库，[spring](http://lib.csdn.net/base/17)会自动在JDK动态代理和CGLIB之间转换

说明：

JDK代理是不需要以来第三方的库，只要要JDK环境就可以进行代理，它有几个要求  
\* 实现InvocationHandler   
\* 使用Proxy.newProxyInstance产生代理对象  
\* 被代理的对象必须要实现接口  
CGLib 必须依赖于CGLib的类库，但是它需要类来实现任何接口代理的是指定的类生成一个子类，覆盖其中的方法，是一种继承但是针对接口编程的环境下推荐使用JDK的代理

# Tomcat调优

## 配置详解

### tomcat/service.xml



<server>元素是整个文件的根元素

Port：Tomcat服务器监听用于关闭Tomcat服务器的命令（必须）  
Shutdown：发送到端口上用于关闭Tomcat服务器的命令。

**<Connector>元素：**  
连接器，负责接收客户的请求，以及向客户端回送响应的消息

URIEncoding：用于解码URL的字符编码，没有指定默认值为ISO-8859-1

connectionTimeout：设置连接的超时值，以毫秒为单位。默认值为60000=60秒

minSpareThreads Tomcat初始化时创建的 socket 线程数

maxSpareThreads：允许存在空闲线程的最大数目，默认值为50

maxThread：最大线程数，默认值为200

acceptCount：当所有的可能处理的线程都正在使用时，在队列中排队请求的最大数目。当队列已满，任何接收到的请求都会被拒绝，默认值为10

<Connector port="8081" protocol="HTTP/1.1"

connectionTimeout="20000"

redirectPort="8443" acceptCount="700" maxThreads="500"/>

## 性能优化

1.jvm内存优化

在bin/catalina.sh配置文件中增加jvm设置

export JAVA\_OPTS="-server

-Xms1800M -Xmx1800M -Xss128k -Xmn1024M -XX:SurvivorRatio=8

-XX:PermSize=128M

-XX:MaxPermSize=128M

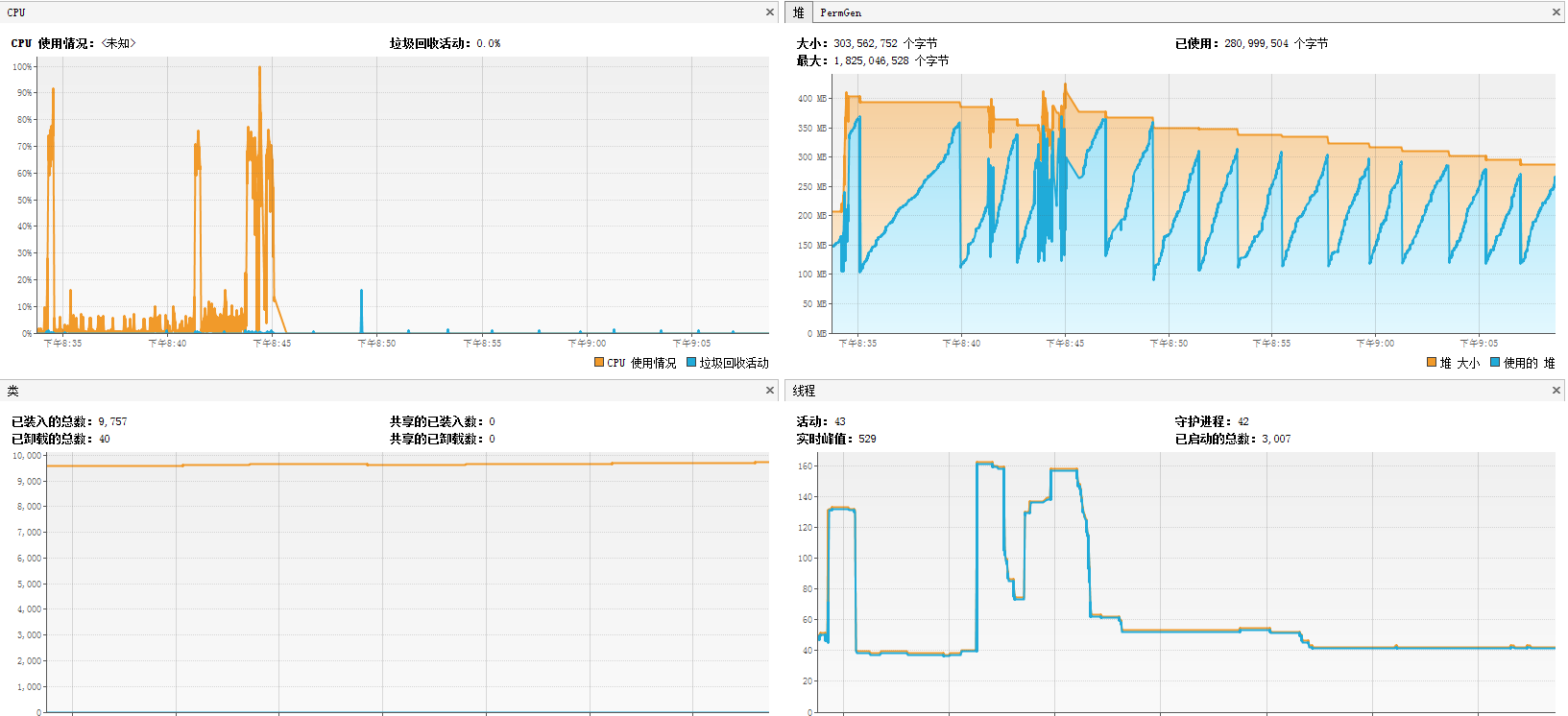
-XX:+UseConcMarkSweepGC

-Djava.awt.headless=true

-XX:+DisableExplicitGC"

