# Jvm

## CPU占用率过高

1. jps -m 非常方便直接定位所有的Java进程pid

[root@cdn ldw]# jps -m | grep 28081

1. jstack -l pid > jstack.log命令打印栈信息，如果因某些原因无法成功打印，可以使用kill -pid，输出到jvm日志
2. 查找占用CPU资源最高的进程id，使用top -H -p pid 查看进程pid的所有的线程，默认是按照%CPU高~低排序。
3. pid 转成 16 进制小写， obase 是输出格式，可以指定 ibase =16 表示16进制输入

echo "obase=16; 50740" | bc  这个语句是大写，要变成小写。

1. 根据转换的16 进制 PID 0x12c4 去jstack.log日志中查找基本能定位到具体哪行代码的问题。

"reportThirdException" daemon prio=10 tid=0x00007f8bd450b800 nid=0x12c4 runnable [0x00007f8b906ac000]

java.lang.Thread.State: RUNNABLE

at com.xxx.interfaces.util.NoticeMonitorSysHelper$ThreadStatue$1.run(NoticeMonitorSysHelper.java:167)  at java.lang.Thread.run(Thread.java:722)

## 引用类型

### 强引用

一般引用。

### 软引用

SoftReference 引用，在失去了引用对象后，被放入ReferenceQueue，在内存不足时被清除，内存足时保留。

### 弱引用

WeakReference 在GC 时必然会清除。

### 虚引用

PhantomReference 对 GC 没有影响，只是个记录，GC 该怎么做就怎么做。

# IO

RandomAccessFile，这个类是以 java 二进制对字符处理的，是经过了 JVM 编码的，而流不是这样，流是文本编码。

## 各种IO服务器的特点

阻塞式的 IO ，客户端每次连接，服务端都要启动一个线程来处理连接，这样连接的数量越多，服务端运行的线程数越多，线程数膨胀之后，服务器性能下降，线程是系统宝贵资源，是有系统开销的。而采用线程池会发生排队现象，当客户端任务太多后，或者当前任务占用了工作线程，导致没有空闲线程处理当前任务，出现排队，客户端长时间得不到响应。

若采用 select 机制，客户端线程不用阻塞等待或者排队，而且 socket 的数量是有限制的，而 select 在大多数系统上都是基于 epoll 系统调用，不受连接句柄数限制，只受线程句柄数限制。

## NIO

### buffer

mark() 在当前位置做标记，并返回该buffer 的引用。

filp() limit置为当前位置，position置0，mark 清除。

rewind() 重来，position置0，mark 清除，其它不变。

clear() buffer恢复到初始状态，position 置0，limit置为最大capacity，内容不会清除，还可以再次读取。

reset() 如果没有设置 mark() ，这个会抛异常，返回上一个 mark() 位置。

arrayOffset() ，对于可写的的 buffer ，都是有一个数组来备份的，这个函数返回buffer第一个元素在数组中的偏移量。

array() 获取 buffer 的备份数组，不是全部都存在备份数组。

read() 函数不会出现缓存中有数据，但是返回 -1 的情况。

Buffer中字符遵循文本编码格式，转码最重要的就是实际的存储格式，我们字符串存储，或者字符串默认转 byte 流都是 utf-8 格式，中文的utf-8 可能是三个字节，而 getChar() 每次取出两个字节，这就导致中文编码会出问题。而如果是英文字母，它只占一个字节，如果我们一次取出两个字节，也会出问题。因此编码最重要的就是逻辑编码存储和逻辑编码解码要兼容。

### DirectByteBuffer

1. DirectByteBuffer(**int** cap) {           // 同包可访问这个函数
2. **super**(-1, 0, cap, cap);
3. **boolean** pa = VM.isDirectMemoryPageAligned();  //是否页对齐
4. **int** ps = Bits.pageSize();    //获取pageSize大小
5. **long** size = Math.max(1L, (**long**) cap + (pa ? ps : 0));  //如果是页对齐的话，那么
6. 就加上一页的大小
7. Bits.reserveMemory(size, cap);   //保留总分配内存(按页分配)的大小和实际内存的大
8. 小
10. **long** base = 0;
11. **try** {
12. base = unsafe.allocateMemory(size);  //实际分配内存
13. } **catch** (OutOfMemoryError x) {
14. Bits.unreserveMemory(size, cap);
15. **throw** x;
16. }
17. unsafe.setMemory(base, size, (**byte**) 0);   //初始化内存，返回基地址
18. //计算地址
19. **if** (pa && (base % ps != 0)) {
20. // Round up to page boundary
21. address = base + ps - (base & (ps - 1));
22. } **else** {
23. address = base;
24. }
25. //构建Cleaner对象用于跟踪DirectByteBuffer对象的垃圾回收，以实现当DirectByteBuffer被垃圾回收时，堆外内存也会被释放，Cleaner时个链表
26. cleaner = Cleaner.create(**this**, **new** Deallocator(base, size, cap));
27. att = **null**;
28. }

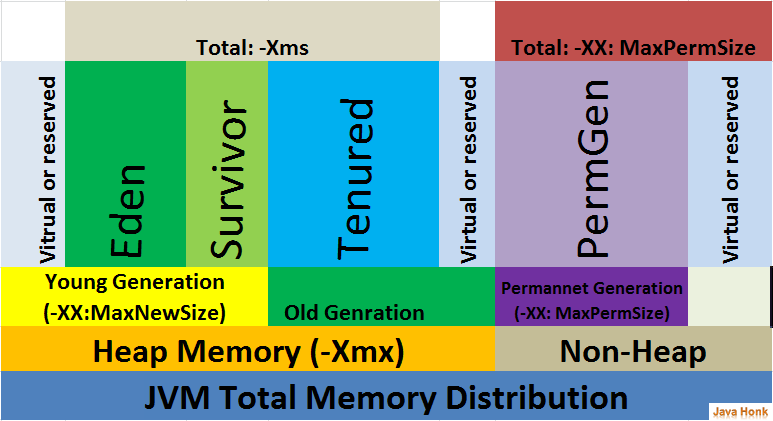
Bits.reserveMemory(size, cap) 这个函数在直接内存回收或者分配时都会调用，如果直接内存不够，jlra.tryHandlePendingReference()会触发一次非堵塞的Reference#tryHandlePending(false)。该方法会将已经被JVM垃圾回收的DirectBuffer对象的堆外内存释放。如果在进行一次堆外内存资源回收后，还不够进行本次堆外内存分配的话，则System.gc()会触发一个full gc，当然前提是你没有显示的设置-XX:+DisableExplicitGC来禁用显式GC。并且你需要知道，调用System.gc()并不能够保证full gc马上就能被执行。

所以在后面打代码中，会进行最多9次尝试，看是否有足够的可用堆外内存来分配堆外内存。并且每次尝试之前，都对延迟等待时间，已给JVM足够的时间去完成full gc操作。如果9次尝试后依旧没有足够的可用堆外内存来分配本次堆外内存，则抛出OutOfMemoryError("Direct buffer memory”)异常。注意，这里之所以用使用full gc的很重要的一个原因是：System.gc()会对新生代的老生代都会进行内存回收，这样会比较彻底地回收DirectByteBuffer对象以及他们关联的堆外内存. DirectByteBuffer对象本身其实是很小的，但是它后面可能关联了一个非常大的堆外内存，因此我们通常称之为冰山对象.我们做ygc的时候会将新生代里的不可达的DirectByteBuffer对象及其堆外内存回收了，但是无法对old里的DirectByteBuffer对象及其堆外内存进行回收，这也是我们通常碰到的最大的问题。( 并且堆外内存多用于生命期中等或较长的对象 )如果有大量的DirectByteBuffer对象移到了old，但是又一直没有做cms gc或者full gc，而只进行ygc，那么我们的物理内存可能被慢慢耗光，但是我们还不知道发生了什么，因为heap明明剩余的内存还很多(前提是我们禁用了System.gc – JVM参数DisableExplicitGC)。

Cleaner是PhantomReference的子类，并通过自身的next和prev字段维护的一个双向链表。PhantomReference的作用在于跟踪垃圾回收过程，并不会对对象的垃圾回收过程造成任何的影响。所以cleaner = Cleaner.create(this, new Deallocator(base, size, cap)); 用于对当前构造的DirectByteBuffer对象的垃圾回收过程进行跟踪。当DirectByteBuffer对象从pending状态 ——> enqueue状态时，会触发Cleaner的clean()，而Cleaner的clean()的方法会实现通过unsafe对堆外内存的释放。虽然Cleaner不会调用到Reference.clear()，但Cleaner的clean()方法调用了remove(this)，即将当前Cleaner从Cleaner链表中移除，这样当clean()执行完后，Cleaner就是一个无引用指向的对象了，也就是可被GC回收的对象。

### zerocopy

1. 内存结构



注意在Java8以后PermGen被MetaSpace代替，运行时可自动扩容，并且默认是无限大

2. 堆外内存

2.1. 广义的堆外内存

除了堆栈内存，剩下的就都是堆外内存了，包括了jvm本身在运行过程中分配的内存，codecache，jni里分配的内存，DirectByteBuffer分配的内存等等

2.2. 狭义的堆外内存 - DirectByteBuffer

而作为java开发者，我们常说的堆外内存溢出了，其实是狭义的堆外内存，这个主要是指java.nio.DirectByteBuffer在创建的时候分配内存，我们这篇文章里也主要是讲狭义的堆外内存，因为它和我们平时碰到的问题比较密切

为啥要使用堆外内存。通常因为：

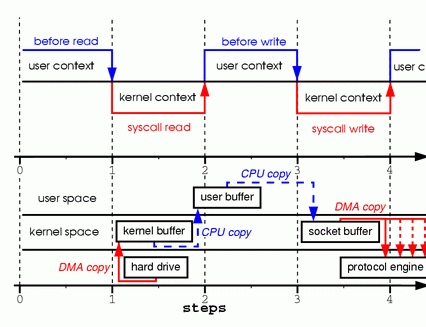
在进程间可以共享，减少虚拟机间的复制

对垃圾回收停顿的改善：如果应用某些长期存活并大量存在的对象，经常会出发YGC或者FullGC，可以考虑把这些对象放到堆外。过大的堆会影响Java应用的性能。如果使用堆外内存的话，堆外内存是直接受操作系统管理( 而不是虚拟机 )。这样做的结果就是能保持一个较小的堆内内存，以减少垃圾收集对应用的影响。

在某些场景下可以提升程序I/O操纵的性能。少去了将数据从堆内内存拷贝到堆外内存的步骤。

目前，JAVA的IO方式有基于流的传统IO还有基于块的NIO方式（虽然文件读取其实不是严格意义上的NIO，哈哈）。面向流意味着从流中一次可以读取一个或多个字节，拿到读取的这些做什么你说了算，这里没有任何缓存（这里指的是使用流没有任何缓存，接收或者发送的数据是缓存到操作系统中的，流就像一根水管从操作系统的缓存中读取数据）而且只能顺序从流中读取数据，如果需要跳过一些字节或者再读取已经读过的字节，你必须将从流中读取的数据先缓存起来。面向块的处理方式有些不同，数据是先被 读/写到buffer中的，根据需要你可以控制读取什么位置的数据。这在处理的过程中给用户多了一些灵活性，然而，你需要额外做的工作是检查你需要的数据是否已经全部到了buffer中，你还需要保证当有更多的数据进入buffer中时，buffer中未处理的数据不会被覆盖。

4. Linux下普通IO



1. 当调用 read 系统调用时，通过 DMA（Direct Memory Access）将数据 copy 到内核模式

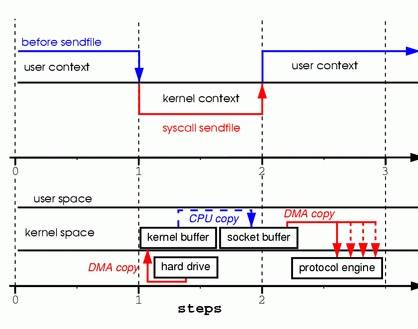
2. 然后由 CPU 控制将内核模式数据 copy 到用户模式下的 buffer 中

3. read 调用完成后，write 调用首先将用户模式下 buffer 中的数据 copy 到内核模式下的 socket buffer 中

4. 最后通过 DMA copy 将内核模式下的 socket buffer 中的数据 copy 到网卡设备中传送。

从上面的过程可以看出，数据白白从内核模式到用户模式走了一圈，浪费了两次 copy(第一次，从kernel模式拷贝到user模式；第二次从user模式再拷贝回kernel模式，即上面4次过程的第2和3步骤。)，而这两次 copy 都是 CPU copy，即占用CPU资源

4.2. sendfile模式



通过 sendfile 传送文件只需要一次系统调用，当调用 sendfile 时：

1. 首先通过 DMA copy 将数据从磁盘读取到 kernel buffer 中

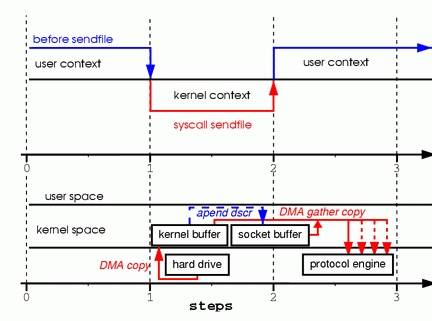
2. 然后通过 CPU copy 将数据从 kernel buffer copy 到 sokcet buffer 中

3. 最终通过 DMA copy 将 socket buffer 中数据 copy 到网卡 buffer 中发送

sendfile 与 read/write 方式相比，少了 一次模式切换一次 CPU copy。但是从上述过程中也可以发现从 kernel buffer 中将数据 copy 到socket buffer 是没必要的。

4.3. sendfile模式改进

Linux2.4 内核对sendFile模式进行了改进：



改进后的处理过程如下：

DMA copy 将磁盘数据 copy 到 kernel buffer 中

2.向 socket buffer 中追加当前要发送的数据在 kernel buffer 中的位置和偏移量

DMA gather copy 根据 socket buffer 中的位置和偏移量直接将 kernel buffer 中的数据 copy 到网卡上。

经过上述过程，数据只经过了 2 次 copy 就从磁盘传送出去了。（事实上这个 Zero copy 是针对内核来讲的，数据在内核模式下是 Zero－copy 的）。

当前许多高性能 http server 都引入了 sendfile 机制，如 nginx，lighttpd 等。

5. Java零拷贝实现的变化

Zero-Copy技术省去了将操作系统的read buffer拷贝到程序的buffer，以及从程序buffer拷贝到socket buffer的步骤，直接将read buffer拷贝到socket buffer. Java NIO中的FileChannal.transferTo()方法就是这样的实现

public void transferTo(long position,long count,WritableByteChannel target);

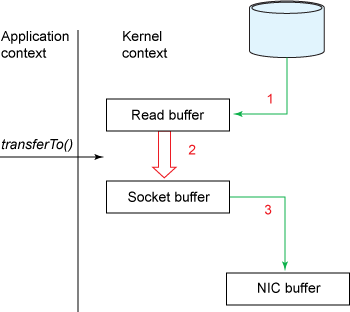
transferTo()方法将数据从一个channel传输到另一个可写的channel上，其内部实现依赖于操作系统对zero copy技术的支持。在unix操作系统和各种linux的发型版本中，这种功能最终是通过sendfile()系统调用实现。下边就是这个方法的定义：

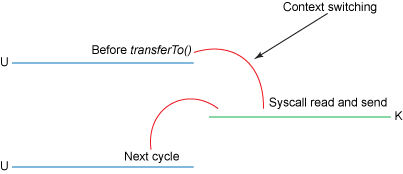
#include <sys/socket.h>

ssize\_t sendfile(int out\_fd, int in\_fd, off\_t \*offset, size\_t count);

5.1. Linux 2.4之前的底层实现

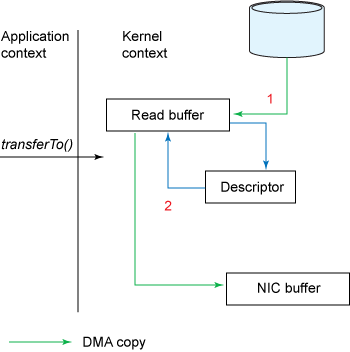
和之前所述一样，我们用下面两幅图更清楚的展示一下发生的复制以及内核态用户态切换：





内核、用户态切换的次数只有两次，将数据的复制只有三次(只有一次用到cpu资源) 在Linux2.4之后，我们可以将这仅有的一次cpu复制也去掉

5.2. Linux 2.4之后的底层实现



在内核为2.4或者以上版本的linux系统上，socket缓冲区描述符将被用来满足这个需求。这个方式不仅减少了内核用户态间的切换，而且也省去了那次需要cpu参与的复制过程。

从用户角度来看依旧是调用transferTo()方法，但是其本质发生了变化：调用transferTo方法后数据被DMA从文件复制到了内核的一个缓冲区中。数据不再被复制到socket关联的缓冲区中了，仅仅是将一个描述符（包含了数据的位置和长度等信息）追加到socket关联的缓冲区中。DMA直接将内核中的缓冲区中的数据传输给协议引擎，消除了仅剩的一次需要cpu周期的数据复制。

# 泛型

**class Fruit** {}  
**class Apple extends Fruit** {}  
**class Orange extends Fruit**{}

1. 说明 List<Fruit> 与 List<Apple> 之间为什么是非继承关系。

不存在继承关系，这里的类型是List，Fruit 和 Apple只是参数，f(ax), 和f(bx)不能保证它们之间功能是否有关系，是不能确定的，真正的功能不是参数决定的，而是函数自身决定的，这道理类似。假如有一个方法，有这么一句

If(P1.class == P2.class) …

如果都是 Apple或者 Orange 运行良好，否则这方法就是不正常的。说白了，参数不能决定函数的功能，参数之间有什么关系，跟函数之间都什么关系是搭不上边的。

1. 说明 List<? super Fruit> 与 List<? extends Fruit> 的区别及 List<? super Fruit> 存在的必要性。

**List**<? **extends Fruit**> **extends**

**List**<? **super Fruit**> **supers**

Extendes 的意思是扩展，上边界。而super得意思是抽象，是下边界。这是在拿数据得区别，存数据是没有任何区别的。它们往里面放数据是相同的，但是取数据是不一样的。Extendes 是一个闭区间 ( ,Fruit] 。而 super 是一个开区间， (Fruite, ) 这就导致了使用 super 是取出得类型只能用 object 接收，而extends 可用 Fruit 来接收，

1. 说明 <? extends Fruit> 与 <T extends Fruit> 区别。
2. 说明 原生List 与 List<?> 区别。
3. 解释自限定泛型 class SelfBound<T extends SelfBound<T>>{ }。

# Hashmap

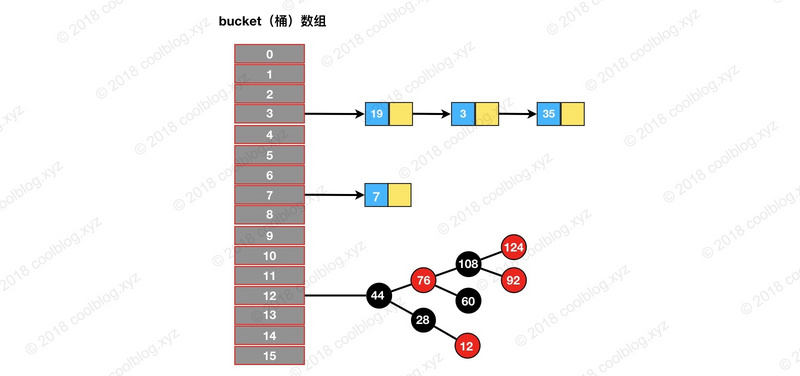
## 1.概述

本篇文章我们来聊聊大家日常开发中常用的一个集合类 - HashMap。HashMap 最早出现在 JDK 1.2中，底层基于散列算法实现。HashMap 允许 null 键和 null 值，在计算哈键的哈希值时，null 键哈希值为 0。HashMap 并不保证键值对的顺序，这意味着在进行某些操作后，键值对的顺序可能会发生变化。另外，需要注意的是，HashMap 是非线程安全类，在多线程环境下可能会存在问题。

在本篇文章中，我将会对 HashMap 中常用方法、重要属性及相关方法进行分析。需要说明的是，HashMap 源码中可分析的点很多，本文很难一一覆盖，请见谅。

## 2.原理

上一节说到 HashMap 底层是基于散列算法实现，散列算法分为散列再探测和拉链式。HashMap 则使用了拉链式的散列算法，并在 JDK 1.8 中引入了红黑树优化过长的链表。数据结构示意图如下：

[](http://www.coolblog.xyz/)

对于拉链式的散列算法，其数据结构是由数组和链表（或树形结构）组成。在进行增删查等操作时，首先要定位到元素的所在桶的位置，之后再从链表中定位该元素。比如我们要查询上图结构中是否包含元素35，步骤如下：

1. 定位元素35所处桶的位置，index = 35 % 16 = 3
2. 在3号桶所指向的链表中继续查找，发现35在链表中。

上面就是 HashMap 底层数据结构的原理，HashMap 基本操作就是对拉链式散列算法基本操作的一层包装。不同的地方在于 JDK 1.8 中引入了红黑树，底层数据结构由数组+链表变为了数组+链表+红黑树，不过本质并未变。好了，原理部分先讲到这，接下来说说源码实现。

## 3.源码分析

本篇文章所分析的源码版本为 JDK 1.8。与 JDK 1.7 相比，JDK 1.8 对 HashMap 进行了一些优化。比如引入红黑树解决过长链表效率低的问题。重写 resize 方法，移除了 alternative hashing 相关方法，避免重新计算键的 hash 等。不过本篇文章并不打算对这些优化进行分析，本文仅会分析 HashMap 常用的方法及一些重要属性和相关方法。如果大家对红黑树感兴趣，可以阅读我的另一篇文章 - [红黑树详细分析](http://www.coolblog.xyz/2018/01/11/%E7%BA%A2%E9%BB%91%E6%A0%91%E8%AF%A6%E7%BB%86%E5%88%86%E6%9E%90/)。

### 3.1 构造方法

#### 3.1.1 构造方法分析

HashMap 的构造方法不多，只有四个。HashMap 构造方法做的事情比较简单，一般都是初始化一些重要变量，比如 loadFactor 和 threshold。而底层的数据结构则是延迟到插入键值对时再进行初始化。HashMap 相关构造方法如下：

/\*\* 构造方法 1 \*/

**public** **HashMap**() {

**this**.loadFactor = DEFAULT\_LOAD\_FACTOR; // all other fields defaulted

}

/\*\* 构造方法 2 \*/

**public** **HashMap**(**int** initialCapacity) {

**this**(initialCapacity, DEFAULT\_LOAD\_FACTOR);

}

/\*\* 构造方法 3 \*/

**public** **HashMap**(**int** initialCapacity, **float** loadFactor) {

**if** (initialCapacity < 0)

**throw** **new** IllegalArgumentException("Illegal initial capacity: " +

initialCapacity);

**if** (initialCapacity > MAXIMUM\_CAPACITY)

initialCapacity = MAXIMUM\_CAPACITY;

**if** (loadFactor <= 0 || Float.isNaN(loadFactor))

**throw** **new** IllegalArgumentException("Illegal load factor: " +

loadFactor);

**this**.loadFactor = loadFactor;

**this**.threshold = tableSizeFor(initialCapacity);

}

/\*\* 构造方法 4 \*/

**public** **HashMap**(Map<? extends K, ? extends V> m) {

**this**.loadFactor = DEFAULT\_LOAD\_FACTOR;

putMapEntries(m, **false**);

}

上面4个构造方法中，大家平时用的最多的应该是第一个了。第一个构造方法很简单，仅将 loadFactor 变量设为默认值。构造方法2调用了构造方法3，而构造方法3仍然只是设置了一些变量。构造方法4则是将另一个 Map 中的映射拷贝一份到自己的存储结构中来，这个方法不是很常用。

上面就是对构造方法简单的介绍，构造方法本身并没什么太多东西，所以就不说了。接下来说说构造方法所初始化的几个的变量。

#### 3.1.2 初始容量、负载因子、阈值

我们在一般情况下，都会使用无参构造方法创建 HashMap。但当我们对时间和空间复杂度有要求的时候，使用默认值有时可能达不到我们的要求，这个时候我们就需要手动调参。在 HashMap 构造方法中，可供我们调整的参数有两个，一个是初始容量 initialCapacity，另一个负载因子 loadFactor。通过这两个设定这两个参数，可以进一步影响阈值大小。但初始阈值 threshold 仅由 initialCapacity 经过移位操作计算得出。他们的作用分别如下：

| **名称** | **用途** |
| --- | --- |
| initialCapacity | HashMap 初始容量 |
| loadFactor | 负载因子 |
| threshold | 当前 HashMap 所能容纳键值对数量的最大值，超过这个值，则需扩容 |

相关代码如下：

/\*\* The default initial capacity - MUST be a power of two. \*/

**static** **final** **int** DEFAULT\_INITIAL\_CAPACITY = 1 << 4;

/\*\* The load factor used when none specified in constructor. \*/

**static** **final** **float** DEFAULT\_LOAD\_FACTOR = 0.75f;

**final** **float** loadFactor;

/\*\* The next size value at which to resize (capacity \* load factor). \*/

**int** threshold;

如果大家去看源码，会发现 HashMap 中没有定义 initialCapacity 这个变量。这个也并不难理解，从参数名上可看出，这个变量表示一个初始容量，只是构造方法中用一次，没必要定义一个变量保存。但如果大家仔细看上面 HashMap 的构造方法，会发现存储键值对的数据结构并不是在构造方法里初始化的。这就有个疑问了，既然叫初始容量，但最终并没有用与初始化数据结构，那传这个参数还有什么用呢？这个问题我先不解释，给大家留个悬念，后面会说明。

默认情况下，HashMap 初始容量是16，负载因子为 0.75。这里并没有默认阈值，原因是阈值可由容量乘上负载因子计算而来（注释中有说明），即threshold = capacity \* loadFactor。但当你仔细看构造方法3时，会发现阈值并不是由上面公式计算而来，而是通过一个方法算出来的。这是不是可以说明 threshold 变量的注释有误呢？还是仅这里进行了特殊处理，其他地方遵循计算公式呢？关于这个疑问，这里也先不说明，后面在分析扩容方法时，再来解释这个问题。接下来，我们来看看初始化 threshold 的方法长什么样的的，源码如下：

/\*\*

\* Returns a power of two size for the given target capacity.

\*/

**static** **final** **int** **tableSizeFor**(**int** cap) {

**int** n = cap - 1;

n |= n >>> 1;

n |= n >>> 2;

n |= n >>> 4;

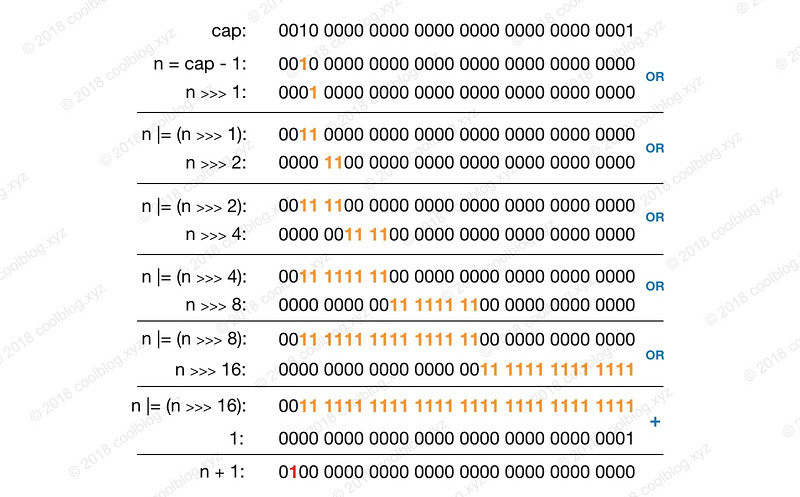
n |= n >>> 8;

n |= n >>> 16;

**return** (n < 0) ? 1 : (n >= MAXIMUM\_CAPACITY) ? MAXIMUM\_CAPACITY : n + 1;

}

上面的代码长的有点不太好看，反正我第一次看的时候不明白它想干啥。不过后来在纸上画画，知道了它的用途。总结起来就一句话：找到大于或等于 cap 的最小2的幂。至于为啥要这样，后面再解释。我们先来看看 tableSizeFor 方法的图解：

[](http://www.coolblog.xyz/)

上面是 tableSizeFor 方法的计算过程图，这里cap = 536,870,913 = 2<sup>29</sup> + 1，多次计算后，算出n + 1 = 1,073,741,824 = 2<sup>30</sup>。通过图解应该可以比较容易理解这个方法的用途，这里就不多说了。

说完了初始阈值的计算过程，再来说说负载因子（loadFactor）。对于 HashMap 来说，负载因子是一个很重要的参数，该参数反应了 HashMap 桶数组的使用情况（假设键值对节点均匀分布在桶数组中）。通过调节负载因子，可使 HashMap 时间和空间复杂度上有不同的表现。当我们调低负载因子时，HashMap 所能容纳的键值对数量变少。扩容时，重新将键值对存储新的桶数组里，键的键之间产生的碰撞会下降，链表长度变短。此时，HashMap 的增删改查等操作的效率将会变高，这里是典型的拿空间换时间。相反，如果增加负载因子（负载因子可以大于1），HashMap 所能容纳的键值对数量变多，空间利用率高，但碰撞率也高。这意味着链表长度变长，效率也随之降低，这种情况是拿时间换空间。至于负载因子怎么调节，这个看使用场景了。一般情况下，我们用默认值就可以了。

### 3.2 查找

HashMap 的查找操作比较简单，查找步骤与原理篇介绍一致，即先定位键值对所在的桶的位置，然后再对链表或红黑树进行查找。通过这两步即可完成查找，该操作相关代码如下：

**public** V **get**(Object key) {

Node<K,V> e;

**return** (e = getNode(hash(key), key)) == **null** ? **null** : e.value;

}

**final** Node<K,V> **getNode**(**int** hash, Object key) {

Node<K,V>[] tab; Node<K,V> first, e; **int** n; K k;

// 1. 定位键值对所在桶的位置

**if** ((tab = table) != **null** && (n = tab.length) > 0 &&

(first = tab[(n - 1) & hash]) != **null**) {

**if** (first.hash == hash && // always check first node

((k = first.key) == key || (key != **null** && key.equals(k))))

**return** first;

**if** ((e = first.next) != **null**) {

// 2. 如果 first 是 TreeNode 类型，则调用黑红树查找方法

**if** (first **instanceof** TreeNode)

**return** ((TreeNode<K,V>)first).getTreeNode(hash, key);

// 2. 对链表进行查找

**do** {

**if** (e.hash == hash &&

((k = e.key) == key || (key != **null** && key.equals(k))))

**return** e;

} **while** ((e = e.next) != **null**);

}

}

**return** **null**;

}

查找的核心逻辑是封装在 getNode 方法中的，getNode 方法源码我已经写了一些注释，应该不难看懂。我们先来看看查找过程的第一步 - 确定桶位置，其实现代码如下：

// index = (n - 1) & hash

first = tab[(n - 1) & hash]

这里通过(n - 1)& hash即可算出桶的在桶数组中的位置，可能有的朋友不太明白这里为什么这么做，这里简单解释一下。HashMap 中桶数组的大小 length 总是2的幂，此时，(n - 1) & hash 等价于对 length 取余。但取余的计算效率没有位运算高，所以(n - 1) & hash也是一个小的优化。举个例子说明一下吧，假设 hash = 185，n = 16。计算过程示意图如下：

[](http://www.coolblog.xyz/)

上面的计算并不复杂，这里就不多说了。

在上面源码中，除了查找相关逻辑，还有一个计算 hash 的方法。这个方法源码如下：

/\*\*

\* 计算键的 hash 值

\*/

**static** **final** **int** **hash**(Object key) {

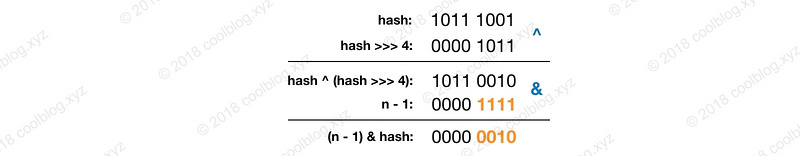
**int** h;

**return** (key == **null**) ? 0 : (h = key.hashCode()) ^ (h >>> 16);

}

看这个方法的逻辑好像是通过位运算重新计算 hash，那么这里为什么要这样做呢？为什么不直接用键的 hashCode 方法产生的 hash 呢？大家先可以思考一下，我把答案写在下面。

这样做有两个好处，我来简单解释一下。我们再看一下上面求余的计算图，图中的 hash 是由键的 hashCode 产生。计算余数时，由于 n 比较小，hash 只有低4位参与了计算，高位的计算可以认为是无效的。这样导致了计算结果只与低位信息有关，高位数据没发挥作用。为了处理这个缺陷，我们可以上图中的 hash 高4位数据与低4位数据进行异或运算，即 hash ^ (hash >>> 4)。通过这种方式，让高位数据与低位数据进行异或，以此加大低位信息的随机性，变相的让高位数据参与到计算中。此时的计算过程如下：

[](http://www.coolblog.xyz/)

在 Java 中，hashCode 方法产生的 hash 是 int 类型，32 位宽。前16位为高位，后16位为低位，所以要右移16位。

上面所说的是重新计算 hash 的一个好处，除此之外，重新计算 hash 的另一个好处是可以增加 hash 的复杂度。当我们覆写 hashCode 方法时，可能会写出分布性不佳的 hashCode 方法，进而导致 hash 的冲突率比较高。通过移位和异或运算，可以让 hash 变得更复杂，进而影响 hash 的分布性。这也就是为什么 HashMap 不直接使用键对象原始 hash 的原因了。

### 3.3 遍历

和查找查找一样，遍历操作也是大家使用频率比较高的一个操作。对于 遍历 HashMap，我们一般都会用下面的方式：

**for**(Object key : map.keySet()) {

// do something

}

或

**for**(HashMap.Entry entry : map.entrySet()) {

// do something

}

从上面代码片段中可以看出，大家一般都是对 HashMap 的 key 集合或 Entry 集合进行遍历。上面代码片段中用 foreach 遍历 keySet 方法产生的集合，在编译时会转换成用迭代器遍历，等价于：

Set keys = map.keySet();

Iterator ite = keys.iterator();

**while** (ite.hasNext()) {

Object key = ite.next();

// do something

}

大家在遍历 HashMap 的过程中会发现，多次对 HashMap 进行遍历时，遍历结果顺序都是一致的。但这个顺序和插入的顺序一般都是不一致的。产生上述行为的原因是怎样的呢？大家想一下原因。我先把遍历相关的代码贴出来，如下：

**public** Set<K> **keySet**() {

Set<K> ks = keySet;

**if** (ks == **null**) {

ks = **new** KeySet();

keySet = ks;

}

**return** ks;

}

/\*\*

\* 键集合

\*/

**final** **class** **KeySet** **extends** **AbstractSet**<**K**> {

**public** **final** **int** **size**() { **return** size; }

**public** **final** **void** **clear**() { HashMap.**this**.clear(); }

**public** **final** Iterator<K> **iterator**() { **return** **new** KeyIterator(); }

**public** **final** **boolean** **contains**(Object o) { **return** containsKey(o); }

**public** **final** **boolean** **remove**(Object key) {

**return** removeNode(hash(key), key, **null**, **false**, **true**) != **null**;

}

// 省略部分代码

}

/\*\*

\* 键迭代器

\*/

**final** **class** **KeyIterator** **extends** **HashIterator**

**implements** **Iterator**<**K**> {

**public** **final** K **next**() { **return** nextNode().key; }

}

**abstract** **class** **HashIterator** {

Node<K,V> next; // next entry to return

Node<K,V> current; // current entry

**int** expectedModCount; // for fast-fail

**int** index; // current slot

HashIterator() {

expectedModCount = modCount;

Node<K,V>[] t = table;

current = next = **null**;

index = 0;

**if** (t != **null** && size > 0) { // advance to first entry

// 寻找第一个包含链表节点引用的桶

**do** {} **while** (index < t.length && (next = t[index++]) == **null**);

}

}

**public** **final** **boolean** **hasNext**() {

**return** next != **null**;

}

**final** Node<K,V> **nextNode**() {

Node<K,V>[] t;

Node<K,V> e = next;

**if** (modCount != expectedModCount)

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

**if** (e == **null**)

**throw** **new** NoSuchElementException();

**if** ((next = (current = e).next) == **null** && (t = table) != **null**) {

// 寻找下一个包含链表节点引用的桶

**do** {} **while** (index < t.length && (next = t[index++]) == **null**);

}

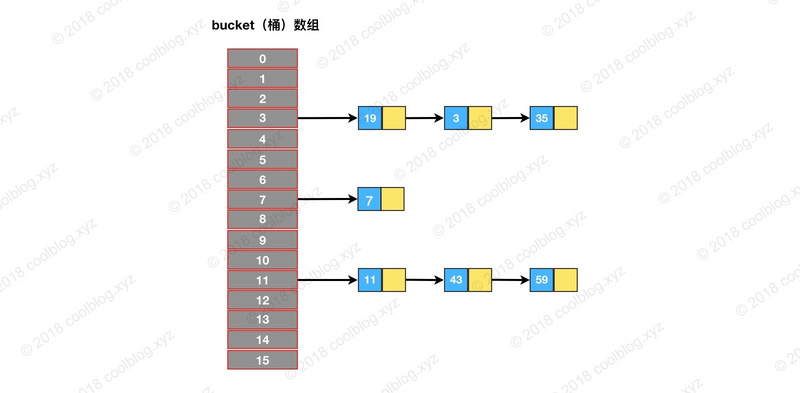
**return** e;

}

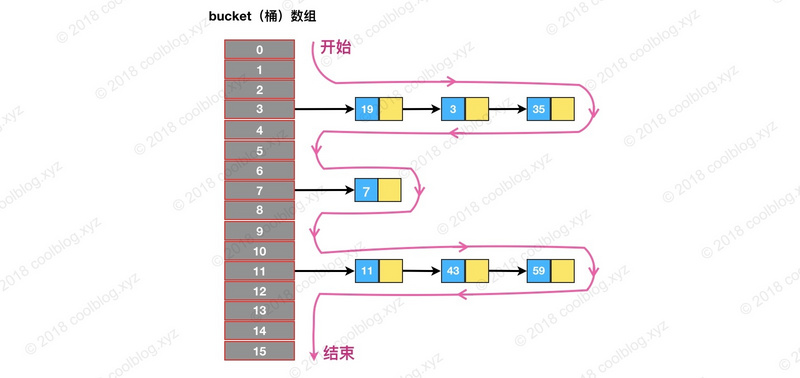
//省略部分代码

}

如上面的源码，遍历所有的键时，首先要获取键集合KeySet对象，然后再通过 KeySet 的迭代器KeyIterator进行遍历。KeyIterator 类继承自HashIterator类，核心逻辑也封装在 HashIterator 类中。HashIterator 的逻辑并不复杂，在初始化时，HashIterator 先从桶数组中找到包含链表节点引用的桶。然后对这个桶指向的链表进行遍历。遍历完成后，再继续寻找下一个包含链表节点引用的桶，找到继续遍历。找不到，则结束遍历。举个例子，假设我们遍历下图的结构：

[](http://www.coolblog.xyz/)

HashIterator 在初始化时，会先遍历桶数组，找到包含链表节点引用的桶，对应图中就是3号桶。随后由 nextNode 方法遍历该桶所指向的链表。遍历完3号桶后，nextNode 方法继续寻找下一个不为空的桶，对应图中的7号桶。之后流程和上面类似，直至遍历完最后一个桶。以上就是 HashIterator 的核心逻辑的流程，对应下图：

[](http://www.coolblog.xyz/)

遍历上图的最终结果是 19 -> 3 -> 35 -> 7 -> 11 -> 43 -> 59，为了验证正确性，简单写点测试代码跑一下看看。测试代码如下：

/\*\*

\* 应在 JDK 1.8 下测试，其他环境下不保证结果和上面一致

\*/

**public** **class** **HashMapTest** {

**@Test**

**public** **void** **testTraversal**() {

HashMap<Integer, String> map = **new** HashMap(16);

map.put(7, "");

map.put(11, "");

map.put(43, "");

map.put(59, "");

map.put(19, "");

map.put(3, "");

map.put(35, "");

System.out.println("遍历结果：");

**for** (Integer key : map.keySet()) {

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

}

}

}

遍历结果如下：  
[](http://www.coolblog.xyz/)

在本小节的最后，抛两个问题给大家。在 JDK 1.8 版本中，为了避免过长的链表对 HashMap 性能的影响，特地引入了红黑树优化性能。但在上面的源码中并没有发现红黑树遍历的相关逻辑，这是为什么呢？对于被转换成红黑树的链表该如何遍历呢？大家可以先想想，然后可以去源码或本文后续章节中找答案。

### 3.4 插入

#### 3.4.1 插入逻辑分析

通过前两节的分析，大家对 HashMap 低层的数据结构应该了然于心了。即使我不说，大家也应该能知道 HashMap 的插入流程是什么样的了。首先肯定是先定位要插入的键值对属于哪个桶，定位到桶后，再判断桶是否为空。如果为空，则将键值对存入即可。如果不为空，则需将键值对接在链表最后一个位置，或者更新键值对。这就是 HashMap 的插入流程，是不是觉得很简单。当然，大家先别高兴。这只是一个简化版的插入流程，真正的插入流程要复杂不少。首先 HashMap 是变长集合，所以需要考虑扩容的问题。其次，在 JDK 1.8 中，HashMap 引入了红黑树优化过长链表，这里还要考虑多长的链表需要进行优化，优化过程又是怎样的问题。引入这里两个问题后，大家会发现原本简单的操作，现在略显复杂了。在本节中，我将先分析插入操作的源码，扩容、树化（链表转为红黑树，下同）以及其他和树结构相关的操作，随后将在独立的两小结中进行分析。接下来，先来看一下插入操作的源码：

**public** V **put**(K key, V value) {

**return** putVal(hash(key), key, value, **false**, **true**);

}

**final** V **putVal**(**int** hash, K key, V value, **boolean** onlyIfAbsent,

**boolean** evict) {

Node<K,V>[] tab; Node<K,V> p; **int** n, i;

// 初始化桶数组 table，table 被延迟到插入新数据时再进行初始化

**if** ((tab = table) == **null** || (n = tab.length) == 0)

n = (tab = resize()).length;

// 如果桶中不包含键值对节点引用，则将新键值对节点的引用存入桶中即可

**if** ((p = tab[i = (n - 1) & hash]) == **null**)

tab[i] = newNode(hash, key, value, **null**);

**else** {

Node<K,V> e; K k;

// 如果键的值以及节点 hash 等于链表中的第一个键值对节点时，则将 e 指向该键值对

**if** (p.hash == hash &&

((k = p.key) == key || (key != **null** && key.equals(k))))

e = p;

// 如果桶中的引用类型为 TreeNode，则调用红黑树的插入方法

**else** **if** (p **instanceof** TreeNode)

e = ((TreeNode<K,V>)p).putTreeVal(**this**, tab, hash, key, value);

**else** {

// 对链表进行遍历，并统计链表长度

**for** (**int** binCount = 0; ; ++binCount) {

// 链表中不包含要插入的键值对节点时，则将该节点接在链表的最后

**if** ((e = p.next) == **null**) {

p.next = newNode(hash, key, value, **null**);

// 如果链表长度大于或等于树化阈值，则进行树化操作

**if** (binCount >= TREEIFY\_THRESHOLD - 1) // -1 for 1st

treeifyBin(tab, hash);

**break**;

}

// 条件为 true，表示当前链表包含要插入的键值对，终止遍历

**if** (e.hash == hash &&

((k = e.key) == key || (key != **null** && key.equals(k))))

**break**;

p = e;

}

}

// 判断要插入的键值对是否存在 HashMap 中

**if** (e != **null**) { // existing mapping for key

V oldValue = e.value;

// onlyIfAbsent 表示是否仅在 oldValue 为 null 的情况下更新键值对的值

**if** (!onlyIfAbsent || oldValue == **null**)

e.value = value;

afterNodeAccess(e);

**return** oldValue;

}

}

++modCount;

// 键值对数量超过阈值时，则进行扩容

**if** (++size > threshold)

resize();

afterNodeInsertion(evict);

**return** **null**;

}

插入操作的入口方法是 put(K,V)，但核心逻辑在V putVal(int, K, V, boolean, boolean) 方法中。putVal 方法主要做了这么几件事情：

1. 当桶数组 table 为空时，通过扩容的方式初始化 table
2. 查找要插入的键值对是否已经存在，存在的话根据条件判断是否用新值替换旧值
3. 如果不存在，则将键值对链入链表中，并根据链表长度决定是否将链表转为红黑树
4. 判断键值对数量是否大于阈值，大于的话则进行扩容操作

以上就是 HashMap 插入的逻辑，并不是很复杂，这里就不多说了。接下来来分析一下扩容机制。

#### 3.4.2 扩容机制

在 Java 中，数组的长度是固定的，这意味着数组只能存储固定量的数据。但在开发的过程中，很多时候我们无法知道该建多大的数组合适。建小了不够用，建大了用不完，造成浪费。如果我们能实现一种变长的数组，并按需分配空间就好了。好在，我们不用自己实现变长数组，Java 集合框架已经实现了变长的数据结构。比如 ArrayList 和 HashMap。对于这类基于数组的变长数据结构，扩容是一个非常重要的操作。下面就来聊聊 HashMap 的扩容机制。

在详细分析之前，先来说一下扩容相关的背景知识：

在 HashMap 中，桶数组的长度均是2的幂，阈值大小为桶数组长度与负载因子的乘积。当 HashMap 中的键值对数量超过阈值时，进行扩容。

HashMap 的扩容机制与其他变长集合的套路不太一样，HashMap 按当前桶数组长度的2倍进行扩容，阈值也变为原来的2倍（如果计算过程中，阈值溢出归零，则按阈值公式重新计算）。扩容之后，要重新计算键值对的位置，并把它们移动到合适的位置上去。以上就是 HashMap 的扩容大致过程，接下来我们来看看具体的实现：

**final** Node<K,V>[] resize() {

Node<K,V>[] oldTab = table;

**int** oldCap = (oldTab == **null**) ? 0 : oldTab.length;

**int** oldThr = threshold;

**int** newCap, newThr = 0;

// 如果 table 不为空，表明已经初始化过了

**if** (oldCap > 0) {

// 当 table 容量超过容量最大值，则不再扩容

**if** (oldCap >= MAXIMUM\_CAPACITY) {

threshold = Integer.MAX\_VALUE;

**return** oldTab;

}

// 按旧容量和阈值的2倍计算新容量和阈值的大小

**else** **if** ((newCap = oldCap << 1) < MAXIMUM\_CAPACITY &&

oldCap >= DEFAULT\_INITIAL\_CAPACITY)

newThr = oldThr << 1; // double threshold

} **else** **if** (oldThr > 0) // initial capacity was placed in threshold

/\*

\* 初始化时，将 threshold 的值赋值给 newCap，

\* HashMap 使用 threshold 变量暂时保存 initialCapacity 参数的值

\*/

newCap = oldThr;

**else** { // zero initial threshold signifies using defaults

/\*

\* 调用无参构造方法时，桶数组容量为默认容量，

\* 阈值为默认容量与默认负载因子乘积

\*/

newCap = DEFAULT\_INITIAL\_CAPACITY;

newThr = (**int**)(DEFAULT\_LOAD\_FACTOR \* DEFAULT\_INITIAL\_CAPACITY);

}

// newThr 为 0 时，按阈值计算公式进行计算

**if** (newThr == 0) {

**float** ft = (**float**)newCap \* loadFactor;

newThr = (newCap < MAXIMUM\_CAPACITY && ft < (**float**)MAXIMUM\_CAPACITY ?