## 微信项目压力测试心得

当以50000个请求，100个并发方式访问，203服务器

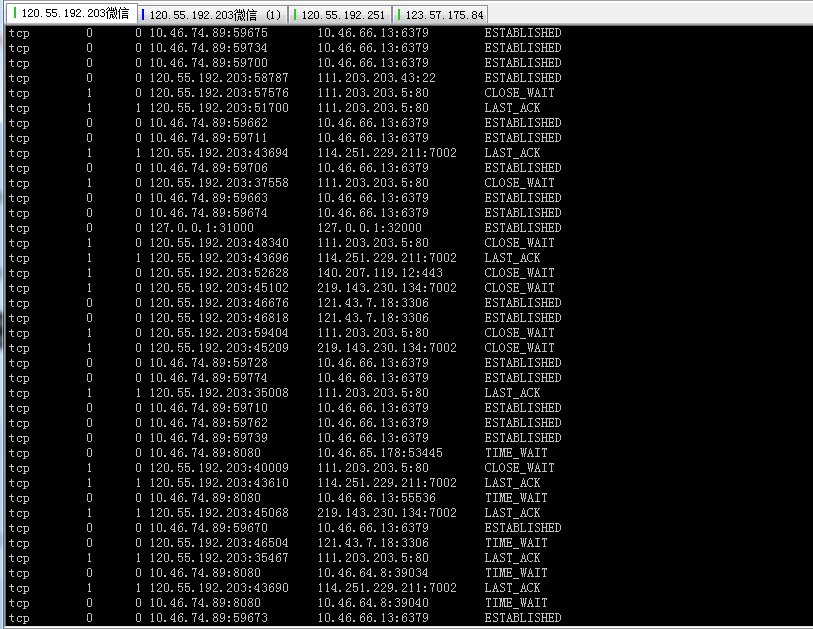


服务器的cpu、内存，线程等信息报告

客户端大量请求被拒绝，提示信息：

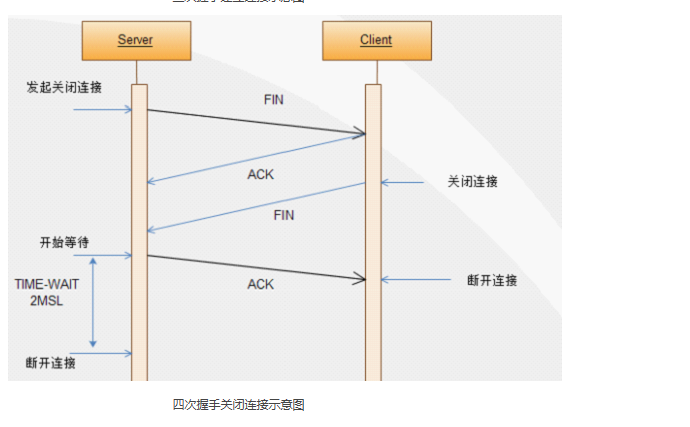
异常No buffer space available (maximum connections reached?)

使用netstat –an查看服务器，显示如下



发现存在大量处于TIME\_WAIT状态的TCP连接

原因：



而四次握手关闭连接示意图中，TCP协议中，关闭TCP连接的是Server端（当然，关闭都可以由任意一方发起），当Server端发起关闭连接请求时，向Client端发送一个FIN报文，Client端收到FIN报文时，很可能还有数据需要发送，所以并不会立即关闭SOCKET，所以先回复一个ACK报文，告诉Server端，“你发的FIN报文我收到了”。当Client端的所有报文都发送完毕之后，Client端向Server端发送一个FIN报文，此时Client端进入关闭状态，不在发送数据。

Server端收到FIN报文后，就知道可以关闭连接了，但是网络是不可靠的，Client端并不知道Server端要关闭，所以Server端发送ACK后进入TIME\_WAIT状态，如果Client端没有收到ACK则Server段可以重新发送。Client端收到ACK后，就知道可以断开连接了。Server端等待了2MSL（Max Segment Lifetime，最大报文生存时间）后依然没有收到回复，则证明Client端已正常断开，此时，Server端也可以断开连接了。2MSL的TIME\_WAIT等待时间就是由此而来。

我们知道了TIME\_WAIT的由来，TIME\_WAIT 状态最大保持时间是2 \* MSL，在1-4分钟之间，所以当系统并发过大，Client-Server连接数过多，Server端会在1-4分钟之内积累大量处于TIME\_WAIT状态的无法释放的socket连接，导致服务器效率急剧下降，甚至耗完服务器的所有资源，最终导致No buffer space available (maximum connections reached?): connect

问题的发生。

解决办法

1、文件描述符

a.查看文件描述符大小

ulimit -n

b.修改文件描述符

vi /etc/security/limits.conf

添加：

soft nofile 65535

hard nofile 65535

2、调整linux内核参数

vi /etc/sysctl.conf

sysctl 命令

-w 临时改变某个指定参数的值，如

# sysctl -w net.ipv4.ip\_forward=1

-a 显示所有的系统参数

-p从指定的文件加载系统参数,默认从/etc/sysctl.conf 文件中加载，一般用来使添加的配置立即生效

net.ipv4.tcp\_syn\_retries = 1

net.ipv4.tcp\_synack\_retries = 1

net.ipv4.tcp\_keepalive\_time = 600

net.ipv4.tcp\_keepalive\_probes = 3

net.ipv4.tcp\_keepalive\_intvl =15

net.ipv4.tcp\_retries2 = 5

net.ipv4.tcp\_fin\_timeout = 2 #系統默认的TIMEOUT时间

net.ipv4.tcp\_max\_tw\_buckets = 36000

net.ipv4.tcp\_tw\_recycle = 1#开启TCP连接中TIME\_WAIT sockets的快速回收 默认为0 #示关闭

#et.ipv4.tcp\_tw\_reuse = 1#启重用，允许将TIME\_WAIT sockets重新用于新的TCP连接 #认为0表示关闭

net.ipv4.tcp\_max\_orphans = 32768

net.ipv4.tcp\_syncookies = 1#防止网络攻击：半连接，默认是0

net.ipv4.tcp\_max\_syn\_backlog = 16384 #半连接队列的长度

net.ipv4.tcp\_wmem = 8192 131072 16777216

net.ipv4.tcp\_rmem = 32768 131072 16777216

net.ipv4.tcp\_mem = 786432 1048576 1572864

net.ipv4.ip\_local\_port\_range = 1024 65000

net.ipv4.ip\_conntrack\_max = 65536

net.ipv4.netfilter.ip\_conntrack\_max=65536

net.ipv4.netfilter.ip\_conntrack\_tcp\_timeout\_established=180

net.core.somaxconn = 16384#连接队列长度

net.core.netdev\_max\_backlog = 16384#据包tcp处理速度

vm.swappiness=5 #控制内存到磁盘交换数据积极性，默认60，值越高越积极

#hadoop服务器设置为0，mysql设置为5

# ProtoBuf

（Google Protocol Buffer）

# NIO

## Buffer

-- 缓冲区

所谓的缓冲区其实就是内存中一段连续的空间，可以向其中写入/读取数据，从而实现数据的保持、共享和处理。

Buffer

ByteBuffer

ShortBuffer

IntBuffer

LongBuffer

FloatBuffer

DoubleBuffer

CharBuffer

### 1.如何创建缓冲区对象

allocate(int capacity);//创建出指定大小的缓冲区，单位取决于创建出的缓冲区的类型

wrap(byte [] buf)

wrap(byte [] buf,int offset,int length)

### 2.如何向缓冲区中写入数据

put(byte b)

put(byte [] bs)

put(byte[]bs,i,len)

put(ByteBuffer buf)

channel.read(buf)//从通道中获取数据填入到buf中

### 3.如何从缓冲区中读取数据

get()//从缓冲区中获取一个单位的数据

get(byte [] bs)//从缓冲区中获取数据填入到数组中，数组有多大就试图获取多少数据往里写

get(byte []bs,int offset,int length)//从缓冲区中获取数据填入到数组中，从数组指定位置开始填入指定数量

get(int index)//获取指定位置上的数据

channel.write(ByteBuffer buf) //将缓冲区中的数据写入到通道中

### 4.缓冲区中位置属性

**capacity - 容量**

**position - 当前位置**

**limit - 限制位置**

缓冲区在刚创建出来时capacity等于缓冲区的大小。postion为0，limit等于capacity。

在向缓冲区写入数据时，会在position指向的位置写入数据，写入后postion+1指向下一个写入位置在从缓冲区中读取数据时，从position指向的位置读取数据，postion+1指向下一个读取位置。，由此可知postion的最大值capacit-1。

limit指定限制位置，在写入数据时，position的值不能超过limit，通过limit可以指定最多写到哪个位置。在从缓冲区中读取数据时，limit可以限制position最大的位置，从而限制最多读取到哪里。

int capacity()

返回此缓冲区的容量。

int position()

返回此缓冲区的位置。

Buffer position(int newPosition)

设置此缓冲区的位置。

int limit()

返回此缓冲区的限制。

Buffer limit(int newLimit)

设置此缓冲区的限制。

### 5.其他方法

flip();//反转缓冲区，一般用在向缓冲区中写入完数据后试图从中读取数据之前调用，将limit设置到当前postion位置，将postion设置为0。

hasRemaining();//通常用在循环读取时，判断position和limit之间有没有元素，从而控制循环的边界。

clear();//通常用在读取完成后清空缓冲区准备重复使用时。limit被设置为和capacity相同，position设置为0，但是注意数据并不真正清空，而是知道再往里写入数据时覆盖旧的数据。

compact();//压缩此缓冲区。次方用来将已经读取的数据干掉，将未读的数据挪到缓冲区的最开始位置，再向其中写数据时追加到已有数据后面。此方法将所有未读取的数据拷贝到Buffer起始处，然后将position设置到最后一个未读取元素后面。limit设置为和capacity一致。

rewind();//重绕次缓冲区。再读取时postion置为0，从新进行读取。

mark();在当前position位置上设置一个标记，可以在后续需要是跳转回到这个标记位

reset();回到之前的标记位操作

array();返回缓冲区底层的数组。

### 6.分散 和 聚合

分散：所谓的分散其实就是从一个通道中获取数据依次写入到多个缓冲区中的过程。

ByteBuffer bufs [] = {buf1,buf2,buf3};

channel.read(bufs);

聚合：将多个缓冲区中的数据依次写出到一个通道中区

ByteBuffer bufs [] = {buf1,buf2,buf3};

channel.write(bufs);

## Charset

Charset forName(String cname)

编码：

ByteBuffer encode(CharBuffer)