(**int**)ft : Integer.MAX\_VALUE);

}

threshold = newThr;

// 创建新的桶数组，桶数组的初始化也是在这里完成的

Node<K,V>[] newTab = (Node<K,V>[])**new** Node[newCap];

table = newTab;

**if** (oldTab != **null**) {

// 如果旧的桶数组不为空，则遍历桶数组，并将键值对映射到新的桶数组中

**for** (**int** j = 0; j < oldCap; ++j) {

Node<K,V> e;

**if** ((e = oldTab[j]) != **null**) {

oldTab[j] = **null**;

**if** (e.next == **null**)

newTab[e.hash & (newCap - 1)] = e;

**else** **if** (e **instanceof** TreeNode)

// 重新映射时，需要对红黑树进行拆分

((TreeNode<K,V>)e).split(**this**, newTab, j, oldCap);

**else** { // preserve order

Node<K,V> loHead = **null**, loTail = **null**;

Node<K,V> hiHead = **null**, hiTail = **null**;

Node<K,V> next;

// 遍历链表，并将链表节点按原顺序进行分组

**do** {

next = e.next;

**if** ((e.hash & oldCap) == 0) {

**if** (loTail == **null**)

loHead = e;

**else**

loTail.next = e;

loTail = e;

}

**else** {

**if** (hiTail == **null**)

hiHead = e;

**else**

hiTail.next = e;

hiTail = e;

}

} **while** ((e = next) != **null**);

// 将分组后的链表映射到新桶中

**if** (loTail != **null**) {

loTail.next = **null**;

newTab[j] = loHead;

}

**if** (hiTail != **null**) {

hiTail.next = **null**;

newTab[j + oldCap] = hiHead;

}

}

}

}

}

**return** newTab;

}

上面的源码有点长，希望大家耐心看懂它的逻辑。上面的源码总共做了3件事，分别是：

1. 计算新桶数组的容量 newCap 和新阈值 newThr
2. 根据计算出的 newCap 创建新的桶数组，桶数组 table 也是在这里进行初始化的
3. 将键值对节点重新映射到新的桶数组里。如果节点是 TreeNode 类型，则需要拆分红黑树。如果是普通节点，则节点按原顺序进行分组。

上面列的三点中，创建新的桶数组就一行代码，不用说了。接下来，来说说第一点和第三点，先说说 newCap 和 newThr 计算过程。该计算过程对应 resize 源码的第一和第二个条件分支，如下：

// 第一个条件分支

**if** ( oldCap > 0) {

// 嵌套条件分支

**if** (oldCap >= MAXIMUM\_CAPACITY) {...}

**else** **if** ((newCap = oldCap << 1) < MAXIMUM\_CAPACITY &&

oldCap >= DEFAULT\_INITIAL\_CAPACITY) {...}

}

**else** **if** (oldThr > 0) {...}

**else** {...}

// 第二个条件分支

**if** (newThr == 0) {...}

通过这两个条件分支对不同情况进行判断，进而算出不同的容量值和阈值。它们所覆盖的情况如下：

分支一：

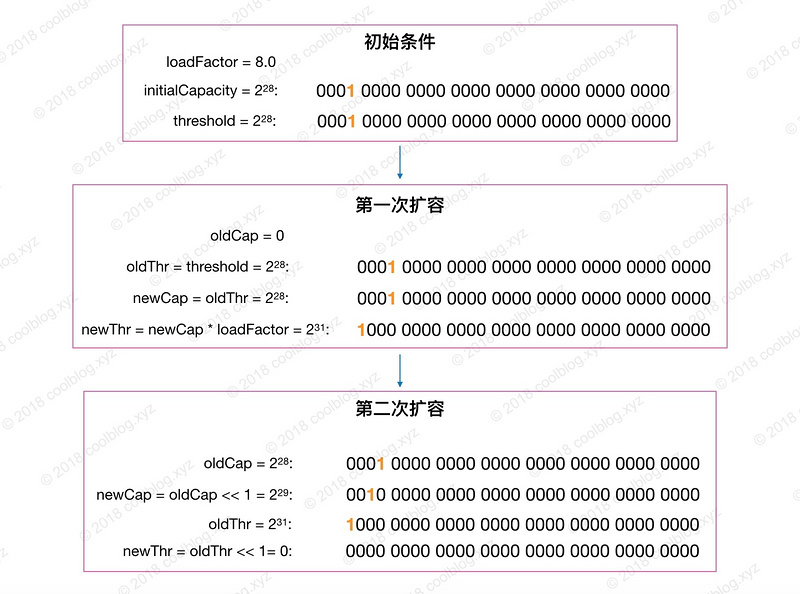
| **条件** | **覆盖情况** | **备注** |
| --- | --- | --- |
| oldCap > 0 | 桶数组 table 已经被初始化 |  |
| oldThr > 0 | threshold > 0，且桶数组未被初始化 | 调用 HashMap(int) 和 HashMap(int, float) 构造方法时会产生这种情况，此种情况下 newCap = oldThr，newThr 在第二个条件分支中算出 |
| oldCap == 0 && oldThr == 0 | 桶数组未被初始化，且 threshold 为 0 | 调用 HashMap() 构造方法会产生这种情况。 |

这里把oldThr > 0情况单独拿出来说一下。在这种情况下，会将 oldThr 赋值给 newCap，等价于newCap = threshold = tableSizeFor(initialCapacity)。我们在初始化时传入的 initialCapacity 参数经过 threshold 中转最终赋值给了 newCap。这也就解答了前面提的一个疑问：initialCapacity 参数没有被保存下来，那么它怎么参与桶数组的初始化过程的呢？

嵌套分支：

| **条件** | **覆盖情况** | **备注** |
| --- | --- | --- |
| oldCap >= 230 | 桶数组容量大于或等于最大桶容量 230 | 这种情况下不再扩容 |
| newCap < 230 && oldCap > 16 | 新桶数组容量小于最大值，且旧桶数组容量大于 16 | 该种情况下新阈值 newThr = oldThr << 1，移位可能会导致溢出 |

这里简单说明一下移位导致的溢出情况，当 loadFactor小数位为 0，整数位可被2整除且大于等于8时，在某次计算中就可能会导致 newThr 溢出归零。见下图：

[](http://www.coolblog.xyz/)

分支二：

| **条件** | **覆盖情况** | **备注** |
| --- | --- | --- |
| newThr == 0 | 第一个条件分支未计算 newThr 或嵌套分支在计算过程中导致 newThr 溢出归零 |  |

说完 newCap 和 newThr 的计算过程，接下来再来分析一下键值对节点重新映射的过程。

在 JDK 1.8 中，重新映射节点需要考虑节点类型。对于树形节点，需先拆分红黑树再映射。对于链表类型节点，则需先对链表进行分组，然后再映射。需要的注意的是，分组后，组内节点相对位置保持不变。关于红黑树拆分的逻辑将会放在下一小节说明，先来看看链表是怎样进行分组映射的。

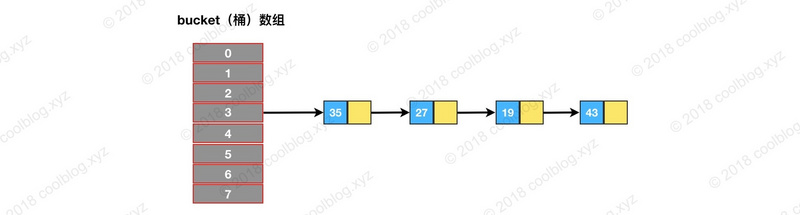
我们都知道往底层数据结构中插入节点时，一般都是先通过模运算计算桶位置，接着把节点放入桶中即可。事实上，我们可以把重新映射看做插入操作。在 JDK 1.7 中，也确实是这样做的。但在 JDK 1.8 中，则对这个过程进行了一定的优化，逻辑上要稍微复杂一些。在详细分析前，我们先来回顾一下 hash 求余的过程：

[](http://www.coolblog.xyz/)

上图中，桶数组大小 n = 16，hash1 与 hash2 不相等。但因为只有后4位参与求余，所以结果相等。当桶数组扩容后，n 由16变成了32，对上面的 hash 值重新进行映射：

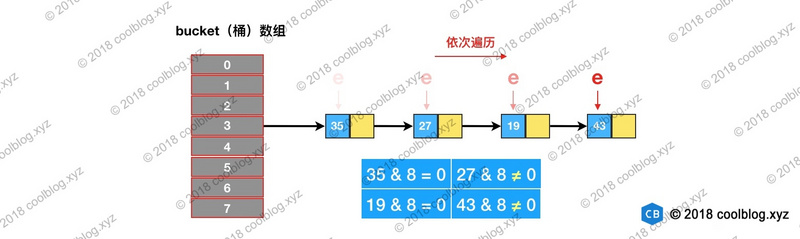
[](http://www.coolblog.xyz/)

扩容后，参与模运算的位数由4位变为了5位。由于两个 hash 第5位的值是不一样，所以两个 hash 算出的结果也不一样。上面的计算过程并不难理解，继续往下分析。

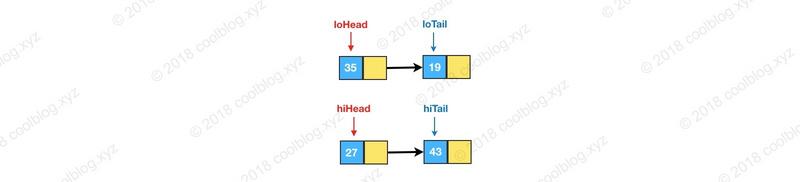
[](http://www.coolblog.xyz/)

假设我们上图的桶数组进行扩容，扩容后容量 n = 16，重新映射过程如下:

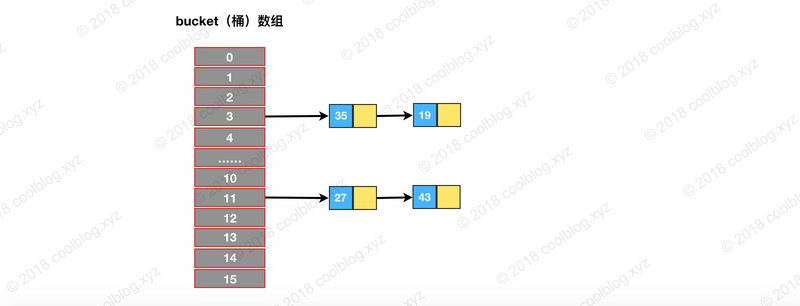
依次遍历链表，并计算节点 hash & oldCap 的值。如下图所示



如果值为0，将 loHead 和 loTail 指向这个节点。如果后面还有节点 hash & oldCap 为0的话，则将节点链入 loHead 指向的链表中，并将 loTail 指向该节点。如果值为非0的话，则让 hiHead 和 hiTail 指向该节点。完成遍历后，可能会得到两条链表，此时就完成了链表分组：

[](http://www.coolblog.xyz/)

最后再将这两条链接存放到相应的桶中，完成扩容。如下图：

[](http://www.coolblog.xyz/)

从上图可以发现，重新映射后，两条链表中的节点顺序并未发生变化，还是保持了扩容前的顺序。以上就是 JDK 1.8 中 HashMap 扩容的代码讲解。另外再补充一下，JDK 1.8 版本下 HashMap 扩容效率要高于之前版本。如果大家看过 JDK 1.7 的源码会发现，JDK 1.7 为了防止因 hash 碰撞引发的拒绝服务攻击，在计算 hash 过程中引入随机种子。以增强 hash 的随机性，使得键值对均匀分布在桶数组中。在扩容过程中，相关方法会根据容量判断是否需要生成新的随机种子，并重新计算所有节点的 hash。而在 JDK 1.8 中，则通过引入红黑树替代了该种方式。从而避免了多次计算 hash 的操作，提高了扩容效率。

本小节的内容讲就先讲到这，接下来，来讲讲链表与红黑树相互转换的过程。

#### 3.4.3 链表树化、红黑树链化与拆分

JDK 1.8 对 HashMap 实现进行了改进。最大的改进莫过于在引入了红黑树处理频繁的碰撞，代码复杂度也随之上升。比如，以前只需实现一套针对链表操作的方法即可。而引入红黑树后，需要另外实现红黑树相关的操作。红黑树是一种自平衡的二叉查找树，本身就比较复杂。本篇文章中并不打算对红黑树展开介绍，本文仅会介绍链表树化需要注意的地方。至于红黑树详细的介绍，如果大家有兴趣，可以参考我的另一篇文章 - [红黑树详细分析](http://www.coolblog.xyz/2018/01/11/%E7%BA%A2%E9%BB%91%E6%A0%91%E8%AF%A6%E7%BB%86%E5%88%86%E6%9E%90/)。

在展开说明之前，先把树化的相关代码贴出来，如下：

**static** **final** **int** TREEIFY\_THRESHOLD = 8;

/\*\*

\* 当桶数组容量小于该值时，优先进行扩容，而不是树化

\*/

**static** **final** **int** MIN\_TREEIFY\_CAPACITY = 64;

**static** **final** **class** **TreeNode**<**K**,**V**> **extends** **LinkedHashMap**.**Entry**<**K**,**V**> {

TreeNode<K,V> parent; // red-black tree links

TreeNode<K,V> left;

TreeNode<K,V> right;

TreeNode<K,V> prev; // needed to unlink next upon deletion

**boolean** red;

TreeNode(**int** hash, K key, V val, Node<K,V> next) {

**super**(hash, key, val, next);

}

}

/\*\*

\* 将普通节点链表转换成树形节点链表

\*/

**final** **void** **treeifyBin**(Node<K,V>[] tab, **int** hash) {

**int** n, index; Node<K,V> e;

// 桶数组容量小于 MIN\_TREEIFY\_CAPACITY，优先进行扩容而不是树化

**if** (tab == **null** || (n = tab.length) < MIN\_TREEIFY\_CAPACITY)

resize();

**else** **if** ((e = tab[index = (n - 1) & hash]) != **null**) {

// hd 为头节点（head），tl 为尾节点（tail）

TreeNode<K,V> hd = **null**, tl = **null**;

**do** {

// 将普通节点替换成树形节点

TreeNode<K,V> p = replacementTreeNode(e, **null**);

**if** (tl == **null**)

hd = p;

**else** {

p.prev = tl;

tl.next = p;

}

tl = p;

} **while** ((e = e.next) != **null**); // 将普通链表转成由树形节点链表

**if** ((tab[index] = hd) != **null**)

// 将树形链表转换成红黑树

hd.treeify(tab);

}

}

TreeNode<K,V> **replacementTreeNode**(Node<K,V> p, Node<K,V> next) {

**return** **new** TreeNode<>(p.hash, p.key, p.value, next);

}

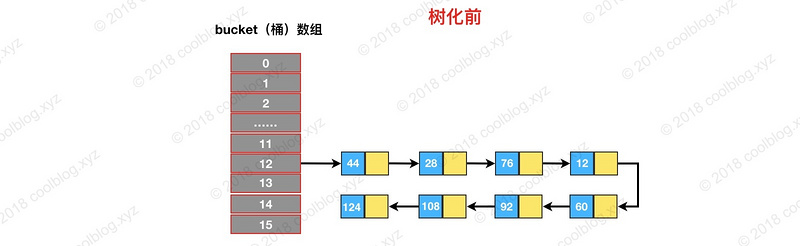
在扩容过程中，树化要满足两个条件：

1. 链表长度大于等于 TREEIFY\_THRESHOLD
2. 桶数组容量大于等于 MIN\_TREEIFY\_CAPACITY

第一个条件比较好理解，这里就不说了。这里来说说加入第二个条件的原因，个人觉得原因如下：

当桶数组容量比较小时，键值对节点 hash 的碰撞率可能会比较高，进而导致链表长度较长。这个时候应该优先扩容，而不是立马树化。毕竟高碰撞率是因为桶数组容量较小引起的，这个是主因。容量小时，优先扩容可以避免一些列的不必要的树化过程。同时，桶容量较小时，扩容会比较频繁，扩容时需要拆分红黑树并重新映射。所以在桶容量比较小的情况下，将长链表转成红黑树是一件吃力不讨好的事。

回到上面的源码中，我们继续看一下 treeifyBin 方法。该方法主要的作用是将普通链表转成为由 TreeNode 型节点组成的链表，并在最后调用 treeify 是将该链表转为红黑树。TreeNode 继承自 Node 类，所以 TreeNode 仍然包含 next 引用，原链表的节点顺序最终通过 next 引用被保存下来。我们假设树化前，链表结构如下：

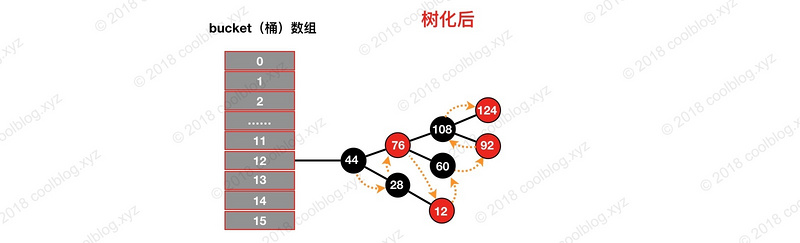
[](http://www.coolblog.xyz/)

HashMap 在设计之初，并没有考虑到以后会引入红黑树进行优化。所以并没有像 TreeMap 那样，要求键类实现 comparable 接口或提供相应的比较器。但由于树化过程需要比较两个键对象的大小，在键类没有实现 comparable 接口的情况下，怎么比较键与键之间的大小了就成了一个棘手的问题。为了解决这个问题，HashMap 是做了三步处理，确保可以比较出两个键的大小，如下：

1. 比较键与键之间 hash 的大小，如果 hash 相同，继续往下比较
2. 检测键类是否实现了 Comparable 接口，如果实现调用 compareTo 方法进行比较
3. 如果仍未比较出大小，就需要进行仲裁了，仲裁方法为 tieBreakOrder（大家自己看源码吧）

tie break 是网球术语，可以理解为加时赛的意思，起这个名字还是挺有意思的。

通过上面三次比较，最终就可以比较出孰大孰小。比较出大小后就可以构造红黑树了，最终构造出的红黑树如下：

[](http://www.coolblog.xyz/)

橙色的箭头表示 TreeNode 的 next 引用。由于空间有限，prev 引用未画出。可以看出，链表转成红黑树后，原链表的顺序仍然会被引用仍被保留了（红黑树的根节点会被移动到链表的第一位），我们仍然可以按遍历链表的方式去遍历上面的红黑树。这样的结构为后面红黑树的切分以及红黑树转成链表做好了铺垫，我们继续往下分析。

##### **红黑树拆分**

扩容后，普通节点需要重新映射，红黑树节点也不例外。按照一般的思路，我们可以先把红黑树转成链表，之后再重新映射链表即可。这种处理方式是大家比较容易想到的，但这样做会损失一定的效率。不同于上面的处理方式，HashMap 实现的思路则是上好佳（上好佳请把广告费打给我）。如上节所说，在将普通链表转成红黑树时，HashMap 通过两个额外的引用 next 和 prev 保留了原链表的节点顺序。这样再对红黑树进行重新映射时，完全可以按照映射链表的方式进行。这样就避免了将红黑树转成链表后再进行映射，无形中提高了效率。

以上就是红黑树拆分的逻辑，下面看一下具体实现吧：

// 红黑树转链表阈值

**static** **final** **int** UNTREEIFY\_THRESHOLD = 6;

**final** **void** **split**(HashMap<K,V> map, Node<K,V>[] tab, **int** index, **int** bit) {

TreeNode<K,V> b = **this**;

// Relink into lo and hi lists, preserving order

TreeNode<K,V> loHead = **null**, loTail = **null**;

TreeNode<K,V> hiHead = **null**, hiTail = **null**;

**int** lc = 0, hc = 0;

/\*

\* 红黑树节点仍然保留了 next 引用，故仍可以按链表方式遍历红黑树。

\* 下面的循环是对红黑树节点进行分组，与上面类似

\*/

**for** (TreeNode<K,V> e = b, next; e != **null**; e = next) {

next = (TreeNode<K,V>)e.next;

e.next = **null**;

**if** ((e.hash & bit) == 0) {

**if** ((e.prev = loTail) == **null**)

loHead = e;

**else**

loTail.next = e;

loTail = e;

++lc;

}

**else** {

**if** ((e.prev = hiTail) == **null**)

hiHead = e;

**else**

hiTail.next = e;

hiTail = e;

++hc;

}

}

**if** (loHead != **null**) {

// 如果 loHead 不为空，且链表长度小于等于 6，则将红黑树转成链表

**if** (lc <= UNTREEIFY\_THRESHOLD)

tab[index] = loHead.untreeify(map);

**else** {

tab[index] = loHead;

/\*

\* hiHead == null 时，表明扩容后，

\* 所有节点仍在原位置，树结构不变，无需重新树化

\*/

**if** (hiHead != **null**)

loHead.treeify(tab);

}

}

// 与上面类似

**if** (hiHead != **null**) {

**if** (hc <= UNTREEIFY\_THRESHOLD)

tab[index + bit] = hiHead.untreeify(map);

**else** {

tab[index + bit] = hiHead;

**if** (loHead != **null**)

hiHead.treeify(tab);

}

}

}

从源码上可以看得出，重新映射红黑树的逻辑和重新映射链表的逻辑基本一致。不同的地方在于，重新映射后，会将红黑树拆分成两条由 TreeNode 组成的链表。如果链表长度小于 UNTREEIFY\_THRESHOLD，则将链表转换成普通链表。否则根据条件重新将 TreeNode 链表树化。举个例子说明一下，假设扩容后，重新映射上图的红黑树，映射结果如下：

[](http://www.coolblog.xyz/)

##### **红黑树链化**

前面说过，红黑树中仍然保留了原链表节点顺序。有了这个前提，再将红黑树转成链表就简单多了，仅需将 TreeNode 链表转成 Node 类型的链表即可。相关代码如下：

**final** Node<K,V> **untreeify**(HashMap<K,V> map) {

Node<K,V> hd = **null**, tl = **null**;

// 遍历 TreeNode 链表，并用 Node 替换

**for** (Node<K,V> q = **this**; q != **null**; q = q.next) {

// 替换节点类型

Node<K,V> p = map.replacementNode(q, **null**);

**if** (tl == **null**)

hd = p;

**else**

tl.next = p;

tl = p;

}

**return** hd;

}

Node<K,V> **replacementNode**(Node<K,V> p, Node<K,V> next) {

**return** **new** Node<>(p.hash, p.key, p.value, next);

}

上面的代码并不复杂，不难理解，这里就不多说了。到此扩容相关内容就说完了，不知道大家理解没。

### 3.5 删除

如果大家坚持看完了前面的内容，到本节就可以轻松一下。当然，前提是不去看红黑树的删除操作。不过红黑树并非本文讲解重点，本节中也不会介绍红黑树相关内容，所以大家不用担心。

HashMap 的删除操作并不复杂，仅需三个步骤即可完成。第一步是定位桶位置，第二步遍历链表并找到键值相等的节点，第三步删除节点。相关源码如下：

**public** V **remove**(Object key) {

Node<K,V> e;

**return** (e = removeNode(hash(key), key, **null**, **false**, **true**)) == **null** ?

**null** : e.value;

}

**final** Node<K,V> **removeNode**(**int** hash, Object key, Object value,

**boolean** matchValue, **boolean** movable) {

Node<K,V>[] tab; Node<K,V> p; **int** n, index;

**if** ((tab = table) != **null** && (n = tab.length) > 0 &&

// 1. 定位桶位置

(p = tab[index = (n - 1) & hash]) != **null**) {

Node<K,V> node = **null**, e; K k; V v;

// 如果键的值与链表第一个节点相等，则将 node 指向该节点

**if** (p.hash == hash &&

((k = p.key) == key || (key != **null** && key.equals(k))))

node = p;

**else** **if** ((e = p.next) != **null**) {

// 如果是 TreeNode 类型，调用红黑树的查找逻辑定位待删除节点

**if** (p **instanceof** TreeNode)

node = ((TreeNode<K,V>)p).getTreeNode(hash, key);

**else** {

// 2. 遍历链表，找到待删除节点

**do** {

**if** (e.hash == hash &&

((k = e.key) == key ||

(key != **null** && key.equals(k)))) {

node = e;

**break**;

}

p = e;

} **while** ((e = e.next) != **null**);

}

}

// 3. 删除节点，并修复链表或红黑树

**if** (node != **null** && (!matchValue || (v = node.value) == value ||

(value != **null** && value.equals(v)))) {

**if** (node **instanceof** TreeNode)

((TreeNode<K,V>)node).removeTreeNode(**this**, tab, movable);

**else** **if** (node == p)

tab[index] = node.next;

**else**

p.next = node.next;

++modCount;

--size;

afterNodeRemoval(node);

**return** node;

}

}

**return** **null**;

}

删除操作本身并不复杂，有了前面的基础，理解起来也就不难了，这里就不多说了。

### 3.6 其他细节

前面的内容分析了 HashMap 的常用操作及相关的源码，本节内容再补充一点其他方面的东西。

#### 被 transient 所修饰 table 变量

如果大家细心阅读 HashMap 的源码，会发现桶数组 table 被申明为 transient。transient 表示易变的意思，在 Java 中，被该关键字修饰的变量不会被默认的序列化机制序列化。我们再回到源码中，考虑一个问题：桶数组 table 是 HashMap 底层重要的数据结构，不序列化的话，别人还怎么还原呢？

这里简单说明一下吧，HashMap 并没有使用默认的序列化机制，而是通过实现readObject/writeObject两个方法自定义了序列化的内容。这样做是有原因的，试问一句，HashMap 中存储的内容是什么？不用说，大家也知道是键值对。所以只要我们把键值对序列化了，我们就可以根据键值对数据重建 HashMap。有的朋友可能会想，序列化 table 不是可以一步到位，后面直接还原不就行了吗？这样一想，倒也是合理。但序列化 talbe 存在着两个问题：

1. table 多数情况下是无法被存满的，序列化未使用的部分，浪费空间
2. 同一个键值对在不同 JVM 下，所处的桶位置可能是不同的，在不同的 JVM 下反序列化 table 可能会发生错误。

以上两个问题中，第一个问题比较好理解，第二个问题解释一下。HashMap 的get/put/remove等方法第一步就是根据 hash 找到键所在的桶位置，但如果键没有覆写 hashCode 方法，计算 hash 时最终调用 Object 中的 hashCode 方法。但 Object 中的 hashCode 方法是 native 型的，不同的 JVM 下，可能会有不同的实现，产生的 hash 可能也是不一样的。也就是说同一个键在不同平台下可能会产生不同的 hash，此时再对在同一个 table 继续操作，就会出现问题。

综上所述，大家应该能明白 HashMap 不序列化 table 的原因了。

### 3.7 总结

本章对 HashMap 常见操作相关代码进行了详细分析，并在最后补充了一些其他细节。在本章中，插入操作一节的内容说的最多，主要是因为插入操作涉及的点特别多，一环扣一环。包含但不限于“table 初始化、扩容、树化”等，总体来说，插入操作分析起来难度还是很大的。好在，最后分析完了。

本章篇幅虽比较大，但仍未把 HashMap 所有的点都分析到。比如，红黑树的增删查等操作。当然，我个人看来，以上的分析已经够了。毕竟大家是类库的使用者而不是设计者，没必要去弄懂每个细节。所以如果某些细节实在看不懂的话就跳过吧，对我们开发来说，知道 HashMap 大致原理即可。

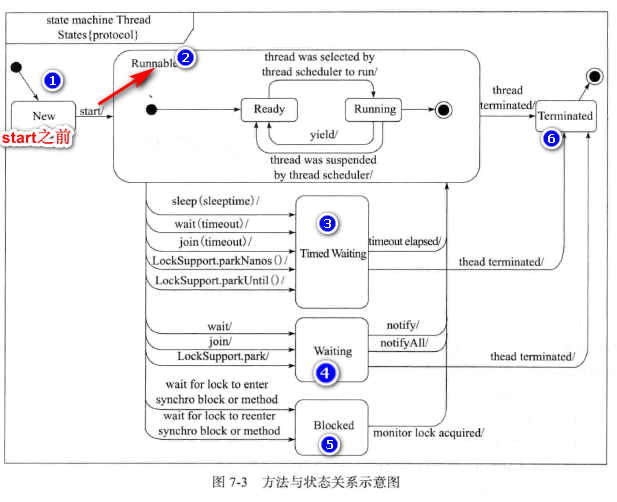
好了，本章到此结束。

## 4.写在最后

写到这里终于可以松一口气了，这篇文章前前后后花了我一周多的时间。在我写这篇文章之前，对 HashMap 认识仅限于原理层面，并未深入了解。一开始，我觉得关于 HashMap 没什么好写的，毕竟大家对 HashMap 多少都有一定的了解。但等我深入阅读 HashMap 源码后，发现之前的认知是错的。不是没什么可写的，而是可写的点太多了，不知道怎么写了。JDK 1.8 版本的 HashMap 实现上比之前版本要复杂的多，想弄懂众多的细节难度还是不小的。仅自己弄懂还不够，还要写出来，难度就更大了，本篇文章基本上是在边读源码边写的状态下完成的。由于时间和能力有限，加之文章篇幅比较大，很难保证不出错分析过程及配图不出错。如果有错误，希望大家指出来，我会及时修改，这里先谢谢大家。

# JAVA多线程

## 线程状态



线程有如下6种状态，由java.lang.State枚举给出

### NEW

创建后尚未启动的线程处于这个状态。意思是这个线程没有被start()启动，或者说还根本不是一个真正意义上的线程，从本质上讲这只是创建了一个Java外壳，还没有真正的线程来运行。不代表调用了start()，状态就立即改变，中间还有一些步骤，如果在这个启动的过程中有另一个线程来获取它的状态，其实是不确定的，要看那些中间步骤是否已经完成了。

### ****RUNNABLE（可运行）****

RUNNABLE状态包括了操作系统线程状态中的Running和Ready，也就是处于此状态的线程可能正在运行，也可能正在等待系统资源，如等待CPU为它分配时间片，如等待网络IO读取数据。

RUNNABLE状态也可以理解为存活着正在尝试征用CPU的线程（有可能这个瞬间并没有占用CPU，但是它可能正在发送指令等待系统调度）。由于在真正的系统中，并不是开启一个线程后，CPU就只为这一个线程服务，它必须使用许多调度算法来达到某种平衡，不过这个时候线程依然处于RUNNABLE状态。

### ****BLOCKED（阻塞）****

BLOCKED称为阻塞状态，或者说线程已经被挂起，它“睡着”了，原因通常是它在等待一个“锁”，当尝试进入一个synchronized语句块/方法时，锁已经被其它线程占有，就会被阻塞，直到另一个线程走完临界区或发生了相应锁对象的wait()操作后，它才有机会去争夺进入临界区的权利

在Java代码中，需要考虑synchronized的粒度问题，否则一个线程长时间占用锁，其它争抢锁的线程会一直阻塞，直到拥有锁的线程释放锁

处于BLOCKED状态的线程，即使对其调用 thread.interrupt()也无法改变其阻塞状态，因为interrupt()方法只是设置线程的中断状态，即做一个标记，不能唤醒处于阻塞状态的线程

**注意：**ReentrantLock.lock()操作后进入的是WAITING状态，其内部调用的是LockSupport.park()方法

### ****WAITING**（**无限期等待）****

处于这种状态的线程不会被分配CPU执行时间，它们要等待显示的被其它线程唤醒。这种状态通常是指一个线程拥有对象锁后进入到相应的代码区域后，调用相应的“锁对象”的wait()方法操作后产生的一种结果。变相的实现还有LockSupport.park()、Thread.join()等，它们也是在等待另一个事件的发生，也就是描述了等待的意思。

以下方法会让线程陷入无限期等待状态：

（1）没有设置timeout参数的Object.wait()

（2）没有设置timeout参数的Thread.join()

（3）LockSupport.park()

**注意：**

LockSupport.park(Object blocker) 会挂起当前线程，参数blocker是用于设置当前线程的“volatile Object parkBlocker 成员变量”

parkBlocker 是用于记录线程是被谁阻塞的，可以通过LockSupport.getBlocker()获取到阻塞的对象，用于监控和分析线程用的。

**“阻塞”与“等待”的区别：**

（1）“阻塞”状态是等待着获取到一个排他锁，进入“阻塞”状态都是被动的，离开“阻塞”状态是因为其它线程释放了锁，不阻塞了；

（2）“等待”状态是在等待一段时间 或者 唤醒动作的发生，进入“等待”状态是主动的

如主动调用Object.wait()，如无法获取到ReentraantLock，主动调用LockSupport.park()，如主线程主动调用 subThread.join()，让主线程等待子线程执行完毕再执行

离开“等待”状态是因为其它线程发生了唤醒动作或者到达了等待时间

### ****TIMED\_WAITING（限期等待）****

处于这种状态的线程也不会被分配CPU执行时间，不过无需等待被其它线程显示的唤醒，在一定时间之后它们会由系统自动的唤醒。

以下方法会让线程进入TIMED\_WAITING限期等待状态：

（1）Thread.sleep()方法

（2）设置了timeout参数的Object.wait()方法

（3）设置了timeout参数的Thread.join()方法

（4）LockSupport.parkNanos()方法

（5）LockSupport.parkUntil()方法

### ****TERMINATED（结束）****

已终止线程的线程状态，线程已经结束执行。换句话说，run()方法走完了，线程就处于这种状态。其实这只是Java语言级别的一种状态，在操作系统内部可能已经注销了相应的线程，或者将它复用给其他需要使用线程的请求，而在Java语言级别只是通过Java代码看到的线程状态而已。

## VisualVM状态

通过VisualVM监控JVM时，可以通过“线程”标签页查看JVM的线程信息，VisualVM的线程状态如下：

[https://images2015.cnblogs.com/blog/677054/201703/677054-20170325171859830-198146769.jpg](http://images2015.cnblogs.com/blog/677054/201703/677054-20170325171859096-1292818011.jpg)

通过dump thread stack，并与VisualVM监控信息中的线程名称对应，找到的VisualVM每种线程状态的线程堆栈如下：（请关注重点信息）

### **运行**

|  |
| --- |
| "http-bio-8080-Acceptor-0" daemon prio=6 tid=0x000000000d7b4800 nid=0xa264 runnable [0x000000001197e000]       java.lang.Thread.State: **RUNNABLE**             at java.net.DualStackPlainSocketImpl.accept0(Native Method) |

### **休眠**

|  |
| --- |
| "Druid-ConnectionPool-Destory-293325558" daemon prio=6 tid=0x000000000d7ad000 nid=0x9c94 **waiting on condition** [0x000000000bf0f000]       java.lang.Thread.State: **TIMED\_WAITING (sleeping)** |

### **等待**

|  |
| --- |
| "Finalizer" daemon prio=8 tid=0x0000000009349000 nid=0xa470 in **Object.wait()** [0x000000000a82f000]       java.lang.Thread.State: **WAITING (on object monitor)**             at java.lang.Object.wait(Native Method) |

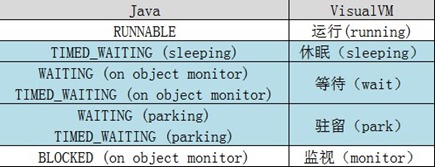
### **驻留**

|  |
| --- |
| "http-bio-8080-exec-2" daemon prio=6 tid=0x000000000d7b8000 nid=0x9264 **waiting on condition** [0x000000000ee4e000]       java.lang.Thread.State:**WAITING (parking)**             at sun.misc.Unsafe.park(Native Method) |

### **监视**

|  |
| --- |
| "Thread-1" prio=6 tid=0x000000000a8a1800 nid=0xfdb4 **waiting for monitor entry** [0x000000000b4de000]       java.lang.Thread.State: **BLOCKED (on object monitor)** |

**“VisualVM线程监控线程状态”与“Java线程状态”对应关系总结：**

[](http://images2015.cnblogs.com/blog/677054/201703/677054-20170325175601080-727037896.jpg)

可以看出，VisualVM的线程状态将“WAITING”和“TIMED\_WAITING”两个状态根据行程状态的原因做了细化（其实java的thread stack dump上已经细化了）

如造成“TIMED\_WAITING”状态的原因可能是 ：

Thread.sleep()  --  休眠

Object.wait(timeout)  --  等待

LockSupport.parkUntil(deadline)  --  驻留

## 同步技术

信号量、互斥量、临界区、事件

CPU支持如下操作，虽然从语义上讲是多步，但是CPU可以在一个指令上完成这些操作，所以这些操作是原子的。

1. 测试并设置（Test and Set）

检测互斥时使用，检测为 1 返回接着测试、测试为零设置为1，进入临界区

1. 获取并增加 (Fetch and Increment)

原子加减吧？

1. 交换 (Swap)
2. 比较并交换 (Compare and Swap)

cas操作对ABA问题无解

1. 加载连接/条件存储 (LL/SC)

### [CAS](https://www.cnblogs.com/xrq730/p/4976007.html)技术

上面两种都是阻塞技术，CAS非阻塞的技术

Unsafe简单讲一下这个类。Java无法直接访问底层操作系统，而是通过本地（native）方法来访问。不过尽管如此，JVM还是开了一个后门，JDK中有一个类Unsafe，它提供了硬件级别的原子操作。

这个类尽管里面的方法都是public的，但是并没有办法使用它们，JDK API文档也没有提供任何关于这个类的方法的解释。总而言之，对于Unsafe类的使用都是受限制的，只有授信的代码才能获得该类的实例，当然JDK库里面的类是可以随意使用的。

从第一行的描述可以了解到Unsafe提供了硬件级别的操作，比如说获取某个属性在内存中的位置，比如说修改对象的字段值，即使它是私有的。不过Java本身就是为了屏蔽底层的差异，对于一般的开发而言也很少会有这样的需求。

举两个例子，比方说：

public native long staticFieldOffset(Field paramField);

这个方法可以用来获取给定的paramField的内存地址偏移量，这个值对于给定的field是唯一的且是固定不变的。再比如说：

Public native int arrayBaseOffset(Class paramClass);

public native int arrayIndexScale(Class paramClass);

前一个方法是用来获取数组第一个元素的偏移地址，后一个方法是用来获取数组的转换因子即数组中元素的增量地址的。最后看三个方法：

public native long allocateMemory(long paramLong);

public native long reallocateMemory(long paramLong1, long paramLong2);

public native void freeMemory(long paramLong);

分别用来分配内存，扩充内存和释放内存的。

CAS，Compare and Swap即比较并交换，设计并发算法时常用到的一种技术，java.util.concurrent包全完建立在CAS之上，没有CAS也就没有此包，可见CAS的重要性。

当前的处理器基本都支持CAS，只不过不同的厂家的实现不一样罢了。CAS有三个操作数：内存值V、旧的预期值A、要修改的值B，当且仅当预期值A和内存值V相同时，将内存值修改为B并返回true，否则什么都不做并返回false。这是从硬件上保证的。

**CAS软件层面的 Unsafe实现的如下**

public final native boolean compareAndSwapObject(Object paramObject1, long paramLong, Object paramObject2, Object paramObject3);

public final native boolean compareAndSwapInt(Object paramObject, long paramLong, int paramInt1, int paramInt2);

public final native boolean compareAndSwapLong(Object paramObject, long paramLong1, long paramLong2, long paramLong3);

java.util.concurrent.atomic包下的原子操作类都是基于CAS实现的，下面拿AtomicInteger分析一下，首先是AtomicInteger类变量的定义：

|  |
| --- |
| 1 private static final Unsafe unsafe = Unsafe.getUnsafe();  2 private static final long valueOffset;  3  4 static {  5 try {  6 valueOffset = unsafe.objectFieldOffset  7 (AtomicInteger.class.getDeclaredField("value"));  8 } catch (Exception ex) { throw new Error(ex); }  9 }  10  11 private volatile int value |

关于这段代码中出现的几个成员属性：

1、Unsafe是CAS的核心类，前面已经讲过了

2、valueOffset表示的是变量值在内存中的偏移地址，因为Unsafe就是根据内存偏移地址获取数据的原值的

3、value是用volatile修饰的，这是非常关键的

下面找一个方法getAndIncrement来研究一下AtomicInteger是如何实现的，比如我们常用的addAndGet方法：

|  |
| --- |
| 1 public final int addAndGet(int delta) {  2 for (;;) {  3 int current = get();  4 int next = current + delta;  5 if (compareAndSet(current, next))  6 return next;  7 }  8 }  1 public final int get() {  2 return value;  3 } |

这段代码如何在不加锁的情况下通过CAS实现线程安全，我们不妨考虑一下方法的执行：

1、AtomicInteger里面的value原始值为3，即主内存中AtomicInteger的value为3，根据Java内存模型，线程1和线程2各自持有一份value的副本，值为3

2、线程1运行到第三行获取到当前的value为3，线程切换

3、线程2开始运行，获取到value为3，利用CAS对比内存中的值也为3，比较成功，修改内存，此时内存中的value改变比方说是4，线程切换

4、线程1恢复运行，利用CAS比较发现自己的value为3，内存中的value为4，得到一个重要的结论-->此时value正在被另外一个线程修改，所以我不能去修改它

5、线程1的compareAndSet失败，循环判断，因为value是volatile修饰的，所以它具备可见性的特性，线程2对于value的改变能被线程1看到，只要线程1发现当前获取的value是4，内存中的value也是4，说明线程2对于value的修改已经完毕并且线程1可以尝试去修改它

6、最后说一点，比如说此时线程3也准备修改value了，没关系，因为比较-交换是一个原子操作不可被打断，线程3修改了value，线程1进行compareAndSet的时候必然返回的false，这样线程1会继续循环去获取最新的value并进行compareAndSet，直至获取的value和内存中的value一致为止

整个过程中，利用CAS机制保证了对于value的修改的线程安全性。

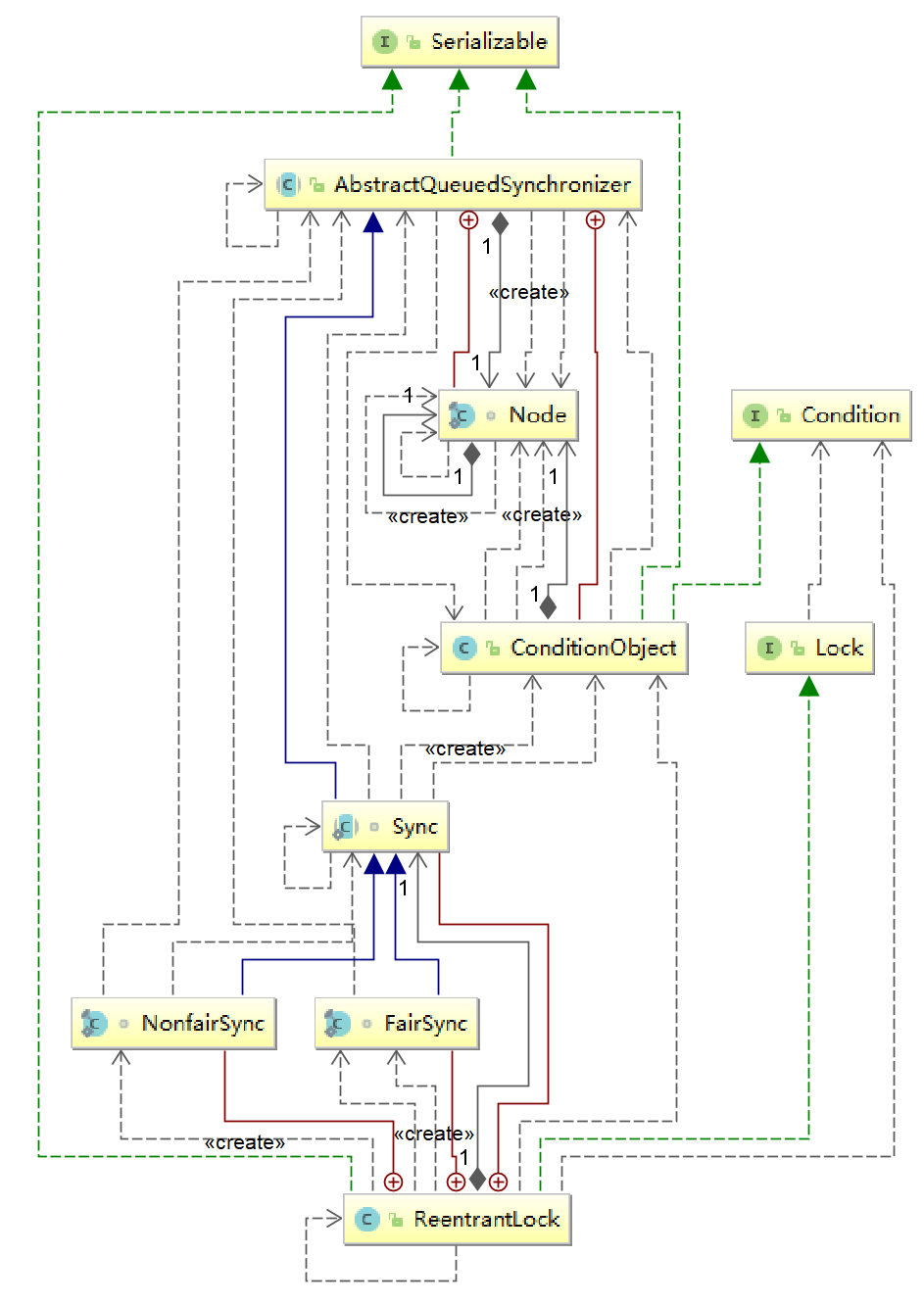
**CAS的缺点**

CAS看起来很美，但这种操作显然无法涵盖并发下的所有场景，并且CAS从语义上来说也不是完美的，存在这样一个逻辑漏洞：如果一个变量V初次读取的时候是A值，并且在准备赋值的时候检查到它仍然是A值，那我们就能说明它的值没有被其他线程修改过了吗？如果在这段期间它的值曾经被改成了B，然后又改回A，那CAS操作就会误认为它从来没有被修改过。这个漏洞称为CAS操作的"ABA"问题。java.util.concurrent包为了解决这个问题，提供了一个带有标记的原子引用类"AtomicStampedReference"，它可以通过控制变量值的版本来保证CAS的正确性。不过目前来说这个类比较"鸡肋"，大部分情况下ABA问题并不会影响程序并发的正确性，如果需要解决ABA问题，使用传统的互斥同步可能回避原子类更加高效。

### ReentrentLock实现细节

|  |
| --- |
| AQS是关键类。Condition接口，给了等待通知等功能，Lock接口给了锁定解锁功能。AbstractOwnableSynchronizer 记录了排它线程，只有这一个作用，它是AQS的父类。ConditionObjec类为AQS提供了 condition条件下的 wait queue 等待队列。AQS自己维护了一个同步队列。注意AQS的同步队列也是需要等待的，condition条件下的等待队列，只有Condition能唤醒，这俩队列是协同工作的。AQS定义两种资源共享方式：Exclusive（独占，只有一个线程能执行，如ReentrantLock）和Share（共享，多个线程可同时执行，如Semaphore/CountDownLatch）。  不同的自定义同步器争用共享资源的方式也不同。自定义同步器在实现时只需要实现共享资源state的获取与释放方式即可，至于具体线程等待队列的维护（如获取资源失败入队/唤醒出队等），AQS已经在顶层实现好了。  自定义同步器实现时主要实现以下几种方法：  isHeldExclusively()：该线程是否正在独占资源。只有用到condition才需要去实现它。  tryAcquire(int)：独占方式。尝试获取资源，成功则返回true，失败则返回false。  tryRelease(int)：独占方式。尝试释放资源，成功则返回true，失败则返回false。  tryAcquireShared(int)：共享方式。尝试获取资源。负数表示失败；0表示成功，但没有剩余可用资源；正数表示成功，且有剩余资源。  tryReleaseShared(int)：共享方式。尝试释放资源，如果释放后允许唤醒后续等待结点返回true，否则返回false。  　　以ReentrantLock为例，state初始化为0，表示未锁定状态。A线程lock()时，会调用tryAcquire()独占该锁并将state+1。此后，其他线程再tryAcquire()时就会失败，直到A线程unlock()到state=0（即释放锁）为止，其它线程才有机会获取该锁。当然，释放锁之前，A线程自己是可以重复获取此锁的（state会累加），这就是可重入的概念。但要注意，获取多少次就要释放多么次，这样才能保证state是能回到零态的。  　　再以CountDownLatch以例，任务分为N个子线程去执行，state也初始化为N（注意N要与线程个数一致）。这N个子线程是并行执行的，每个子线程执行完后countDown()一次，state会CAS减1。等到所有子线程都执行完后(即state=0)，会unpark()主调用线程，然后主调用线程就会从await()函数返回，继续后余动作。  　　一般来说，自定义同步器要么是独占方法，要么是共享方式，他们也只需实现tryAcquire、tryRelease、tryAcquireShared、tryReleaseShared中的一种即可。但AQS也支持自定义同步器同时实现独占和共享两种方式，如ReentrantReadWriteLock。 |

#### 类图



#### 依赖类型接口

Serializable类比较简单，分析Node和Condition。

Node的作用是对AQS中线程的封装类，它指明了当前线程在队列中的状态。

static final class Node {

static final Node SHARED = new Node();

static final Node EXCLUSIVE = null;

static final int CANCELLED = 1;

static final int SIGNAL = -1;

static final int CONDITION = -2;

static final int PROPAGATE = -3;

volatile int waitStatus;

volatile Thread thread;

}

waitStatus：

1、CANCELLED，值为1 。场景：当该线程等待超时或者被中断，需要从同步队列中取消等待，则该线程被置1，即被取消（这里该线程在取消之前是等待状态）。节点进入了取消状态则不再变化；

2、SIGNAL，值为-1。场景：后继的节点处于等待状态，当前节点的线程如果释放了同步状态或者被取消（当前节点状态置为-1），将会通知后继节点，使后继节点的线程得以运行；

3、CONDITION，值为-2。场景：节点处于等待队列中，节点线程等待在Condition上，当其他线程对Condition调用了signal()方法后，该节点从等待队列中转移到同步队列中，加入到对同步状态的获取中；

4、PROPAGATE，值为-3。场景：表示下一次的共享状态会被无条件的传播下去；

5、INITIAL，值为0，初始状态。

Node prev：前驱节点，当节点加入同步队列的时候被设置（尾部添加）

Node next：后继节点

Node nextWaiter：等待节点的后继节点。如果当前节点是共享的，那么这个字段是一个SHARED常量，也就是说节点类型（独占和共享）和等待队列中的后继节点共用一个字段。（注：比如说当前节点A是共享的，那么它的这个字段是shared，也就是说在这个等待队列中，A节点的后继节点也是shared。如果A节点不是共享的，那么它的nextWaiter就不是一个SHARED常量，即是独占的。）

Thread thread：获取同步状态的线程

#### 源码分析

ReentrantLock默认是非公平的，从构造函数可以看出来

|  |
| --- |
| public ReentrantLock(boolean fair) {  sync = fair ? new FairSync() : new NonfairSync(); } |

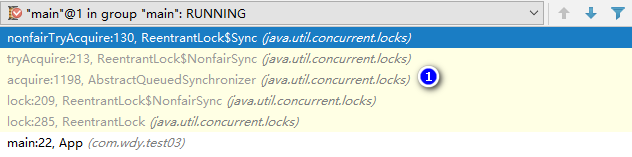
share为0表示锁没被占用，独占模式每冲重入一次，share值加 1 。Semaphore 中share为信号量。读写锁中，share 初始为65536，每次重入左移一位。

##### 非公平Lock() 算法：

1. 第一次lock() 直接尝试设置share = 1 成功立即返回。不成功调用acquire(1) 尝试获取锁，参数1 是share要被设置的值，acquire(1)有如下两个判断条件

if (!tryAcquire(arg) && acquireQueued(addWaiter(Node.EXCLUSIVE), arg))

第一个判断条件调用链如下：acquire()->tryAcquire()->nonfairTryAcquire()

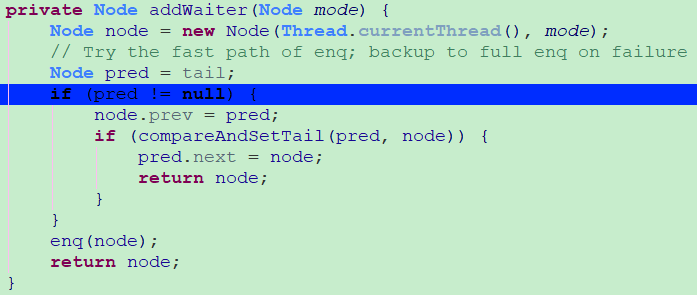


nonfairTryAcquire()方法，该方法的逻辑为：

判断share == 0是否成立，成立直接改share = 1返回。之所以要判断，其它线程释放了锁，这次有可能成功。

share != 0 ,判断当前占用锁的线程是不是自己，如果是自己，share值加 1 返回。一个线程多次重入，条件 1 都是成立的，第二个判断条件就不需要执行了。如果有多个线程，第二个条件才需要执行。下面看第二个条件

acquireQueued(addWaiter(Node.EXCLUSIVE), arg)



addWaiter 先测试tail是不是空的，如果不空，说明队列不空，直接加到队列尾部，如果空的，在eqn()函数中处理。enq() 函数先造一个空节点作为CHL队列头，然后把node加到空节点的后面，作为队尾，返回值是封装了当前线程的节点。下面分析

acquireQueued(**final Node** *node*, **int** *arg*)

这个函数的目的是挂起线程或者尝试获取锁

第一个if 逻辑简单。分析 shouldParkAfterFailedAcquire(Node pred, Node node)，这个函数入参包括 pred 和当前 node。通过前置节点状态，可以知道如何处理当前节点，如果当前节点是空节点的后继，由于空节点的 waitStatus = 0 ，会先循环一次将前置节点的waitStatus 设置为 SIGNAL = -1，表示当前节点要挂起。在收到前置节点的通知后再运行。这个挂起功能是LockSupport 类提供的，下面分析这个类。

**LockSupport是线程阻塞的关键：**

LockSupport类是Java6(JSR166-JUC)引入的一个类，提供了基本的线程同步原语。LockSupport实际上是调用了Unsafe类的两个函数：

park：阻塞当前线程。

unpark: 使给定的线程停止阻塞(Unblock the given thread blocked )。

public native void unpark(Thread jthread);

public native void park(boolean isAbsolute, long time);

isAbsolute参数是指明时间是绝对的，还是相对的。仅仅两个简单的接口，就为上层提供了强大的同步原语。先来解析下两个函数是做什么的。

unpark函数为线程提供“许可(permit)”，线程调用park函数则等待“许可”。这个有点像信号量，但是这个“许可”是不能叠加的，“许可”是一次性的。比如线程B连续调用了三次unpark函数，当线程A调用park函数就使用掉这个“许可”，如果线程A再次调用park，则进入等待状态。注意，unpark函数可以先于park调用。比如线程B调用unpark函数，给线程A发了一个“许可”，那么当线程A调用park时，它发现已经有“许可”了，那么它会马上再继续运行。实际上，park函数即使没有“许可”，有时也会无理由地返回，这点等下再解析。

**park和unpark的灵活之处**

上面已经提到，unpark函数可以先于park调用，这个正是它们的灵活之处。一个线程它有可能在别的线程unPark之前，或者之后，或者同时调用了park，那么因为park的特性，它可以不用担心自己的park的时序问题，否则，如果park必须要在unpark之前，那么给编程带来很大的麻烦！考虑一下，两个线程同步，要如何处理？

在Java5里是用wait/notify/notifyAll来同步的。wait/notify机制有个很蛋疼的地方是，比如线程B要用notify通知线程A，那么线程B要确保线程A已经在wait调用上等待了，否则线程A可能永远都在等待。编程的时候就会很蛋疼。另外，是调用notify，还是notifyAll？notify只会唤醒一个线程，如果错误地有两个线程在同一个对象上wait等待，那么又悲剧了。为了安全起见，貌似只能调用notifyAll了。

park/unpark模型真正解耦了线程之间的同步，线程之间不再需要一个Object或者其它变量来存储状态，不再需要关心对方的状态。

**HotSpot里park/unpark的实现**

每个java线程都有一个Parker实例，Parker类是这样定义的：

|  |
| --- |
| class Parker : public os::PlatformParker {  private:  volatile int \_counter ; //计数  Parker \* FreeNext ; //指向下一个Parker  JavaThread \* AssociatedWith ; // 指向parker所属的线程。    public:  Parker() : PlatformParker() {  \_counter = 0 ; //初始化为0  FreeNext = NULL ;  AssociatedWith = NULL ;  }  protected:  ~Parker() { ShouldNotReachHere(); }  public:  // For simplicity of interface with Java, all forms of park (indefinite,  // relative, and absolute) are multiplexed into one call.  void park(bool isAbsolute, jlong time);  void unpark();    // Lifecycle operators  static Parker \* Allocate (JavaThread \* t) ;  static void Release (Parker \* e) ;  private:  static Parker \* volatile FreeList ;  static volatile int ListLock ;    }; |

Parker继承了PlatformParker。实现如下：

|  |
| --- |
| class PlatformParker : public CHeapObj<mtInternal> {  protected:  enum {  REL\_INDEX = 0,  ABS\_INDEX = 1  };  int \_cur\_index; // 条件变量数组下标，which cond is in use: -1, 0, 1  pthread\_mutex\_t \_mutex [1] ; //pthread互斥锁  pthread\_cond\_t \_cond [2] ; // pthread条件变量数组,一个用于相对时间，一个用于绝对时间。    public: // TODO-FIXME: make dtor private  ~PlatformParker() { guarantee (0, "invariant") ; }    public:  PlatformParker() {  int status;  status = pthread\_cond\_init (&\_cond[REL\_INDEX], os::Linux::condAttr());  assert\_status(status == 0, status, "cond\_init rel");  status = pthread\_cond\_init (&\_cond[ABS\_INDEX], NULL);  assert\_status(status == 0, status, "cond\_init abs");  status = pthread\_mutex\_init (\_mutex, NULL);  assert\_status(status == 0, status, "mutex\_init");  \_cur\_index = -1; // mark as unused  }  }; |

PlatformParker主要看三个成员变量，\_cur\_index, \_mutex, \_cond。其中mutex和cond就是很熟悉的glibc nptl包中符合posix标准的线程同步工具，一个互斥锁一个条件变量。再看thread和Parker的关系，在hotspot的Thread类的NameThread内部类中有一个 Parker成员变量。在创建线程的时候就会生成一个parker实例。

|  |
| --- |
| // JSR166 per-thread parker  private:    Parker\*    \_parker; |

再看park() 函数的实现

|  |
| --- |
| void Parker::park(bool isAbsolute, jlong time) {    //原子交换，如果\_counter > 0,则将\_counter置为0，直接返回，否则\_counter为0  if (Atomic::xchg(0, &\_counter) > 0) return;  //获取当前线程  Thread\* thread = Thread::current();  assert(thread->is\_Java\_thread(), "Must be JavaThread");  //下转型为java线程  JavaThread \*jt = (JavaThread \*)thread;    //如果当前线程设置了中断标志，调用park则直接返回，所以如果在park之前调用了  //interrupt就会直接返回，这也正说明了为什么我们在wait的时候，中断线程会抛出异//常  if (Thread::is\_interrupted(thread, false)) {  return;  }    // 高精度绝对时间变量  timespec absTime;  //如果time小于0，或者isAbsolute是true并且time等于0则直接返回。时间为 0 不用等，直接回去  if (time < 0 || (isAbsolute && time == 0) ) { // don't wait at all  return;  }  //如果time大于0，则根据是否是高精度定时计算定时时间  if (time > 0) {  unpackTime(&absTime, isAbsolute, time);  }      //进入安全点避免死锁  ThreadBlockInVM tbivm(jt);      //如果当前线程设置了中断标志，或者获取mutex互斥锁失败则直接返回  //由于Parker是每个线程都有的，所以\_counter cond mutex都是每个线程都有的，  //mutex不是所有线程共享的所以加锁失败只有两种情况，第一unpark已经加锁这时只需要返回即可，第二调用pthread\_mutex\_trylock出错。对于第一种情况就类似是unpark先调用的情况，所以直接返回。  if (Thread::is\_interrupted(thread, false) || pthread\_mutex\_trylock(\_mutex) != 0) {  return;  }    int status ;  //如果\_counter大于0，说明unpark已经调用完成了将\_counter置为了1，  //现在只需将\_counter置0，解锁，返回  if (\_counter > 0) { // no wait needed  \_counter = 0;  status = pthread\_mutex\_unlock(\_mutex);  assert (status == 0, "invariant");  OrderAccess::fence();  return;  }      OSThreadWaitState osts(thread->osthread(), false /\* not Object.wait() \*/);  jt->set\_suspend\_equivalent();  // cleared by handle\_special\_suspend\_equivalent\_condition() or java\_suspend\_self()    assert(\_cur\_index == -1, "invariant");  //如果time等于0，说明是相对时间也就是isAbsolute是fasle(否则前面就直接返回了),则直接挂起  if (time == 0) {  \_cur\_index = REL\_INDEX; // arbitrary choice when not timed  status = pthread\_cond\_wait (&\_cond[\_cur\_index], \_mutex) ;  } else { //如果time非0  //判断isAbsolute是false还是true，false的话使用\_cond[0]，否则用\_cond[1]  \_cur\_index = isAbsolute ? ABS\_INDEX : REL\_INDEX;  //使用条件变量使得当前线程挂起。  status = os::Linux::safe\_cond\_timedwait (&\_cond[\_cur\_index], \_mutex, &absTime) ;  //如果挂起失败则销毁当前的条件变量重新初始化。  if (status != 0 && WorkAroundNPTLTimedWaitHang) {  pthread\_cond\_destroy (&\_cond[\_cur\_index]) ;  pthread\_cond\_init (&\_cond[\_cur\_index], isAbsolute ? NULL : os::Linux::condAttr());  }  }    //如果pthread\_cond\_wait成功则以下代码都是线程被唤醒后执行的。  \_cur\_index = -1;  assert\_status(status == 0 || status == EINTR ||  status == ETIME || status == ETIMEDOUT,  status, "cond\_timedwait");    #ifdef ASSERT  pthread\_sigmask(SIG\_SETMASK, &oldsigs, NULL);  #endif  //将\_counter变量重新置为1  \_counter = 0 ;  //解锁  status = pthread\_mutex\_unlock(\_mutex) ;  assert\_status(status == 0, status, "invariant") ;  // 使用内存屏障使\_counter对其它线程可见  OrderAccess::fence();    // 如果在park线程挂起的时候调用了stop或者suspend则还需要将线程挂起不能返回  if (jt->handle\_special\_suspend\_equivalent\_condition()) {  jt->java\_suspend\_self();  }  } |

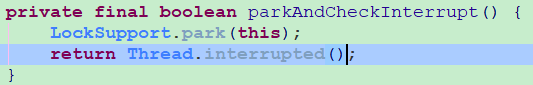
再看unpark函数

|  |
| --- |
| void Parker::unpark() {  int s, status ;  //加互斥锁  status = pthread\_mutex\_lock(\_mutex);  assert (status == 0, "invariant") ;  s = \_counter;  \_counter = 1; //将\_counter置1  //如果\_counter是0则说明调用了park或者没调用(初始为counter0）  //这也说明park和unpark调用没有先后顺序。  if (s < 1) {  // 说明当前parker对应的线程挂起了，因为\_cur\_index初始是-1，并且等待条件变量的线程被唤醒  //后也会将\_cur\_index重置-1  if (\_cur\_index != -1) {  //如果设置了WorkAroundNPTLTimedWaitHang先调用signal再调用unlock，否则相反  //这两个先后顺序都可以，在hotspot在Linux下默认使用这种方式  //即先调用signal再调用unlock  if (WorkAroundNPTLTimedWaitHang) {  status = pthread\_cond\_signal (&\_cond[\_cur\_index]);  assert (status == 0, "invariant");  status = pthread\_mutex\_unlock(\_mutex);  assert (status == 0, "invariant");  } else {  status = pthread\_mutex\_unlock(\_mutex);  assert (status == 0, "invariant");  status = pthread\_cond\_signal (&\_cond[\_cur\_index]);  assert (status == 0, "invariant");  }  } else { //如果\_cur\_index == -1说明线程没在等待条件变量，则直接解锁  pthread\_mutex\_unlock(\_mutex);  assert (status == 0, "invariant") ;  }  } else {//如果\_counter == 1,说明线程调用了一次或多次unpark但是没调用park，则直接解锁  pthread\_mutex\_unlock(\_mutex);  assert (status == 0, "invariant") ;  } |

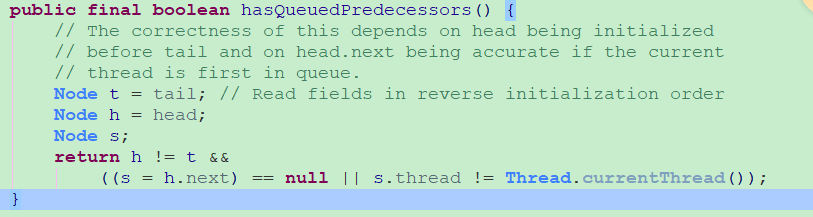
unpark主要是根据counter和cur\_index判断当前线程是否挂在条件变量上，如果是则signal，否则就什么也不做。所以park和unpark和核心就是counter cur\_index, mutex,cond，通过使用条件变量对counter进行操作，在调用park的时候如果counter是0则会去执行挂起的流程，否则返回，在挂起恢复后再将counter置为0。在unpark的时候如果counter是0则会执行唤醒的流程，否则不执行唤醒流程，并且不管什么情况始终将counter置为1。

上面分析完 park() 和 unpark() 函数的实现，我们知道 Thread 是如何被阻塞的了，

if (shouldParkAfterFailedAcquire(p, node) &&parkAndCheckInterrupt()) 函数 shouldParkAfterFailedAcquire(p, node) 给出了当前线程是否应该被挂起， parkAndCheckInterrupt()，函数是用来挂起线程的，如果第一个条件满足，那么第二个条件使用LockSupport挂起线程，在线程被重新启动后立刻检测是否被中断过。



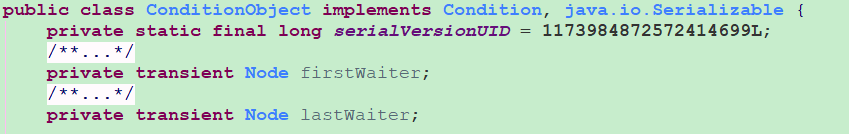
##### 公平Lock算法



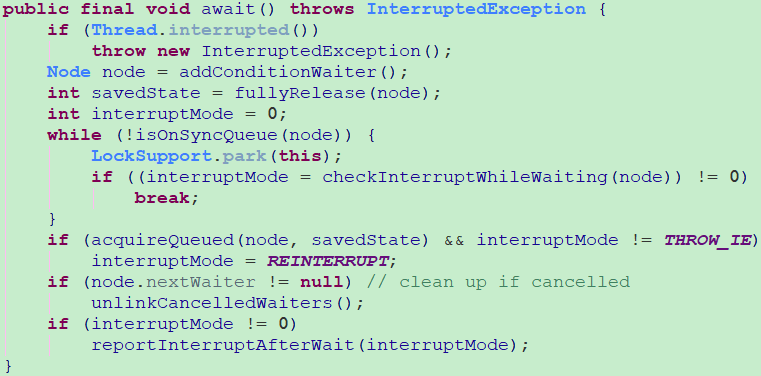
公平Lock() 算法，不再直接尝试获取锁，在尝试之前先判断有没有前驱节点。如果当前节点有驱节点，只有一种情况，就是当前节点在头节点后面。那么 t != h，并且 h.next 不空时，一定不是当前节点。这是公平锁主要的不同点。

##### Condition队列

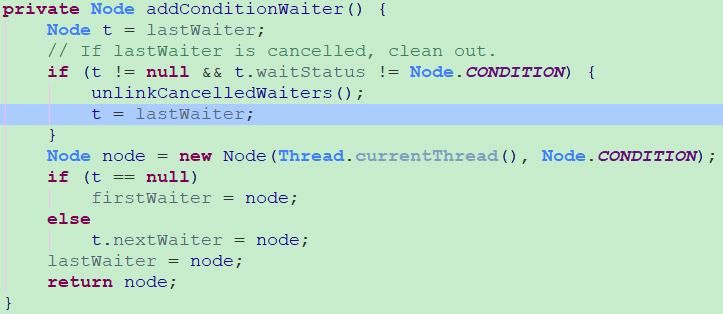
ConditionObject实现了一个队列，我们称这个队列为等待队列，每个Condition对象都有这样一个队列



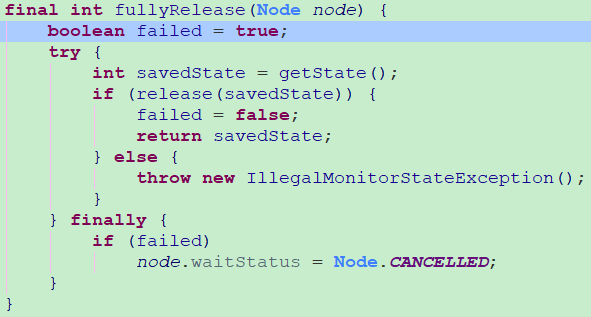
线程 1 调用 condition1.await() 方法即可将当前线程 1 包装成 Node 加入到当前Condition条件队列中，然后阻塞在这里，不继续往下执行，条件队列是一个单向链表具体实现如下



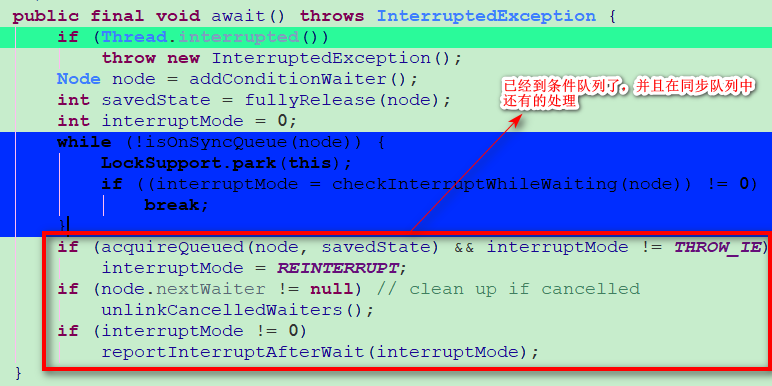
第一步addConditionWaiter就是要将线程加入到Condition的条件队列末尾，并且去掉不处于Node.CONDITION状态的节点（线程），返回值为当前线程的node，当前线程是没有next的。



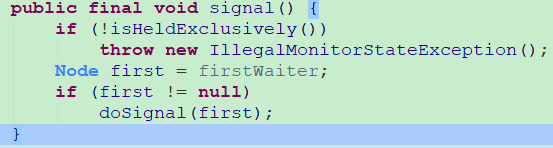
线程加入Condition条件队列后 fullyRelease() 释放锁, 成功后尝试唤醒下一个节点 (线程) ，这个非常重要的，await() 之前时获取到锁了的，如果不释放，其它线程都会阻塞。



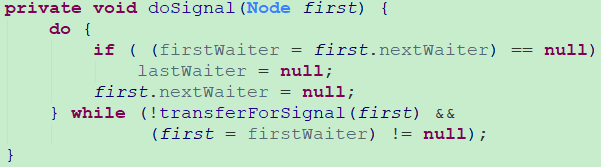
释放锁后，若不在同步队列，挂起该线程，这里要注意，被挂起的这个线程已经释放了锁，却能被挂起，说明拥有锁不是挂起线程的必要条件，像suspend 、sleep 等并不需要获取锁。Condition 却实现为必须先拥有锁，才能await() 使线程安全考虑。线程不在同步队列，并且已经被加入到条件队列了，那确实应该挂起，如果已经加入条件队列的线程却还在同步队列中，那就是之前的fullyRelease() 函数出了问题，就要让当前线程尝试获取锁，来抛出异常，后面红框中的三个 if 就是干这事的。fullyRelease() 处理很简单，现在持有锁的一定是当前线程，只要将持有锁的线程置为 null 。看下面代码



下面看 condition 唤醒阻塞。第一步就要调用 isHeldExclusively() 判断是否持有锁。没持有锁，直接抛出异常。



如果持有锁，就要将该节点从条件队列转移到同步队列，doSignal() 函数完成这个功能。



首先调整条件队列的firstWaiter，将取出的第一个waiter 放入同步队列，transferForSignal(Node node) 完成这个工作，下面分析这个函数。

|  |
| --- |
| final boolean transferForSignal(Node node) {  //将该waiter的状态设置为初始值 0，失败直接返回  if (!compareAndSetWaitStatus(node, Node.CONDITION, 0))  return false;  //将该waiter 加入同步队列尾部  Node p = enq(node);  //加入队尾后，判断是否应该挂起这个watier，不需要挂起，则通知成功  int ws = p.waitStatus;  if (ws > 0 || !compareAndSetWaitStatus(p, ws, Node.SIGNAL))  LockSupport.unpark(node.thread);  return true;  } |

如果返回成功，则表示条件队列的第一个waiter接受了这个signal ，可以返回了，但是如果失败，还要条件队列中的第二个 waiter 。源码如下

|  |
| --- |
| !transferForSignal(first) && (first = firstWaiter) != null |

不成功后，让 first = 指向第二个waiter，接着通知。但是什么情况下会不能成功呢，我比较好奇。还有就是signal不会主动释放锁。

### Synchronized

Synchronized关键字使用了两种同步技术——互斥量和临界区这两种技术保证并发的安全性。

一个对象的Synchronized修饰方法，同一时刻只能由一个线程访问，这个是使用了互斥量。如果没有取得这个互斥量，是不能执行Synchronized方法的。这是临界区办不到的，临界区只能保证不对线程对同一段代码的串行访问，但是对多个线程，同时访问多个临界区是无法控制的。

代码块的同步是由 monitorenter 和 monitorexit 指令来实现的，而方法的同步是隐式，JVM可以从方法常量池中的方法表结构(method\_info Structure) 中的 ACC\_SYNCHRONIZED 访问标志区分一个方法是否同步方法。当方法调用时，调用指令将会检查方法的ACC\_SYNCHRONIZED 访问标志是否被设置，如果设置了，执行线程将先持有monitor（虚拟机规范中用的是管程一词）， 然后再执行方法，最后再方法完成(无论是正常完成还是非正常完成)时释放monitor。在方法执行期间，执行线程持有了monitor，其他任何线程都无法再获得同一个monitor。如果一个同步方法执行期间抛出了异常，并且在方法内部无法处理此异常，那这个同步方法所持有的monitor将在异常抛到方法之外时自动释放。下面我们看看Synchronized的字节码：

下面这段代码

|  |
| --- |
| public class SyncMethod {  public int i;  public synchronized void syncTask(){  i++;  }  } |

使用javap反编译后的字节码如下：

|  |
| --- |
| public synchronized void syncTask();  descriptor: ()V  //方法标识ACC\_PUBLIC代表public修饰，ACC\_SYNCHRONIZED指明该方法为同步方法  flags: ACC\_PUBLIC, ACC\_SYNCHRONIZED  Code:  stack=3, locals=1, args\_size=1 |

从字节码中可以看出，synchronized修饰的方法并没有monitorenter指令和monitorexit指令，取得代之的确实是ACC\_SYNCHRONIZED标识，该标识指明了该方法是一个同步方法，JVM通过该ACC\_SYNCHRONIZED访问标志来辨别一个方法是否声明为同步方法，从而执行相应的同步调用。这便是synchronized锁在同步代码块和同步方法上实现的基本原理。同时我们还必须注意到的是在Java早期版本中，synchronized属于重量级锁，效率低下，因为监视器（monitor）是依赖于底层的操作系统的Mutex Lock来实现的，而操作系统实现线程之间的切换时需要从用户态转换到核心态，这个状态之间的转换需要相对比较长的时间，时间成本相对较高，这也是为什么早期的synchronized效率低的原因。庆幸的是在Java 6之后Java官方对从JVM层面对synchronized较大优化，所以现在的synchronized锁效率也优化得很不错了，Java 6之后，为了减少获得锁和释放锁所带来的性能消耗，引入了轻量级锁和偏向锁，接下来我们将简单了解一下Java官方在JVM层面对synchronized锁的优化。

#### synchronized优化

锁的状态总共有四种，无锁状态、偏向锁、轻量级锁和重量级锁。随着锁的竞争，锁可以从偏向锁升级到轻量级锁，再升级的重量级锁，但是锁的升级是单向的，也就是说只能从低到高升级，不会出现锁的降级，关于重量级锁，前面我们已详细分析过，下面我们将介绍偏向锁和轻量级锁以及JVM的其他优化手段，这里并不打算深入到每个锁的实现和转换过程更多地是阐述Java虚拟机所提供的每个锁的核心优化思想，毕竟涉及到具体过程比较繁琐，如需了解详细过程可以查阅《深入理解Java虚拟机原理》。

偏向锁

偏向锁是Java 6之后加入的新锁，它是一种针对加锁操作的优化手段，经过研究发现，在大多数情况下，锁不仅不存在多线程竞争，而且总是由同一线程多次获得，因此为了减少同一线程获取锁(会涉及到一些CAS操作,耗时)的代价而引入偏向锁。偏向锁的核心思想是，如果一个线程获得了锁，那么锁就进入偏向模式，此时Mark Word 的结构也变为偏向锁结构，当这个线程再次请求锁时，无需再做任何同步操作，即获取锁的过程，这样就省去了大量有关锁申请的操作，从而也就提供程序的性能。所以，对于没有锁竞争的场合，偏向锁有很好的优化效果，毕竟极有可能连续多次是同一个线程申请相同的锁。但是对于锁竞争比较激烈的场合，偏向锁就失效了，因为这样场合极有可能每次申请锁的线程都是不相同的，因此这种场合下不应该使用偏向锁，否则会得不偿失，需要注意的是，偏向锁失败后，并不会立即膨胀为重量级锁，而是先升级为轻量级锁。下面我们接着了解轻量级锁。

轻量级锁

倘若偏向锁失败，虚拟机并不会立即升级为重量级锁，它还会尝试使用一种称为轻量级锁的优化手段(1.6之后加入的)，此时Mark Word 的结构也变为轻量级锁的结构。轻量级锁能够提升程序性能的依据是“对绝大部分的锁，在整个同步周期内都不存在竞争”，注意这是经验数据。需要了解的是，轻量级锁所适应的场景是线程交替执行同步块的场合，如果存在同一时间访问同一锁的场合，就会导致轻量级锁膨胀为重量级锁。

自旋锁

轻量级锁失败后，虚拟机为了避免线程真实地在操作系统层面挂起，还会进行一项称为自旋锁的优化手段。这是基于在大多数情况下，线程持有锁的时间都不会太长，如果直接挂起操作系统层面的线程可能会得不偿失，毕竟操作系统实现线程之间的切换时需要从用户态转换到核心态，这个状态之间的转换需要相对比较长的时间，时间成本相对较高，因此自旋锁会假设在不久将来，当前的线程可以获得锁，因此虚拟机会让当前想要获取锁的线程做几个空循环(这也是称为自旋的原因)，一般不会太久，可能是50个循环或100循环，在经过若干次循环后，如果得到锁，就顺利进入临界区。如果还不能获得锁，那就会将线程在操作系统层面挂起，这就是自旋锁的优化方式，这种方式确实也是可以提升效率的。最后没办法也就只能升级为重量级锁了。

锁消除

消除锁是虚拟机另外一种锁的优化，这种优化更彻底，Java虚拟机在JIT编译时(可以简单理解为当某段代码即将第一次被执行时进行编译，又称即时编译)，通过对运行上下文的扫描，去除不可能存在共享资源竞争的锁，通过这种方式消除没有必要的锁，可以节省毫无意义的请求锁时间，如下StringBuffer的append是一个同步方法，但是在add方法中的StringBuffer属于一个局部变量，并且不会被其他线程所使用，因此StringBuffer不可能存在共享资源竞争的情景，JVM会自动将锁消除。

|  |
| --- |
| public class StringBufferRemoveSync {  public void add(String str1, String str2) {  //StringBuffer是线程安全,由于sb只会在append方法中使用,不可能被其他线程引用  //因此sb属于不可能共享的资源,JVM会自动消除内部的锁  StringBuffer sb = new StringBuffer();  sb.append(str1).append(str2);  }  public static void main(String[] args) {  StringBufferRemoveSync rmsync = new StringBufferRemoveSync();  for (int i = 0; i < 10000000; i++) {  rmsync.add("abc", "123");  }  }  } |

#### synchronized的可重入性

从互斥锁的设计上来说，当一个线程试图操作一个由其他线程持有的对象锁的临界资源时，将会处于阻塞状态，但当一个线程再次请求自己持有对象锁的临界资源时，这种情况属于重入锁，请求将会成功，在java中synchronized是基于原子性的内部锁机制，是可重入的，因此在一个线程调用synchronized方法的同时在其方法体内部调用该对象另一个synchronized方法，也就是说一个线程得到一个对象锁后再次请求该对象锁，是允许的，这就是synchronized的可重入性。如下：

public class AccountingSync implements Runnable{

static AccountingSync instance=new AccountingSync();

static int i=0;

static int j=0;

@Override

public void run() {

for(int j=0;j<1000000;j++){

//this,当前实例对象锁

synchronized(this){

i++;

increase();//synchronized的可重入性

}

}

}

public synchronized void increase(){

j++;

}

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

Thread t1=new Thread(instance);

Thread t2=new Thread(instance);

t1.start();t2.start();

t1.join();t2.join();

System.out.println(i);

}

}

正如代码所演示的，在获取当前实例对象锁后进入synchronized代码块执行同步代码，并在代码块中调用了当前实例对象的另外一个synchronized方法，再次请求当前实例锁时，将被允许，进而执行方法体代码，这就是重入锁最直接的体现，需要特别注意另外一种情况，当子类继承父类时，子类也是可以通过可重入锁调用父类的同步方法。注意由于synchronized是基于monitor实现的，因此每次重入，monitor中的计数器仍会加1。

线程中断与synchronized

线程中断

正如中断二字所表达的意义，在线程运行(run方法)中间打断它，在Java中，提供了以下3个有关线程中断的方法

//中断线程（实例方法）

public void Thread.interrupt();

//判断线程是否被中断（实例方法）

public boolean Thread.isInterrupted();

//判断是否被中断并清除当前中断状态（静态方法）

public static boolean Thread.interrupted();

当一个线程处于被阻塞状态或者试图执行一个阻塞操作时，使用Thread.interrupt()方式中断该线程，注意此时将会抛出一个InterruptedException的异常，同时中断状态将会被复位(由中断状态改为非中断状态)，如下代码将演示该过程：

public class InterruputSleepThread3 {

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

Thread t1 = new Thread() {

@Override

public void run() {

//while在try中，通过异常中断就可以退出run循环

try {

while (true) {

//当前线程处于阻塞状态，异常必须捕捉处理，无法往外抛出

TimeUnit.SECONDS.sleep(2);

}

} catch (InterruptedException e) {

System.out.println("Interruted When Sleep");

boolean interrupt = this.isInterrupted();

//中断状态被复位

System.out.println("interrupt:"+interrupt);

}

}

};

t1.start();

TimeUnit.SECONDS.sleep(2);

//中断处于阻塞状态的线程

t1.interrupt();

/\*\*

\* 输出结果:

Interruted When Sleep

interrupt:false

\*/

}

}

如上述代码所示，我们创建一个线程，并在线程中调用了sleep方法从而使用线程进入阻塞状态，启动线程后，调用线程实例对象的interrupt方法中断阻塞异常，并抛出InterruptedException异常，此时中断状态也将被复位。这里有些人可能会诧异，为什么不用Thread.sleep(2000);而是用TimeUnit.SECONDS.sleep(2);其实原因很简单，前者使用时并没有明确的单位说明，而后者非常明确表达秒的单位，事实上后者的内部实现最终还是调用了Thread.sleep(2000);，但为了编写的代码语义更清晰，建议使用TimeUnit.SECONDS.sleep(2);的方式，注意TimeUnit是个枚举类型。ok~，除了阻塞中断的情景，我们还可能会遇到处于运行期且非阻塞的状态的线程，这种情况下，直接调用Thread.interrupt()中断线程是不会得到任响应的，如下代码，将无法中断非阻塞状态下的线程：

public class InterruputThread {

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

Thread t1=new Thread(){

@Override

public void run(){

while(true){

System.out.println("未被中断");

}

}

};

t1.start();

TimeUnit.SECONDS.sleep(2);

t1.interrupt();

/\*\*

\* 输出结果(无限执行):

未被中断

未被中断

未被中断

......

\*/

}

}

虽然我们调用了interrupt方法，但线程t1并未被中断，因为处于非阻塞状态的线程需要我们手动进行中断检测并结束程序，改进后代码如下：

public class InterruputThread {

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

Thread t1=new Thread(){

@Override

public void run(){

while(true){

//判断当前线程是否被中断

if (this.isInterrupted()){

System.out.println("线程中断");

break;

}

}

System.out.println("已跳出循环,线程中断!");

}

};

t1.start();

TimeUnit.SECONDS.sleep(2);

t1.interrupt();

/\*\*

\* 输出结果:

线程中断

已跳出循环,线程中断!