ByteBuffer encode(String)

解码：

CharBuffer decode(ByteBuffer)

## Channel -- 通道

带表数据传输的通道。

即可以读取数据也可以写出数据

读写操作都可以是异步进行的

通道总是将数据写入到一个Buffer或者从Buffer中向通道写入数据

FileChannel

DatagramChannel

SocketChannel

ServerSocketChannel

传统的Socket/ServerSocket编程中可能产生阻塞的地方在于 accept connect read write

NIO中的SocketChannel/ServerSocketChannel可以以非阻塞的方式实现TCP通信

## Selector -- 选择器

Selector叫做选择器，可以在它身上注册多个非阻塞的通道，

通过Selector来统一管理这些通道，选择已经就绪可以处理的通道来处理，

从而只需要很少的线程就可以管理多个通道了。特别适合于开启了很多个通道但是每个

通道中数据流量都很少的情况。可以大大的减少线程的数量和减少内存消耗避免cpu无意义的轮询。

select();//此方法阻塞，等待有就绪的通道再进行处理

阻塞 -- 非阻塞 === 基于线程

同步 -- 异步 === 任务执行过程

BIO -- 同步阻塞IO - jdk1.0、1.1

NIO -- 同步非阻塞IO - jdk1.4

AIO -- 异步非阻塞IO - jdk7

# Jdk反射

**描述**：对代表类的类进行处理的过程就叫做反射。本质上就是剖析类获取类内部的信息进行操作的过程叫做反射。

## Class类 -- 属性 方法

1、获取Class：

通过类的class属性获取

通过对象.getClass()来获取

通过Class类的静态方法 forNam(String clzStr);来获取

2、方法

cast(Object obj) 进行类型强制转换

getClassLoader() 获取加载当前类的类加载器

Constructor<T> getConstructor(Class<?>... parameterTypes) //获取构造器

Constructor<?>[] getConstructors() //获取构造器

Field getField(String name)

Field[] getFields() //获取成员属性

Method getMethod(String name, Class<?>... parameterTypes)

Method[] getMethods() //获取成员方法

Class<?>[] getInterfaces()//获取类实现的所有接口的数组

String getName() //获取类的名字

String getSimpleName()

Class<? super T> getSuperclass()

3、创建对象

T newInstance()//调用无参构造创建该类的一个对象 必须有无参的构造方法才能调用，通过构造器创建对象

## Field ： 代表成员属性

String getName()

Class<?> getType()

get(xxx)

set(xxx)

setAccessible()实现暴力反射，开启后连私有的属性都可以访问。

## Method ： 代表成员方法的

String getName()

Class<?>[] getParameterTypes()

Class<?> getReturnType()

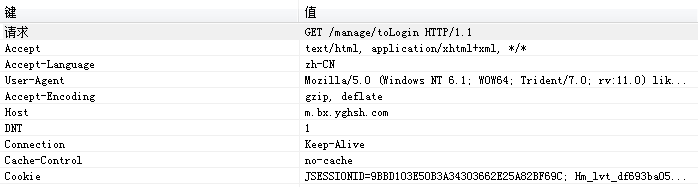
Object invoke(Object obj, Object... args) //执行当前方法

setAccessible()实现暴力反射，开启后连私有的属性都可以访问。

# HTTP协议详解

## 请求篇

HTTP请求是由三个部分组成，分别是：请求行、头信息、消息体



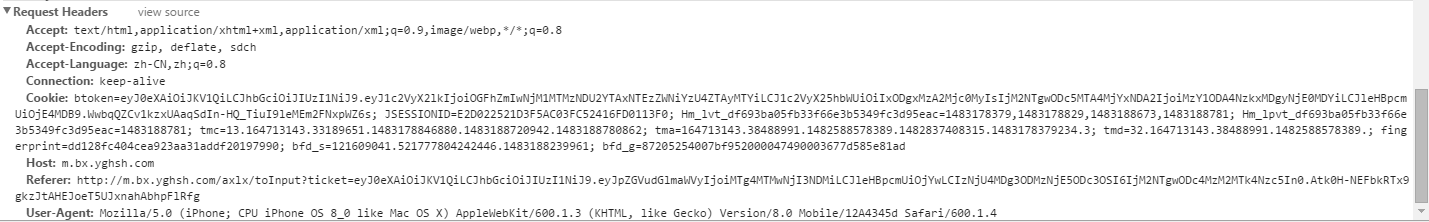
分为三个部分：

### 请求行：

GET /manage/toLogin HTTP/1.1

GET代表的是请求的方法，/manage/toLogin这个是uri，HTTP/1.1协议和版本

### 头信息：



HTTP Request header

#### 1、Cache 头域

　　If-Modified-Since

　　作用： 把浏览器端缓存页面的最后修改时间发送到服务器去，服务器会把这个时间与服务器上实际文件的最后修改时间进行对比。如果时间一致，那么返回304，客户端就直接使用本地缓存文件。如果时间不一致，就会返回200和新的文件内容。客户端接到之后，会丢弃旧文件，把新文件缓存起来，并显示在浏览器中。

　　例如：If-Modified-Since: Thu, 09 Feb 2012 09:07:57 GMT

If-None-Match

作用: If-None-Match和ETag一起工作，工作原理是在HTTP Response中添加ETag信息。 当用户再次请求该资源时，将在HTTP Request 中加入If-None-Match信息(ETag的值)。如果服务器验证资源的ETag没有改变（该资源没有更新），将返回一个304状态告诉客户端使用本地缓存文件。否则将返回200状态和新的资源和Etag.  使用这样的机制将提高网站的性能