\*/

}

}

是的，我们在代码中使用了实例方法isInterrupted判断线程是否已被中断，如果被中断将跳出循环以此结束线程,注意非阻塞状态调用interrupt()并不会导致中断状态重置。综合所述，可以简单总结一下中断两种情况，一种是当线程处于阻塞状态或者试图执行一个阻塞操作时，我们可以使用实例方法interrupt()进行线程中断，执行中断操作后将会抛出interruptException异常(该异常必须捕捉无法向外抛出)并将中断状态复位，另外一种是当线程处于运行状态时，我们也可调用实例方法interrupt()进行线程中断，但同时必须手动判断中断状态，并编写中断线程的代码(其实就是结束run方法体的代码)。有时我们在编码时可能需要兼顾以上两种情况，那么就可以如下编写：

public void run(){

try {

//判断当前线程是否已中断,注意interrupted方法是静态的,执行后会对中断状态进行复位

while (!Thread.interrupted()) {

TimeUnit.SECONDS.sleep(2);

}

} catch (InterruptedException e) {

}

}

中断与synchronized

事实上线程的中断操作对于正在等待获取的锁对象的synchronized方法或者代码块并不起作用，也就是对于synchronized来说，如果一个线程在等待锁，那么结果只有两种，要么它获得这把锁继续执行，要么它就保存等待，即使调用中断线程的方法，也不会生效。演示代码如下

/\*\*

\* Created by zejian on 2017/6/2.

\* Blog : http://blog.csdn.net/javazejian [原文地址,请尊重原创]

\*/

public class SynchronizedBlocked implements Runnable{

public synchronized void f() {

System.out.println("Trying to call f()");

while(true) // Never releases lock

Thread.yield();

}

/\*\*

\* 在构造器中创建新线程并启动获取对象锁

\*/

public SynchronizedBlocked() {

//该线程已持有当前实例锁

new Thread() {

public void run() {

f(); // Lock acquired by this thread

}

}.start();

}

public void run() {

//中断判断

while (true) {

if (Thread.interrupted()) {

System.out.println("中断线程!!");

break;

} else {

f();

}

}

}

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

SynchronizedBlocked sync = new SynchronizedBlocked();

Thread t = new Thread(sync);

//启动后调用f()方法,无法获取当前实例锁处于等待状态

t.start();

TimeUnit.SECONDS.sleep(1);

//中断线程,无法生效

t.interrupt();

}

}

我们在SynchronizedBlocked构造函数中创建一个新线程并启动获取调用f()获取到当前实例锁，由于SynchronizedBlocked自身也是线程，启动后在其run方法中也调用了f()，但由于对象锁被其他线程占用，导致t线程只能等到锁，此时我们调用了t.interrupt();但并不能中断线程。

等待唤醒机制与synchronized

所谓等待唤醒机制本篇主要指的是notify/notifyAll和wait方法，在使用这3个方法时，必须处于synchronized代码块或者synchronized方法中，否则就会抛出IllegalMonitorStateException异常，这是因为调用这几个方法前必须拿到当前对象的监视器monitor对象，也就是说notify/notifyAll和wait方法依赖于monitor对象，在前面的分析中，我们知道monitor 存在于对象头的Mark Word 中(存储monitor引用指针)，而synchronized关键字可以获取 monitor ，这也就是为什么notify/notifyAll和wait方法必须在synchronized代码块或者synchronized方法调用的原因。

synchronized (obj) {

obj.wait();

obj.notify();

obj.notifyAll();

}

需要特别理解的一点是，与sleep方法不同的是wait方法调用完成后，线程将被暂停，但wait方法将会释放当前持有的监视器锁(monitor)，直到有线程调用notify/notifyAll方法后方能继续执行，而sleep方法只让线程休眠并不释放锁。同时notify/notifyAll方法调用后，并不会马上释放监视器锁，而是在相应的synchronized(){}/synchronized方法执行结束后才自动释放锁。

1. 线程状态及状态转换

    当多个线程同时请求某个对象监视器时，对象监视器会设置几种状态用来区分请求的线程：

Contention List：所有请求锁的线程将被首先放置到该竞争队列

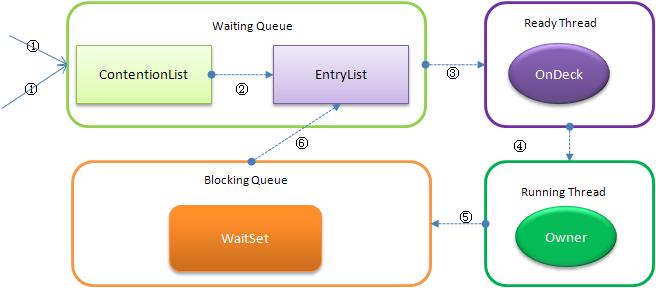
Entry List：Contention List中那些有资格成为候选人的线程被移到Entry List

Wait Set：那些调用wait方法被阻塞的线程被放置到Wait Set

OnDeck：任何时刻最多只能有一个线程正在竞争锁，该线程称为OnDeck

Owner：获得锁的线程称为Owner

!Owner：释放锁的线程

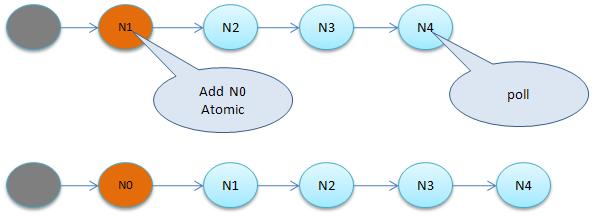
下图反映了个状态转换关系：  


　　新请求锁的线程将首先被加入到ConetentionList中，当某个拥有锁的线程（Owner状态）调用unlock之后，如果发现EntryList为空则从ContentionList中移动线程到EntryList，下面说明下ContentionList和EntryList的实现方式：

1.1 Contention List虚拟队列

　　Contention List并不是一个真正的Queue，而只是一个虚拟队列，原因在于ContentionList是由Node及其next指针逻辑构成，并不存在一个Queue的数据结构。ContentionList是一个后进先出（LIFO）的队列，每次新加入Node时都会在队头进行，通过CAS改变第一个节点的的指针为新增节点，同时设置新增节点的next指向后续节点，而取得操作则发生在队尾。显然，该结构其实是个Lock-Free的队列。

　　因为只有Owner线程才能从队尾取元素，也即线程出列操作无争用，当然也就避免了CAS的ABA问题。



ps.这里怎么看都像FIFO，查了很久，没找到关于Contention List 的详细说明，这里先不纠结了。

1.2 EntryList

EntryList与ContentionList逻辑上同属等待队列，ContentionList会被线程并发访问，为了降低对ContentionList队尾的争用，而建立EntryList。Owner线程在unlock时会从ContentionList中迁移线程到EntryList，并会指定EntryList中的某个线程（一般为Head）为Ready（OnDeck）线程。Owner线程并不是把锁传递给OnDeck线程，只是把竞争锁的权利交给OnDeck，OnDeck线程需要重新竞争锁。这样做虽然牺牲了一定的公平性，但极大的提高了整体吞吐量，在Hotspot中把OnDeck的选择行为称之为“竞争切换”。

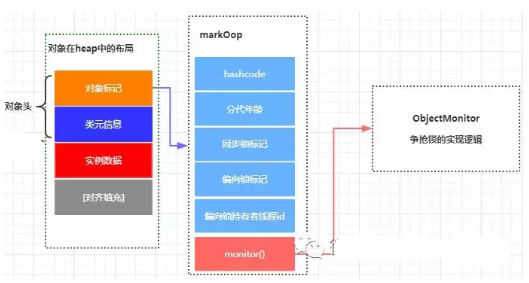
OnDeck线程获得锁后即变为owner线程，无法获得锁则会依然留在EntryList中，考虑到公平性，在EntryList中的位置不发生变化（依然在队头）。如果Owner线程被wait方法阻塞，则转移到WaitSet队列；如果在某个时刻被notify/notifyAll唤醒，则再次转移到EntryList。

#### 获取锁过程

1. 第一步，检查MarkWord里面是不是放的自己的ThreadId ,如果是，表示当前线程是处于 “偏向锁”.跳过轻量级锁直接执行同步体。
2. 第二步，如果MarkWord不是自己的ThreadId,锁升级，这时候，用CAS来执行切换，新的线程根据MarkWord里面现有的ThreadId，通知之前线程暂停，之前线程将Markword的内容置为空。
3. 第三步，两个线程都把对象的HashCode复制到自己新建的用于存储锁的记录空间，接着开始通过CAS操作，把共享对象的MarKword的内容修改为自己新建的记录空间的地址的方式竞争MarkWord.
4. 第四步，第三步中成功执行CAS的获得资源，失败的则进入自旋.
5. 第五步，自旋的线程在自旋过程中，成功获得资源(即之前获的资源的线程执行完成并释放了共享资源)，则整个状态依然处于轻量级锁的状态，如果自旋失败 第六步，进入重量级锁的状态，这个时候，自旋的线程进行阻塞，等待之前线程执行完成并唤醒自己.

#### C++源码分析

##### 基本类介绍



在分析源代码之前需要了解oop, oopDesc, markOop等相关概念。Java中的每一个Object在JVM内部都会有一个native的C++对象oop/oopDesc与之对应。在hotspot源码 oopDesc定义如下：

|  |
| --- |
| //openjdk\hotspot\src\share\vm\oops\oop.hpp  class **oopDesc** {  friend class VMStructs;  private:  volatile markOop \_mark; //Mark Word字段  union \_metadata {  Klass\* \_klass; //对象类型元数据的指针  narrowKlass \_compressed\_klass;  } \_metadata;  // Fast access to barrier set. Must be initialized.  static BarrierSet\* \_bs;  public:  markOop mark() const { return \_mark; }  markOop\* mark\_addr() const { return (markOop\*) &\_mark; }  void set\_mark(volatile markOop m) { \_mark = m; }  void release\_set\_mark(markOop m);  markOop cas\_set\_mark(markOop new\_mark, markOop old\_mark);  // Used only to re-initialize the mark word (e.g., of promoted  // objects during a GC) -- requires a valid klass pointer  void init\_mark();  Klass\* klass() const;  Klass\* klass\_or\_null() const volatile;  Klass\*\* klass\_addr();  narrowKlass\* compressed\_klass\_addr();  ....省略...  } |

markOop就是我们所说的Mark Word，用于存储锁的标识。hotspot源码 markOop.hpp文件代码片段

|  |
| --- |
| class markOopDesc: public oopDesc  {  private:  // Conversion  uintptr\_t value() const { return (uintptr\_t) this; }  public:  // Constants  //锁等都会有枚举  enum {  age\_bits = 4,  lock\_bits = 2,  biased\_lock\_bits = 1,  max\_hash\_bits = BitsPerWord - age\_bits - lock\_bits - biased\_lock\_bits,  hash\_bits = max\_hash\_bits > 31 ? 31 : max\_hash\_bits,  cms\_bits = LP64\_ONLY(1) NOT\_LP64(0),  epoch\_bits = 2 }; ...  } |

markOopDesc继承自oopDesc，并且扩展了自己的monitor方法，这个方法返回一个ObjectMonitor指针对象，

|  |
| --- |
| 1 ObjectMonitor\* monitor() const {  2 assert(has\_monitor(), "check");  3 return (ObjectMonitor\*) (value() ^ monitor\_value);  4 } |

其中value()这样定义：

 1 uintptr\_t value() const { return (uintptr\_t) this; }

可知：将this转换成一个指针宽度的整数（uintptr\_t），然后进行"异或"位操作。

monitor\_value是常量

1 enum { locked\_value = 0,//00偏向锁

2 unlocked\_value = 1,//01无锁

3 monitor\_value = 2,//10监视器锁，又叫重量级锁

4 marked\_value = 3,//11GC标记

5 biased\_lock\_pattern = 5 //101偏向锁

6 };

指针低2位00，异或10，结果还是10.（拿一个模板为00的数，异或一个二位数=数本身。因为异或：“相同为0，不同为1”.操作）

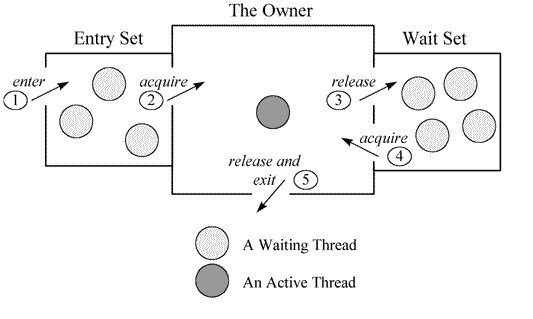
每个对象都存在着一个 ObjectMonitor与之关联，ObjectMonitor就是所谓的监视器，实际上可以理解为一个同步工具。在Hotspot中，是通过如下调用链获取到ObjectMonitor的，通过对象找mark word ，再找到 ObjectMonitor

oopDesc--继承-->markOopDesc--方法monitor()-->ObjectMonitor-->enter、exit 获取、释放锁

如下是ObjectMonitor的构造函数：

|  |
| --- |
| ObjectMonitor() {  \_header = NULL; //指向markOop对象  \_count = 0;  \_waiters = 0, //等待线程数  \_recursions = 0; //重入次数  \_object = NULL; //监视器锁寄生的对象。锁不是平白出现的，而是寄托存储于对象中。  \_owner = NULL; //指向获得ObjectMonitor对象的线程或基础锁  \_WaitSet = NULL; //处于wait状态的线程，会被加入到wait set；  \_WaitSetLock = 0 ;  \_Responsible = NULL ;  \_succ = NULL ;  \_cxq = NULL ;  FreeNext = NULL ;  \_EntryList = NULL ; //处于等待锁block状态的线程，会被加入到entry set；  \_SpinFreq = 0 ;  \_SpinClock = 0 ;  OwnerIsThread = 0 ; // \_owner is (Thread \*) vs SP/BasicLock  \_previous\_owner\_tid = 0; // 监视器前一个拥有者线程的ID  } |

ObjectMonitor中有两个队列，\_WaitSet 和 \_EntryList，用来保存ObjectWaiter对象列表( 每个等待锁的线程都会被封装成ObjectWaiter对象)，\_owner指向持有ObjectMonitor对象的线程，当多个线程同时访问一段同步代码时，首先会进入 \_EntryList 集合，当线程获取到对象的monitor 后进入 \_Owner 区域并把monitor中的owner变量设置为当前线程同时monitor中的计数器count加1，若线程调用 wait() 方法，将释放当前持有的monitor，owner变量恢复为null，count自减1，同时该线程进入 WaitSe t集合中等待被唤醒。若当前线程执行完毕也将释放monitor(锁)并复位变量的值，以便其他线程进入获取monitor(锁)。如下图所示



##### 锁获取流程

Synchronized 通过 monitorenter、monitorexit指令来获取和释放互斥锁.。涉及到如下两个函数

|  |
| --- |
| InterpreterRuntime::monitorenter(JavaThread\* thread, BasicObjectLock\* elem)  InterpreterRuntime::monitorexit(JavaThread\* thread, BasicObjectLock\* elem) |

先看InterpreterRuntime::monitorenter函数，具体实现如下：

|  |
| --- |
| IRT\_ENTRY\_NO\_ASYNC(void, InterpreterRuntime::monitorenter(JavaThread\* thread, BasicObjectLock\* elem))  #ifdef ASSERT  thread->last\_frame().interpreter\_frame\_verify\_monitor(elem);  #endif  if (PrintBiasedLockingStatistics) {  Atomic::inc(BiasedLocking::slow\_path\_entry\_count\_addr());  }  Handle h\_obj(thread, elem->obj());  assert(Universe::heap()->is\_in\_reserved\_or\_null(h\_obj()),  "must be NULL or an object");  if (UseBiasedLocking) {//标识虚拟机是否开启偏向锁功能,默认开启  // Retry fast entry if bias is revoked to avoid unnecessary inflation  ObjectSynchronizer::fast\_enter(h\_obj, elem->lock(), true, CHECK);  } else {  ObjectSynchronizer::slow\_enter(h\_obj, elem->lock(), CHECK);  }  assert(Universe::heap()->is\_in\_reserved\_or\_null(elem->obj()),  "must be NULL or an object");  #ifdef ASSERT  thread->last\_frame().interpreter\_frame\_verify\_monitor(elem);  #endif  IRT\_END |

先看一下入参：

1、JavaThread thread指向java中的当前线程；  
2、BasicObjectLock基础对象锁：包含一个BasicLock和一个指向Object对象的指针oop。

|  |
| --- |
| class BasicObjectLock VALUE\_OBJ\_CLASS\_SPEC {  friend class VMStructs;  private:  BasicLock \_lock; // the lock, must be double word aligned  oop \_obj; // object holds the lock;  public:  // Manipulation  oop obj() const { return \_obj; }  void set\_obj(oop obj) { \_obj = obj; }  BasicLock\* lock() { return &\_lock; }  // Note: Use frame::interpreter\_frame\_monitor\_size() for the size of BasicObjectLocks  // in interpreter activation frames since it includes machine-specific padding.  static int size() { return sizeof(BasicObjectLock)/wordSize; }  // GC support  void oops\_do(OopClosure\* f) { f->do\_oop(&\_obj); }  static int obj\_offset\_in\_bytes() { return offset\_of(BasicObjectLock, \_obj); }  static int lock\_offset\_in\_bytes() { return offset\_of(BasicObjectLock, \_lock); }  }; |

BasicLock类型\_lock对象主要用来保存：指向Object对象的对象头数据；

|  |
| --- |
| class BasicLock VALUE\_OBJ\_CLASS\_SPEC {  friend class VMStructs;  private:  volatile markOop \_displaced\_header;//markOop是不是很熟悉？1.2节中讲解对象头时就是分析的markOop源码  public:  markOop displaced\_header() const { return \_displaced\_header; }  void set\_displaced\_header(markOop header) { \_displaced\_header = header; }  void print\_on(outputStream\* st) const;  // move a basic lock (used during deoptimization  void move\_to(oop obj, BasicLock\* dest);  static int displaced\_header\_offset\_in\_bytes() { return offset\_of(BasicLock, \_displaced\_header); }  }; |

###### 偏向锁获取逻辑

接着分析InterpreterRuntime::monitorenter()函数，UseBiasedLocking是在JVM启动的时候，是否启动偏向锁的标识

如果支持偏向锁,则执行 ObjectSynchronizer::fast\_enter的逻辑

如果不支持偏向锁,则执行 ObjectSynchronizer::slow\_enter逻辑，绕过偏向锁，直接进入轻量级锁

ObjectSynchronizer::fast\_enter的实现在 synchronizer.cpp文件中，代码如下

|  |
| --- |
| void ObjectSynchronizer::fast\_enter(Handle obj, BasicLock\* lock, bool attempt\_rebias, TRAPS) {  if (UseBiasedLocking) {  if (!SafepointSynchronize::is\_at\_safepoint()) {  BiasedLocking::Condition cond = BiasedLocking::revoke\_and\_rebias(obj, attempt\_rebias, THREAD);  if (cond == BiasedLocking::BIAS\_REVOKED\_AND\_REBIASED) {  return;  }  } else {  assert(!attempt\_rebias, "can not rebias toward VM thread");  BiasedLocking::revoke\_at\_safepoint(obj);  }  assert(!obj->mark()->has\_bias\_pattern(), "biases should be revoked by now");  }  //轻量级锁  slow\_enter (obj, lock, THREAD) ;  } |

**偏向锁的获取由BiasedLocking::revoke\_and\_rebias**

1、通过markOop mark = obj->mark()获取对象的markOop数据mark，即对象头的Mark Word；

2、判断mark是否为可偏向状态，即mark的偏向锁标志位为 **1**，锁标志位为 **01**；

3、判断mark中JavaThread的状态：如果为空，则进入步骤（4）；如果指向当前线程，则执行同步代码块；如果指向其它线程，进入步骤（5）；

4、通过CAS原子指令设置mark中JavaThread为当前线程ID，如果执行CAS成功，则执行同步代码块，否则进入步骤（5）；

5、如果执行CAS失败，表示当前存在多个线程竞争锁，当达到全局安全点（safepoint），获得偏向锁的线程被挂起，撤销偏向锁，并升级为轻量级，升级完成后被阻塞在安全点的线程继续执行同步代码块；

|  |
| --- |
| BiasedLocking::Condition BiasedLocking::revoke\_and\_rebias(Handle obj, bool attempt\_rebias, TRAPS) {  assert(!SafepointSynchronize::is\_at\_safepoint(), "must not be called while at safepoint");  markOop mark = obj->mark(); //获取锁对象的对象头  //判断mark是否为可偏向状态，即mark的偏向锁标志位为1，锁标志位为 01，线程id为null  if (mark->is\_biased\_anonymously() && !attempt\_rebias) {  //这个分支是进行对象的hashCode计算时会进入，在一个非全局安全点进行偏向锁撤销  markOop biased\_value = mark;  //创建一个非偏向的markword  markOop unbiased\_prototype = markOopDesc::prototype()->set\_age(mark->age());  //Atomic:cmpxchg\_ptr是CAS操作，通过cas重新设置偏向锁状态  markOop res\_mark = (markOop) Atomic::cmpxchg\_ptr(unbiased\_prototype, obj->mark\_addr(), mark);  if (res\_mark == biased\_value) {//如果CAS成功，返回偏向锁撤销状态  return BIAS\_REVOKED;  }  } else if (mark->has\_bias\_pattern()) {//如果锁对象为可偏向状态（biased\_lock:1, lock:01，不管线程id是否为空）,尝试重新偏向  Klass\* k = obj->klass();  markOop prototype\_header = k->prototype\_header();  //如果已经有线程对锁对象进行了全局锁定，则取消偏向锁操作  if (!prototype\_header->has\_bias\_pattern()) {  markOop biased\_value = mark;  //CAS 更新对象头markword为非偏向锁  markOop res\_mark = (markOop) Atomic::cmpxchg\_ptr(prototype\_header, obj->mark\_addr(), mark);  assert(!(\*(obj->mark\_addr()))->has\_bias\_pattern(), "even if we raced, should still be revoked");  return BIAS\_REVOKED; //返回偏向锁撤销状态  } else if (prototype\_header->bias\_epoch() != mark->bias\_epoch()) {  //如果偏向锁过期，则进入当前分支  if (attempt\_rebias) {//如果允许尝试获取偏向锁  assert(THREAD->is\_Java\_thread(), "");  markOop biased\_value = mark;  markOop rebiased\_prototype = markOopDesc::encode((JavaThread\*) THREAD, mark->age(), prototype\_header->bias\_epoch());  //通过CAS 操作， 将本线程的 ThreadID 、时间错、分代年龄尝试写入对象头中  markOop res\_mark = (markOop) Atomic::cmpxchg\_ptr(rebiased\_prototype, obj->mark\_addr(), mark);  if (res\_mark == biased\_value) { //CAS成功，则返回撤销和重新偏向状态  return BIAS\_REVOKED\_AND\_REBIASED;  }  } else {//不尝试获取偏向锁，则取消偏向锁  //通过CAS操作更新分代年龄  markOop biased\_value = mark;  markOop unbiased\_prototype = markOopDesc::prototype()->set\_age(mark->age());  markOop res\_mark = (markOop) Atomic::cmpxchg\_ptr(unbiased\_prototype, obj->mark\_addr(), mark);  if (res\_mark == biased\_value) { //如果CAS操作成功，返回偏向锁撤销状态  return BIAS\_REVOKED;  }  }  }  }  ...//省略  } |

当到达一个全局安全点时，这时会根据偏向锁的状态来判断是否需要撤销偏向锁只有当其它线程尝试竞争偏向锁时，持有偏向锁的线程才会释放锁，偏向锁的撤销由BiasedLocking::revoke\_at\_safepoint方法实现：

|  |
| --- |
| void BiasedLocking::revoke\_at\_safepoint(Handle h\_obj) {  assert(SafepointSynchronize::is\_at\_safepoint(), "must only be called while at safepoint");  oop obj = h\_obj();  //更新撤销偏向锁计数，并返回偏向锁撤销次数和偏向次数  HeuristicsResult heuristics = update\_heuristics(obj, false);  if (heuristics == HR\_SINGLE\_REVOKE) {//可偏向且未达到批量处理的阈值(下面会单独解释)  revoke\_bias(obj, false, false, NULL); //撤销偏向锁  } else if ((heuristics == HR\_BULK\_REBIAS) ||  (heuristics == HR\_BULK\_REVOKE)) {//如果是多次撤销或者多次偏向  //批量撤销  bulk\_revoke\_or\_rebias\_at\_safepoint(obj, (heuristics == HR\_BULK\_REBIAS), false, NULL);  }  clean\_up\_cached\_monitor\_info();  } |

偏向锁的释放，需要等待全局安全点（在这个时间点上没有正在执行的字节码），首先暂停拥有偏向锁的线程，然后检查持有偏向锁的线程是否还活着，如果线程不处于活动状态，则将对象头设置成无锁状态。如果线程仍然活着，则会遍历偏向对象的锁记录。栈帧中的锁记录和对象头的Mark Word要么重新偏向其他线程，要么恢复到无锁，或者标记对象不适合作为偏向锁。然后重新争锁，多个线程竞争偏向锁导致偏向锁升级为轻量级锁，会尝试获取轻量级锁，其入口位于ObjectSynchronizer::slow\_enter。

|  |
| --- |
| void ObjectSynchronizer::slow\_enter(Handle obj, BasicLock\* lock, TRAPS) {  markOop mark = obj->mark();  assert(!mark->has\_bias\_pattern(), "should not see bias pattern here");  if (mark->is\_neutral()) { //如果当前是无锁状态, markword的biase\_lock:0，lock:01  //直接把mark保存到BasicLock对象的\_displaced\_header字段  lock->set\_displaced\_header(mark);  //通过CAS将mark word更新为指向BasicLock对象的指针，更新成功表示获得了轻量级锁  if (mark == (markOop) Atomic::cmpxchg\_ptr(lock, obj()->mark\_addr(), mark)) {  TEVENT (slow\_enter: release stacklock) ;  return ;  }  // Fall through to inflate() ...  }  //如果markword处于加锁状态、且markword中的ptr指针指向当前线程的栈帧，表示为重入操作，不需要争抢锁  else if (mark->has\_locker() && THREAD->is\_lock\_owned((address)mark->locker())) {  assert(lock != mark->locker(), "must not re-lock the same lock");  assert(lock != (BasicLock\*)obj->mark(), "don't relock with same BasicLock");  lock->set\_displaced\_header(NULL);  return;  }  #if 0  // The following optimization isn't particularly useful.  if (mark->has\_monitor() && mark->monitor()->is\_entered(THREAD)) {  lock->set\_displaced\_header (NULL) ;  return ;  }  #endif  //代码执行到这里，说明有多个线程竞争轻量级锁，轻量级锁通过`inflate`进行膨胀升级为重量级锁  lock->set\_displaced\_header(markOopDesc::unused\_mark());  ObjectSynchronizer::inflate(THREAD, obj())->enter(THREAD);  } |

1、markOop mark = obj->mark()方法获取对象的markOop数据mark；

2、mark->is\_neutral()方法判断mark是否为无锁状态：mark的偏向锁标志位为 **0**，锁标志位为 **01**；

3、如果mark处于无锁状态，则进入步骤（4），否则执行步骤（6）；

4、把mark保存到BasicLock对象的\_displaced\_header字段；

5、通过CAS尝试将Mark Word更新为指向BasicLock对象的指针，如果更新成功，表示竞争到锁，则执行同步代码，否则执行步骤（6）；

6、如果当前mark处于加锁状态，且mark中的ptr指针指向当前线程的栈帧，则执行同步代码，否则说明有多个线程竞争轻量级锁，轻量级锁需要膨胀升级为重量级锁；

**假设线程A和B同时执行到临界区if (mark->is\_neutral())**：

1、线程AB都把Mark Word复制到各自的\_displaced\_header字段，该数据保存在线程的栈帧上，是线程私有的；

2、Atomic::cmpxchg\_ptr原子操作保证只有一个线程可以把指向栈帧的指针复制到Mark Word，假设此时线程A执行成功，并返回继续执行同步代码块；

3、线程B执行失败，退出临界区，通过ObjectSynchronizer::inflate方法开始膨胀锁；

###### 轻量级锁的释放逻辑

轻量级锁的释放是通过 monitorexit调用

|  |
| --- |
| IRT\_ENTRY\_NO\_ASYNC(void, InterpreterRuntime::monitorexit(JavaThread\* thread, BasicObjectLock\* elem))  #ifdef ASSERT  thread->last\_frame().interpreter\_frame\_verify\_monitor(elem);  #endif  Handle h\_obj(thread, elem->obj());  assert(Universe::heap()->is\_in\_reserved\_or\_null(h\_obj()),  "must be NULL or an object");  if (elem == NULL || h\_obj()->is\_unlocked()) {  THROW(vmSymbols::java\_lang\_IllegalMonitorStateException());  }  ObjectSynchronizer::slow\_exit(h\_obj(), elem->lock(), thread);  // Free entry. This must be done here, since a pending exception might be installed on  // exit. If it is not cleared, the exception handling code will try to unlock the monitor again.  elem->set\_obj(NULL);  #ifdef ASSERT  thread->last\_frame().interpreter\_frame\_verify\_monitor(elem);  #endif  IRT\_END |

这段代码中主要是通过 ObjectSynchronizer::slow\_exit来执行

void ObjectSynchronizer::slow\_exit(oop object, BasicLock\* lock, TRAPS) {

fast\_exit (object, lock, THREAD) ;

}

最终通过ObjectSynchronizer::slow\_exit--->调用ObjectSynchronizer::fast\_exit完成。

|  |
| --- |
| void ObjectSynchronizer::fast\_exit(oop object, BasicLock\* lock, TRAPS) {  assert(!object->mark()->has\_bias\_pattern(), "should not see bias pattern here");  // if displaced header is null, the previous enter is recursive enter, no-op  markOop dhw = lock->displaced\_header(); //获取锁对象中的对象头  markOop mark ;  if (dhw == NULL) {  // Recursive stack-lock.  // Diagnostics -- Could be: stack-locked, inflating, inflated.  mark = object->mark() ;  assert (!mark->is\_neutral(), "invariant") ;  if (mark->has\_locker() && mark != markOopDesc::INFLATING()) {  assert(THREAD->is\_lock\_owned((address)mark->locker()), "invariant") ;  }  if (mark->has\_monitor()) {  ObjectMonitor \* m = mark->monitor() ;  assert(((oop)(m->object()))->mark() == mark, "invariant") ;  assert(m->is\_entered(THREAD), "invariant") ;  }  return ;  }  mark = object->mark() ; //获取线程栈帧中锁记录(LockRecord)中的markword  // If the object is stack-locked by the current thread, try to  // swing the displaced header from the box back to the mark.  if (mark == (markOop) lock) {  assert (dhw->is\_neutral(), "invariant") ;  //通过CAS尝试将Displaced Mark Word替换回对象头，如果成功，表示锁释放成功。  if ((markOop) Atomic::cmpxchg\_ptr (dhw, object->mark\_addr(), mark) == mark) {  TEVENT (fast\_exit: release stacklock) ;  return;  }  }  //锁膨胀，调用重量级锁的释放锁方法  ObjectSynchronizer::inflate(THREAD, object)->exit (true, THREAD) ;  } |

1、确保处于偏向锁状态时不会执行这段逻辑；

2、取出在获取轻量级锁时保存在BasicLock对象的mark数据dhw；

3、通过CAS尝试把dhw替换到当前的Mark Word，如果CAS成功，说明成功的释放了锁，否则执行步骤（4）；

4、如果CAS失败，说明有其它线程在尝试获取该锁，这时需要将该锁升级为重量级锁，并释放；

轻量级锁的释放也比较简单，就是将当前线程栈帧中锁记录空间中的Mark Word替换到锁对象的对象头中，如果成功表示锁释放成功。否则，锁膨胀成重量级锁，实现重量级锁的释放锁逻辑

###### 重量级锁

重量级锁通过对象内部的监视器（monitor）实现，其中monitor的本质是依赖于底层操作系统的Mutex Lock实现，操作系统实现线程之间的切换需要从用户态到内核态的切换，切换成本非常高。

锁的膨胀过程通过ObjectSynchronizer::inflate函数实现

|  |
| --- |
| ObjectMonitor \* ATTR ObjectSynchronizer::inflate (Thread \* Self, oop object) {  // Inflate mutates the heap ...  // Relaxing assertion for bug 6320749.  assert (Universe::verify\_in\_progress() ||  !SafepointSynchronize::is\_at\_safepoint(), "invariant") ;  for (;;) { //通过无意义的循环实现自旋操作  const markOop mark = object->mark() ;  assert (!mark->has\_bias\_pattern(), "invariant") ;  if (mark->has\_monitor()) {//has\_monitor是markOop.hpp中的方法，如果为true表示当前锁已经是重量级锁了  ObjectMonitor \* inf = mark->monitor() ;//获得重量级锁的对象监视器直接返回  assert (inf->header()->is\_neutral(), "invariant");  assert (inf->object() == object, "invariant") ;  assert (ObjectSynchronizer::verify\_objmon\_isinpool(inf), "monitor is invalid");  return inf ;  }  if (mark == markOopDesc::INFLATING()) {//膨胀等待，表示存在线程正在膨胀，通过continue进行下一轮的膨胀  TEVENT (Inflate: spin while INFLATING) ;  ReadStableMark(object) ;  continue ;  }  if (mark->has\_locker()) {//表示当前锁为轻量级锁，以下是轻量级锁的膨胀逻辑  ObjectMonitor \* m = omAlloc (Self) ;//获取一个可用的ObjectMonitor  // Optimistically prepare the objectmonitor - anticipate successful CAS  // We do this before the CAS in order to minimize the length of time  // in which INFLATING appears in the mark.  m->Recycle();  m->\_Responsible = NULL ;  m->OwnerIsThread = 0 ;  m->\_recursions = 0 ;  m->\_SpinDuration = ObjectMonitor::Knob\_SpinLimit ; // Consider: maintain by type/class  /\*\*将object->mark\_addr()和mark比较，如果这两个值相等，则将object->mark\_addr()  改成markOopDesc::INFLATING()，相等返回是mark，不相等返回的是object->mark\_addr()\*\*/  markOop cmp = (markOop) Atomic::cmpxchg\_ptr (markOopDesc::INFLATING(), object->mark\_addr(), mark) ;  if (cmp != mark) {//CAS失败  omRelease (Self, m, true) ;//释放监视器  continue ; // 重试  }  markOop dmw = mark->displaced\_mark\_helper() ;  assert (dmw->is\_neutral(), "invariant") ;  //CAS成功以后，设置ObjectMonitor相关属性  m->set\_header(dmw) ;  m->set\_owner(mark->locker());  m->set\_object(object);  // TODO-FIXME: assert BasicLock->dhw != 0.  guarantee (object->mark() == markOopDesc::INFLATING(), "invariant") ;  object->release\_set\_mark(markOopDesc::encode(m));  if (ObjectMonitor::\_sync\_Inflations != NULL) ObjectMonitor::\_sync\_Inflations->inc() ;  TEVENT(Inflate: overwrite stacklock) ;  if (TraceMonitorInflation) {  if (object->is\_instance()) {  ResourceMark rm;  tty->print\_cr("Inflating object " INTPTR\_FORMAT " , mark " INTPTR\_FORMAT " , type %s",  (void \*) object, (intptr\_t) object->mark(),  object->klass()->external\_name());  }  }  return m ; //返回ObjectMonitor  }  //如果是无锁状态  assert (mark->is\_neutral(), "invariant");  ObjectMonitor \* m = omAlloc (Self) ; ////获取一个可用的ObjectMonitor  //设置ObjectMonitor相关属性  m->Recycle();  m->set\_header(mark);  m->set\_owner(NULL);  m->set\_object(object);  m->OwnerIsThread = 1 ;  m->\_recursions = 0 ;  m->\_Responsible = NULL ;  m->\_SpinDuration = ObjectMonitor::Knob\_SpinLimit ; // consider: keep metastats by type/class  /\*\*将object->mark\_addr()和mark比较，如果这两个值相等，则将object->mark\_addr()  改成markOopDesc::encode(m)，相等返回是mark，不相等返回的是object->mark\_addr()\*\*/  if (Atomic::cmpxchg\_ptr (markOopDesc::encode(m), object->mark\_addr(), mark) != mark) {  //CAS失败，说明出现了锁竞争，则释放监视器重行竞争锁  m->set\_object (NULL) ;  m->set\_owner (NULL) ;  m->OwnerIsThread = 0 ;  m->Recycle() ;  omRelease (Self, m, true) ;  m = NULL ;  continue ;  // interference - the markword changed - just retry.  // The state-transitions are one-way, so there's no chance of  // live-lock -- "Inflated" is an absorbing state.  }  if (ObjectMonitor::\_sync\_Inflations != NULL) ObjectMonitor::\_sync\_Inflations->inc() ;  TEVENT(Inflate: overwrite neutral) ;  if (TraceMonitorInflation) {  if (object->is\_instance()) {  ResourceMark rm;  tty->print\_cr("Inflating object " INTPTR\_FORMAT " , mark " INTPTR\_FORMAT " , type %s",  (void \*) object, (intptr\_t) object->mark(),  object->klass()->external\_name());  }  }  return m ; //返回ObjectMonitor对象  }  } |

膨胀过程的实现比较复杂，大概实现过程如下：

1、整个膨胀过程在自旋下完成；

2、mark->has\_monitor()方法判断当前是否为重量级锁（上图18-25行），即Mark Word的锁标识位为 **10**，如果当前状态为重量级锁，执行步骤（3），否则执行步骤（4）；

3、mark->monitor()方法获取指向ObjectMonitor的指针，并返回，说明膨胀过程已经完成；

4、如果当前锁处于膨胀中（上图33-37行），说明该锁正在被其它线程执行膨胀操作，则当前线程就进行自旋等待锁膨胀完成，这里需要注意一点，虽然是自旋操作，但不会一直占用cpu资源，每隔一段时间会通过os::NakedYield方法放弃cpu资源，或通过park方法挂起；如果其他线程完成锁的膨胀操作，则退出自旋并返回；

5、如果当前是轻量级锁状态（上图58-138行），即锁标识位为 **00**，膨胀过程如下：

通过omAlloc方法，获取一个可用的ObjectMonitor monitor，并重置monitor数据；

通过CAS尝试将Mark Word设置为markOopDesc:INFLATING，标识当前锁正在膨胀中，如果CAS失败，说明同一时刻其它线程已经将Mark Word设置为markOopDesc:INFLATING，当前线程进行自旋等待膨胀完成；

如果CAS成功，设置monitor的各个字段：\_header、\_owner和\_object等，并返回；

6、如果是无锁（中立，上图150-186行），重置监视器值；

###### monitor竞争

当锁膨胀完成并返回对应的monitor时，并不表示该线程竞争到了锁，真正的锁竞争发生在ObjectMonitor::enter方法中。

|  |
| --- |
| void ATTR ObjectMonitor::enter(TRAPS) {  // The following code is ordered to check the most common cases first  // and to reduce RTS->RTO cache line upgrades on SPARC and IA32 processors.  Thread \* const Self = THREAD ;  void \* cur ;  cur = Atomic::cmpxchg\_ptr (Self, &\_owner, NULL) ;  if (cur == NULL) {//CAS成功  // Either ASSERT \_recursions == 0 or explicitly set \_recursions = 0.  assert (\_recursions == 0 , "invariant") ;  assert (\_owner == Self, "invariant") ;  // CONSIDER: set or assert OwnerIsThread == 1  return ;  }  if (cur == Self) {//重入锁  // TODO-FIXME: check for integer overflow! BUGID 6557169.  \_recursions ++ ;  return ;  }  if (Self->is\_lock\_owned ((address)cur)) {  assert (\_recursions == 0, "internal state error");  \_recursions = 1 ;  // Commute owner from a thread-specific on-stack BasicLockObject address to  // a full-fledged "Thread \*".  \_owner = Self ;  OwnerIsThread = 1 ;  return ;  }  // We've encountered genuine contention.  assert (Self->\_Stalled == 0, "invariant") ;  Self->\_Stalled = intptr\_t(this) ;  // Try one round of spinning \*before\* enqueueing Self  // and before going through the awkward and expensive state  // transitions. The following spin is strictly optional ...  // Note that if we acquire the monitor from an initial spin  // we forgo posting JVMTI events and firing DTRACE probes.  if (Knob\_SpinEarly && TrySpin (Self) > 0) {  assert (\_owner == Self , "invariant") ;  assert (\_recursions == 0 , "invariant") ;  assert (((oop)(object()))->mark() == markOopDesc::encode(this), "invariant") ;  Self->\_Stalled = 0 ;  return ;  }  assert (\_owner != Self , "invariant") ;  assert (\_succ != Self , "invariant") ;  assert (Self->is\_Java\_thread() , "invariant") ;  JavaThread \* jt = (JavaThread \*) Self ;  assert (!SafepointSynchronize::is\_at\_safepoint(), "invariant") ;  assert (jt->thread\_state() != \_thread\_blocked , "invariant") ;  assert (this->object() != NULL , "invariant") ;  assert (\_count >= 0, "invariant") ;  // Prevent deflation at STW-time. See deflate\_idle\_monitors() and is\_busy().  // Ensure the object-monitor relationship remains stable while there's contention.  Atomic::inc\_ptr(&\_count);  EventJavaMonitorEnter event;  { // Change java thread status to indicate blocked on monitor enter.  JavaThreadBlockedOnMonitorEnterState jtbmes(jt, this);  DTRACE\_MONITOR\_PROBE(contended\_\_enter, this, object(), jt);  if (JvmtiExport::should\_post\_monitor\_contended\_enter()) {  JvmtiExport::post\_monitor\_contended\_enter(jt, this);  }  OSThreadContendState osts(Self->osthread());  ThreadBlockInVM tbivm(jt);  Self->set\_current\_pending\_monitor(this);  // TODO-FIXME: change the following for(;;) loop to straight-line code.  for (;;) {  jt->set\_suspend\_equivalent();  // cleared by handle\_special\_suspend\_equivalent\_condition()  // or java\_suspend\_self()  EnterI (THREAD) ;  ...省略...139 } |

1、通过CAS尝试把monitor的\_owner字段设置为当前线程；

2、如果设置之前的\_owner指向当前线程，说明当前线程再次进入monitor，即重入锁，执行\_recursions ++ ，记录重入的次数；

3、如果之前的\_owner指向的地址在当前线程中，这种描述有点拗口，换一种说法：之前\_owner指向的BasicLock在当前线程栈上，说明当前线程是第一次进入该monitor，设置\_recursions为1，\_owner为当前线程，该线程成功获得锁并返回；

4、如果获取锁失败，则等待锁的释放；

###### monitor等待

monitor竞争失败的线程，通过自旋执行ObjectMonitor::EnterI方法等待锁的释放，EnterI方法的部分逻辑实现如下：

|  |
| --- |
| ObjectWaiter node(Self) ;  Self->\_ParkEvent->reset() ;  node.\_prev = (ObjectWaiter \*) 0xBAD ;  node.TState = ObjectWaiter::TS\_CXQ ;  // Push "Self" onto the front of the \_cxq.  // Once on cxq/EntryList, Self stays on-queue until it acquires the lock.  // Note that spinning tends to reduce the rate at which threads  // enqueue and dequeue on EntryList|cxq.  ObjectWaiter \* nxt ;  for (;;) {  node.\_next = nxt = \_cxq ;  if (Atomic::cmpxchg\_ptr (&node, &\_cxq, nxt) == nxt) break ;  // Interference - the CAS failed because \_cxq changed. Just retry.  // As an optional optimization we retry the lock.  if (TryLock (Self) > 0) {  assert (\_succ != Self , "invariant") ;  assert (\_owner == Self , "invariant") ;  assert (\_Responsible != Self , "invariant") ;  return ;  }  } |

1、当前线程被封装成ObjectWaiter对象node，状态设置成ObjectWaiter::TS\_CXQ；

2、在for循环中，通过CAS把node节点push到\_cxq列表中，同一时刻可能有多个线程把自己的node节点push到\_cxq列表中；

3、node节点push到\_cxq列表之后，通过自旋尝试获取锁，如果还是没有获取到锁，则通过park将当前线程挂起，等待被唤醒，实现如下：

|  |
| --- |
| int ObjectMonitor::TryLock (Thread \* Self) {  for (;;) {  void \* own = \_owner ;  if (own != NULL) return 0 ;  if (Atomic::cmpxchg\_ptr (Self, &\_owner, NULL) == NULL) {//CAS成功，获取锁  // Either guarantee \_recursions == 0 or set \_recursions = 0.  assert (\_recursions == 0, "invariant") ;  assert (\_owner == Self, "invariant") ;  // CONSIDER: set or assert that OwnerIsThread == 1  return 1 ;  }  // The lock had been free momentarily, but we lost the race to the lock.  // Interference -- the CAS failed.  // We can either return -1 or retry.  // Retry doesn't make as much sense because the lock was just acquired.  if (true) return -1 ;  }  } |

其本质就是通过CAS设置monitor的\_owner字段为当前线程，如果CAS成功，则表示该线程获取了锁，跳出自旋操作，执行同步代码，否则继续被挂起；

###### monitor释放

当某个持有锁的线程执行完同步代码块时，会进行锁的释放，给其它线程机会执行同步代码，在HotSpot中，通过退出monitor的方式实现锁的释放，并通知被阻塞的线程，具体实现位于ObjectMonitor::exit方法中。

|  |
| --- |
| void ATTR ObjectMonitor::exit(bool not\_suspended, TRAPS) {  Thread \* Self = THREAD ;  if (THREAD != \_owner) {  if (THREAD->is\_lock\_owned((address) \_owner)) {  // Transmute \_owner from a BasicLock pointer to a Thread address.  // We don't need to hold \_mutex for this transition.  // Non-null to Non-null is safe as long as all readers can  // tolerate either flavor.  assert (\_recursions == 0, "invariant") ;  \_owner = THREAD ;  \_recursions = 0 ;  OwnerIsThread = 1 ;  } else {  // NOTE: we need to handle unbalanced monitor enter/exit  // in native code by throwing an exception.  // TODO: Throw an IllegalMonitorStateException ?  TEVENT (Exit - Throw IMSX) ;  assert(false, "Non-balanced monitor enter/exit!");  if (false) {  THROW(vmSymbols::java\_lang\_IllegalMonitorStateException());  }  return;  }  }  if (\_recursions != 0) {  \_recursions--; // this is simple recursive enter  TEVENT (Inflated exit - recursive) ;  return ;  }  ...省略... |

1、如果是重量级锁的释放，monitor中的\_owner指向当前线程，即THREAD == \_owner；

2、根据不同的策略（由QMode指定），从cxq或EntryList中获取头节点，通过ObjectMonitor::ExitEpilog方法唤醒该节点封装的线程，唤醒操作最终由unpark完成，实现如下：

|  |
| --- |
| void ObjectMonitor::ExitEpilog (Thread \* Self, ObjectWaiter \* Wakee) {  assert (\_owner == Self, "invariant") ;  // Exit protocol:  // 1. ST \_succ = wakee  // 2. membar #loadstore|#storestore;  // 2. ST \_owner = NULL  // 3. unpark(wakee)  \_succ = Knob\_SuccEnabled ? Wakee->\_thread : NULL ;  ParkEvent \* Trigger = Wakee->\_event ;  // Hygiene -- once we've set \_owner = NULL we can't safely dereference Wakee again.  // The thread associated with Wakee may have grabbed the lock and "Wakee" may be  // out-of-scope (non-extant).  Wakee = NULL ;  // Drop the lock  OrderAccess::release\_store\_ptr (&\_owner, NULL) ;  OrderAccess::fence() ; // ST \_owner vs LD in unpark()  if (SafepointSynchronize::do\_call\_back()) {  TEVENT (unpark before SAFEPOINT) ;  }  DTRACE\_MONITOR\_PROBE(contended\_\_exit, this, object(), Self);  Trigger->unpark() ;  // Maintain stats and report events to JVMTI  if (ObjectMonitor::\_sync\_Parks != NULL) {  ObjectMonitor::\_sync\_Parks->inc() ;  }  } |

被唤醒的线程，继续执行monitor的竞争；

##### 总结

本文重点介绍了Synchronized原理以及JVM对Synchronized的优化。简单来说解决三种场景：

1）只有一个线程进入临界区，偏向锁

2）多个线程交替进入临界区，轻量级锁

3）多线程同时进入临界区，重量级锁

## Exchanger的使用

Exchanger 用于两个线程之间交换数据，两个线程要分别拿到 Exchanger 对象，exchanger 会阻塞线程，直到另一个与之交换数据的线程调用 exchange 方法才返回，回导致死锁。

## countdown latch

countdown latch 的读法

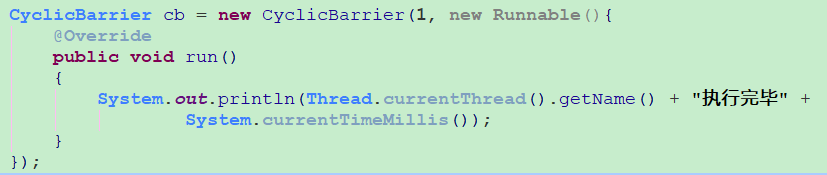
countdown 单词音标是:

英[ˈkaʊntdaʊn] 美[ˈkaʊntˌdaʊn]

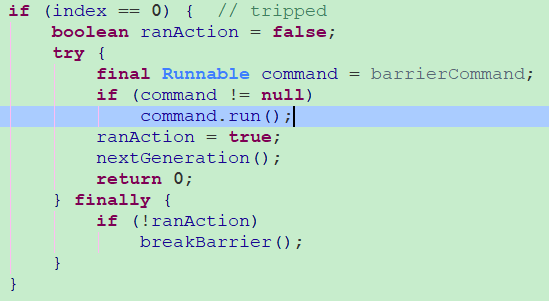
latch 单词音标是:

英[lætʃ] 美[lætʃ]

## cyclicBarrier



构造函数中 1 表示，屏障数量，每次调用await就会减少一个，等于零时，调用signalAll 通知全部运行，第二个参数是一个 Runnable，注意这个不会重新启动一个线程，而是由打破屏障的线程，调用 run() 方法，直接运行。



屏障打破后，会重置屏障，下面还可以继续使用。

## phaser

phaser移相器，他会产生一道一道的屏障，当全部参与者到达第一道屏障后，屏障打开，屏障编号 + 1. 每次调用arrive() 使得到达屏障的参与者数量 +1， 每次调用 register() 使得屏障参与者上限 +1. 每次调用 await() 会阻塞线程，只有全部 arrive() 到达屏障后，被阻带的线程才会unpark() 。

**功能简介：**

* Phaser是jdk1.7提供的类似于CyclicBarrier和CountDownLatch的同步机制。原理跟它们完全不同，phaser没有使用AQS队列
* 它支持更灵活的使用方式：1.使用过程中可以随时注册和注销参与者；2.不同于CyclicBarrier，分离出"到达"和"等待"机制；3.支持结束，默认情况下，当没有参与者的时候Phaser就结束了；4.支持层级Phaser结构；5.提供针对内部状态的监控方法；

**源码分析：**

* **先看一下内部结构：**

**Java代码  [收藏代码](javascript:void())**

1. /\*\*
2. \* 主状态，分为4部分：
3. \*
4. \* 未到达计数  -- 还没有到达栅栏的参与者计数。  (bits  0-15)
5. \* parties     -- 栅栏全部参与者的计数。        (bits 16-31)
6. \* phase       -- 栅栏当前所处的阶段            (bits 32-62)
7. \* terminated  -- 栅栏的结束标记                (bit  63 / sign)
8. \*
9. \* 一个没有注册参与者的phaser的主状态中会有0个参与者计数
10. \* 和1个未到达计数。
11. \*/
12. **private** **volatile** **long** state;
13. **private** **static** **final** **int**  MAX\_PARTIES     = 0xffff;
14. **private** **static** **final** **int**  MAX\_PHASE       = Integer.MAX\_VALUE;
15. **private** **static** **final** **int**  PARTIES\_SHIFT   = 16;
16. **private** **static** **final** **int**  PHASE\_SHIFT     = 32;
17. **private** **static** **final** **int**  UNARRIVED\_MASK  = 0xffff;      // to mask ints
18. **private** **static** **final** **long** PARTIES\_MASK    = 0xffff0000L; // to mask longs
19. **private** **static** **final** **long** TERMINATION\_BIT = 1L << 63;
20. // some special values
21. **private** **static** **final** **int**  ONE\_ARRIVAL     = 1;
22. **private** **static** **final** **int**  ONE\_PARTY       = 1 << PARTIES\_SHIFT;
23. **private** **static** **final** **int**  EMPTY           = 1;

       主状态是Phaser中的一个重要的域，它是一个long型值，内部包含了4部分内容：未到达栅栏的参与者的计数、栅栏的全部参与者计数、当前栅栏处于的阶段(和CyclicBarrier每次用完后会产生一个新的generation的行为类似)、结束标记。

       主状态这样设计(将状态封装到一个原子的long域)可以从两方面提高性能，一个是对状态的编解码简单高效、另一个是可以减小竞争窗口(空间)。

**Java代码  [收藏代码](javascript:void())**

1. /\*\*
2. \* 当前phaser的父phaser, 如果没有父phaser，这个域为null。
3. \*/
4. **private** **final** Phaser parent;
5. /\*\*
6. \* phaser树的根节点. 如果当前phaser不在一棵树内，这个域等于自身。
7. \*/
8. **private** **final** Phaser root;

**Java代码  [收藏代码](javascript:void())**

1. **private** **final** AtomicReference<QNode> evenQ;
2. **private** **final** AtomicReference<QNode> oddQ;
3. **private** AtomicReference<QNode> queueFor(**int** phase) {
4. **return** ((phase & 1) == 0) ? evenQ : oddQ;
5. }

       Phaser中使用Treiber Stack结构来保存等待线程，为了在一些情况下避免竞争，Phaser内部使用了2个Treiber Stack，evenQ和addQ，分别在内部phase为偶数和奇数下交替使用。

       再看下QNode这个类，先看下结构：

**Java代码  [收藏代码](javascript:void())**

1. **static** **final** **class** QNode **implements** ForkJoinPool.ManagedBlocker {
2. **final** Phaser phaser;
3. **final** **int** phase;
4. **final** **boolean** interruptible;
5. **final** **boolean** timed;
6. **boolean** wasInterrupted;
7. **long** nanos;
8. **long** lastTime;
9. **volatile** Thread thread; // nulled to cancel wait
10. QNode next;
11. QNode(Phaser phaser, **int** phase, **boolean** interruptible,
12. **boolean** timed, **long** nanos) {
13. **this**.phaser = phaser;
14. **this**.phase = phase;
15. **this**.interruptible = interruptible;
16. **this**.nanos = nanos;
17. **this**.timed = timed;
18. **this**.lastTime = timed ? System.nanoTime() : 0L;
19. thread = Thread.currentThread();
20. }

       QNode内部结构很简单，就是保存了一些线程等待的相关信息，还有指向下一个QNode的域。这里要注意的是QNode实现了ForkJoinPool.ManagedBlocker，作用就是当包含ForkJoinWorkerThread的QNode阻塞的时候，ForkJoinPool内部会增加一个工作线程来保证并行度(具体的内容可以回头看一下ForkJoin框架的分析文章)。

      继续看下QNode中的方法，首先是isReleasable方法：

**Java代码  [收藏代码](javascript:void())**

1. **public** **boolean** isReleasable() {
2. **if** (thread == **null**)
3. **return** **true**;
4. **if** (phaser.getPhase() != phase) {
5. thread = **null**;
6. **return** **true**;
7. }
8. **if** (Thread.interrupted())
9. wasInterrupted = **true**;
10. **if** (wasInterrupted && interruptible) {
11. thread = **null**;
12. **return** **true**;
13. }
14. **if** (timed) {
15. **if** (nanos > 0L) {
16. **long** now = System.nanoTime();
17. nanos -= now - lastTime;
18. lastTime = now;
19. }
20. **if** (nanos <= 0L) {
21. thread = **null**;
22. **return** **true**;
23. }
24. }
25. **return** **false**;
26. }

       isReleasable方法中的逻辑比较简单：当QNode中的thread为null、或者和phaser的阶段值不相等、或者被中断、或者等待超时，方法都返回true。

       再看下block方法：

**Java代码  [收藏代码](javascript:void())**

1. **public** **boolean** block() {
2. **if** (isReleasable())
3. **return** **true**;
4. **else** **if** (!timed)
5. LockSupport.park(**this**);
6. **else** **if** (nanos > 0)
7. LockSupport.parkNanos(**this**, nanos);
8. **return** isReleasable();
9. }

       block方法就是一个阻塞的过程。

* **通过一些示例来分析下主要功能源码。**

**首先看一个示例，使用方式类似于Count为1的CountDownLatch：**

**Java代码  [收藏代码](javascript:void())**

1. **public** **static** **void** main(String[] args) {
3. **final** Phaser phaser = **new** Phaser(1);
4. **for**(**int** i=0;i<10;i++){
5. phaser.register();
6. **new** Thread(**new** Runnable() {
7. @Override
8. **public** **void** run() {
9. phaser.arriveAndAwaitAdvance();
10. System.out.println(Thread.currentThread() +" start!!!!");
11. }
12. }).start();
13. }
14. **try** {
15. TimeUnit.SECONDS.sleep(5);
16. phaser.arriveAndDeregister();
17. } **catch** (InterruptedException e) {
18. e.printStackTrace();
19. }
21. }

       上面的示例会在main线程启动后，sleep5秒，然后上面启动的10个线程才会打印。

       示例中首先构造了一个Phaser，我们先看下构造方法：

**Java代码  [收藏代码](javascript:void())**

1. /\*\*
2. \* 创建一个没有初始参与者的phaser，默认没有父级phaser，初始
3. \* phase值为0。如果有任何线程想要使用这个phaser，都必须先
4. \* 注册这个phaser。
5. \*/
6. **public** Phaser() {
7. **this**(**null**, 0);
8. }
9. /\*\*
10. \* 创建一个有初始参与者(未到达)数量的phaser，默认没有父级phaser，初始
11. \* phase值为0。
12. \*/
13. **public** Phaser(**int** parties) {
14. **this**(**null**, parties);
15. }
17. **public** Phaser(Phaser parent) {
18. **this**(parent, 0);
19. }
20. /\*\*
21. \* 创建一个有给定父级phaser和初始参与者(未到达)数量的phaser，
22. \* 如果给定的父级phaser不为null，并且给定的参与者数量大于0，
23. \* 当前的子phaser相当于注册了父phaser。
24. \*/
25. **public** Phaser(Phaser parent, **int** parties) {
26. //parties不能超过65535
27. **if** (parties >>> PARTIES\_SHIFT != 0)
28. **throw** **new** IllegalArgumentException("Illegal number of parties");
29. **int** phase = 0;
30. **this**.parent = parent;
31. **if** (parent != **null**) {
32. //如果父级Phaser不为空。
33. **final** Phaser root = parent.root;
34. **this**.root = root;
35. //共享父级的线程等待队列。
36. **this**.evenQ = root.evenQ;
37. **this**.oddQ = root.oddQ;
38. **if** (parties != 0)
39. //如果当前phaser的参与者不为0，那么注册一个参与者到父级，注意这里是一个。
40. phase = parent.doRegister(1);
41. }
42. **else** {
43. //如果父级为空。
44. //root就是自身，
45. **this**.root = **this**;
46. **this**.evenQ = **new** AtomicReference<QNode>();
47. **this**.oddQ = **new** AtomicReference<QNode>();
48. }
49. **this**.state = (parties == 0) ? (**long**)EMPTY :
50. ((**long**)phase << PHASE\_SHIFT) |
51. ((**long**)parties << PARTIES\_SHIFT) |
52. ((**long**)parties);
53. }

       构造方法很简单，只要注意两点：1.当有父级Phaser的时候，子Phaser会共享父Phaser的线程等待队列。如果子Phaser的参与者不为0，那么会以一个参与者的身份注册到父Phaser上，注意是一个!!；2.最后初始化state的时候，如果没有参与者，state就是EMPTY，等于1。

       示例中接下来会在每次新建线程之前调用register方法来注册参与者，看下这个方法：

**Java代码  [收藏代码](javascript:void())**

1. /\*\*
2. \* 添加一个新的未到达的参与者到当前phaser。如果当前正在onAdvance方法，
3. \* 的执行过程中，这个方法会等待其完成再返回。如果当前phaser有父phaser，
4. \* 并且当前phaser之前没有注册的参与者，phaser会注册到父phaser上。
5. \* 如果当前phaser结束了，那么方法不会产生任何作用，并返回一个负数。
6. \*/
7. **public** **int** register() {
8. **return** doRegister(1);
9. }

       register内部调用的是doRegister，看下这个方法：

**Java代码  [收藏代码](javascript:void())**

1. **private** **int** doRegister(**int** registrations) {
2. //调整主状态，将给定的数值加到总参与者和未到达参数者数量上。
3. **long** adj = ((**long**)registrations << PARTIES\_SHIFT) | registrations;
4. **final** Phaser parent = **this**.parent;
5. **int** phase;
6. **for** (;;) {
7. **long** s = state;
8. **int** counts = (**int**)s;
9. **int** parties = counts >>> PARTIES\_SHIFT;
10. **int** unarrived = counts & UNARRIVED\_MASK;
11. **if** (registrations > MAX\_PARTIES - parties)
12. //注册的参与者数量和已存在的参与者数量加起来不能超过最大参与者数量。
13. **throw** **new** IllegalStateException(badRegister(s));
14. **else** **if** ((phase = (**int**)(s >>> PHASE\_SHIFT)) < 0)
15. //如果phaser已经结束，那么直接退出循环。
16. **break**;
17. **else** **if** (counts != EMPTY) {
18. //如果不是第一个注册。
19. **if** (parent == **null** || reconcileState() == s) {
20. **if** (unarrived == 0)
21. // 如果当前未到达数量为0，说明需要进入下一阶段了，这里要等待一下root进入下一阶段。
22. root.internalAwaitAdvance(phase, **null**);
23. //否则原子更新主状态。
24. **else** **if** (UNSAFE.compareAndSwapLong(**this**, stateOffset,
25. s, s + adj))
26. **break**;
27. }
28. }
29. **else** **if** (parent == **null**) {
30. // 第一个root注册(没有父级)。
31. // 算出下一个主状态。
32. **long** next = ((**long**)phase << PHASE\_SHIFT) | adj;
33. // 原子更新主状态。
34. **if** (UNSAFE.compareAndSwapLong(**this**, stateOffset, s, next))
35. **break**;
36. }
37. **else** {
38. //第一个子phaser的注册，需要加锁。
39. **synchronized** (**this**) {
40. **if** (state == s) {               //检测一下状态有没有变化。
41. parent.doRegister(1);       //由于是第一次注册，所以需要向父类注册一下。
42. **do** {                        //更新到下一个主状态。
43. phase = (**int**)(root.state >>> PHASE\_SHIFT);
44. // assert phase < 0 || (int)state == EMPTY;
45. } **while** (!UNSAFE.compareAndSwapLong
46. (**this**, stateOffset, state,
47. ((**long**)phase << PHASE\_SHIFT) | adj));
48. **break**;
49. }
50. }
51. }
52. }
53. **return** phase;
54. }

       总结一下doRegister方法：

              1.当前不是第一个注册者(参与者)。如果当前主状态中未到达数量为0，说明参与者已经全部到达栅栏，当前Phaser正在进入下一阶段过程中，需要等待这一过程完成(可能会阻塞)；否则会原子更新当前的主状态，加一个总参与者数量和一个未到达参与者数量。(过程中如果parent不为null，需要调用reconcileState调整一下当前主状态，和root的主状态保持一致)

              2.当前是第一个注册者且当前Phaser没有父级Phaser。直接原子更新当前主状态，加一个总参与者数量和一个未到达参与者数量。

              3.当前是第一个注册者且当前Phaser有父级Phaser。需要加锁操作，首先向父级Phaser注册一个参与者，然后原子更新主状态，加一个总参与者数量和一个未到达参与者数量。

       doRegister方法中会调用reconcileState来调整状态，看下这个方法实现：

**Java代码  [收藏代码](javascript:void())**

1. **private** **long** reconcileState() {
2. **final** Phaser root = **this**.root;
3. **long** s = state;
4. **if** (root != **this**) {
5. **int** phase, u, p;
6. // CAS root phase with current parties; possibly trip unarrived
7. **while** ((phase = (**int**)(root.state >>> PHASE\_SHIFT)) !=
8. (**int**)(s >>> PHASE\_SHIFT) &&
9. !UNSAFE.compareAndSwapLong
10. (**this**, stateOffset, s,
11. s = (((**long**)phase << PHASE\_SHIFT) |
12. (s & PARTIES\_MASK) |
13. ((p = (**int**)s >>> PARTIES\_SHIFT) == 0 ? EMPTY :
14. (u = (**int**)s & UNARRIVED\_MASK) == 0 ? p : u))))
15. s = state;
16. }
17. **return** s;
18. }

       可见，这个方法要做的事情就是将当前Phaser和root Phaser的phase值调整为一致的。

       doRegister方法中，如果当前Phaser正在进入下一阶段过程中，需要等待这个过程完成，会调用internalAwaitAdvance方法，看下这个方法：

**Java代码  [收藏代码](javascript:void())**

1. /\*\* cpu核数 \*/
2. **private** **static** **final** **int** NCPU = Runtime.getRuntime().availableProcessors();
3. /\*\*
4. \* 单个参与者阻塞等待栅栏进入下一个阶段之前的自旋次数。
5. \* 在多核处理器下，一次性完全的阻塞和唤醒一大批线程通常比较慢，
6. \* 所以我们这里使用了一个可调整的自旋次数值在避免这种情况。
7. \* 当一个参与者线程在internalAwaitAdvance方法中阻塞之前发现了
8. \* 其他到达的线程，并且有cpu资源可用，那么这个参与者线程会在阻塞
9. \* 之前自旋SPINS\_PER\_ARRIVAL或者更多次。
10. \*/
11. **static** **final** **int** SPINS\_PER\_ARRIVAL = (NCPU < 2) ? 1 : 1 << 8;
13. **private** **int** internalAwaitAdvance(**int** phase, QNode node) {
14. releaseWaiters(phase-1);          // 清空不用的等待线程队列(Treiber Stack)。
15. **boolean** queued = **false**;           // 入队标识。
16. **int** lastUnarrived = 0;            // 用于在发生变化时增加自旋次数。
17. **int** spins = SPINS\_PER\_ARRIVAL;
18. **long** s;
19. **int** p;
20. **while** ((p = (**int**)((s = state) >>> PHASE\_SHIFT)) == phase) {
21. **if** (node == **null**) {
22. **int** unarrived = (**int**)s & UNARRIVED\_MASK;
23. //如果未到达参与者数量发生了变化，且变化后的未到达数量小于cpu核数，需要增加自旋次数。
24. **if** (unarrived != lastUnarrived &&
25. (lastUnarrived = unarrived) < NCPU)
26. spins += SPINS\_PER\_ARRIVAL;
27. //获取并清除当前线程中断标记。
28. **boolean** interrupted = Thread.interrupted();
29. **if** (interrupted || --spins < 0) {
30. //如果当前线程被中断，或者自旋次数用完。创建一个(不可中断的)节点。
31. node = **new** QNode(**this**, phase, **false**, **false**, 0L);
32. node.wasInterrupted = interrupted;
33. }
34. }
35. **else** **if** (node.isReleasable()) // done or aborted
36. **break**;
37. **else** **if** (!queued) {           // 将节点加入队列首部。
38. AtomicReference<QNode> head = (phase & 1) == 0 ? evenQ : oddQ;
39. QNode q = node.next = head.get();
40. **if** ((q == **null** || q.phase == phase) &&
41. (**int**)(state >>> PHASE\_SHIFT) == phase) // avoid stale enq
42. queued = head.compareAndSet(q, node);
43. }
44. **else** {
45. **try** {
46. //阻塞等待。
47. ForkJoinPool.managedBlock(node);
48. } **catch** (InterruptedException ie) {
49. node.wasInterrupted = **true**;
50. }
51. }
52. }
53. **if** (node != **null**) {
54. **if** (node.thread != **null**)
55. node.thread = **null**;       // avoid need for unpark()
56. **if** (node.wasInterrupted && !node.interruptible)
57. //不可中断模式下要传递中断。
58. Thread.currentThread().interrupt();
59. **if** (p == phase && (p = (**int**)(state >>> PHASE\_SHIFT)) == phase)
60. **return** abortWait(phase); // possibly clean up on abort
61. }
62. releaseWaiters(phase);
63. **return** p;
64. }

       internalAwaitAdvance中的主要逻辑过程就是当前(参与者)线程等待Phaser进入下一个阶段(就是phase值变化)。有些细节需要注意一下。

       如果传入的node为null，过程如下：

       第1步，等待分两个阶段，自旋等待和阻塞等待，首先自旋给定的次数，如果自旋过程中未到达参与者数量发生变化，且变化后的为未到达参与者数量小于CPU处理器核数，那么自选次数会增加SPINS\_PER\_ARRIVAL次。如果自旋次数用完后或者当前线程被中断了，那么会创建一个不可中断模式的节点，节点中保存当前线程及其他信息。

       第2步，将上面创建的节点加入线程等待队列的首部(类似于压栈，因为线程等待队列就是Treiber Stack)。

       第3步，当前线程开始阻塞等待。

       第4步，当前线程被唤醒后，如果是不可中断模式的节点，需要向上层传递中断状态；如果当前phaser还是没有进入下一阶段，那么调用abortWait，做放弃等待操作。

       如果传入的node不为null，过程和上面类似，只是没有第1步。

       注意整个internalAwaitAdvance过程的前后都会清空一下当前不用的等待线程队列(两个奇偶队列交替使用)，看下这个方法：

**Java代码  [收藏代码](javascript:void())**

1. **private** **void** releaseWaiters(**int** phase) {
2. QNode q;   // first element of queue
3. Thread t;  // its thread
4. AtomicReference<QNode> head = (phase & 1) == 0 ? evenQ : oddQ;
5. **while** ((q = head.get()) != **null** &&
6. q.phase != (**int**)(root.state >>> PHASE\_SHIFT)) {
7. **if** (head.compareAndSet(q, q.next) &&
8. (t = q.thread) != **null**) {
9. q.thread = **null**;
10. LockSupport.unpark(t);
11. }
12. }
13. }

       逻辑很简单，就是把等待线程队列里面的节点都移除了，如果节点有线程的话，将线程唤醒。

       internalAwaitAdvance中还可能会调用一个放弃等待的abortWait方法，看下：

**Java代码  [收藏代码](javascript:void())**

1. **private** **int** abortWait(**int** phase) {
2. AtomicReference<QNode> head = (phase & 1) == 0 ? evenQ : oddQ;
3. **for** (;;) {
4. Thread t;
5. QNode q = head.get();
6. **int** p = (**int**)(root.state >>> PHASE\_SHIFT);
7. **if** (q == **null** || ((t = q.thread) != **null** && q.phase == p))
8. **return** p;
9. **if** (head.compareAndSet(q, q.next) && t != **null**) {
10. q.thread = **null**;
11. LockSupport.unpark(t);
12. }
13. }
14. }

       这个方法和releaseWaiters方法有些区别：

       releaseWaiters方法是用来清空当前不适用的等待线程队列的；abortWait方法是将当前正在使用的队列中由于超时或者中断不在等待当前phaser的下一阶段的节点移除。

       再回头看我们的示例，for循环中新建的线程在运行时会先调用arriveAndAwaitAdvance方法，然后会在这个方法上等待，直到主线程调用了arriveAndDeregister。看下arriveAndAwaitAdvance这个方法：

**Java代码  [收藏代码](javascript:void())**

1. **public** **int** arriveAndAwaitAdvance() {
2. **final** Phaser root = **this**.root;
3. **for** (;;) {
4. //获取主状态
5. **long** s = (root == **this**) ? state : reconcileState();
6. //获取phase值
7. **int** phase = (**int**)(s >>> PHASE\_SHIFT);
8. **int** counts = (**int**)s;
9. //获取当前未到达参与者计数，就是之前的未到达计数减1。
10. **int** unarrived = (counts & UNARRIVED\_MASK) - 1;
11. **if** (phase < 0)
12. **return** phase; //如果当前phaser已经结束，退出。
13. **else** **if** (counts == EMPTY || unarrived < 0) {
14. **if** (reconcileState() == s)
15. **throw** **new** IllegalStateException(badArrive(s)); //非法状态。
16. }
17. //主状态中未到达参与者的计数减1。
18. **else** **if** (UNSAFE.compareAndSwapLong(**this**, stateOffset, s,
19. s -= ONE\_ARRIVAL)) {
20. **if** (unarrived != 0)
21. //如果还有未到达的参与者，等待。
22. **return** root.internalAwaitAdvance(phase, **null**);
23. **if** (root != **this**)
24. //如果当前是子级Phaser，要等待父级进入下一阶段。
25. **return** parent.arriveAndAwaitAdvance();
26. **long** n = s & PARTIES\_MASK;  // base of next state
27. //这里算出来的是总参与者的数量。
28. **int** nextUnarrived = (**int**)n >>> PARTIES\_SHIFT;
29. //调用onAdvance方法。
30. **if** (onAdvance(phase, nextUnarrived))
31. //如果onAdvance方法返回true，给主状态中设置结束标记。
32. n |= TERMINATION\_BIT;
33. **else** **if** (nextUnarrived == 0)
34. //如果总参与者数量变为0，那么将主状态设置为没有参与者的特殊状态。
35. n |= EMPTY;
36. **else**
37. //否则，重置未到达参与者数量。
38. n |= nextUnarrived;
39. //算出下一个phase值。
40. **int** nextPhase = (phase + 1) & MAX\_PHASE;
41. //设置到主状态上。
42. n |= (**long**)nextPhase << PHASE\_SHIFT;
43. //原子更新主状态。
44. **if** (!UNSAFE.compareAndSwapLong(**this**, stateOffset, s, n))
45. //如果发生竞争，返回phase值，如果当前phaser结束，返回负数。
46. **return** (**int**)(state >>> PHASE\_SHIFT); // terminated
47. //清空上一阶段使用的线程等待队列。
48. releaseWaiters(phase);
49. //最后返回上面算出来的nextPhase值。
50. **return** nextPhase;
51. }
52. }
53. }

       arriveAndAwaitAdvance方法中的主要逻辑如下：

              首先将主状态中的未到达参与者数量减1，然后判断未到达参与者数量是否为0。如果不为0，当前线程会等待其他参与者到来；如果为0，说明当前(线程)是最后一个参与者，那么会继续算出下一个阶段的主状态，然后更新到Phaser中。计算过程中会通过调用onAdvance方法，判断当前Phaser是否结束，还会重置未到达参与者数量等。

       看一下arriveAndAwaitAdvance方法中调用的onAdvance方法：

**Java代码  [收藏代码](javascript:void())**

1. **protected** **boolean** onAdvance(**int** phase, **int** registeredParties) {
2. **return** registeredParties == 0;
3. }

       onAdvance方法的作用就是来控制Phaser是否结束，默认行为是当总参与者数量为0时，Phaser就结束了。具体使用时可以覆盖这个方法，实现合适的策略来控制Phaser的结束时机。

       回到示例的最后，主线程休眠5秒中，然后调用了arriveAndDeregister方法，看下这个方法：

**Java代码  [收藏代码](javascript:void())**

1. **public** **int** arriveAndDeregister() {
2. **return** doArrive(**true**);
3. }

       这个方法表示一个参与者到达栅栏，并且将自己从phaser上注销。内部调用了doArrive方法：

**Java代码  [收藏代码](javascript:void())**

1. **private** **int** doArrive(**boolean** deregister) {
2. **int** adj = deregister ? ONE\_ARRIVAL|ONE\_PARTY : ONE\_ARRIVAL;
3. **final** Phaser root = **this**.root;
4. **for** (;;) {
5. **long** s = (root == **this**) ? state : reconcileState();
6. **int** phase = (**int**)(s >>> PHASE\_SHIFT);
7. **int** counts = (**int**)s;
8. **int** unarrived = (counts & UNARRIVED\_MASK) - 1;
9. **if** (phase < 0)
10. **return** phase;
11. **else** **if** (counts == EMPTY || unarrived < 0) {
12. **if** (root == **this** || reconcileState() == s)
13. **throw** **new** IllegalStateException(badArrive(s));
14. }
15. **else** **if** (UNSAFE.compareAndSwapLong(**this**, stateOffset, s, s-=adj)) {
16. **if** (unarrived == 0) {
17. **long** n = s & PARTIES\_MASK;  // base of next state
18. **int** nextUnarrived = (**int**)n >>> PARTIES\_SHIFT;
19. **if** (root != **this**)
20. //这里注意下：如果当前子phaser中没有参与者了，就要于从父phaser中将当前子phaser注销。
21. **return** parent.doArrive(nextUnarrived == 0);
22. **if** (onAdvance(phase, nextUnarrived))
23. n |= TERMINATION\_BIT;
24. **else** **if** (nextUnarrived == 0)
25. n |= EMPTY;
26. **else**
27. n |= nextUnarrived;
28. n |= (**long**)((phase + 1) & MAX\_PHASE) << PHASE\_SHIFT;
29. UNSAFE.compareAndSwapLong(**this**, stateOffset, s, n);
30. releaseWaiters(phase);
31. }
32. **return** phase;
33. }
34. }
35. }

       可见，这个方法中的逻辑和arriveAndAwaitAdvance方法类似，区别是如果参数deregister为true，会在主状态中减去一个参与者计数。

       示例中使用到的方法都分析过了，再回头看示例，会发现逻辑细节也清晰了：

*首先，构造了有1个参与者的Phaser。*

*其次，开启10个线程，每个线程作为1个参与者注册到上面创建的Phaser上。然后启动线程，在线程执行具体逻辑前等待其他参与者到达栅栏(arriveAndAwaitAdvance)。*

*最后，主线程到达栅栏并将自己注销。这时，所有的参与者(一共11个)都已经到达栅栏，上面的10个参与者(线程)就可以通过栅栏，执行具体逻辑了。*

**再看一个示例，使用方式类似于Count为N的CountDownLatch：**

**Java代码  [收藏代码](javascript:void())**

1. **public** **static** **void** main(String[] args) {
2. **final** Phaser phaser = **new** Phaser();
3. **for**(**int** i=0;i<10;i++){
4. phaser.register();
5. **new** Thread(**new** Runnable() {
6. @Override
7. **public** **void** run() {
8. **try** {
9. TimeUnit.SECONDS.sleep(**new** Random().nextInt(5));
10. System.out.println(Thread.currentThread() +" is ready!");
11. phaser.arriveAndAwaitAdvance();
12. System.out.println(Thread.currentThread() +" start!!!!");
13. } **catch** (InterruptedException e) {
14. e.printStackTrace();
15. }
16. }
17. }).start();
18. }
19. }

       这个示例要更简单一些，就是等待所有的线程都ready后，然后一起start。涉及到的Phaser方法都分析过了。

**继续看一个示例，和第一个示例差不多，但有一点区别：第一个示例中线程只执行了一次逻辑，现在要求线程执行若干次逻辑：**

**Java代码  [收藏代码](javascript:void())**

1. **public** **static** **void** main(String[] args) {
2. **final** Phaser phaser = **new** Phaser(1){
3. **protected** **boolean** onAdvance(**int** phase, **int** registeredParties) {
4. System.out.println("now phase is " + phase);
5. **return** phase >= 5 || registeredParties == 0;
6. }
7. };
8. **for**(**int** i=0;i<5;i++){
9. phaser.register();
10. **final** **int** time = i;
11. **new** Thread(**new** Runnable() {
12. @Override
13. **public** **void** run() {
14. **do**{
15. phaser.arriveAndAwaitAdvance();
16. System.out.println(Thread.currentThread() +" start-"+time+"!!!!");
17. }**while**(!phaser.isTerminated());
18. }
19. }).start();
20. }
21. **try** {
22. TimeUnit.SECONDS.sleep(5);
23. System.out.println("arriveAndDeregister...");
24. phaser.arriveAndDeregister();
25. } **catch** (InterruptedException e) {
26. e.printStackTrace();
27. }
28. }

       本示例和第一个示例区别只是覆盖了onAdvance方法，加上了阶段限制逻辑：当phase大于等于5时，Phaser就结束了，也就是说，线程里面的逻辑会执行6次(phase从0开始)。

       前面提高过，第一个示例对Phaser的使用方式类似于Count为1的CountDownLatch；而本示例中的Phaser在第一次使用时相当于Count为1的CountDownLatch，当主线程到达并注销自己(相当于countDown)，前面启动的子线程就会(通过栅栏)执行逻辑了。但随后的使用又相当于是一个CyclicBarrier，示例中第一次的参与者包括主线程，一共是6个。主线程在Phaser中注销后，后面就只有5个参与者，所以每当这个5个线程都达到栅栏时，它们会一起通过，所以相当于一个CyclicBarrier了。同时我们可以看到，onAdvance中也可以定制类似于CyclicBarrier的barrierAction的逻辑，我们这里是打印了一个语句。

       示例输出如下：

**Java代码  [收藏代码](javascript:void())**

1. arriveAndDeregister...
2. now phase is 0
3. Thread[Thread-4,5,main] start-4!!!!
4. Thread[Thread-1,5,main] start-1!!!!
5. Thread[Thread-0,5,main] start-0!!!!
6. Thread[Thread-2,5,main] start-2!!!!
7. Thread[Thread-3,5,main] start-3!!!!
8. now phase is 1
9. Thread[Thread-0,5,main] start-0!!!!
10. Thread[Thread-4,5,main] start-4!!!!
11. Thread[Thread-2,5,main] start-2!!!!
12. Thread[Thread-3,5,main] start-3!!!!
13. Thread[Thread-1,5,main] start-1!!!!
14. now phase is 2
15. Thread[Thread-1,5,main] start-1!!!!
16. Thread[Thread-3,5,main] start-3!!!!
17. Thread[Thread-2,5,main] start-2!!!!
18. Thread[Thread-4,5,main] start-4!!!!
19. Thread[Thread-0,5,main] start-0!!!!
20. now phase is 3
21. Thread[Thread-0,5,main] start-0!!!!
22. Thread[Thread-3,5,main] start-3!!!!
23. Thread[Thread-1,5,main] start-1!!!!
24. Thread[Thread-2,5,main] start-2!!!!
25. Thread[Thread-4,5,main] start-4!!!!
26. now phase is 4
27. Thread[Thread-4,5,main] start-4!!!!
28. Thread[Thread-0,5,main] start-0!!!!
29. Thread[Thread-1,5,main] start-1!!!!
30. Thread[Thread-3,5,main] start-3!!!!
31. Thread[Thread-2,5,main] start-2!!!!
32. now phase is 5
33. Thread[Thread-2,5,main] start-2!!!!
34. Thread[Thread-3,5,main] start-3!!!!
35. Thread[Thread-4,5,main] start-4!!!!
36. Thread[Thread-0,5,main] start-0!!!!
37. Thread[Thread-1,5,main] start-1!!!!