#### 2、Client 头域

**Accept**

　　作用： 浏览器端可以接受的媒体类型,

　 例如：  Accept: text/html  代表浏览器可以接受服务器回发的类型为 text/html  也就是我们常说的html文档,

　　如果服务器无法返回text/html类型的数据，服务器应该返回一个406错误(non acceptable)

　　通配符 \* 代表任意类型

　　例如  Accept: \*/\*  代表浏览器可以处理所有类型，(一般浏览器发给服务器都是发这个)

**Accept-Encoding**：

　　作用： 浏览器申明自己接收的编码方法，通常指定压缩方法，是否支持压缩，支持什么压缩方法（gzip，deflate），（注意：这不是只字符编码）;

　　例如： Accept-Encoding: gzip, deflate

**Accept-Language**

　　作用： 浏览器申明自己接收的语言。

　　语言跟字符集的区别：中文是语言，中文有多种字符集，比如big5，gb2312，gbk等等；

　　例如： Accept-Language: en-us

**User-Agent**

　　作用：告诉HTTP服务器， 客户端使用的操作系统和浏览器的名称和版本.

　　我们上网登陆论坛的时候，往往会看到一些欢迎信息，其中列出了你的操作系统的名称和版本，你所使用的浏览器的名称和版本，这往往让很多人感到很神奇，实际上，服务器应用程序就是从User-Agent这个请求报头域中获取到这些信息User-Agent请求报头域允许客户端将它的操作系统、浏览器和其它属性告诉服务器。

　　例如： User-Agent: Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 5.1; Trident/4.0; CIBA; .NET CLR 2.0.50727; .NET CLR 3.0.4506.2152; .NET CLR 3.5.30729; .NET4.0C; InfoPath.2; .NET4.0E)

**Accept-Charset**

　　作用：浏览器申明自己接收的字符集，这就是本文前面介绍的各种字符集和字符编码，如gb2312，utf-8（通常我们说Charset包括了相应的字符编码方案）；

#### 3、Cookie/Login 头域

　　Cookie:

　　作用： 最重要的header, 将cookie的值发送给HTTP 服务器

#### 4、Entity头域

　　Content-Length

　　作用：发送给HTTP服务器数据的长度。

　　例如： Content-Length: 38

　　Content-Type

　　作用：

　　例如：Content-Type: application/x-www-form-urlencoded

#### 5、Miscellaneous 头域

　　Referer:

　　作用： 提供了Request的上下文信息的服务器，告诉服务器我是从哪个链接过来的，比如从我主页上链接到一个朋友那里，他的服务器就能够从HTTP Referer中统计出每天有多少用户点击我主页上的链接访问他的网站。

　　例如: Referer:http://translate.google.cn/?hl=zh-cn&tab=wT

　　Transport 头域

　　Connection

　　例如：　Connection: keep-alive   当一个网页打开完成后，客户端和服务器之间用于传输HTTP数据的TCP连接不会关闭，如果客户端再次访问这个服务器上的网页，会继续使用这一条已经建立的连接

　　例如：  Connection: close  代表一个Request完成后，客户端和服务器之间用于传输HTTP数据的TCP连接会关闭， 当客户端再次发送Request，需要重新建立TCP连接。