* 最后，看一下示例中未涉及到的代码。

**Java代码  [收藏代码](javascript:void())**

1. **private** **static** **int** unarrivedOf(**long** s) {
2. **int** counts = (**int**)s;
3. **return** (counts == EMPTY) ? 0 : counts & UNARRIVED\_MASK;
4. }
5. **private** **static** **int** partiesOf(**long** s) {
6. **return** (**int**)s >>> PARTIES\_SHIFT;
7. }
8. **private** **static** **int** phaseOf(**long** s) {
9. **return** (**int**)(s >>> PHASE\_SHIFT);
10. }
11. **private** **static** **int** arrivedOf(**long** s) {
12. **int** counts = (**int**)s;
13. **return** (counts == EMPTY) ? 0 :
14. (counts >>> PARTIES\_SHIFT) - (counts & UNARRIVED\_MASK);
15. }

       这组方法可以从主状态中获取各个状态，是一组"拆包"方法。在具体功能方法内部可能不会调用这些方法，而是手工内联进去。

**Java代码  [收藏代码](javascript:void())**

1. **private** String badArrive(**long** s) {
2. **return** "Attempted arrival of unregistered party for " +
3. stateToString(s);
4. }
6. **private** String badRegister(**long** s) {
7. **return** "Attempt to register more than " +
8. MAX\_PARTIES + " parties for " + stateToString(s);
9. }
10. **private** String stateToString(**long** s) {
11. **return** **super**.toString() +
12. "[phase = " + phaseOf(s) +
13. " parties = " + partiesOf(s) +
14. " arrived = " + arrivedOf(s) + "]";
15. }

       这badArrice和badRegister方法用于在逻辑中出现非法参与注册情况时，提供提示消息。stateToString方法将state转化为易懂的字符串形式，用于支持前面两个方法和toString方法。

**Java代码  [收藏代码](javascript:void())**

1. **public** **int** bulkRegister(**int** parties) {
2. **if** (parties < 0)
3. **throw** **new** IllegalArgumentException();
4. **if** (parties == 0)
5. **return** getPhase();
6. **return** doRegister(parties);
7. }
8. **public** **final** **int** getPhase() {
9. **return** (**int**)(root.state >>> PHASE\_SHIFT);
10. }

       相对于register的批量注册方法。

**Java代码  [收藏代码](javascript:void())**

1. **public** **int** arrive() {
2. **return** doArrive(**false**);
3. }

       和arriveAndDeregister类似，只是不会注销当前参与者。

**Java代码  [收藏代码](javascript:void())**

1. **public** **int** awaitAdvance(**int** phase) {
2. **final** Phaser root = **this**.root;
3. **long** s = (root == **this**) ? state : reconcileState();
4. **int** p = (**int**)(s >>> PHASE\_SHIFT);
5. **if** (phase < 0)
6. **return** phase;
7. **if** (p == phase)
8. **return** root.internalAwaitAdvance(phase, **null**);
9. **return** p;
10. }

       等待phase值表示的阶段，如果当前phase和给定的phase不一致，直接返回当前的phase值。

**Java代码  [收藏代码](javascript:void())**

1. **public** **int** awaitAdvanceInterruptibly(**int** phase)
2. **throws** InterruptedException {
3. **final** Phaser root = **this**.root;
4. **long** s = (root == **this**) ? state : reconcileState();
5. **int** p = (**int**)(s >>> PHASE\_SHIFT);
6. **if** (phase < 0)
7. **return** phase;
8. **if** (p == phase) {
9. QNode node = **new** QNode(**this**, phase, **true**, **false**, 0L);
10. p = root.internalAwaitAdvance(phase, node);
11. **if** (node.wasInterrupted)
12. **throw** **new** InterruptedException();
13. }
14. **return** p;
15. }

       awaitAdvance方法逻辑一样，但支持中断。

**Java代码  [收藏代码](javascript:void())**

1. **public** **int** awaitAdvanceInterruptibly(**int** phase,
2. **long** timeout, TimeUnit unit)
3. **throws** InterruptedException, TimeoutException {
4. **long** nanos = unit.toNanos(timeout);
5. **final** Phaser root = **this**.root;
6. **long** s = (root == **this**) ? state : reconcileState();
7. **int** p = (**int**)(s >>> PHASE\_SHIFT);
8. **if** (phase < 0)
9. **return** phase;
10. **if** (p == phase) {
11. QNode node = **new** QNode(**this**, phase, **true**, **true**, nanos);
12. p = root.internalAwaitAdvance(phase, node);
13. **if** (node.wasInterrupted)
14. **throw** **new** InterruptedException();
15. **else** **if** (p == phase)
16. **throw** **new** TimeoutException();
17. }
18. **return** p;
19. }

       awaitAdvance方法逻辑一样，但支持中断和超时。

**Java代码  [收藏代码](javascript:void())**

1. **public** **void** forceTermination() {
2. // Only need to change root state
3. **final** Phaser root = **this**.root;
4. **long** s;
5. **while** ((s = root.state) >= 0) {
6. //给主状态中添加结束标记。
7. **if** (UNSAFE.compareAndSwapLong(root, stateOffset,
8. s, s | TERMINATION\_BIT)) {
9. //清空奇偶线程等待队列。
10. releaseWaiters(0);
11. releaseWaiters(1);
12. **return**;
13. }
14. }
15. }

       强制结束Phaser。

       最后看一些支持监控的方法：

**Java代码  [收藏代码](javascript:void())**

1. **public** **final** **int** getPhase() {
2. **return** (**int**)(root.state >>> PHASE\_SHIFT);
3. }
5. **public** **int** getRegisteredParties() {
6. **return** partiesOf(state);
7. }
9. **public** **int** getArrivedParties() {
10. **return** arrivedOf(reconcileState());
11. }
13. **public** **int** getUnarrivedParties() {
14. **return** unarrivedOf(reconcileState());
15. }
17. **public** Phaser getParent() {
18. **return** parent;
19. }
21. **public** Phaser getRoot() {
22. **return** root;
23. }
25. **public** **boolean** isTerminated() {
26. **return** root.state < 0L;
27. }

       方法都非常简单，不啰嗦了。

## 线程池

创建线程是消耗系统资源的，create\_thread() 系统调用会陷入内核态，创建和销毁线程的代价非常大，如果频繁地为任务创建和销毁线程，显然是不合适的。

# 动态代理

## Jdkd动态代理

Jdk 动态代理，需要借助于 jdk 的 java.lang.reflect 工具包，完成字节码的自动生成。其中 Proxy 类的

**Proxy.newProxyInstance()**

方法，返回一个代理对象。该函数需要一个类型为 InvocationHandler 的参数，InvocationHandler 是一个接口，它的 invoke() 方法，实现了对被代理对象的增强。如下是ProxyHelper 实现了 InvocationHandler 接口，另外提供了一个方法，让我们获取一个代理类。

|  |
| --- |
| package com.wdyspring;  import java.lang.reflect.InvocationHandler; import java.lang.reflect.Method; import java.lang.reflect.Proxy;  /\*\*  \* @创建人：吴德运  \* @创建时间 2019/1/16 17:40  \* @当前项目 spring  \* @描述：  \*/ public class ProxyHelper implements InvocationHandler {  private Object target; //被代理对象   /\*\*  \* 这个函数是被代理类隐式调用的，我们不会直接使用它  \* @param proxy 代理类对象，这里没被使用  \* @param method target 中的方法  \* @param args target 中方法的参数  \* @throws Throwable  \*/  @Override  public Object invoke(Object *proxy*, Method *method*, Object[] *args*) throws Throwable  {  System.*out*.println("增强实现开始");  Object object = *method*.invoke(target, *args*);  System.*out*.println("增强实现结束");  return object;  }   /\*\*  \* 返回 target 的代理类,这个代理类跟 target 的接口类型是一样的。注意返回的类型不是ProxyHelper  \* ProxyHelper 的 invoke()方法会被代理类调用，从而实现对 target 的增强。  \*/  public Object getProxy(Object *target*)  {  this.target = *target*;  return Proxy.newProxyInstance(*target*.getClass().getClassLoader(), *target*.getClass().getInterfaces(), this);  } } |

使用如下，调用getProxy 方法，获取 Cat 的代理类，该代理类跟 Cat 的接口类型是一样的，不是 Cat ，这点要注意，因为 newProxyInstance() 函数的参数是接口数组，所以代理类是接口的实现，这也要求 jdk 动态代理，必须是基于接口的。

|  |
| --- |
| **IAnnimal** annimal = (**IAnnimal**) **new** ProxyHelper().getProxy(**new** Cat());  annimal.say();  //这个是错误的，Cat annimal = (Cat) new ProxyHelper().getProxy(new Cat()); |

## Jdk源码分析

最终会到 Proxy 类的 generateClassFile 方法中，添加三个方法，根据被代理类的接口，把被代理类都增强了。

|  |
| --- |
| private byte[] generateClassFile() {  //增加 hashcode、equals、toString方法  addProxyMethod(hashCodeMethod, Object.class);  addProxyMethod(equalsMethod, Object.class);  addProxyMethod(toStringMethod, Object.class);  //增加接口方法  for (Class<?> intf : interfaces) {  for (Method m : intf.getMethods()) {  addProxyMethod(m, intf);  }  } |

## cjlib动态代理

cjlib使用的是对方法建索引，直接找到相关方法，jdk动态代理，生成字节码文件速度更快，运行速度比cjlib可能慢一些，但是不确定。

 cglib是针对类来实现代理的，原理是对指定的业务类生成一个子类，并覆盖其中业务方法实现代理。因为采用的是继承，所以不能对final修饰的类进行代理。

|  |
| --- |
| public class CglibHelper implements MethodInterceptor {  private Object target;//被代理对象   //获取代理类对象  public Object getProxyObject(Object target) {  this.target = target; //给业务对象赋值  Enhancer enhancer = new Enhancer(); //创建加强器，用来创建动态代理类  enhancer.setSuperclass(this.target.getClass()); //为下面生成的代理类指定父类   //设置回调：对于代理类上所有方法的调用，都会调用CallBack，而Callback则需要实现intercept()方法进行拦  enhancer.setCallback(this);    // 创建动态代理类对象并返回  return enhancer.create();  }  // 实现回调方法  public Object intercept(Object obj, Method method, Object[] args, MethodProxy proxy) throws Throwable {  System.out.println("预处理——————");  proxy.invokeSuper(obj, args); //调用业务类（父类中）的方法  System.out.println("调用后操作——————");  return null;  } } |

使用如下，cglib使用的是继承，不需要接口

|  |
| --- |
| Cat cat = new Cat(); Cat catProxy = (Cat)new CglibHelper().getProxyObject(cat); catProxy.say(); |

## 源码分析

然后调用父类AbstractClassGenerator方法：create()直接生成超类的子类：

# Spring 源码

　spring的jar包只有20个左右，每个都有相应的功能，一个jar还可能依赖了若干其他jar，所以，搞清楚它们之间的关系，配置maven依赖就可以简洁明了，下面举个例子，要在普通java工程使用spring框架，需要哪些jar呢？只要一个

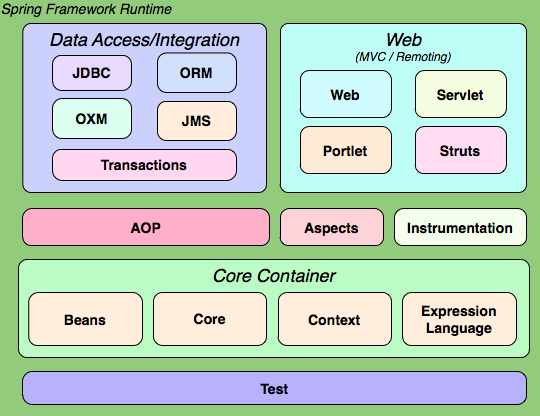
|  |  |
| --- | --- |
| 1  2  3  4  5 | <dependency>      <groupId>org.springframework</groupId>      <artifactId>spring-context</artifactId>      <version>3.2.17.RELEASE</version>  </dependency> |

　　那要在web工程中引入spring mvc呢？也只要配置一个依赖

|  |  |
| --- | --- |
| 1  2  3  4  5 | <dependency>      <groupId>org.springframework</groupId>      <artifactId>spring-webmvc</artifactId>      <version>3.2.17.RELEASE</version>  </dependency> |

　　为什么可以这样配置？接下来我们以spring 3.2.17.RELEASE版本为例，介绍spring框架结构，spring 4稍有不同，将在最后介绍

　　spring官网给出了一张spring3的结构图



　　图中将spring分为5个部分：core、aop、data access、web、test，图中每个圆角矩形都对应一个jar，如果在maven中配置，所有这些jar的“groupId”都是“org.springframework”，每个jar有一个不同的“artifactId”，另外，“instrumentation”有两个jar，还有一个“spring-context-support”图中没有列出，所以spring3的jar包一共是19个

　　下面介绍这5个部分的jar以及依赖关系

core

　　core部分包含4个模块

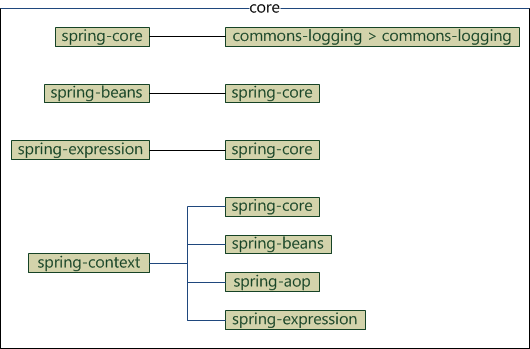
spring-core：依赖注入IoC与DI的最基本实现

spring-beans：Bean工厂与bean的装配

spring-context：spring的context上下文即IoC容器

spring-expression：spring表达式语言

　　它们的完整依赖关系



　　因为spring-core依赖了commons-logging，而其他模块都依赖了spring-core，所以整个spring框架都依赖了commons-logging，如果有自己的日志实现如log4j，可以排除对commons-logging的依赖，没有日志实现而排除了commons-logging依赖，编译报错

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | <dependency>      <groupId>org.springframework</groupId>      <artifactId>spring-context</artifactId>      <version>3.2.17.RELEASE</version>      <exclusions>          <exclusion>              <groupId>commons-logging</groupId>              <artifactId>commons-logging</artifactId>          </exclusion>      </exclusions>  </dependency> |

aop

　　aop部分包含4个模块

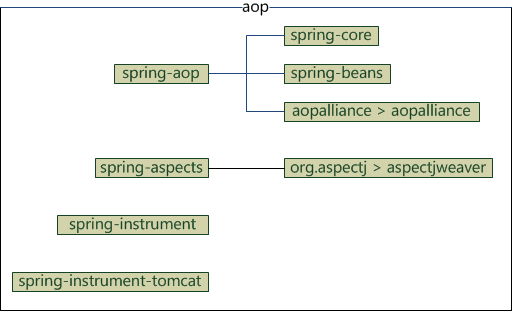
spring-aop：面向切面编程

spring-aspects：集成AspectJ

spring-instrument：提供一些类级的工具支持和ClassLoader级的实现，用于服务器

spring-instrument-tomcat：针对tomcat的instrument实现

　　它们的依赖关系



data access

　　data access部分包含5个模块

spring-jdbc：jdbc的支持

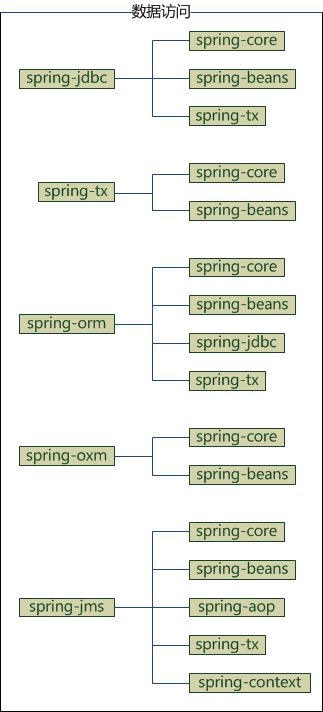
spring-tx：事务控制

spring-orm：对象关系映射，集成orm框架

spring-oxm：对象xml映射

spring-jms：java消息服务

　　它们的依赖关系



web

　　web部分包含4个模块

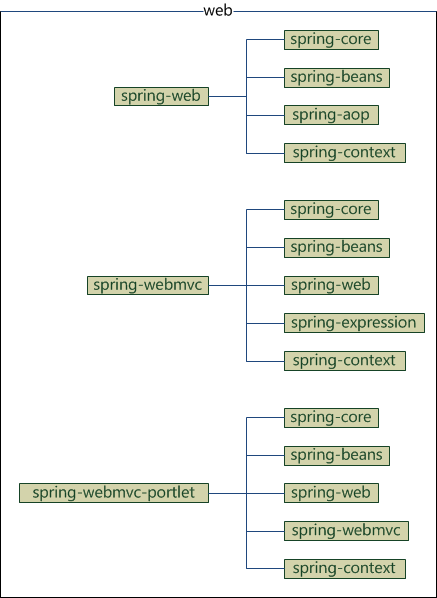
spring-web：基础web功能，如文件上传

spring-webmvc：mvc实现

spring-webmvc-portlet：基于portlet的mvc实现

spring-struts：与struts的集成，不推荐，spring4不再提供

　　它们的依赖关系



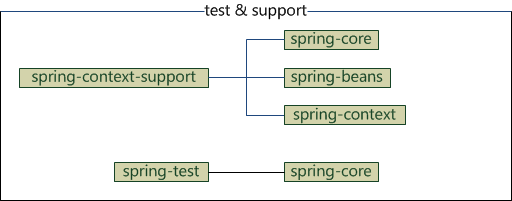
test

　　test部分只有一个模块，我将spring-context-support也放在这吧

spring-test：spring测试，提供junit与mock测试功能

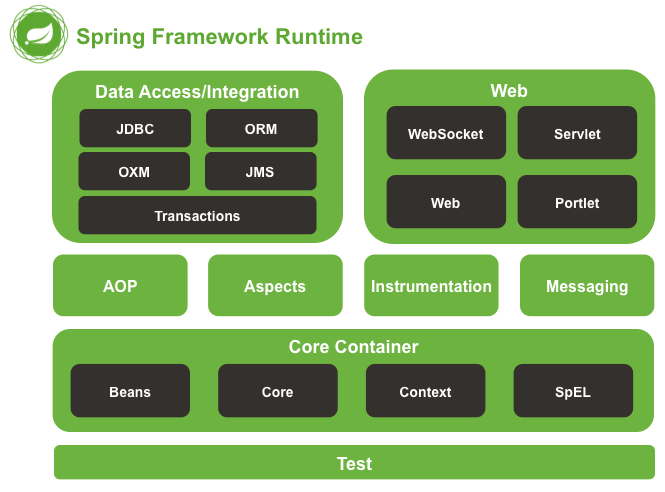
spring-context-support：spring额外支持包，比如邮件服务、视图解析等

　　它们的依赖关系



　　到这里，spring3的介绍就完了，看着这些图我相信你在maven中配置spring依赖时不会再混乱了

下面介绍spring4，与spring3结构基本相同，下面是官网给出的结构图

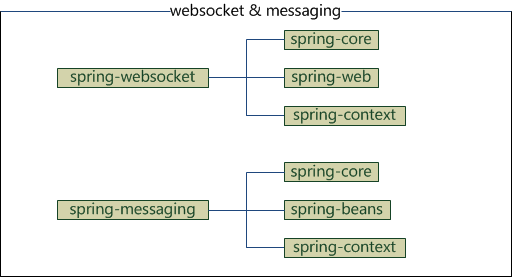


　　可以看到，图中去掉了spring3的struts，添加了messaging和websocket，其他模块保持不变，因此，spring4的jar有20个

spring-websocket：为web应用提供的高效通信工具

spring-messaging：用于构建基于消息的应用程序

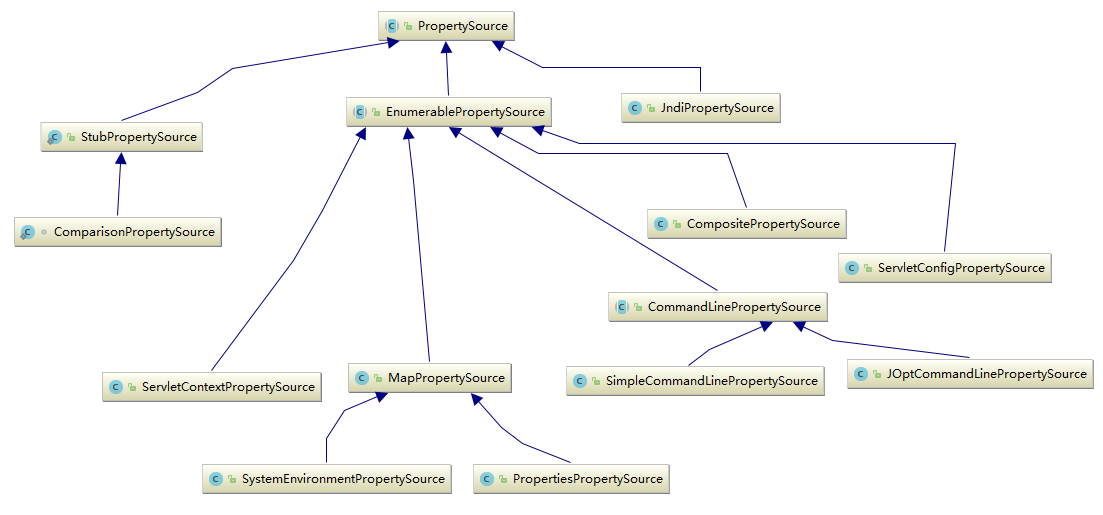
　　它们的依赖关系



Spring AOP：Spring的面向切面编程，提供AOP（面向切面编程）的实现

Spring Aspects：Spring提供的对AspectJ框架的整合  
Spring Beans：Spring IOC的基础实现，包含访问配置文件、创建和管理bean等。  
Spring Context：在基础IOC功能上提供扩展服务，此外还提供许多企业级服务的支持，有邮件服务、任务调度、JNDI定位，EJB集成、远程访问、缓存以及多种视图层框架的支持。  
Spring Context Support：Spring context的扩展支持，用于MVC方面。  
Spring Core：Spring的核心工具包  
Spring expression：Spring表达式语言  
Spring Framework Bom：  
Spring Instrument：Spring对服务器的代理接口  
Spring Instrument Tomcat：Spring对tomcat连接池的集成  
Spring JDBC：对JDBC 的简单封装  
Spring JMS：为简化jms api的使用而做的简单封装  
Spring Messaging：  
Spring orm：整合第三方的orm实现，如hibernate，ibatis，jdo以及spring 的jpa实现  
Spring oxm：Spring对于object/xml映射的支持，可以让JAVA与XML之间来回切换  
Spring test：对JUNIT等测试框架的简单封装  
Spring tx：为JDBC、Hibernate、JDO、JPA等提供的一致的声明式和编程式事务管理。  
Spring web：包含Web应用开发时，用到Spring框架时所需的核心类，包括自动载入WebApplicationContext特性的类、Struts与JSF集成类、文件上传的支持类、Filter类和大量工具辅助类。  
Spring webmvc：包含SpringMVC框架相关的所有类。包含国际化、标签、Theme、视图展现的FreeMarker、JasperReports、 Tiles、Velocity、XSLT相关类。当然，如果你的应用使用了独立的MVC框架，则无需这个JAR文件里的任何类。  
Spring webmvc portlet：Spring MVC的增强

Spring websocket：提供 Socket通信， web端的推送功能



## Spring core

├─asm Spring将ASM代码重新打包，提供ASM依赖。

├─cglib  cglib生成的动态代理类命名规则

│ ├─core

│ └─proxy

├─core

│ ├─annotation注解解析器和注解工具

│ ├─codec 编解码工具

│ ├─convert 数据类型之间相互转换工具

│ │ ├─converter

│ │ └─support

│ ├─env

│ ├─io 处理各种resource，文件，网络，等来源

│ │ ├─buffer netty框架

│ │ └─support

│ ├─log

│ ├─serializer 序列化

│ │ └─support

│ ├─style 格式化输出

│ ├─task 任务

│ │ └─support

│ └─type

│ ├─classreading

│ └─filter

├─lang 条件编译

├─objenesis

└─util

├─backoff

├─comparator

├─concurrent

├─function

├─unit

└─xml

**Env:**

Spring3.1开始提供的新的属性管理API，提供配置读取和环境划分能力，主要接口：PropertySource和Environment

MapPropertySource:属性来自于一个Map

ResourcePropertySource:属性来自于一个properties文件

ServletContextPropertySource:属性来自ServletContext上下文初始化参数

CompositePropertySource:提供了组合PropertySource的功能，查找顺序就是注册顺序。

**Task:**

Java SE 5.0引入了ThreadPoolExecutor、ScheduledThreadPoolExecutor。Spring 2.x借助ConcurrentTaskExecutor和ThreadPoolTaskExecutor能够通过IoC配置形式自定义它们暴露的各个属性。

TaskExecutor接口，是对java.util.concurrent.Executor接口的扩展。给其他组件提供线程池的抽象。

例如ApplicationEventMulticaster组件、JMS的AbstractMessageListenerContainer和对Quartz的整合都使用了TaskExecutor抽象来提供线程池。

从类图上很明显可以看出分为同步Executor和异步Executor。同步比较简单，就是调用一下的run方法。下面主要分析异步。

SimpleAsyncTaskExecutor:每次调用都启动一个新线程。但是，它还是支持对并发总数设限，当超过线程并发总数限制时，阻塞新的调用，直到有位置被释放。

ConcurrentTaskExecutor:对Java 5 java.util.concurrent.Executor类的适配，暴露了Executor的配置参数作为bean属性，可以配置自己的Executor，可以设置任务装饰器。

\*\*ThreadPoolTaskExecutor \*\*:对java.util.concurrent.ThreadPoolExecutor的包装。

ThreadPoolTaskScheduler：对java.util.concurrent.ScheduledExecutorService的封装，并提供了@Scheduled 注解,极大简化了ScheduledExecutorService的操作。  
使用方法：

## Spring-beans

├─annotation

├─factory

│ ├─annotation

│ ├─config

│ ├─groovy

│ ├─parsing

│ ├─serviceloader

│ ├─support

│ ├─wiring

│ └─xml

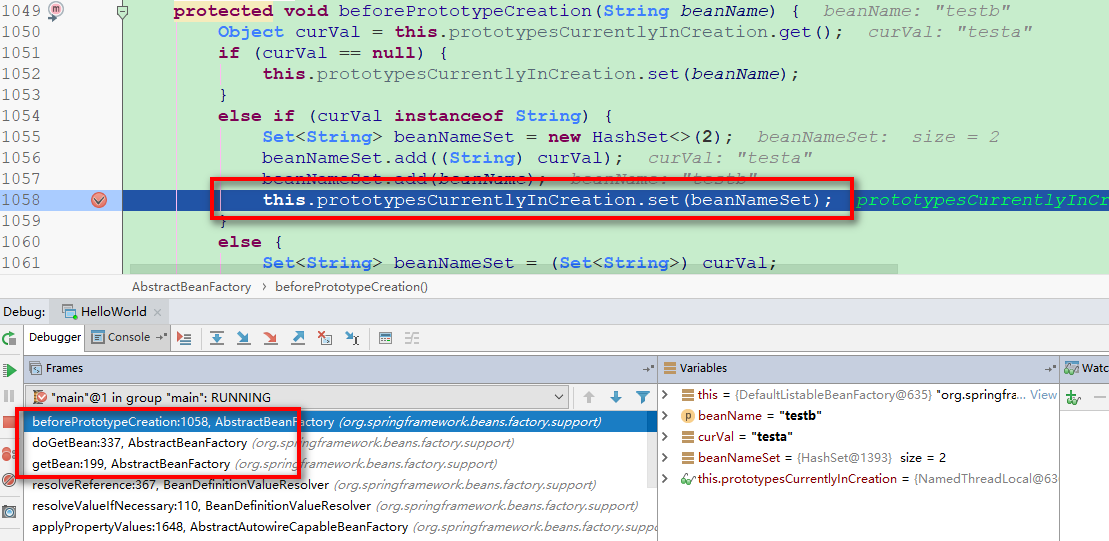
├─propertyeditors

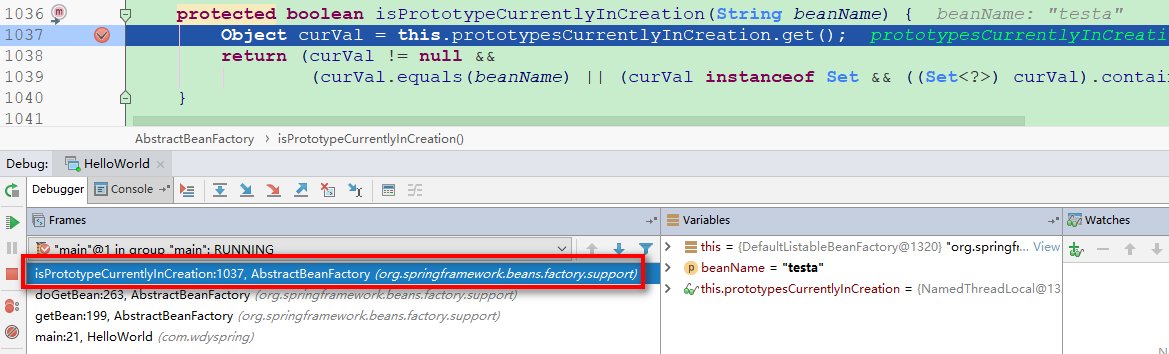
└─support

主要是beanfactory 的实现，它是一个容器。提供bean define实例。Beanfactory 对标签的处理是 parseDefaultElement(Element ele, BeanDefinitionParserDelegate delegate) 这个函数实现的，这里的 ele 是 xml 的节点，节点由顶至底依次解析。这个函数实现了 import, alias, bean, beans 四个节点依次解析。Beans是前面三个的父节点。

|  |
| --- |
| BeanFactory和ApplicationContext对待bean后置处理器稍有不同。ApplicationContext会自动检测在配置文件中实现了BeanPostProcessor接口的所有bean，并把它们注册为后置处理器，然后在容器创建bean的适当时候调用它。部署一个后置处理器同部署其他的bean并没有什么区别。而使用BeanFactory实现的时候，bean 后置处理器必须通过下面类似的代码显式地去注册：    BeanPostPrcessorImpl beanPostProcessor = new BeanPostPrcessorImpl();  Resource resource = new FileSystemResource("applicationContext.xml");  ConfigurableBeanFactory factory = new XmlBeanFactory(resource);  factory.addBeanPostProcessor(beanPostProcessor); |

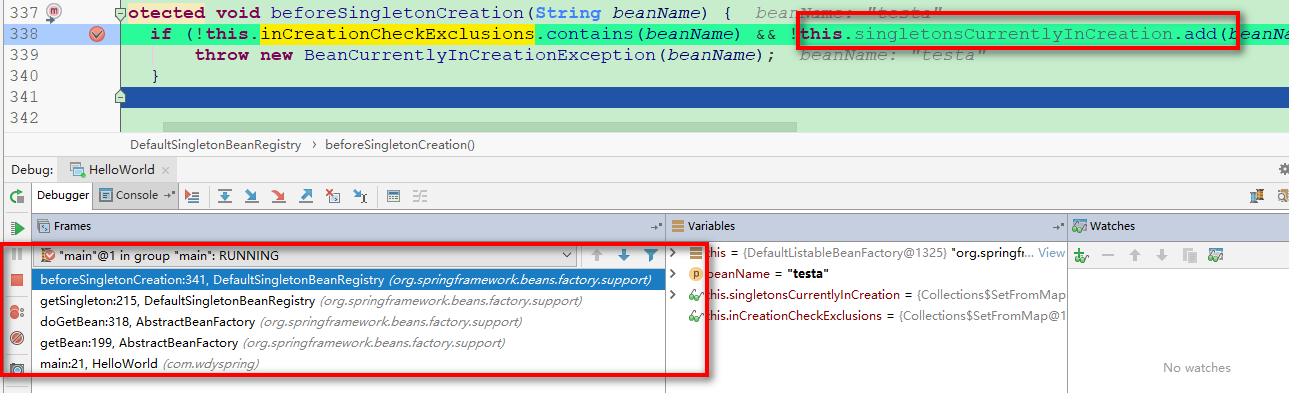
正在创建的非单例 bean ，都放进线程本地 prototypesCurrentlyInCreation 中，每次引用非单例 bean ，先从 prototypesCurrentlyInCreation 中查看，看看被引用的非单例 bean 是不是还在创建中，如果是，抛出异常。对于单例 bean ，是可以破圈的，在单例 bean 创建之前，先暴漏 factoryObj。

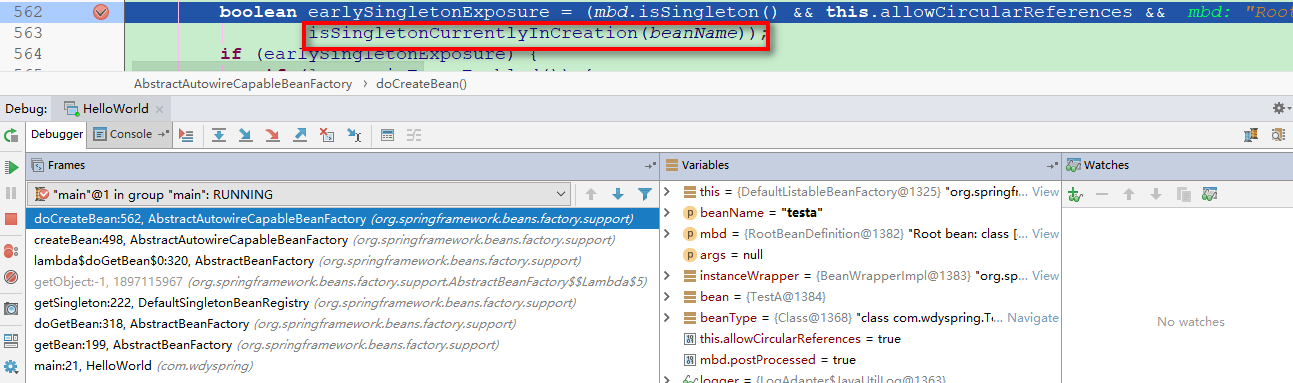




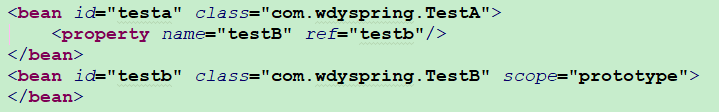
单例在创建过程中，会加到线程本地 singletonsCurrentlyInCreation ，然后填充属性，在填充的过程中如果发现被填充的对象在 singletonsCurrentlyInCreation 中存在，说明正在创建，这时就要提早暴露除 FactoryObj ，避免循环依赖。







最后要说明的是，单例 bean 依赖原型 bean 则原型 bean 只有一个，不可能是多个，这是由于单例 bean 是初始化的时候注入的，只有一个原型 bean 被注入。如本例， TestA 是单例， TestB 是原型，多次取 TestA 后，发现是同一个 TestB





## Spring-context

这个主要是对 beanFactory 的增强，增加了网络功能、注解功能

<context: annotation-config/>

## Spring aop

Spring aop有两种方式，一种是系统自带的，一种是 AspectJ 包集成。系统自带如下

|  |
| --- |
| <!-- 配置切面的Bean -->  <bean id="sysAspect" class="com.example.aop.SysAspect"/>  <!-- 配置AOP -->  <aop:config>  <!-- 配置切点表达式 -->  <aop:pointcut id="pointcut" expression="execution(public \* com.example.controller.\*Controller.\*(..))"/>  <!-- 配置切面及配置 -->  <aop:aspect order="3" ref="sysAspect">  <!-- 前置通知 -->  <aop:before method="beforMethod" pointcut-ref="pointcut" />  <!-- 后置通知 -->  <aop:after method="afterMethod" pointcut-ref="pointcut"/>  <!-- 返回通知 -->  <aop:after-returning method="afterReturnMethod" pointcut-ref="pointcut" returning="result"/>  <!-- 异常通知 -->  <aop:after-throwing method="afterThrowingMethod" pointcut-ref="pointcut" throwing="ex"/>  <aop:around method="aroundMethod" pointcut-ref="pointcut"/>  </aop:aspect>  </aop:config> |

AspectJ 增强如下：

|  |
| --- |
| <aop:aspectj-autoproxy proxy-target-class="false" expose-proxy="false"/>  proxy-target-class 表示代理的是否为类本身，true 说明用的是 cglib 。expose-proxy 表示是否代理私有方法，默认只代理public 方法 |

Aop 大致流程是，spring 启动的时候，会根据配置，创建 pointCut 和 advisor 类 bean ，在使用的时候，获取到 bean 后， spring 会返回一个 proxy 对象，这样就可以使用了。

## spring-jdbc

spring jdbc 有两个关键点，一个是 datasource 他应该是数据库的客户端程序，还有个一就是 mybatis 。一般情况下，有 database 就可以工作，不需要 mybatis 。在构建 sqlSessionFactory 的时候，会把 mybatis 的配置和datasource 合一。所以如果我们要换数据源，就要在sqlSessionFactory 上想办法。Mapper 接口，在用的时候，spring 返回的是代理类，datasource跟 mapper 之间没关系的。

## Spring –web

springframework

├─http

│ ├─client

│ │ ├─reactive

│ │ └─support

│ ├─codec

│ │ ├─json

│ │ ├─multipart

│ │ ├─protobuf

│ │ ├─support

│ │ └─xml

│ ├─converter

│ │ ├─cbor

│ │ ├─feed

│ │ ├─json

│ │ ├─protobuf

│ │ ├─smile

│ │ ├─support

│ │ └─xml

│ └─server

│ └─reactive

├─remoting

│ ├─caucho

│ ├─httpinvoker

│ └─jaxws

└─web

├─accept

├─bind

│ ├─annotation

│ └─support

├─client

│ └─support

├─context

│ ├─annotation

│ ├─request

│ │ └─async

│ └─support

├─cors

│ └─reactive

├─filter

│ └─reactive

├─jsf

│ └─el

├─method

│ ├─annotation

│ └─support

├─multipart

│ ├─commons

│ └─support

├─server

│ ├─adapter

│ ├─handler

│ ├─i18n

│ └─session

└─util

└─pattern

Spring-web 分为三个子模块，分别为 http、remoting、web，下面分别分析这三个子模块。先分析 http 模块：

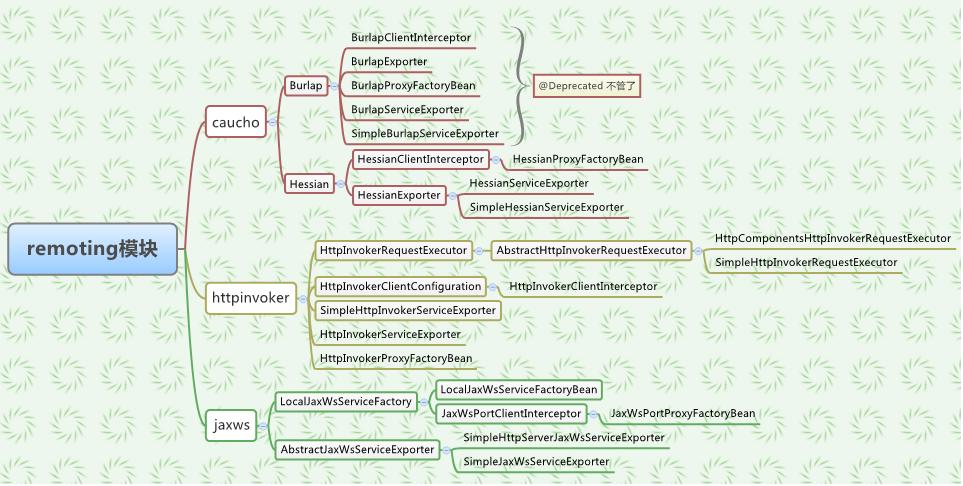
### http子模块

http 模块，提供了 http 的各种封装，包括 request、resopnes、参数转换、编码等等基础功能，总体来说比较简单。

### Remoting 子模块

该子模块主要支持远程过程调用，远程过程调用是 IPC的一种，远程过程调用机制有很多，包括 RPC机制、消息机制等。

分为三部分：caucho、httpinvoker、jaxws，其总体构造图如下：



1. Hessian/Burlap 方式

    客户端通过Hessian/BurlapProxyFactoryBean生成的代理对象负责通过HTTP协议与远程服务通信。在服务端，HessianServiceExporter是一个Spring MVC的控制器，可以通过接收请求并将这些请求翻译成对POJO对象的调用将POJO输出为一个服务。

需要添加两处配置

（1）Spring配置文件处理URL。用来分发Hessian的URL请求给适当的Hessian服务Bean。

（2）在web.xml文件配置Spring的DispatcherServlet，并把应用部署为web应用。

1. HttpInvoker 的远程调用方式

可以看出HttpInvoker和Hessian方式的区别仅仅在于客户端的ProxyFactoryBean和服务端的ServiceExporter的区别。

1. Java API for XML Web Services（JAX-WS）是Java程序设计语言一个用来创建Web服务的API。JAX-WS是sun的Java企业平台一部分。JAX-WS规范是一组XML web services的[JAVA API](http://baike.baidu.com/view/3911786.htm)，JAX-WS允许开发者可以选择RPC-oriented或者message-oriented 来实现自己的web services。
2. 三者的区别

相同点：客户端都是通过\*\*ProxyFactoryBean创建代理的方式调用远程服务。

          通信方式：JAX-WS，Hessian/Burlap 和HttpInvoker通过Http协议。

不同点：

JAX-WS： 基于HTTP协议。

Hessian/Burlap ：基于HTTP协议；Hessian使用二进制格式，Burlap使用XML格式传输文件；使用自己的序列化方式。

HttpInvoker ：基于HTTP协议，使用java的序列化机制；只能用于Spring框架。

1. Java远程通讯可选技术及原理

**RPC**是一种思想，它也是一种进程通信技术(IPC)，支持RPC 的应用级协议有RMI、MINA、ESB、Burlap、Hessian、SOAP、EJB和JMS等。底层需要做的就是将流从一个进程传到另一个进程，基于传输[协议](http://cisco.chinaitlab.com/List_11.html)和网络IO来实现。下面我们来看看这些应用级协议

#### RMI

RMI是个典型的为java定制的远程通信协议，我们都知道，在single vm中，我们可以通过直接调用java object instance来实现通信，那么在远程通信时，如果也能按照这种方式当然是最好了， RMI正是朝着这个目标而诞生的。  
来看下基于RMI的一次完整的远程通信过程的原理：  
1、客户端发起请求，请求转交至RMI客户端的stub类；  
2、stub类将请求的接口、方法、参数等信息进行序列化；  
3、基于socket将序列化后的流传输至[服务器](http://server.chinaitlab.com/)端；  
4、[服务器](http://server.chinaitlab.com/)端接收到流后转发至相应的skelton类；  
5、skelton类将请求的信息反序列化后调用实际的处理类；  
6、处理类处理完毕后将结果返回给skelton类；  
7、Skelton类将结果序列化，通过socket将流传送给客户端的stub；  
8、stub在接收到流后反序列化，将反序列化后的[Java](http://java.chinaitlab.com/) Object返回给调用者。

#### XML-RPC

XML-RPC也是一种和RMI类似的远程调用的协议，它和RMI的不同之处在于它以标准的xml格式来定义请求的信息(请求的对象、方法、参数等)，这样的好处是什么呢，就是在跨语言通讯的时候也可以使用。  
来看下XML-RPC协议的一次远程通信过程：  
1、客户端发起请求，按照XML-RPC协议将请求信息进行填充；  
2、填充完毕后将xml转化为流，通过传输协议进行传输；  
3、接收到在接收到流后转换为xml，按照XML-RPC协议获取请求的信息并进行处理；  
4、处理完毕后将结果按照XML-RPC协议写入xml中并返回。

#### Binary-RPC

Binary-RPC看名字就知道和XML-RPC是差不多的了，不同之处仅在于传输的标准格式由XML转为了二进制的格式。  
同样来回答问题：

#### SOAP

SOAP原意为Simple Object Access Protocol，是一个用于分布式环境的、轻量级的、基于XML进行信息[交换](http://cisco.chinaitlab.com/List_7.html)的通信协议，可以认为SOAP是XML RPC的高级版，两者的原理完全相同，都是http+XML，不同的仅在于两者定义的XML规范不同，SOAP也是Webservice采用的服务调用协议标准，因此在此就不多加阐述了。

#### CORBA

Common Object Request Broker Architecture（公用对象请求代理[调度]程序体系结构），是一组用来定义“分布式对象系统”的标准，由OMG(Object Menagement Group)作为发起和标准制定单位。CORBA的目的是定义一套协议，符合这个协议的对象可以互相交互，不论它们是用什么样的语言写的，不论它们运行于什么样的机器和操作系统。  
CORBA在我看来是个类似于SOA的体系架构，涵盖可选的远程通信协议，但其本身不能列入通信协议这里来讲，而且CORBA基本淘汰，再加上对CORBA也不怎么懂，在此就不进行阐述了。

#### JMS

JMS不是RPC，但它确实是个远程通信协议，在其他的语言体系中也存在着类似JMS的东西，可以统一的将这类机制称为消息机制，而消息机制呢，通常是高并发、分布式领域推荐的一种通信机制，这里的主要一个问题是容错（详细见ErLang论文）。  
来看JMS中的一次远程通信的过程：  
1、客户端将请求转化为符合JMS规定的Message；  
2、通过JMS API将Message放入JMS Queue或Topic中；  
3、如为JMS Queue，则发送中相应的目标Queue中，如为Topic，则发送给订阅了此Topic的JMS Queue。  
4、处理端则通过轮训JMS Queue，来获取消息，接收到消息后根据JMS协议来解析Message并处理。  
回答问题：  
1、传输的标准格式是？  
      JMS规定的Message。  
2、怎么样将请求转化为传输的流？  
      将参数信息放入Message中即可。  
3、怎么接收和处理流？  
      轮训JMS Queue来接收Message，接收到后进行处理，处理完毕后仍然是以Message的方式放入Queue中发送或Multicast。  
4、传输协议是？  
      不限。  
基于JMS也是常用的实现远程异步调用的方法之一。

可选实现技术  
当然，在上面的原理中并没有介绍到所有的java领域可选的远程通信协议了，例如还有EJB采用的ORMI、Spring自己定义的一个简单的Http Invoker等等。

#### 远程过程调用框架

知名的有：JBoss-Remoting、Spring-Remoting、Hessian、Burlap、XFire(Axis)、ActiveMQ、Mina、Mule、EJB3等等，分布式服务框架其实是包含了解决分布式领域以及应用层面领域两方面问题的。  
当然，你也可以自己根据远程网络通信原理(transport protocol+Net IO)去实现自己的通讯框架或library。  
那么在了解这些远程通讯的框架或library时，会带着什么问题去学习呢？  
1、是基于什么协议实现的？  
2、怎么发起请求？  
3、怎么将请求转化为符合协议的格式的？  
4、使用什么传输协议传输？  
5、响应端基于什么机制来接收请求？  
6、怎么将流还原为传输格式的？  
7、处理完毕后怎么回应？

##### JBoss-Remoting

Jboss-remoting是由jboss编写的一个java领域的远程通讯框架，基于此框架，可以很简单的实现基于多种传输协议的java对象的RPC。  
直接来回答问题：  
1、是基于什么协议实现的？  
      JBoss-Remoting是个通讯框架，因此它支持多种协议方式的通信，例如纯粹的socket+io方式、rmi方式、http+io方式等。  
2、怎么发起请求？  
      在JBoss-Remoting中，只需将需要发起的请求参数对象传入jboss-remoting的InvocationRequest对象即可，也可根据协议基于InvocationRequest封装符合需求的InvocationRequest对象。  
3、怎么将请求转化为符合协议的格式的？  
      JBoss-Remoting基于Java串行化机制或JBoss自己的串行化实现来将请求转化为对象字节流。  
4、使用什么传输协议传输？  
      支持多种传输协议，例如socket、http等。  
5、响应端基于什么机制来接收请求？  
      响应端只需将自己的处理对象注册到JBoss-Remoting提供的server端的Connector对象中即可。  
6、怎么将流还原为传输格式的？  
      JBoss-Remoting基于java串行化机制或jboss自己的串行化实现来将请求信息还原为java对象。  
7、处理完毕后怎么回应？  
      处理完毕后将结果对象直接返回即可，jboss-remoting会将此对象按照协议进行序列化，返回至调用端。  
另外，jboss-remoting支持多种通信方式，例如同步/异步/单向通信等。

##### Spring-Remoting

Spring-remoting是Spring提供java领域的远程通讯框架，基于此框架，同样也可以很简单的将普通的spring bean以某种远程协议的方式来发布，同样也可以配置spring bean为远程调用的bean。  
1、是基于什么协议实现的？  
      和JBoss-Remoting一样，作为一个远程通讯的框架，Spring通过集成多种远程通讯的library，从而实现了对多种协议的支持，例如rmi、http+io、xml-rpc、binary-rpc等。  
2、怎么发起请求？  
      在Spring中，由于其对于远程调用的bean采用的是proxy实现，发起请求完全是通过服务接口调用的方式。  
3、怎么将请求转化为符合协议的格式的？  
      Spring按照协议方式将请求的对象信息转化为流，例如Spring Http Invoker是基于Spring自己定义的一个协议来实现的，传输协议上采用的为http，请求信息是基于java串行化机制转化为流进行传输。  
4、使用什么传输协议传输？  
      支持多种传输协议，例如rmi、http等等。  
5、响应端基于什么机制来接收请求？  
      响应端遵循协议方式来接收请求，对于使用者而言，则只需通过spring的配置方式将普通的spring bean配置为响应端或者说提供服务端。  
6、怎么将流还原为传输格式的？  
      按照协议方式来进行还原。  
7、处理完毕后怎么回应？  
      处理完毕后直接返回即可，spring-remoting将根据协议方式来做相应的序列化。

##### Hessian

Hessian是由caucho提供的一个基于binary-RPC实现的远程通讯library。  
1、是基于什么协议实现的？  
      基于Binary-RPC协议实现。  
2、怎么发起请求？  
      需通过Hessian本身提供的API来发起请求。  
3、怎么将请求转化为符合协议的格式的？  
      Hessian通过其自定义的串行化机制将请求信息进行序列化，产生二进制流。  
4、使用什么传输协议传输？  
      Hessian基于Http协议进行传输。  
5、响应端基于什么机制来接收请求？  
      响应端根据Hessian提供的API来接收请求。  
6、怎么将流还原为传输格式的？  
      Hessian根据其私有的串行化机制来将请求信息进行反序列化，传递给使用者时已是相应的请求信息对象了。  
7、处理完毕后怎么回应？  
      处理完毕后直接返回，hessian将结果对象进行序列化，传输至调用端。

##### Burlap

Burlap也是有caucho提供，它和hessian的不同在于，它是基于XML-RPC协议的。  
1、是基于什么协议实现的？  
      基于XML-RPC协议实现。  
2、怎么发起请求？  
      根据Burlap提供的API。  
3、怎么将请求转化为符合协议的格式的？  
      将请求信息转化为符合协议的XML格式，转化为流进行传输。  
4、使用什么传输协议传输？  
      Http协议。  
5、响应端基于什么机制来接收请求？  
      监听Http请求。  
6、怎么将流还原为传输格式的？  
      根据XML-RPC协议进行还原。  
7、处理完毕后怎么回应？  
      返回结果写入XML中，由Burlap返回至调用端。

##### XFire、Axis

XFire、Axis是Webservice的实现框架，WebService可算是一个完整的SOA架构实现标准了，因此采用XFire、Axis这些也就意味着是采用webservice方式了。  
1、是基于什么协议实现的？  
      基于SOAP协议。  
2、怎么发起请求？  
      获取到远端service的proxy后直接调用。  
3、怎么将请求转化为符合协议的格式的？  
      将请求信息转化为遵循SOAP协议的XML格式，由框架转化为流进行传输。  
4、使用什么传输协议传输？  
      Http协议。  
5、响应端基于什么机制来接收请求？  
      监听Http请求。  
6、怎么将流还原为传输格式的？  
      根据SOAP协议进行还原。  
7、处理完毕后怎么回应？  
      返回结果写入XML中，由框架返回至调用端。

##### ActiveMQ

ActiveMQ是JMS的实现，基于JMS这类消息机制实现远程通讯是一种不错的选择，毕竟消息机制本身的功能使得基于它可以很容易的去实现同步/异步/单向调用等，而且消息机制从容错角度上来说也是个不错的选择，这是Erlang能够做到容错的重要基础。  
1、是基于什么协议实现的？  
      基于JMS协议。  
2、怎么发起请求？  
      遵循JMS API发起请求。  
3、怎么将请求转化为符合协议的格式的？  
      不太清楚，猜想应该是二进制流。  
4、使用什么传输协议传输？  
      支持多种传输协议，例如socket、http等等。  
5、响应端基于什么机制来接收请求？  
      监听符合协议的端口。  
6、怎么将流还原为传输格式的？  
      同问题3。  
7、处理完毕后怎么回应？  
      遵循JMS API生成消息，并写入JMS Queue中。  
基于JMS此类机制实现远程通讯的例子有Spring-Intergration、Mule、Lingo等等。

##### Mina

Mina是Apache提供的通讯框架，在之前一直没有提到网络IO这块，之前提及的框架或library基本都是基于BIO的，而Mina是采用NIO的，NIO在并发量增长时对比BIO而言会有明显的性能提升，而java性能的提升，与其NIO这块与OS的紧密结合是有不小的关系的。  
1、是基于什么协议实现的？  
      基于纯粹的Socket+NIO。  
2、怎么发起请求？  
      通过Mina提供的Client API。  
3、怎么将请求转化为符合协议的格式的？  
      Mina遵循java串行化机制对请求对象进行序列化。  
4、使用什么传输协议传输？  
      支持多种传输协议，例如socket、http等等。  
5、响应端基于什么机制来接收请求？  
      以NIO的方式监听协议端口。  
6、怎么将流还原为传输格式的？  
      遵循java串行化机制对请求对象进行反序列化。  
7、处理完毕后怎么回应？  
      遵循Mina API进行返回。  
MINA是NIO方式的，因此支持异步调用是毫无悬念的。

##### EJB

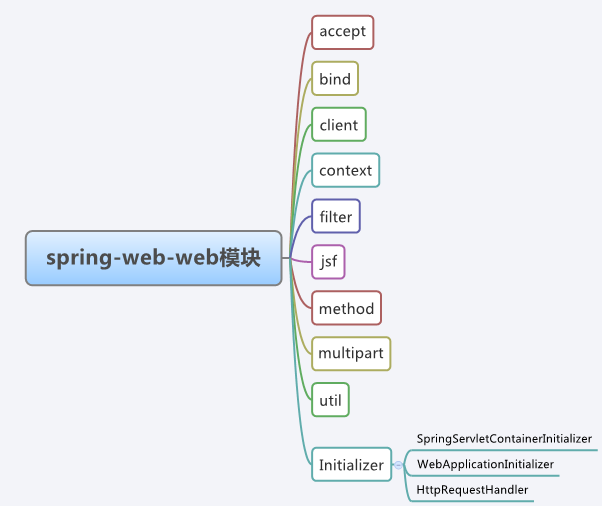
EJB最突出的在于其分布式，EJB采用的是ORMI协议，和RMI协议是差不多的，但EJB在分布式通讯的[安全](http://security.chinaitlab.com/)控制、transport pool、smart proxy等方面的突出使得其在分布式领域是不可忽视的力量。  
1、是基于什么协议实现的？  
      基于ORMI协议。  
2、怎么发起请求？  
      EJB调用。  
3、怎么将请求转化为符合协议的格式的？  
      遵循java串行化机制对请求对象进行序列化。  
4、使用什么传输协议传输？  
      Socket。  
5、响应端基于什么机制来接收请求？  
      监听协议端口。  
6、怎么将流还原为传输格式的？  
      遵循java串行化机制对请求对象进行反序列化。  
7、处理完毕后怎么回应？  
      直接返回处理对象即可。

在之前的分布式服务框架系列的文章中对于jndi有误导的嫌疑，在这篇blog中也顺带的提下jndi的机制，由于JNDI取决于具体的实现，在这里只能是讲解下jboss的jndi的实现了。  
在将对象实例绑定到jboss jnp server后，当远程端采用context.lookup()方式获取远程对象实例并开始调用时，jboss jndi的实现方法是从jnp server上获取对象实例，将其序列化回本地，然后在本地进行反序列化，之后在本地进行类调用。  
通过这个机制，就可以知道了，本地其实是必须有绑定到jboss上的对象实例的class的，否则反序列化的时候肯定就失败了，而远程通讯需要做到的是在远程执行某动作，并获取到相应的结果，可见纯粹基于JNDI是无法实现远程通讯的。  
但JNDI也是实现分布式服务框架一个很关键的技术点，因为可以通过它来实现透明化的远端和本地调用，就像ejb，另外它也是个很好的隐藏实际部署机制(就像datasource)等的方案。

总结  
由上一系列的分析可知，在远程通讯领域中，涉及的知识点还是相当的多的，例如有：通信协议(Socket/tcp/http/udp/rmi/xml-rpc etc.)、消息机制、网络IO（BIO/NIO/AIO）、MultiThread、本地调用与远程调用的透明化方案（涉及java classloader、Dynamic Proxy、Unit Test etc.）、异步与同步调用、网络通信处理机制（自动重连、广播、异常、池处理等等）、Java Serialization (各种协议的私有序列化机制等)、各种框架的实现原理（传输格式、如何将传输格式转化为流的、如何将请求信息转化为传输格式的、如何接收流的、如何将流还原为传输格式的等等），要精通其中的哪些东西，得根据实际需求来决定了，只有在了解了原理的情况下才能很容易的做出选择，甚至可以根据需求做私有的远程通讯协议，对于从事分布式服务平台或开发较大型的分布式应用的人而言，我觉得至少上面提及的知识点是需要比较了解的。

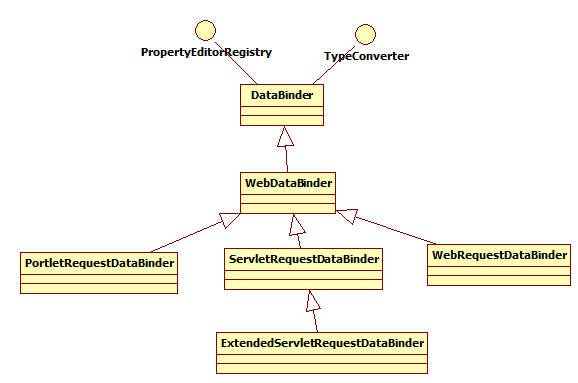
### Web

spring-web的web模块是更高一层的抽象，它封装了快速开发spring-web需要的基础组件。其结构如下：



3. 数据绑定bing部分

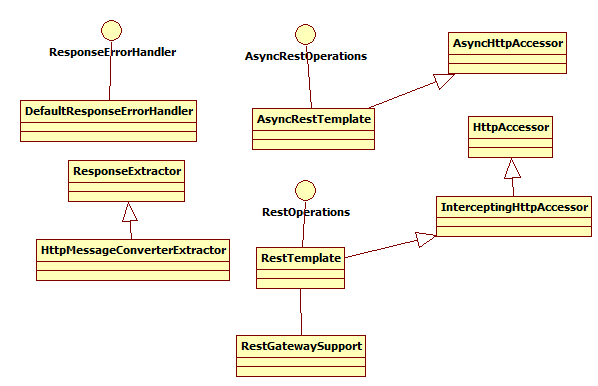
   在这里，绑定的意思是将不同类型的http请求上的数据设置到特定的对象object上如javabean等，支持multipart的绑定。其结构如下：



绑定支持两种方式：编程式和注解式

其中，注解的实现由HandlerMethodInvoker触发HandlerMethodResolver来完成。

4.web客户端client部分



5. 上下文context部分

包含一系列web应用的applicationContext接口和用来启动根web applicationContext的contextLoaderListener。

各种applicationContext层次结构如下图：（图片来源网络，具体链接已经找不到了，请原谅）

spring-context相关内容请参照这方面的源码解析，在这里就不一一赘述了。

6. 过滤器filter

    spring也对filter进行一定程度的封装和实现，其结构如下：

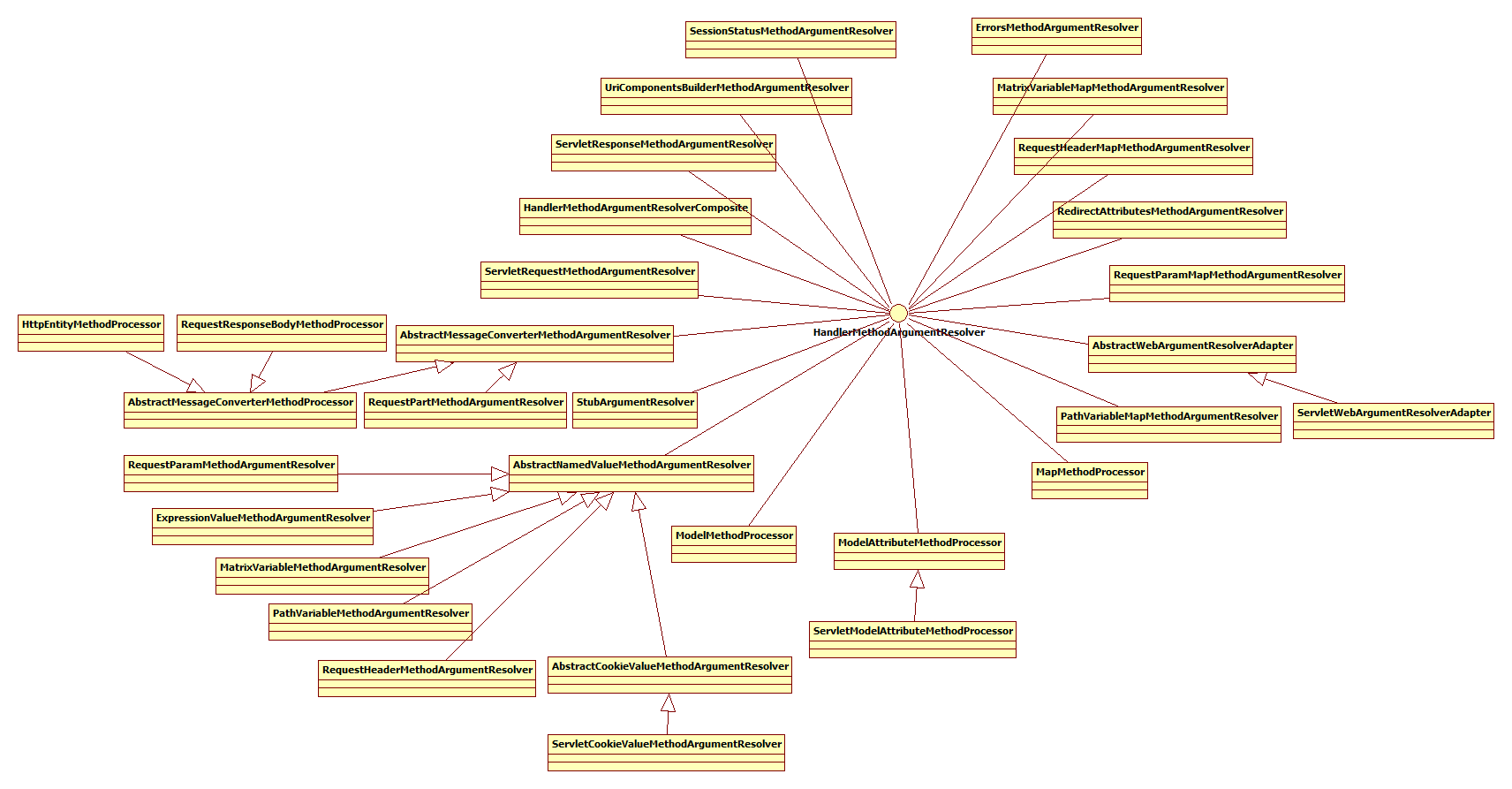


7. jsf部分

支持将jsf的web层和spring的service集成在一起，并支持jsf的el解析，spring的service层驻留在spring的根webapplicationContext中。

8. method部分

   提供给spring mvc使用的方法处理类。



9. multipart部分

一个multipart的解决文件上传的方案框架，MultipartResolver继承实现了 [Apache Commons FileUpload](http://commons.apache.org/proper/commons-fileupload).

10. util部分

    提供了公共的工具类。

小结

   spring-web的web模块是spring框架处理web请求的基础，spring-web的http模块(<http://www.cnblogs.com/davidwang456/p/4421495.html>)封装http协议中client端/server端的request请求和response响应及格式的转换，如json，rss，xml等；spring-web的remoting模块包括jaxws、caucho、httpinvoker等远程调用；spring-web的web模块则在上述模块的基础上对web应用进行了进一步的封装，提供了快速开发web的能力。

## Servlet url-pattern详解

Web项目从读取 web.xml 文件开始，web.xml 是servlet 容器 (catalina) 的配置信息。Web.xml 有两部分，一个是全局的，它在 tomcat/cfg 或者 catalina/cfg 下面。这个是对全局有效的，另一个是自己项目的 web.xml 只对本项目有效，先来分析一下全局 web.xml。

mime-mapping元素占了web.xml 87%的篇幅，作用是根据资源文件的扩展名产生对应的content-type头（也即MIME类型）添加到response header中。

|  |
| --- |
| <servlet-mapping>  <servlet-name>default</servlet-name>  <url-pattern>/</url-pattern>  </servlet-mapping>  <!-- The mappings for the JSP servlet -->  <servlet-mapping>  <servlet-name>jsp</servlet-name>  <url-pattern>\*.jsp</url-pattern>  <url-pattern>\*.jspx</url-pattern>  </servlet-mapping> |

两个默认的 servlet。一个负责缺省匹配，一个负责扩展名 \*.jsp 匹配，优先级都比较低。

首先搞清楚[servlet的url-pattern匹配规则](https://www.cnblogs.com/canger/p/6084846.html)：

1. servlet容器中的匹配规则既不是简单的通配，也不是正则表达式，而是特定的规则。所以不要用通配符或者正则表达式的匹配规则来看待servlet的url-pattern。
2. Servlet 2.5开始，一个servlet可以使用多个url-pattern规则，<servlet-mapping>标签声明了与该servlet相应的匹配规则，每个<url-pattern>标签代表1个匹配规则；
3. 当servlet容器接收到浏览器发起的一个url请求后，容器会用url减去当前应用的上下文路径，以剩余的字符串作为servlet映射，假如url是http://localhost:8080/appDemo/index.html，其应用上下文是appDemo，容器会将http://localhost:8080/appDemo去掉，用剩下的/index.html部分拿来做servlet的映射匹配
4. url-pattern映射匹配过程是有优先顺序的

而且当有一个servlet匹配成功以后，就不会去理会剩下的servlet了。当一个url与多个servlet的匹配规则匹配时，则按照 “ **精确路径 > 最长路径>扩展名 > 缺省匹配**” 这样的优先级匹配到对应的servlet。

**只支持如下四种匹配：**

1. **精确匹配**

　　<url-pattern>中配置的项必须与url完全精确匹配。

<servlet-mapping>

<servlet-name>MyServlet</servlet-name>

<url-pattern>/user/users.html</url-pattern>

<url-pattern>/index.html</url-pattern>

<url-pattern>/user/addUser.action</url-pattern>

</servlet-mapping>

　　当在浏览器中输入如下几种url时，都会被匹配到该servlet  
　　http://localhost:8080/appDemo/user/users.html  
　　http://localhost:8080/appDemo/index.html  
　　http://localhost:8080/appDemo/user/addUser.action

　　注意：

　　http://localhost:8080/appDemo/user/addUser/ 是非法的url，不会被当作http://localhost:8080/appDemo/user/addUser识别

　　另外上述url后面可以跟任意的查询条件，都会被匹配，如

http://localhost:8080/appDemo/user/addUser?username=Tom&age=23 会被匹配到MyServlet。

1. **路径匹配**

　　以“/”字符开头，并以“/\*”结尾的字符串用于路径匹配

<servlet-mapping>

<servlet-name>MyServlet</servlet-name>

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

</servlet-mapping>

　　路径以/user/开始，后面的路径可以任意。比如下面的url都会被匹配。  
　　http://localhost:8080/appDemo/user/users.html  
　　http://localhost:8080/appDemo/user/addUser.action  
　　http://localhost:8080/appDemo/user/updateUser.actionl

1. **扩展名匹配**

　　以“\*.”开头的字符串被用于扩展名匹配

<servlet-mapping>

<servlet-name>MyServlet</servlet-name>

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

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

</servlet-mapping>

　　则任何扩展名为jsp或action的url请求都会匹配，比如下面的url都会被匹配  
　　http://localhost:8080/appDemo/user/users.jsp  
　　http://localhost:8080/appDemo/toHome.action

1. **缺省匹配**

这个是默认的，匹配不上的，都靠它

<servlet-mapping>

<servlet-name>MyServlet</servlet-name>

<url-pattern>/</url-pattern>

</servlet-mapping>

**总结：**

默认的根路径就是指，webapp目录。Default 和 默认 jsp 都把 webapp 认作默认目录。 <mvc:resources mapping=”/persons/{id}” location=”classpath:static/jsp”/> 中 mapping 是要注册的 url 的 path ，我们如果访问这个 path ，会定位到 location 处的静态资源。

Tomcat 提供了缺省配置 “/” 和 \*.jsp 的扩展配置，如果 DispatcherServlet 配置为 “/” ，这是一个缺省配置，那么它的优先级最低，会直接被默认 JspServlet 给拦截。但是如果配置的是 “/\*” ，这是一个路径匹配，优先级比扩展匹配 \*.jsp 高，那么 DispatcherServlet 拦截了这个 jsp 请求，但是我们没有这个 jsp 请求的 handler (Controller 接口等)，所以不能成功，报 404 错误。

还要说一下 WEB-INF  目录，是Java的web应用安全目录，只对服务端开放，对客户端是不可见的，对客户端来说，不存在 WEB-INF  目录。所以访问该目录下的静态资源，只能通过转发，重定向无效。

通常把那些限制访问的资源（比如说jsp源代码）放到Web应用的WEB-INF目录下，对于/web-INF/及其子目录，不允许直接的公共访问，所以就可以起到保护这些代码未经授权的访问和窥视。基于Servlet的声明，WEB-INF不作为Web应用的公共文档树的一部分。客户端直接访问，报 404 错误。

使用<mvc:resources/>元素,把mapping的URI注册到SimpleUrlHandlerMapping的urlMap中,key为mapping的URI pattern值,而value为ResourceHttpRequestHandler,这样就巧妙的把对静态资源的访问由HandlerMapping转到ResourceHttpRequestHandler处理并返回,所以就支持classpath目录,jar包内静态资源的访问.另外需要注意的一点是,不要对SimpleUrlHandlerMapping设置defaultHandler.因为对static uri的defaultHandler就是ResourceHttpRequestHandler,否则无法处理static resources request.

使用<mvc:default-servlet-handler/> 会把"/\*\*" url,注册到SimpleUrlHandlerMapping的urlMap中,把对静态资源的访问由HandlerMapping转到DefaultServletHttpRequestHandler处理并返回. DefaultServletHttpRequestHandler使用就是各个Servlet容器自己的默认Servlet.

补充说明：多个HandlerMapping的执行顺序问题：

DefaultAnnotationHandlerMapping的order属性值是：0

<mvc:resources/ >自动注册的 SimpleUrlHandlerMapping的order属性值是： 2147483646

<mvc:default-servlet-handler/>自动注册 的SimpleUrlHandlerMapping 的order属性值是： 2147483647

spring会先执行order值比较小的。当访问一个a.jpg图片文件时，先通过 DefaultAnnotationHandlerMapping 来找处理器，一定是找不到的，我们没有叫a.jpg的handler。再按order值升序找，由于最后一个 SimpleUrlHandlerMapping 是匹 "/\*\*"的，所以一定会匹配上，再响应图片。

通过 spring 来弄，速度就慢了，通过 DefaultServlet 来弄，它不支持 WEB-INF 目录， 并且如果 DispatcherServlet 设置为缺省配置 “/” ，就会覆盖 DefaultServlet ，导致不起作用，所以如果想快的话，就要手动增加 DefaultServlet 的 url-pattern ，让它不只有缺省配置，还能拦截静态资源，这样它除了缺省匹配，还有扩展匹配，比缺省匹配优先级高，但是想访问 WEB-INF 目录是不可能的。还有就是 mvc:resources 和 mvc:default-servlet-handler 是 spring 配置，不是放在 web.xml 文件中的。

## Spring 注解

开启注解：

<context:annotation-config>  不支持 @component 这个注解，对 @Autoware 等是支持的。如果想往 beanFactory 中自动注册 bean ，需要 <context:component-scan base-package="com.xxx"/> 支持 @component 这个注解

当配置了<mvc:annotation-driven/>后，Spring就知道了我们启用注解驱动。然后Spring通过<context:component-scan/>标签的配置，会自动为我们将扫描到的@Component，[@Controller](https://my.oschina.net/u/1774615)，[@Service](https://my.oschina.net/service)，@Repository等注解标记的组件注册到工厂中，来处理我们的请求。

<mvc:annotation-driven/>它会向Spring容器中注册14个Bean，而其中主要的是下面几个：

1 RequestMappingHandlerMapping

2 BeanNameUrlHandlerMapping

3 RequestMappingHandlerAdapter

4 HttpRequestHandlerAdapter

5 SimpleControllerHandlerAdapter

6 ExceptionHandlerExceptionResolver

7 ResponseStatusExceptionResolver

8 DefaultHandlerExceptionResolver

它使得映射可以顺利进行。一般要和 context:component-scan 连用

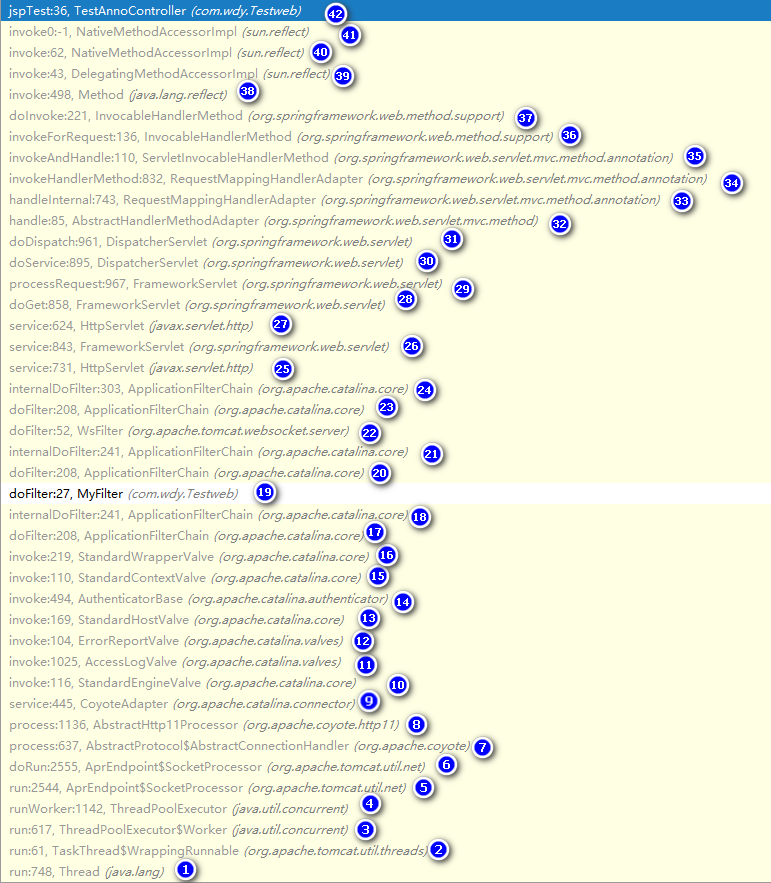
|  |
| --- |
| <mvc: annotation-driven message-codes-resolver="bean name" validator="bean name" conversion-service="bean name" ignoreDefaultModelOnRedirect="true or false">  <mvc:message-converters>  <bean></bean>  </mvc:message-converters>  <mvc:argument-resolvers>  <bean></bean>  </mvc:argument-resolvers>  <mvc:return-value-handlers>  <bean></bean>  </mvc:return-value-handlers>  </mvc:annotation-driven> |

# Spring mvc

## 事务管理

1. 前言：
2. 选择spring作为开发的框架，很大一部分因素是spring框架完善的事务处理机制，spring的事务实现主要分为两种，一种是基于Dao层，另一种是基于Service层，前者是针对单个dao的持久化操作做了事务控制，控制粒度比较小，后者则是基于业务的原则性需求，将一个原子性业务的操作做了事务控制，本文主要针对service层事务配置进行说明：
3. 方法一：基于注解的service层事务配置
5. spring-mybatis.xml中做如下配置：
7. 1、首先配置数据源：
8. **<context:property-placeholder** location="classpath:jdbc.properties" **/>**
9. **<bean** id="dataSource" class="com.alibaba.druid.pool.DruidDataSource" destroy-method="close" lazy-init="false"**>**
10. **<property** name="driverClassName" value="${jdbc.driverClassName}" **/>**
11. **<property** name="url" value="${jdbc.url}" **/>**
12. **<property** name="username" value="${jdbc.username}" **/>**
13. **<property** name="password" value="${jdbc.password}" **/>**
14. **</bean>**
16. 2、事务管理 : DataSourceTransactionManager dataSource:引用上面定义的数据源
17. **<bean** id="transactionManager"
18. class="org.springframework.jdbc.datasource.DataSourceTransactionManager"**>**
19. **<property** name="dataSource" ref="dataSource"**></property>**
20. **</bean>**
22. 3、使用声明式事务 transaction-manager：引用上面定义的事务管理器
23. **<tx:annotation-driven** transaction-manager="transactionManager" **/>**
24. service层方法添加如下注解：
26. @Transactional( rollbackFor={Exception.class})
27. 例：
28. @Service("cmdbOrderService")
29. public class  CmdbOrderServiceImpl implements CmdbOrderService {
30. //@Transactional( rollbackFor={Exception.class})
31. public boolean disposeCmdbOrder(CmdbOrder cmdbOrder) {
32. //........
33. }
34. }
35. 第二种方法：基于AOP代理的service层事务配置
37. 1、首先配置数据源：
38. **<context:property-placeholder** location="classpath:jdbc.properties" **/>**
39. **<bean** id="dataSource" class="com.alibaba.druid.pool.DruidDataSource" destroy-method="close" lazy-init="false"**>**
40. **<property** name="driverClassName" value="${jdbc.driverClassName}" **/>**
41. **<property** name="url" value="${jdbc.url}" **/>**
42. **<property** name="username" value="${jdbc.username}" **/>**
43. **<property** name="password" value="${jdbc.password}" **/>**
44. **</bean>**
46. 2、事务管理 : DataSourceTransactionManager dataSource:引用上面定义的数据源
47. **<bean** id="transactionManager"
48. class="org.springframework.jdbc.datasource.DataSourceTransactionManager"**>**
49. **<property** name="dataSource" ref="dataSource"**></property>**
50. **</bean>**
52. 3 配置事物的具体内容,
53. **<tx:advice** id="transactionAdvice" transaction-manager="transactionManager"**>**
54. **<tx:attributes>**
55. **<tx:method** name="add\*" propagation="REQUIRED" **/>**
56. **<tx:method** name="append\*" propagation="REQUIRED" **/>**
57. **<tx:method** name="insert\*" propagation="REQUIRED" **/>**
58. **<tx:method** name="save\*" propagation="REQUIRED" **/>**
59. **<tx:method** name="batchSave\*" propagation="REQUIRED" **/>**
60. **<tx:method** name="batchDel\*" propagation="REQUIRED" **/>**
61. **<tx:method** name="refreshUnrecovery\*" propagation="REQUIRED" **/>**
62. **<tx:method** name="update\*" propagation="REQUIRED" **/>**
63. **<tx:method** name="modify\*" propagation="REQUIRED" **/>**
64. **<tx:method** name="edit\*" propagation="REQUIRED" **/>**
65. **<tx:method** name="delete\*" propagation="REQUIRED" **/>**
66. **<tx:method** name="remove\*" propagation="REQUIRED" **/>**
67. **<tx:method** name="repair" propagation="REQUIRED" **/>**
68. **<tx:method** name="delAndRepair" propagation="REQUIRED" **/>**
69. **<tx:method** name="create\*" propagation="REQUIRED" **/>**
70. **<tx:method** name="dispose\*" propagation="REQUIRED" **/>**
72. **<tx:method** name="get\*" propagation="SUPPORTS" **/>**
73. **<tx:method** name="find\*" propagation="SUPPORTS" **/>**
74. **<tx:method** name="load\*" propagation="SUPPORTS" **/>**
75. **<tx:method** name="search\*" propagation="SUPPORTS" **/>**
76. **<tx:method** name="datagrid\*" propagation="SUPPORTS" **/>**
78. **<tx:method** name="\*" propagation="SUPPORTS" **/>**
79. **</tx:attributes>**
80. **</tx:advice>**
82. 4 动态数据源事物aop，当service层方法满足上述正则，则开启事物
83. **<aop:config**  proxy-target-class="true"**>**
84. **<aop:pointcut** id="transactionPointcut" expression="execution(\*  com.wutongyu.service..\*Impl.\*(..))" **/>**
85. **<aop:advisor** pointcut-ref="transactionPointcut" advice-ref="transactionAdvice"      order="2"**/>**
86. **</aop:config>**
87. 注意：捕获异常 事务的处理
89. 默认情况下，spring事务只在发生未被捕获的RuntimeException时才回滚，如果有通过try-catch捕获异常，事物不会生效；只有在抛出 RuntimeException时，事物才会生效，如果要求方法内部必须进行捕捉，怎么处理呢？
90. （1）在catch语句中最后增加throw new RuntimeException(),手动抛出异常，以便aop捕获异常再去回滚，并且在service上层要继续捕获这个异常并处理。
91. （2）在service层方法的catch语句中增加：TransactionAspectSupport.currentTransactionStatus().setRollbackOnly()；语句，手动回滚，这样上层就无需去处理异常。

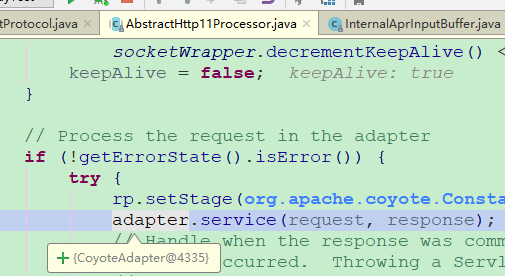
## Tomcat 和 Spring mvc 响应流程



1. 第一步，启动一个工作线程

每一次连接，都是一个 TaskThread类型的任务，1、2、3、4 就是启动这个任务的过程。这个 TaskThread 的本质是 SocketProcessor，在 TaskThread 中用 WrappingRunnable 进行封装 4 中的 runWorker() 调用 SocketProcessor 的 5 run() 方法，并最终在6 doRun() 中启动。

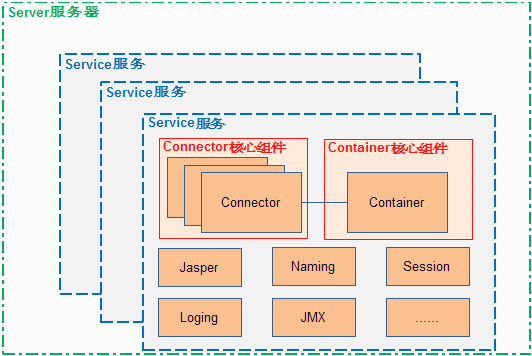
1. 第二步，处理连接，7、8



封装成 request 和 response 交给 CoyoteAdapter 来处理请求，在这里开始进入 catalina 容器中

connector.getService().getContainer().getPipeline().getFirst().invoke(request, response);

这个方法是关键，tomcat 的概念如下。



一个服务，多个Service ，默认Service 是 catalina 。Service 提供多个 connector 和 一个 engine 对外服务，不同的 connector 处理不同协议类型的请求。

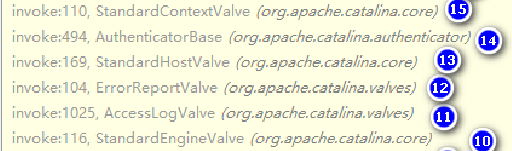
connector.getService().getContainer() 其实是拿到了 StandardEngine ， tomcat 中一共四类容器， 一次是 engine、host、context、wrap 。connector.getService().getContainer().getPipeline().getFirst()这个是拿到了 StandardEngineValue 它是一个基础阀，基础阀

host.getPipeline().getFirst().invoke(request, response)

这里的第一个不再是 StandarHostValue 基础阀，而是 AccessLogValve 这个阀，它是 host 流水线的第一个阀，最后一个阀是 StandarHostValue ，我们看下 AccessLogValve 这个阀的处理

getNext().invoke(request, response)

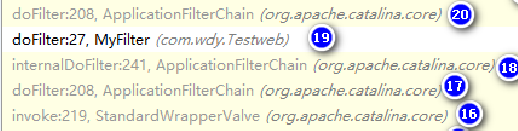
果然，获取流水线上的下一个阀。



11、12、13 才到达 StandarHostValue 这个基础阀，但是后面跟了一个 AuthenticatorBase，经分析这个是 Context 流水线的第一个阀。看 StandarHostValue 中调用

context.getPipeline().getFirst().invoke(request, response);

这是获取了 AuthenticatorBase 经查看确实如此。最终我们到达最后一个容器 Wrap 的基础阀 StandardWrapperValve。这里不会再次获取流水线，而是处理业务了。



下面详细分析 StandardWrapperValve 的 invoke 方法。

ApplicationFilterChain filterChain = factory.createFilterChain(request, wrapper, servlet);

在这里将 filter 组成了一个链。

filterChain.doFilterEvent(*request*.getEvent());

filterChain.doFilter(*request*.getRequest(),  
 *response*.getResponse());

internalDoFilter(*request*,*response*)

对事件和业务进行处理，真正的过滤在 internalDoFilter() 方法中实现，下面是 它的源码分析：

这段代码都在 filterChain 类中，

ApplicationFilterConfig filterConfig = filters[pos++];

filter = filterConfig.getFilter();

这是获取配置，从而从 filter 链中拿出一个 filter ，并且位置向后移动一位。

filter.doFilter(request, response, this);

filter 逻辑执行，this 是 filterChain ，并且在 doFilter() 中一定要调用filterChain.doFilter(servletRequest, servletResponse); 作用是 filter 处理完成后，再次进入 filterChain 这个类的方法，以便调用下一个 filter 。总体流程是

filterChain.doFilter() -> filterChain.internalDoFilter() -> 自定义filter.doFilter() ->自定义filter 中调用 filterChain.doFilter() 依次循环全部 filter 。

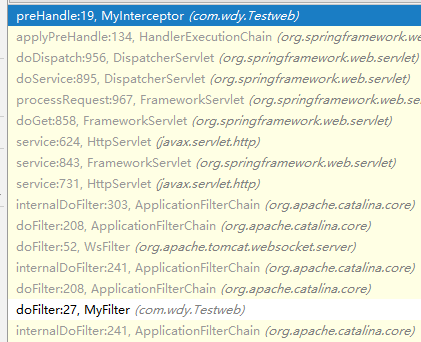
最终这个条件if (pos < n) 不再满足，进入后面代码



另外可以看到 filterChain 类 ApplicationFilterChain 将匹配到的 servlet 作为一个属性给记录了下来，从而进入到 servlet 中

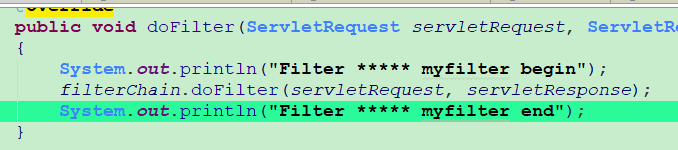


在有拦截器的情况下，栈信息如下

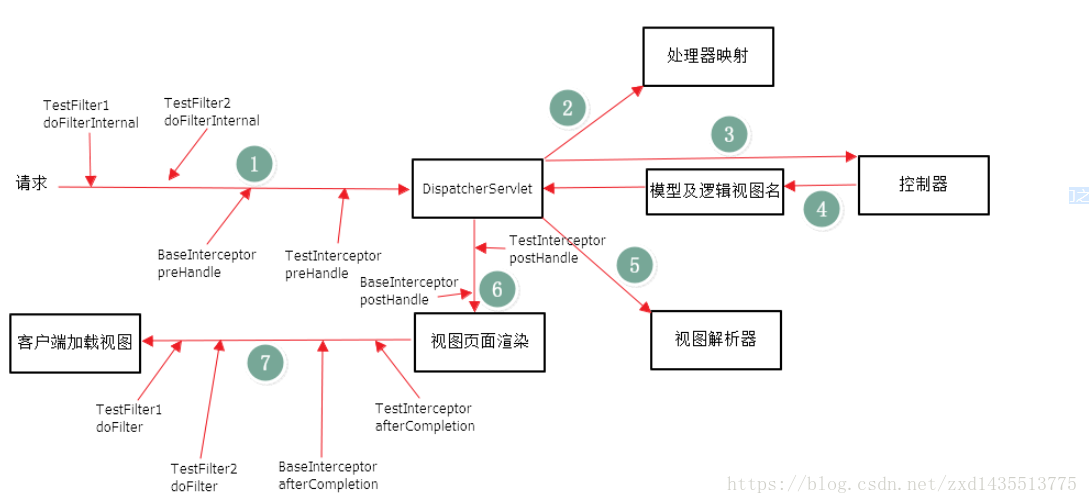


拦截器不是Filter ，但是思想与拦截器一样，它是 spring 的工具，在 Controller 请求的前、中、后对请求进行处理，必须实现的方法为 preHandler()、postHandle()、afterCompletion() ，反别在 Controller 响应前、相应后、响应完成被调用。

有一个问题，看源码



这里调用 doFilter 后最终进入 Controller 中，等Controller 执行完了， doFilter 才会返回，才会打印 end 这句。所以 filter 、servlet 、interceptor 、controller的执行顺序如下



日志信息如下：

Filter \*\*\*\*\* myfilter begin

HandlerInterceptor \*\*\*\*\* invoke preHandle()

Controller \*\*\*\*\* pathName: /ano/jsp

HandlerInterceptor \*\*\*\*\* invoke postHandle()

HandlerInterceptor \*\*\*\*\* invoke afterCompletion()

Filter \*\*\*\*\* myfilter end

这里，客户端加载视图，加载的是 html 文件，视图渲染是将视图文件翻译成浏览器能够访问的格式。

# JSP

Jsp 是 xhtml、java 技术的的结合。

JSP全称是JavaServer Pages，它和servle技术一样，都是SUN公司定义的一种用于开发动态web资源的技术。

JSP这门技术的最大的特点在于，写jsp就像在写html，但：

它相比html而言，html只能为用户提供静态数据，而Jsp技术允许在页面中嵌套java代码，为用户提供动态数据。

相比servlet而言，servlet很难对数据进行排版，而jsp除了可以用java代码产生动态数据的同时，也很容易对数据进行排版。

不管是JSP还是Servlet，虽然都可以用于开发动态web资源。但由于这2门技术各自的特点，在长期的软件实践中，人们逐渐把servlet作为web应用中的控制器组件来使用，而把JSP技术作为数据显示模板来使用。

其原因为，程序的数据通常要美化后再输出：

让jsp既用java代码产生动态数据，又做美化会导致页面难以维护。

让servlet既产生数据，又在里面嵌套html代码美化数据，同样也会导致程序可读性差，难以维护。

因此最好的办法就是根据这两门技术的特点，让它们各自负责各的，servlet只负责响应请求产生数据，并把数据通过转发技术带给jsp，数据的显示jsp来做。

Jsp的运行原理:

目标：

Web服务器是如何调用并执行一个jsp页面的？

Jsp页面中的html排版标签是如何被发送到客户端的？

Jsp页面中的java代码服务器是如何执行的？

Web服务器在调用jsp时，会给jsp提供一些什么java对象？

思考：JSP为什么可以像servlet一样，也可以叫做动态web资源的开发技术？

其实Jsp就是一个Servlet，所以我们要先介绍Servlet的相关技术，当我们第一次访问Jsp的时候，Jsp引擎都会将这个Jsp翻译成一个Servlet，这个文件存放在Tomcat中的work目录中，这里，我们新建一个MyJsp.jsp页面，然后访问以下，我们看一下翻译后的源码：

以下是MyJsp.jsp页面的内容：

[html] view plain copy

<%@ page language="java" import="java.util.\*" pageEncoding="utf-8"%>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">

<html>

  <head>

    <title>My JSP 'MyJsp.jsp' starting page</title>

  </head>

  <body>

    This is my JSP page. <br>

  </body>

</html>

下面是翻译之后的源码：

[java] view plain copy

package org.apache.jsp;

import javax.servlet.\*;

import javax.servlet.http.\*;

import javax.servlet.jsp.\*;

import java.util.\*;

public final class MyJsp\_jsp extends org.apache.jasper.runtime.HttpJspBase

    implements org.apache.jasper.runtime.JspSourceDependent {

  private static final JspFactory \_jspxFactory = JspFactory.getDefaultFactory();

  private static java.util.List \_jspx\_dependants;

  private javax.el.ExpressionFactory \_el\_expressionfactory;

  private org.apache.AnnotationProcessor \_jsp\_annotationprocessor;

  public Object getDependants() {

    return \_jspx\_dependants;

  }

  public void \_jspInit() {

    \_el\_expressionfactory = \_jspxFactory.getJspApplicationContext(getServletConfig().getServletContext()).getExpressionFactory();

    \_jsp\_annotationprocessor = (org.apache.AnnotationProcessor) getServletConfig().getServletContext().getAttribute(org.apache.AnnotationProcessor.class.getName());

  }

  public void \_jspDestroy() {

  }

  public void \_jspService(HttpServletRequest request, HttpServletResponse response)

        throws java.io.IOException, ServletException {

    PageContext pageContext = null;

    HttpSession session = null;

    ServletContext application = null;

    ServletConfig config = null;

    JspWriter out = null;

    Object page = this;

    JspWriter \_jspx\_out = null;

    PageContext \_jspx\_page\_context = null;

    try {

      response.setContentType("text/html;charset=utf-8");

      pageContext = \_jspxFactory.getPageContext(this, request, response,

                null, true, 8192, true);

      \_jspx\_page\_context = pageContext;

      application = pageContext.getServletContext();

      config = pageContext.getServletConfig();

      session = pageContext.getSession();

      out = pageContext.getOut();

      \_jspx\_out = out;

      out.write("\r\n");

      out.write("\r\n");

      out.write("<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.01 Transitional//EN\">\r\n");

      out.write("<html>\r\n");

      out.write("  <head>\r\n");

      out.write("    \r\n");

      out.write("    <title>My JSP 'MyJsp.jsp' starting page</title>\r\n");

      out.write("    \r\n");

      out.write("  </head>\r\n");

      out.write("  \r\n");

      out.write("  <body>\r\n");

      out.write("    This is my JSP page. <br>\r\n");

      out.write("  </body>\r\n");

      out.write("</html>\r\n");

    } catch (Throwable t) {

      if (!(t instanceof SkipPageException)){

        out = \_jspx\_out;

        if (out != null && out.getBufferSize() != 0)

          try { out.clearBuffer(); } catch (java.io.IOException e) {}

        if (\_jspx\_page\_context != null) \_jspx\_page\_context.handlePageException(t);

      }

    } finally {

      \_jspxFactory.releasePageContext(\_jspx\_page\_context);

    }

  }

}

我们看到，这个类继承了org.apache.jasper.runtime.HttpJspBase，要想看到这个类的源码，我们需要下载tomcat的源码，然后找到这个类，源码如下：

[java] view plain copy

/\*

 \* Licensed to the Apache Software Foundation (ASF) under one or more

 \* contributor license agreements.  See the NOTICE file distributed with

 \* this work for additional information regarding copyright ownership.