　　Host（发送请求时，该报头域是必需的）

　　作用: 请求报头域主要用于指定被请求资源的Internet主机和端口号，它通常从HTTP URL中提取出来的

　　例如: 我们在浏览器中输入：http://www.guet.edu.cn/index.html

　　浏览器发送的请求消息中，就会包含Host请求报头域，如下：

　　Host：http://www.guet.edu.cn

　　此处使用缺省端口号80，若指定了端口号，则变成：Host：指定端口号

### 消息体：

头信息与消息体之间是空格分开

## 响应篇

HTTP响应也是由三个部分组成，分别是：状态行、消息报头、响应正文

**状态行**：由HTTP协议版本号， 状态码， 状态消息

HTTP/1.1中定义了5类状态码， 状态码由三位数字组成，第一个数字定义了响应的类别

1XX  提示信息 - 表示请求已被成功接收，继续处理

　　2XX  成功 - 表示请求已被成功接收，理解，接受

　　3XX  重定向 - 要完成请求必须进行更进一步的处理

　　4XX  客户端错误 -  请求有语法错误或请求无法实现

5XX  服务器端错误 -   服务器未能实现合法的请求

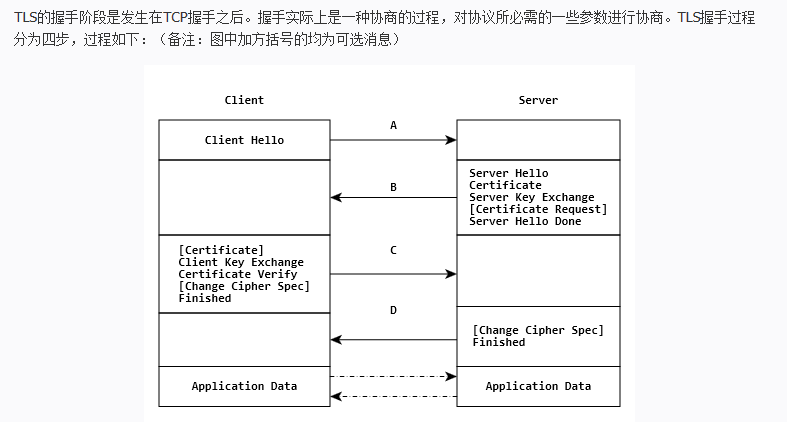
# HTTPS协议

## 概念：

HTTPS = HTTP + TLS

TLS是SSL的3.0升级版

TLS握手



### Client Hello

由于客户端(如浏览器)对一些加解密算法的支持程度不一样，但是在TLS协议传输过程中必须使用同一套加解密算法才能保证数据能够正常的加解密。在TLS握手阶段，客户端首先要告知服务端，自己支持哪些加密算法，所以客户端需要将本地支持的加密套件(Cipher Suite)的列表传送给服务端。除此之外，客户端还要产生一个随机数，这个随机数一方面需要在客户端保存，另一方面需要传送给服务端，客户端的随机数需要跟服务端产生的随机数结合起来产生后面要讲到的Master Secret。

### Server Hello

上图中，从Server Hello到Server Done，有些服务端的实现是每条单独发送，有服务端实现是合并到一起发送。Sever Hello和Server Done都是只有头没有内容的数据。

服务端在接收到客户端的Client Hello之后，服务端需要将自己的证书发送给客户端。这个证书是对于服务端的一种认证。例如，客户端收到了一个来自于称自己是www.alipay.com的数据，但是如何证明对方是合法的alipay支付宝呢？这就是证书的作用，支付宝的证书可以证明它是alipay，而不是财付通。证书是需要申请，并由专门的数字证书认证机构(CA)通过非常严格的审核之后颁发的电子证书。颁发证书的同时会产生一个私钥和公钥。私钥由服务端自己保存，不可泄漏。公钥则是附带在证书的信息中，可以公开的。证书本身也附带一个证书电子签名，这个签名用来验证证书的完整性和真实性，可以防止证书被串改。另外，证书还有个有效期。

在服务端向客户端发送的证书中没有提供足够的信息的时候，还可以向客户端发送一个Server Key Exchange。

此外，对于非常重要的保密数据，服务端还需要对客户端进行验证，以保证数据传送给了安全的合法的客户端。服务端可以向客户端发出Cerficate Request消息，要求客户端发送证书对客户端的合法性进行验证。

跟客户端一样，服务端也需要产生一个随机数发送给客户端。客户端和服务端都需要使用这两个随机数来产生Master Secret。

最后服务端会发送一个Server Hello Done消息给客户端，表示Server Hello消息结束了。

### Client Key Exchange

如果服务端需要对客户端进行验证，在客户端收到服务端的Server Hello消息之后，首先需要向服务端发送客户端的证书，让服务端来验证客户端的合法性。

在此之前的所有TLS握手信息都是明文传送的。在收到服务端的证书等信息之后，客户端会使用一些加密算法(例如：RSA, Diffie-Hellman)产生一个48个字节的Key，这个Key叫PreMaster Secret，很多材料上也被称作PreMaster Key, 最终通过Master secret生成session secret， session secret就是用来对应用数据进行加解密的。PreMaster secret属于一个保密的Key，只要截获PreMaster secret，就可以通过之前明文传送的随机数，最终计算出session secret，所以PreMaster secret使用RSA非对称加密的方式，使用服务端传过来的公钥进行加密，然后传给服务端。

接着，客户端需要对服务端的证书进行检查，检查证书的完整性以及证书跟服务端域名是否吻合。

ChangeCipherSpec是一个独立的协议，体现在数据包中就是一个字节的数据，用于告知服务端，客户端已经切换到之前协商好的加密套件的状态，准备使用之前协商好的加密套件加密数据并传输了。

在ChangecipherSpec传输完毕之后，客户端会使用之前协商好的加密套件和session secret加密一段Finish的数据传送给服务端，此数据是为了在正式传输应用数据之前对刚刚握手建立起来的加解密通道进行验证。

### Server Finish

服务端在接收到客户端传过来的PreMaster加密数据之后，使用私钥对这段加密数据进行解密，并对数据进行验证，也会使用跟客户端同样的方式生成session secret，一切准备好之后，会给客户端发送一个ChangeCipherSpec，告知客户端已经切换到协商过的加密套件状态，准备使用加密套件和session secret加密数据了。之后，服务端也会使用session secret加密后一段Finish消息发送给客户端，以验证之前通过握手建立起来的加解密通道是否成功。

根据之前的握手信息，如果客户端和服务端都能对Finish信息进行正常加解密且消息正确的被验证，则说明握手通道已经建立成功，接下来，双方可以使用上面产生的session secret对数据进行加密传输了。

HTTPS 在 HTTP 应用层的基础上使用安全套接字层作为子层。

## HTTP 和 HTTPS 的不同之处：

1、HTTP 的 URL 以 http:// 开头，而 HTTPS 的 URL 以 https:// 开头

2、HTTP 是不安全的，而 HTTPS 是安全的

3、HTTP 标准端口是 80 ，而 HTTPS 的标准端口是 443

4、在 OSI 网络模型中，HTTP 工作于应用层，而 HTTPS 工作在传输层

5、HTTP 无需加密，而 HTTPS 对传输的数据进行加密

6、HTTP 无需证书，而 HTTPS 需要认证证书

## HTTPS工作方式

使用 HTTPS 连接时，服务器要求有公钥和签名的证书。

当使用 https 连接，服务器响应初始连接，并提供它所支持的加密方法。作为回应，客户端选择一个连接方法，并且客户端和服务器端交换证书验证彼此身份。完成之后，在确保使用相同密钥的情况下传输加密信息，然后关闭连接。为了提供 https 连接支持，服务器必须有一个公钥证书，该证书包含经过证书机构认证的密钥信息，大部分证书都是通过第三方机构授权的，以保证证书是安全的。

SSL 包含如下动作：

1. 验证服务器端
2. 允许客户端和服务器端选择加密算法和密码，确保双方都支持
3. 验证客户端(可选)
4. 使用公钥加密技术来生成共享加密数据
5. 创建一个加密的 SSL 连接
6. 基于该 SSL 连接传递 HTTP 请求

# 加密算法

## 对称加密：

**概念**：对称加密采用了对称密码编码技术，它的特点是文件加密和解密使用相同的密钥，即加密密钥也可以用作解密密钥，这种方法在密码学中叫做对称加密算法，对称加密算法使用起来简单快捷，密钥较短，且破译困难

存在的问题：

1、要求提供一条安全的渠道使通讯双方在首次通讯时协商一个共同的密钥。

2、密钥的数目难于管理。

3、对称加密算法一般不能提供信息完整性的鉴别。它无法验证发送者和接受者的身份

4、对称密钥的管理和分发工作是一件具有潜在危险的和烦琐的过程

常见的有：DES、3DES、Blowfish、IDEA、RC4、RC5、RC6和AES

## 非对称加密：

**概念**：

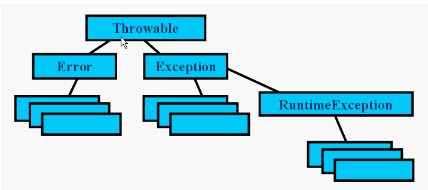
与对称加密算法不同，非对称加密算法需要两个密钥：公开密钥（publickey）和私有密钥（privatekey）。公开密钥与私有密钥是一对，如果用公开密钥对数据进行加密，只有用对应的私有密钥才能解密；如果用私有密钥对数据进行加密，那么只有用对应的公开密钥才能解密。因为加密和解密使用的是两个不同的密钥，所以这种算法叫作非对称加密算法。

## HASH算法

**HASH算法：**常用的摘要算法包括MD5、SHA1，SHA256

# Java异常

## 基本概念



Throwable是所有异常的根，java.lang.Throwable

Error是错误，java.lang.Error

Exception是异常，java.lang.Exception

## Error

常见的error

java.lang.ClassFormatError

类格式错误。当Java虚拟机试图从一个文件中读取Java类，而检测到该文件的内容不符合类的有效格式时抛出。

java.lang.NoClassDefFoundError

未找到类定义错误。当Java虚拟机或者类装载器试图实例化某个类，而找不到该类的定义时抛出该错误。

java.lang.NoSuchMethodError

方法不存在错误。当应用试图调用某类的某个方法，而该类的定义中没有该方法的定义时抛出该错误。

比如：项目中引入了一个jar包，而这个jar包中的某些地方又需要引入另一个jar包的支持，而你并没有引入，这时是可以正常编译的，而当你在运行中调用，就会报找不到方法的错误

java.lang.OutOfMemoryError

内存不足错误。当可用内存不足以让Java虚拟机分配给一个对象时抛出该错误。

## Exception

一般分为Checked异常和Runtime异常，所有RuntimeException类及其子类的实例被称为Runtime异常，不属于该范畴的异常则被称为CheckedException

### ****Checked异常****

只有java语言提供了Checked异常，Java认为Checked异常都是可以被处理的异常，所以Java程序必须显示处理Checked异常。如果程序没有处理Checked异常，该程序在编译时就会发生错误无法编译。这体现了Java的设计哲学：没有完善错误处理的代码根本没有机会被执行。对Checked异常处理方法有两种

1 当前方法知道如何处理该异常，则用try...catch块来处理该异常。

2 当前方法不知道如何处理，则在定义该方法是声明抛出该异常。

我们比较熟悉的Checked异常有

Java.lang.ClassNotFoundException 通过反射获取类的时候，可能会抛出这个异常

Java.lang.NoSuchMetodException 通过反射获取

java.io.IOException

### RunTimeException-非检查异常

Runtime如除数是0和数组下标越界等，其产生频繁，处理麻烦，若显示申明或者捕获将会对程序的可读性和运行效率影响很大。所以由系统自动检测并将它们交给缺省的异常处理程序。当然如果你有处理要求也可以显示捕获它们。

我们比较熟悉的RumtimeException类的子类有

Java.lang.ClassCastException

Java.lang.IndexOutOfBoundsException

Java.lang.NullPointerException

## thow与throws区别

1. throws 是方法可能抛出异常的声明。(用在声明方法时，表示该方法可能要抛出异常) 调用者必须做出处理（捕获或继续抛出）。
2. throw 是语句抛出一个异常。
3. throws可以单独使用，throw的如果是检查异常，必须配合throws使用。

# 内部类

[内部类](https://blog.csdn.net/suifeng3051/article/details/51791812)

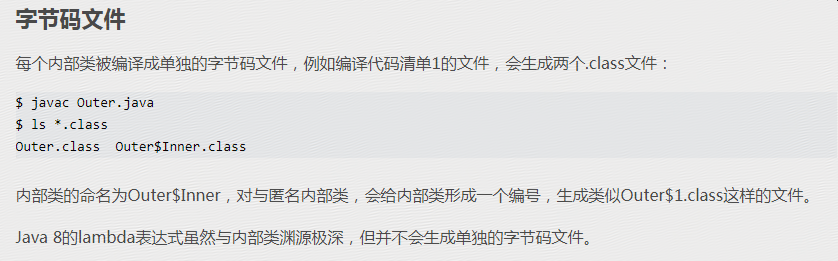
[**Java 内部类实现原理简单分析**](https://blog.csdn.net/wzm112358/article/details/47145995)

**内部类中持有外部类的引用this$0，编译后有构造函数，将外部类和需要使用到外部类的局部变量作为参数。**

内部类可以直接访问外部类的元素，但是外部类不可以直接访问内部类的元素

## 匿名内部类

其实就是一个没有名字的方法内部类，所以它符合方法内部类的所有约束，初次之外，还有一些地方需要注意：



匿名内部类是没有访问修饰符的。

匿名内部类必须继承一个抽象类或者实现一个接口

匿名内部类中不能存在任何静态成员或方法

匿名内部类是没有构造方法的，因为它没有类名。

## 类级内部类

static修饰的内部类，即静态内部类

非静态内部类在编译完成之后会隐含地保存着一个引用，该引用是指向创建它的外围内，但是静态内部类却没有。没有这个引用就意味着：

静态内部类的创建是不需要依赖于外围类，可以直接创建

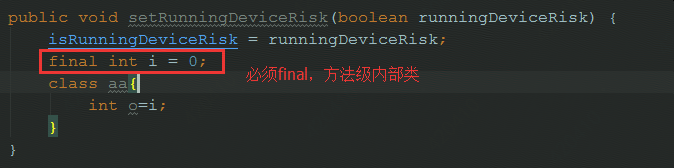
静态内部类不可以使用任何外围类的非static成员变量和方法，而内部类则都可以

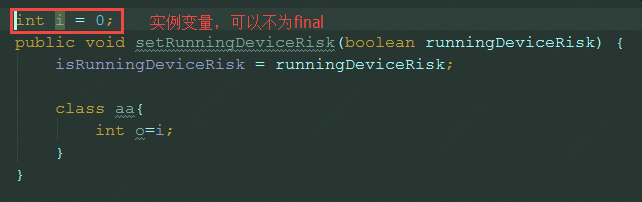
## 方法内部类

内部类只能访问final或相当于final的局部变量

编译器为内部类中用到的所有局部变量在内部类中都生成一个对应的字段。当创建内部类的实例时，局部变量的值会通过内部类的构造函数拷贝到内部类的对应字段中。内部类对外围局部变量的访问实际转化成了对内部类字段的访问。

因为值已经被拷贝到了内部类的字段中，那么在外围函数中对该变量的修改会使整个程序的行为变得有点怪异–看起来你可能使用了过期的数据。因此，不能在外围函数中修改变量。对应的，内部类中也不允许修改该变量（实际上，自动生成的字段被声明为final），否则，外围函数好像在处理过期数据。





# 实际案例

## ThreadLocal引发的血案

我们六艺系统做了一个出行人组件，就在10月28号当天， 线上用户下单我发现有很多订单的出行人列表中莫名奇妙的加入了其他人的出行人信息，因为购票的数据和出行人的数量不一致所以出票不会成功，但是这样的订单越来越多，我开始紧张了起来，一开始我以为是有人在刷接口，通过程序输入的参数，后来通过观察日志发现，入参是没有该参数的，那这样我就很快就定位到是出行人的拦截器有问题了，为了验证是因为没有被回收，我把机器重启了一次，启来以后发现下单没有了。

       可是每一个线程都是隔离的，为什么会被别的线程给共享了呢？我也不卖关子了（当然找这个问题和复现它也挺费劲的），是因为tomcat的线程池使线程局部变量没有被回收，然后下次用户请求虽然请求不是同一个，但是线程还是同一个线程。

        修改的话就是加上了，上面标红的第三条remove操作，在每次用完时候把它清除掉。

         最后我总结了一下：

* 1. 之前登录也没有删除，为什么存的用户信息没有错过呢，因为登录的拦截器每次都会用最新的数据把旧数据覆盖掉；
  2. 不删除还会引起堆内存的这个局部变量没有回收，数据多的时候会导致内存泄漏；
  3. 为什么测试和预发验证的时候没有发现问题呢，因为线上的tomcat线程数默认设置的是1000。所以自己学会压力测试的话还是很有帮助的。

写完这个以后，感觉这个错误这辈子都不会犯了。

                   分享一个链接  <http://blog.csdn.net/lufeng20/article/details/24314381>

## 缩小日志量

1. 字段名称缩短
2. 输出日志使用toString，重写对象的toString方法，定制自己想要打印的字段和格式
3. 现在系统中使用fastJson，这样日志量非常大（若用打标让某些字段不序列化，可能其他用到该对象的地方也会影响）

[Log日志框架的学习五.正确使用日志的10个技巧](http://itindex.net/detail/48502-log-%E6%97%A5%E5%BF%97-%E6%A1%86%E6%9E%B6)

[java日志框架log4j详细配置及与slf4j联合使用教程](https://www.cnblogs.com/ywlaker/p/6124067.html)

[为什么使用 SLF4J 而不是 Log4J 来做 Java 日志](https://www.oschina.net/translate/why-use-sl4j-over-log4j-for-logging)

## 上线三板斧

三板斧：可灰度、可监控、可回滚

## 一张优惠券引发的血案

分布式锁：

Redis的setNx

[一张优惠券引发的血案](https://mp.weixin.qq.com/s/PjpoRZMVHOU1QJ-v1TuvzQ)