 \* The ASF licenses this file to You under the Apache License, Version 2.0

 \* (the "License"); you may not use this file except in compliance with

 \* the License.  You may obtain a copy of the License at

 \*

 \*      http://www.apache.org/licenses/LICENSE-2.0

 \*

 \* Unless required by applicable law or agreed to in writing, software

 \* distributed under the License is distributed on an "AS IS" BASIS,

 \* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.

 \* See the License for the specific language governing permissions and

 \* limitations under the License.

 \*/

package org.apache.jasper.runtime;

import java.io.IOException;

import javax.servlet.ServletConfig;

import javax.servlet.ServletException;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

import javax.servlet.jsp.HttpJspPage;

import javax.servlet.jsp.JspFactory;

import org.apache.jasper.compiler.Localizer;

/\*\*

 \* This is the super class of all JSP-generated servlets.

 \*

 \* @author Anil K. Vijendran

 \*/

public abstract class HttpJspBase

    extends HttpServlet

    implements HttpJspPage

{

    protected HttpJspBase() {

    }

    public final void init(ServletConfig config)

    throws ServletException

    {

        super.init(config);

    jspInit();

        \_jspInit();

    }

    public String getServletInfo() {

    return Localizer.getMessage("jsp.engine.info");

    }

    public final void destroy() {

    jspDestroy();

    \_jspDestroy();

    }

    /\*\*

     \* Entry point into service.

     \*/

    public final void service(HttpServletRequest request, HttpServletResponse response)

    throws ServletException, IOException

    {

        \_jspService(request, response);

    }

    public void jspInit() {

    }

    public void \_jspInit() {

    }

    public void jspDestroy() {

    }

    protected void \_jspDestroy() {

    }

    public abstract void \_jspService(HttpServletRequest request,

                     HttpServletResponse response)

    throws ServletException, IOException;

}

好吧，看到了，继承了HttpServlet类，所以说其实Jsp就是一个Servlet

## Jsp的语法:

1.JSP模版元素

2.JSP表达式

3.JSP脚本片段

4.JSP注释

5.JSP指令

6.JSP标签

7.JSP内置对象

8.如何查找JSP页面中的错误

### Jsp模板元素

JSP页面中的HTML内容称之为JSP模版元素。

JSP模版元素定义了网页的基本骨架，即定义了页面的结构和外观。

### Jsp中的脚本表达式

JSP脚本表达式（expression）用于将程序数据输出到客户端

语法：<%= 变量或表达式 %>

举例：当前时间:<%= new java.util.Date() %>

### Jsp中的脚本

JSP脚本片断（scriptlet）用于在JSP页面中编写多行Java代码。语法：

<%

多行java代码

%>

注意：JSP脚本片断中只支持java代码，不能出现其它模板元素， JSP引擎在翻译JSP页面中，会将JSP脚本片断中的Java代码将被原封不动地放到Servlet的\_jspService方法中。

JSP脚本片断中的Java代码必须严格遵循Java语法，例如，每执行语句后面必须用分号（;）结束。

在一个JSP页面中可以有多个脚本片断，在两个或多个脚本片断之间可以嵌入文本、HTML标记和其他JSP元素。

举例：

[html] view plain copy

<%

    int x = 10;

    out.println(x);

%>

<p>这是JSP页面文本</p>

<%

    int y = 20;

    out.println(y);

%>

多个脚本片断中的代码可以相互访问，犹如将所有的代码放在一对<%%>之中的情况。如：out.println(x);

单个脚本片断中的Java语句可以是不完整的，但是，多个脚本片断组合后的结果必须是完整的Java语句，例如：

[html] view plain copy

<%

    for (int i=1; i<5; i++)

    {

%>

    <H1>www.it315.org</H1>

<%

    }

%>

### Jsp的声明

JSP页面中编写的所有代码，默认会翻译到servlet的service方法中， 而Jsp声明中的java代码被翻译到\_jspService方法的外面。语法：

<%！

java代码

%>

所以，JSP声明可用于定义JSP页面转换成的Servlet程序的静态代码块、成员变量和方法 。

多个静态代码块、变量和函数可以定义在一个JSP声明中，也可以分别单独定义在多个JSP声明中。

JSP隐式对象的作用范围仅限于Servlet的\_jspService方法，所以在JSP声明中不能使用这些隐式对象。

### Jsp注释

四种注释分别为 <!--- --> 这个是 xml 标准注释，会发送给客户端。另外 <%-----%> 这个是 JSP 标准注释，不会发给可客户端， // 和 /\* \*/这两种是 java 注释，也不发给客户端。

### Jsp指令

JSP指令（directive）是为JSP引擎而设计的，它们并不直接产生任何可见输出，而只是告诉引擎如何处理JSP页面中的其余部分。在JSP2.0规范中共定义了三个指令：

page指令

Include指令

taglib指令

JSP指令的基本语法格式：

<%@ 指令 属性名="值" %>

首先我们来看一下page指令的用法

page指令用于定义JSP页面的各种属性，无论page指令出现在JSP页面中的什么地方，它作用的都是整个JSP页面，为了保持程序的可读性和遵循良好的编程习惯，page指令最好是放在整个JSP页面的起始位置。

JSP 2.0规范中定义的page指令的完整语法：

<%@ page

[ language="java" ]

[ extends="package.class" ]

[ import="{package.class | package.\*}, ..." ]

[ session="true | false" ]

[ buffer="none | 8kb | sizekb" ]

[ autoFlush="true | false" ]

[ isThreadSafe="true | false" ]

[ info="text" ]

[ errorPage="relative\_url" ]

[ isErrorPage="true | false" ]

[ contentType="mimeType [ ;charset=characterSet ]" | "text/html ; charset=ISO-8859-1" ]

[ pageEncoding="characterSet | ISO-8859-1" ]

[ isELIgnored="true | false" ]

%>

1.看一下通过page指令设置页面的编码：

举例：<%@ page contentType="text/html;charset=gb2312"%>

这个指令的作用就相当于response.setContentType("text/html;charset=gb2312");

但是这个指令和

<%@ page pageEncoding="gb2312"%>

的区别是：

### pageEncoding是jsp文件本身的编码

contentType的charset是指服务器发送给客户端时的内容编码

JSP要经过两次的“编码”，第一阶段会用pageEncoding，第二阶段会用utf-8至utf-8，第三阶段就是由Tomcat出来的网页， 用的是contentType。

第一阶段是jsp编译成.java，它会根据pageEncoding的设定读取jsp，结果是由指定的编码方案翻译成统一的UTF-8 JAVA源码（即.java），如果pageEncoding设定错了，或没有设定，出来的就是中文乱码。

第二阶段是由JAVAC的JAVA源码至java byteCode的编译，不论JSP编写时候用的是什么编码方案，经过这个阶段的结果全部是UTF-8的encoding的java源码。

JAVAC用UTF-8的encoding读取java源码，编译成UTF-8 encoding的二进制码（即.class），这是JVM对常数字串在二进制码（java encoding）内表达的规范。

第三阶段是Tomcat（或其的application container）载入和执行阶段二的来的JAVA二进制码，输出的结果，也就是在客户端见到的，这时隐藏在阶段一和阶段二的参数contentType就发挥了功效

contentType的設定.

pageEncoding 和contentType的预设都是 ISO8859-1. 而随便设定了其中一个, 另一个就跟着一样了(TOMCAT4.1.27是如此). 但这不是绝对的, 这要看各自JSPC的处理方式. 而pageEncoding不等于contentType, 更有利亚洲区的文字 CJKV系JSP网页的开发和展示, (例pageEncoding=GB2312 不等于 contentType=utf-8)。

jsp文件不像.java，.java在被编译器读入的时候默认采用的是操作系统所设定的locale所对应的编码，比如中国大陆就是GBK，台湾就是BIG5或者MS950。而一般我们不管是在记事本还是在ue中写代码，如果没有经过特别转码的话，写出来的都是本地编码格式的内容。所以编译器采用的方法刚好可以让虚拟机得到正确的资料。

但是jsp文件不是这样，它没有这个默认转码过程，但是指定了pageEncoding就可以实现正确转码了。

因为我们使用了MyEclipse开发工具，只要我们设置其中一个编码就会将该文件的编码以及页面访问编码都设置好了，例如：

我们使用指令：

<%@ page contentType="text/html;charset=gb2312"%>

我们看一下翻译之后的servlet代码：

我们在来看一下这个文件的编码：

可以看到都是gb2312的编码，

同样我们在使用指令：

<%@ page pageEncoding="utf-8"%>

效果和上面一样的，会同时将文件的编码和页面浏览的编码都设置成了utf-8编码，这些动作都是得益于MyEclipse的智能，如果你使用txt文本开发的话，就不会有这样的结果了。

那么当我们同时使用这两个命令的话是什么样的结果呢：

<%@ page contentType="text/html;charset=gb2312"%>

<%@ page pageEncoding="utf-8"%>

这时候我就会发现，文件的编码是utf-8,而页面浏览的编码是gb2312,所以说上面的一条指令是专门设置页面浏览的编码的，下面的指令是专门设置文件的保存编码的。出现以上的结果都是因为使用了MyEclipse开发工具。

### page指令导入java包：

JSP 引擎自动导入下面的包：

java.lang.\*

javax.servlet.\*

javax.servlet.jsp.\*

javax.servlet.http.\*

可以在一条page指令的import属性中引入多个类或包，其中的每个包或类之间使用逗号分隔：

<%@ page import="java.util.Date,java.sql.\*,java.io.\*"%>

上面的语句也可以改写为使用多条page指令的import属性来分别引入各个包或类：

<%@ page import="java.util.Date"%>

<%@ page import="java.sql.\*"%>

<%@ page import="java.io.\*"%>

3.下面在来看一下buffer属性：

<%@ page buffer="4kb" %>

使用这个属性是指定out对象的缓存大小，关于out对象，我们后面会说到，比如我们这里设置了out的缓冲区是4kb，我们可以看一下翻译后的代码：

如果我们想关闭缓冲区的话，只需要设置值为none就可以了

4.下面在来看一下isThreadSafe属性：

这个属性见名知意，是设置是否线程安全的，我们在之前讨论Servlet的时候，说到了Servlet是个单例对象，是线程不安全的，我们那时候可以通过实现一个接口来实现线程安全，这里只需要设置这个属性值就可以控制Jsp翻译之后的Servlet时候线程安全：

<%@ page isThreadSafe="true|false" %>  默认值为true

isThreadSafe=false模式表示它是以Singleton模式运行。

     该模式implements了接口SingleThreadMode,

     该模式同一时刻只有一个实例，不会出现信息同步与否的概念。

     若多个用户同时访问一个这种模式的页面，

     那么先访问者完全执行完该页面后，后访问者才开始执行。

isThreadSafe=true模式表示它以多线程方式运行。

    该模式的信息同步，需访问同步方法(用synchronized标记的)来实现。

我们将值设置成false之后发现翻译Jsp之后的Servlet：

实现了SingleThreadModel接口。

5.下面在来看一下session属性：

<%@ page session="false|true"%> 默认值是true

是指不能在本页使用session.也就是在本页面禁用了session

就是在将Jsp翻译成Servlet的时候不会传递Session对象了，比如我们将他设置成false,查看翻译后的Servlet源码：

我们发现并没有HttpSession对象，所以我们也不能在Jsp页面中使用session对象了

6.下面来看一下errorPage属性

这个属性是设置服务器端出错之后的错误页面的，比如我们在编写服务器代码的时候，突然抛出异常，那么会返回一个500，这样给用户的感觉就很恶心，所以我们要做的人性化一点，就是通过这个属性值，设置一个错误页面，当服务器发生错误的时候都会跳转到这个页面中，比如：

<%@ page errorPage="/error.jsp"%>

然后我们可以在Jsp中添加一个脚本片段：

[html] view plain copy

<%

    int x = 1/0;

%>

这样就会抛出异常，我们访问这个Jsp看一下效果：

转到了错误页面，同时我们观察地址栏可以发现这里面使用的是转发技术，所以我们在书写url地址的时候，那个开始的斜杠代表是当前应用，因为这个地址是给服务器用的(这个原则我们在之前说过了)

这里在拓展一下，如果我们想给应用配置一个全局的错误页面我们该怎么配置呢？

这个想一想肯定是在web.xml文件中做配置：

[html] view plain copy

<!-- 错误页面问题同时jsp中的errorPage命令的优先级比较高 -->

    <error-page>

        <exception-type>Exception</exception-type>

        <location>/error.jsp</location>

    </error-page>

    <error-page>

        <error-code>404</error-code>

        <location>/error.jsp</location>

    </error-page>

    <error-page>

        <error-code>500</error-code>

        <location>/error.jsp</location>

    </error-page>

我们看到这里可以配置抛出的异常类型所对应的错误页面，也可以配置状态码对应的错误页面，而且这里配置的结果的优先级没有errorPage属性配置错误页面的优先级高。

7.同时还有一个属性就是isErrorPages:

<@ page isErrorPages="false|true"%> 默认值是true

是否开启错误页面，就是控制上面的errorPage属性的作用开关的

8.属性isELIgnored

<@ page isELIgnored="false|true"%> 默认值是false

是否在该页面中忽视EL表达式的功能，这个我们一般都不去做设置，因为我们在页面中肯定会使用到EL表达式的

注意：如果一个指令有多个属性，这多个属性可以写在一个指令中，也可以分开写。

例如：

[html] view plain copy

<%@ page contentType="text/html;charset=gb2312"%>

<%@ page import="java.util.Date"%>

也可以写作：

[java] view plain copy

<%@ page contentType="text/html;charset=gb2312" import="java.util.Date"%>

下面在来看一下include指令

include指令很简单，就是实现页面包含的，我们之前在介绍request的时候，说到使用代码进行页面包含，那时候我们说到了，这个代码来实现页面包含是动态包含，而使用include指令来实现页面包含是静态包含，关于静态包含和动态包含，我们在下面介绍jsp:include标签的时候在详细说明

最后来看一下taglib指令

这个指令作用也是很简单的，就是引入标签，这个之后再我们后面接收JSTL的内容的时候在作介绍

Jsp中内置的9个隐式对象

每个JSP 页面在第一次被访问时，WEB容器都会把请求交给JSP引擎（即一个Java程序）去处理。JSP引擎先将JSP翻译成一个\_jspServlet(实质上也是一个servlet) ，然后按照servlet的调用方式进行调用。

由于JSP第一次访问时会翻译成servlet，所以第一次访问通常会比较慢，但第二次访问，JSP引擎如果发现JSP没有变化，就不再翻译，而是直接调用，所以程序的执行效率不会受到影响。

JSP引擎在调用JSP对应的\_jspServlet时，会传递或创建9个与web开发相关的对象供\_jspServlet使用。JSP技术的设计者为便于开发人员在编写JSP页面时获得这些web对象的引用，特意定义了9个相应的变量，开发人员在JSP页面中通过这些变量就可以快速获得这9大对象的引用。

这9个对象分别是哪些，以及作用也是笔试经常考察的知识点。

### Jsp9大对象

request：HttpServletRequest

response：HttpServletResponse

session： HttpSession

application： servletContext

config：servletConfig

out：JspWriter

exception

page：this

pageContext

上面其实很多对象我们都接触过了，之前介绍servlet的时候都介绍过了，关于那个exception是个异常对象，只有当我们的Jsp页面中抛出异常的时候，才会有这个对象的产生，否则是不会传递这个对象的，至于page对象，这个很简单就是当前对象，即jsp翻译后的servlet对象，那么下面就来详细解释一下out对象和pageContext对象了。

out隐式对象用于向客户端发送文本数据。

out对象是通过调用pageContext对象的getOut方法返回的，其作用和用法与ServletResponse.getWriter方法返回的PrintWriter对象非常相似。

JSP页面中的out隐式对象的类型为JspWriter，JspWriter相当于一种带缓存功能的PrintWriter，设置JSP页面的page指令的buffer属性可以调整它的缓存大小，甚至可以关闭它的缓存。

只有向out对象中写入了内容，且满足如下任何一个条件时，out对象才去调用ServletResponse.getWriter方法，并通过该方法返回的PrintWriter对象将out对象的缓冲区中的内容真正写入到Servlet引擎提供的缓冲区中：

1.设置page指令的buffer属性关闭了out对象的缓存功能

2.out对象的缓冲区已满

3.整个JSP页面结束

下面来看一下实例：

[html] view plain copy

<%

        out.print("aaa");

        response.getWriter().write("bbb");

%>

我们访问一下MyJsp.jsp页面：

我们看到了，我们是先向浏览器中输出aaa,然后在输出bbb,但是结果是相反的，原因就是out对象是JspWriter，是带有缓冲的。看下图解释：

所以我们以后再给浏览器输出数据的时候一定要注意，最好不要同时使用这两个对象，一般我们只是用out对象进行输出数据的。

pageContext对象是JSP技术中最重要的一个对象，它代表JSP页面的运行环境，这个对象不仅封装了对其它8大隐式对象的引用，它自身还是一个域对象(之前我们介绍了三个域对象:ServletContext,Session,Request)，这个域对象的生命周期最短，作用域最小，他的作用域就是当前的jsp页面，当然它可以用来保存数据。并且，这个对象还封装了web开发中经常涉及到的一些常用操作，例如引入和跳转其它资源、检索其它域对象中的属性等。

getException方法返回exception隐式对象

getPage方法返回page隐式对象

getRequest方法返回request隐式对象

getResponse方法返回response隐式对象

getServletConfig方法返回config隐式对象

getServletContext方法返回application隐式对象

getSession方法返回session隐式对象

getOut方法返回out隐式对象

pageContext封装其它8大内置对象的意义，思考：如果在编程过程中，把pageContext对象传递给一个普通java对象，那么这个java对象将具有什么功能？

这个我们在后面会介绍自定义标签的时候，这个用途就体现出来了，我们只需要传递一个pageContext对象，就可以操作其他多个对象了，很方便的

pageContext对象中的操作域中数据的方法

public void setAttribute(java.lang.String name,java.lang.Object value)

public java.lang.Object getAttribute(java.lang.String name)

public void removeAttribute(java.lang.String name)

pageContext对象中还封装了访问其它域的方法(和上面的方法不同之处就是多了一个参数，这个参数是可以直接指定相应的域)

public java.lang.Object getAttribute(java.lang.String name,int scope)

public void setAttribute(java.lang.String name, java.lang.Object value,int scope)

public void removeAttribute(java.lang.String name,int scope)

代表各个域的常量

PageContext.APPLICATION\_SCOPE

PageContext.SESSION\_SCOPE

PageContext.REQUEST\_SCOPE

PageContext.PAGE\_SCOPE

下面这个方法是非常重要的，因为这个方法是在所有的域中查找数据，查找顺序是：pageContext->request->session->ServletContext,如果查找不到相应的数据的话，就返回一个空字符串，这个方法和之后要说到的el表达式的功能是一样的

findAttribute方法  （\*重点，查找各个域中的属性）

到此为止，web开发接触到了4个域对象：

pageContext（称之为page域）

request（称之为request域）

session（称之为session域）

servletContext（称之为application域）

这4个域对象是学习web的重点，也是笔试经常考察的知识点。

明确如下问题：

这4个对象的生命周期？

什么是域？为什么把这4个对象叫做域对象呢？

哪种情况下用哪种域对象。

PageContext类中定义了一个forward方法和两个include方法来分别简化和替代RequestDispatcher.forward方法和include方法

传递给这些方法的资源路径都只能是相对路径，如果路径以“/”开头，表示相对于当前WEB应用程序的根目录，否则，表示相对于当前JSP所映射到的访问路径。

## JSP标签库

虽然我们希望JSP页面仅用作数据显示模块，不要嵌套任何java代码引入任何业务逻辑，但在实际开发中不引入一点业务逻辑是不可能的，但引入业务逻辑会导致页面出现难看java代码，怎么办？

Sun公司允许用户开发自定义标签封装页面的java代码，以便jsp页面不出现一行java代码。当然sun公司在jsp页面中也内置了一些标签(这些标签叫做jsp标签)，开发人员使用这些标签可以完成页面的一些常用业务逻辑。

JSP标签也称之为Jsp Action(JSP动作)元素，它用于在JSP页面中提供业务逻辑功能。

<jsp:include>标签

<jsp:forward>标签

<jsp:param>标签

<jsp:useBean>标签

<jsp:setProperty>标签

<jsp:getProperty>标签

### <jsp:include>标签

<jsp:include>标签用于把另外一个资源的输出内容插入进当前JSP页面的输出内容之中，这种在JSP页面执行时的引入方式称之为动态引入。

语法：

<jsp:include page="relativeURL | <%=expression%>" flush="true|false" />

page属性用于指定被引入资源的相对路径，它也可以通过执行一个表达式来获得。

flush属性指定在插入其他资源的输出内容时，是否先将当前JSP页面的已输出的内容刷新到客户端。

<jsp:include>标签是动态引入(和使用代码进行include一样)，<jsp:include>标签涉及到的2个JSP页面会被翻译成2个servlet，这2个servlet的内容在执行时进行合并。 而include指令是静态引入(编译时引入)，涉及到的2个JSP页面会被翻译成一个servlet，其内容是在源文件级别进行合并。不管是<jsp:include>标签，还是include指令，它们都会把两个JSP页面内容合并输出，所以这两个页面不要出现重复的HTML全局架构标签，否则输出给客户端的内容将会是一个格式混乱的HTML文档。

例子：

使用<jsp:include>标签来实现包含页面：

[html] view plain copy

<jsp:include page="/head.jsp"></jsp:include>

<jsp:include page="/foot.jsp"></jsp:include>

我们到tomcat的work目录中看一下：我们看到，会将head.jsp和foot.jsp单独翻译成servlet,这个就是动态包含

下面在看一下使用include指令实现页面包含：

[html] view plain copy

<%@ include file="/head.jsp" %>

这时候我们发现work目录中并不会还单独翻译head.jsp页面了，同时我们看看MyJsp页面翻译的servlet代码：

我们看到在代码中使用静态代码块实现静态页面包含的。

<jsp:include>标签：使用page属性指定被引入资源。

include指令：使用file属性指定被引入资源。

假设myweb应用的根目录下有一个a.jsp文件 如果将a.jsp页面映射成了如下地址：

http://localhost:8080/myweb/dir1/a.html

在a.jsp页面中使用了如下语句引入b.jsp文件：

<jsp:include page="b.jsp" />

请问：b.jsp要位于什么位置，上面的include才不会出错？

http://localhost:8080/myweb/b.jspf

http://localhost:8080/myweb/dir1/b.jspf

假设myweb应用程序的根目录下有一个a.jsp文件，如果将a.jsp页面映射为如下地址：

http://localhost:8080/myweb/dir1/a.html

在a.jsp页面中使用了如下语句引入b.jspf文件：

<%@ include file=“b.jspf”%>

请问： b.jspf要位于什么位置，上面的include才不会出错？

http://localhost:8080/myweb/b.jspf

http://localhost:8080/myweb/dir1/b.jspf

### <jsp:forward>标签

<jsp:forward>标签用于把请求转发给另外一个资源。

语法：

<jsp:forward page="relativeURL | <%=expression%>" />

page属性用于指定请求转发到的资源的相对路径，它也可以通过执行一个表达式来获得。

### <jsp:param>或者<jsp:params>标签

当使用<jsp:include>和<jsp:forward>标签引入或将请求转发给其它资源时，可以使用<jsp:param>标签向这个资源传递参数。

语法1：

<jsp:include page="relativeURL | <%=expression%>">

<jsp:param name="parameterName" value="parameterValue|<%= expression %>" />

</jsp:include>

语法2：

<jsp:forward page="relativeURL | <%=expression%>">

<jsp:param name="parameterName" value="parameterValue|<%= expression %>" />

</jsp:include>

<jsp:param>标签的name属性用于指定参数名，value属性用于指定参数值。在<jsp:include>和<jsp:forward>标签中可以使用多个<jsp:param>标签来传递多个参数。

### <jsp:useBean>标签，<jsp:setProperty>标签，<jsp:getProperty>标签

这三个标签是一起用来操作bean对象的，<jsp:useBean>是用来初始化bean对象的，<jsp:setProperty>标签是用来设置bean对象中的属性值，<jsp:getProperty>标签是用来获取bean对象中的属性值的，例子：

[html] view plain copy

<body>

    <!-- 找到就直接用，找不到实例化 scope默认是page域-->

    <jsp:useBean id="person" class="com.weijia.domain.Person" scope="page"/>

    <!-- 直接设置属性值(8种基本类型的转换) -->

    <jsp:setProperty name="person" property="name" value="xxx"/>

    <!-- 用请求参数给属性赋值(8中基本类型的转换) -->

    <jsp:setProperty name="person" property="name" param="name"/>

    <!-- 获取Person中的属性name的值 -->

    <jsp:getProperty name="person" property="name"/>

  </body>

同时我们定义了一个Bean对象：com.weijia.domain.Person，其中有一个name属性(一定要有get/set方法才叫属性)

对于标签<jsp:useBean>他可以指定在哪个域中创建这个bean对象，他的规则是首先在这个域中查找有没有这个对象，没有就创建，有就直接拿来使用。

对于<jsp:setProperty>标签，可以直接使用value属性设置属性的值，这里面他内部是有一个类型转换的，可以将字符串值转化成8中基本类型的值，其他对象类型的转化是会报错的，比如我们现在Person类中有一个属性birthday是Date型的，那么我们这里就不能直接写成这样：

[html] view plain copy

<jsp:setProperty name="person" property="birthday" value="1990-08-01"/>

这样系统会报类型转化错误，

当然我们可以直接使用脚本表达式给这个属性值传递一个Date类型的对象，这样就不会又错了：

[html] view plain copy

<jsp:setProperty name="person" property="birthday" value="<%=new Date()%>"/>

当然我们也可以使用请求参数来设置属性值：

[html] view plain copy

<jsp:setProperty name="person" property="name" param="name"/>

我们访问：http://localhost:8080/JspDemo/bean.jsp?name=jiangwei,这样就可以进行属性赋值

同时这里的name属性中的填的是<jsp:useBean>标签中的id值，所以说，当我们在使用get/setProperty标签的时候一定是在useBean标签之后

关于取出属性值的标签<jsp:getProperty>的话就简单了。

下面来看一下怎么配置jsp的访问路径

[html] view plain copy

<servlet>

    <servlet-name>SimpleJspServlet</servlet-name>

    <jsp-file>/jsp/simple.jsp</jsp-file>

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

</servlet>

    ……

<servlet-mapping>

    <servlet-name>SimpleJspServlet</servlet-name>

    <url-pattern>/xxx/yyy.html</url-pattern>

</servlet-mapping>

# Html

## Html页面设置

### Lang

设置页面的语言，对大部分标签都有效，特定标签无效。设置后，起作用的标签都会识别为该语言

### Meta

这个标签是网页的元数据标签，html中应该写成 <meta/> 或者 <meta>。

<meta http-equiv="charset" content="iso-8859-1">

<meta http-equiv="expires" content="31 Dec 2008">

http-equiv 是 key ， content 是 value

<**meta** *charset=***"UTF-8"**>

设置编码还可以这样，这里制定的是页面的编码，浏览器会尝试按照指定编码来解码显示。注意文件是二进制字节码，我们看到的文件是解析过的，这个解码影响 url 。

## 表单

表单用于收集不同类型的用户输入。表单本身不可见，但是表单的子标签可见，表单元素是 <form> ，常用的子标签如下：

### Input

<input type="submit" value="OK" />，一定要这样闭合。而不是：<input type="submit" value="OK" ></input>。因为起始标签为必须，而关闭标签是禁止的。 关闭标签设置。<input>禁用关闭标签</input>。闭合的写法：<input type="submit" value="OK" />。<button>的起始标签和关闭标签都是必须的，如 <button>OK</button>。

2、<button>的值并不是写在value属性里，而是在起始、关闭标签之间，如上面的OK。同时<button>的值很广泛，

     有文字、图像、移动、水平线、框架、分组框、音频视频等。见《[button按钮设计初步](http://xuhui-8491.blog.163.com/blog/static/8074589120091129113714184/)》

3、可为button元素添加CSS样式。

     例如，<button style="width:150px;height:50px;border:0;"></button>

     其中：width:150px;height:50px; 为按钮的宽度、高度， 为按钮的背景色，

               border:0; 是去掉默认的边框。见《[button按钮设计初步](http://xuhui-8491.blog.163.com/blog/static/8074589120091129113714184/)》

4、鼠标单击事件、弹出信息的代码可直接写在<button>标签中，方法简单。

      例如，<button onclick="alert('弹出信息的内容');window.open('打开网页的地址')">按钮名称</button>

      其中：alert('弹出信息的内容'); 为单击时弹出的信息，window.open('打开网页的地址') 为打开的网页

无意中把<button>标签放到了<form>标签中，你会发现点击这个button变成了提交，相当于<input type="submit"/>

<html>

<body>

<form action="">

<button> button </button>

<input type="submit" value="input submit"/>

<input type="button" value="input button"/>

</form>

</body>

</html>

Input子元素有多种类型，包括输入框、单选、复选、file、password、reset(重置)、submit等等。

### Button

也是一个按钮，submit、reset、button三种都支持

<form action="demo\_form.html" method="get">  
  First name: <input type="text" name="fname"><br>  
  Last name: <input type="text" name="lname"><br>  
  <button type="submit" value="提交">提交</button>  
  <button type="reset" value="重置">重置</button>  
</form>

### Select

下拉列表

### 表单提交

表单默认的提交方式为，application/x-www-form-urlencoded, charset=utf-8，在提交的时候，会进行编码，并且区分 GET 和 POST 方式。当使用 GET 方法提交的时候，直接加入到 url 中，当使用 POST 方式提交的时候，放到 body 中，使用的是 key 和 value 的格式，没有使用 json 格式，需要用 @RequestParam 来接收，不能使用 @Requestbody 接收。

Form 支持的数据类型为： application/x-www-form-urlencoded、multipart/form-data 和text/plain 其它的数据类型，需要使用 js 脚本提交。

### 框结构

每个框结构包含一个 html 文档。下面是两个 document 文档。

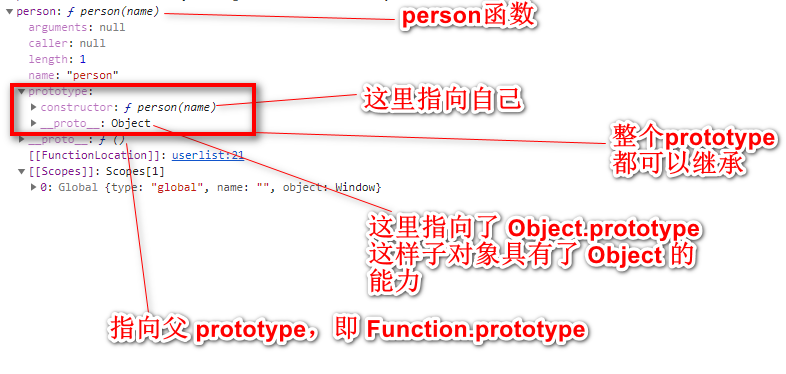
1. <!DOCTYPE html**>**
2. **<html** lang="en"**>**
3. **<meta** charset="UTF-8"**>**
4. **<title>**utf-8 测试**</title>**
6. **<frameSet** cols="%50,%50"**>**
7. **<frame** src="http://www.sina.com"**/>**
8. **<frame** src="http://news.baidu.com"**/>**
9. **</frameSet>**
10. **</html>**

### 层次结构

# Javascript

## 基础概念

#### Prototype



首先，不是每个对象都有 prototype 属性，有没有这个属性是生成这个对象的构造函数决定的。prototype属性，在 new 语句生成的子对象中，会将 \_\_proto\_\_ 指向父prototype 属性，这样只要是父 prototype 的属性，子对象都能使用，调用顺序是，子对象先查看自己的属性，若不存在，通过 \_\_proto\_\_ 查看父 prototype 是否有，再查看祖先prototype，直到 Object.prototype 这是顶级。我们也可以看出，不会主动到自己的 prototype 中去寻找它是为儿子节点准备的。

\_\_proto\_\_ 总是指向它的父 prototype 属性，这样就形成了一个链，叫 原型链。

#### Constructor

constructor 是 prototype 的一个属性，它指向对象的构造函数，如果对象没有 prototype 属性，那么constructor 只能从原型链上去找，这个不是必须的。

## 继承

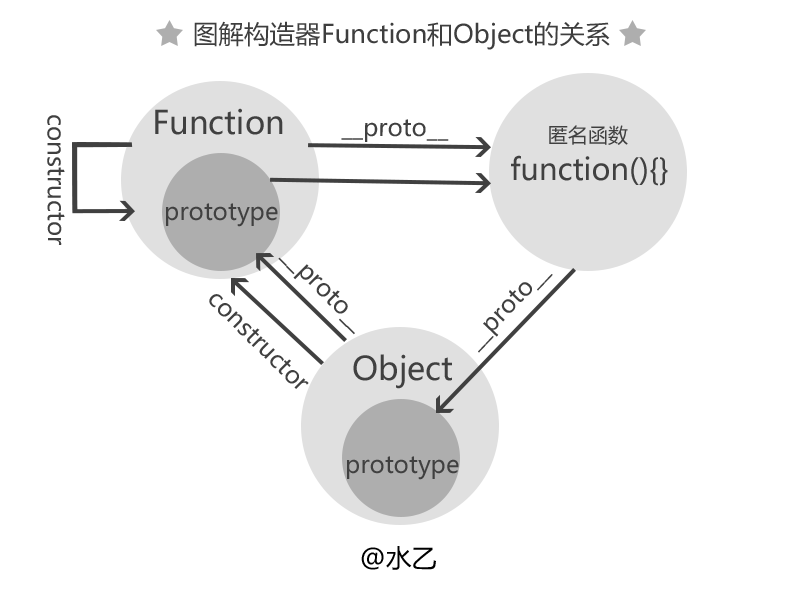
定义：对象 A 继承对象 B的充分必要条件是 A. \_\_proto\_\_ === B. prototype。

性值1：A 继承 B，则 A 获得了 B. prototype 的全部功能。

性值2：new 语句生成的子对象满足继承关系。

子对象 A 直接从父对象 B.prototype 获得功能，A.prototype 是给儿子对象 C 使用的。 A 先找自己的属性，再从原型链 A.\_\_proto\_\_ 往上找属性，找到即可使用，因此 \_\_proto\_\_属性决定向上的继承关系，prototype 决定向下的扩展关系。

## Function 和 Object



Js 一切皆对象。首先，顶级对象 Function 和 Object 的关系是，Function 创造了 Object 。 Object.prototype 也是 Function 的匿名子对象创建的， 并且 Object.\_\_proto\_\_ 指向了 Function.prototype ，Object 是 Function 。Function 创造了自己，但是 Function. \_\_proto\_\_ 没有指向 Function.prototype ，而是指向了匿名函数 A ，而 A. \_\_proto\_\_ 指向了 Object.prototype ， Function 是 Object 的孙子，所以 Function 是 Object 。那么功能的实现是谁完成了的呢？是 Object. prototype 我们看 Object 的原型链

Object.\_\_proto\_\_ |---> Function.\_\_proto\_\_ |---> A. \_\_proto\_\_ --> Object.prototype

可以看出，Object获得了Object.prototype 的功能，那么 Function 看起来就没啥用了，事实不是这样的。必须明确，对象不会凭空产生，一个对象的产生需要分配内存，初始化等，这些功能都需要函数调用，但是Object.prototype 只是一些简单函数调用，不涉及资源管理，所以 Object.\_\_proto\_\_ 指向了 Function.prototype 获取了资源管理的功能，这就是 Function 的作用。这也是为什么只有函数才能 new 操作，因为函数继承了 Function.prototype 的能力，才能够分配内存。而 Object 提供得构造函数，把这个能力去掉了，所以 Object 产生的子孙对象，没法执行 new 操作。因此一切皆为对象强调的是基本的逻辑操作，不涉及资源管理。关键字 function 定义的对象，并不是 Function 的儿子，而是孙子，这点要注意，生成的 function 是一个匿名函数的儿子。

## 基本语言

### Typeof

可以通过typeof判断，string/number/boolean都很清楚，但是typeof在判断到引用类型的时候，返回值只有object/function，无法判断出具体类型

### instanceof

A instanceof  B 充分必要条件是 A 是 Object 子对象， B 是函数子对象，并且

A.\_\_proto\_\_ 链上有一个引用了 B.prototype

|  |
| --- |
| Object.creat(null) 生成的对象，一个属性都没有，但是不是 null ，也不是 Object 的子对象。 |

### new

new Animal('cat') 等价于

{

    var obj = {};

    obj.\_\_proto\_\_ = Animal.prototype;

    var result = Animal.call(obj,"cat");

    return typeof result ==='object'? result : obj;

}

【1】创建一个空对象 obj；

【2】把 obj 的\_\_proto\_\_ 指向构造函数 Animal 的原型对象 prototype，此时便建立了 obj 对象的原型链：obj->Animal.prototype->Object.prototype->null

【3】在 obj 对象的执行环境调用 Animal 函数并传递参数 “ cat ” 。 相当于 var result = obj.Animal("cat")。

         当这句执行完之后，obj 便产生了属性 name 并赋值为 "cat"。关于 call 的用法请参考：[深入理解](http://www.cnblogs.com/onepixel/p/5143863.html) [call、apply 和 bind](http://www.cnblogs.com/onepixel/p/5143863.html)

【4】考察第 3 步的返回值，如果无返回值 或者 返回一个非对象值，则将 obj 作为新对象返回；否则会将 result 作为新对象返回。

根据以上过程，我们发现 cat 其实就是【4】的返回值，因此我们对 cat 对象的认知就多了一些：

cat 的原型链是：cat->Animal.prototype->Object.prototype->null

cat上新增了一个属性：name

### 普通对象注意

1. 对象属性名，不用加引号，有空格就要加引号
2. Var obj = {} 跟 var obj = new Object() 是一样的；同理还有 var arr = [] 跟 var arr = new Array() 都是一样的。四种基本类型。

### 自有属性

自有属性，是在对象中申明的，不是通过继承得到的属性，使用hasOwnProperty("toString") 判断 toString 是不是对象的自有属性。

### 属性枚举

|  |
| --- |
| var  obj = {};  Object.defineProperty(obj, "sex", {      value: "female",      enumerable: false  });  obj.propertyIsEnumerable("sex"); 返回false;  定义对象 obj ，添加属性 sex: “female”, 并且设置属性为不可枚举 |

判断属性可枚举，使用 propertyIsEnumerable 函数。属性可枚举当且仅当属性是自有属性，且 enumerable 属性为 true。

### 属性

对象属性是名字、值、和一组特性(attribute) 构成的，属性的特性也非常重要。

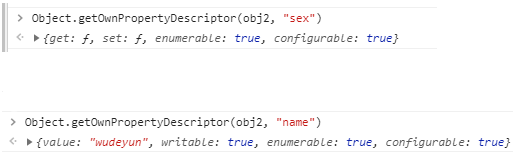
属性分为两种，对象属性和存储器属性，由 get 和set 关键字定义的属性是存储器属性，存储器属性分为 只读属性，只写属性，和读写属性。分别对应于 get 、set关键字。

|  |
| --- |
| var obj =  {  \_name: “wudeyun”,  get name(){return this.\_name},  set name(name){this.\_name = name}  } |

这里我们定义了一个数据属性，\_name 和 一个存储器属性 name 这两个属性都是可读写的，可以通过 obj.\_name 和 obj.name 调用这俩属性。存储器属性也可以看作特性的一种，是读写特性。

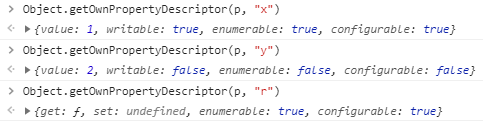
### 属性特性

对于 8.4.7 节中属性的特性，对于数据属性，四个属性分别为： 值(value)、可写性(writable)、可枚举(enumerable)、可配置性 (configurable) 。存储属性的读写由 set 和 get 关键字控制，所以不需要 value 和 writable 属性。它的四个属性是 读、写、可枚举(enumerable)、可配置性 (configurable) 。如下两个属性分别为存储器属性和数据属性：



|  |
| --- |
| 下面构造对象 p ，添加三个属性 x、y、r ，并指定各属性的特性，如下  var p = Object.defineProperties({}, {x:{value:1, writable: true, enumerable:true,configurable:true},y:{value:2, writable:false, enumerable:false, configurable:false},r:{get:function(){return Math.sqrt(this.x\*this.x+this.y\*this.y)}, enumerable:true,configurable:true}}); |

上面测试了x，y是否是 p 的自有属性，尝试改 y 的值失败，因为 y 的值不可改，y不可枚举，x 和 r可枚举。查看 x、y、r属性如下：



### 冻结

#### 扩展性

preventExtensions ，这个函数限制对象添加自有属性，可以在 prototype 上添加属性，可以删除属性，配置属性。

seal()方法封闭一个对象，阻止添加新属性并将所有现有属性标记为不可配置。当前属性的值只要可写就可以改变。

Object.freeze() 方法可以冻结一个对象。一个被冻结的对象再也不能被修改；冻结了一个对象则不能向这个对象添加新的属性，不能删除已有属性，不能修改该对象已有属性的可枚举性、可配置性、可写性，以及不能修改已有属性的值。此外，冻结一个对象后该对象的原型也不能被修改。freeze() 返回和传入的参数相同的对象。

## 编程技巧

### forEach( ) 函数

Array.forEach(function(value , index , array){

//value为遍历的当前元素，index为当前索引，array为正在操作的数组

//do something

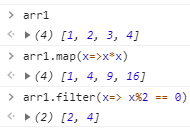
},thisArg) //thisArg为执行回调时的this值

forEach( ) 函数的参数是一个函数对象，返回值是 undefined 。会遍历数组，对每个元素执行该函数，写法有两种。

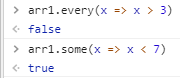
|  |
| --- |
| 方法一：  array.forEach(v=>{  console.log(v);  });  方法二：  array.forEach（function(v){  console.log(v);  }); |

### map() 函数

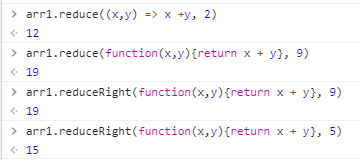
**map()** 方法创建一个新数组，其结果是该数组中的每个元素都调用一个提供的函数后返回的结果



### Every() 和 some()



### Reduce() 和 reduceRight()



## JSON

JSON 是原生对象，提供了 json APIs。JSON 语法是 JS 语法的子集，但是 JSON 不是 JS 。JSON. Stringify() 这个 API 的作用是产生 json 字符串。JSON.parse() 作用是将 json 字符串转化为 JS 对象。JS 对象是 JS 的内存对象，不仅仅是 json。

# Javascript 和 html

## 同源策略

定义：The **same-origin policy** is a critical security mechanism that restricts how a document or script loaded from one [origin](https://developer.mozilla.org/en-US/docs/Glossary/origin) can interact with a resource from another origin. It helps isolate potentially malicious documents, reducing possible attack vectors.

同源策略是一种关键的安全机制，它对从一个源加载的文档或脚本如何与另一个源的资源交互做出规定。 它有助于隔离潜在的恶意文档，减少可能的攻击媒介。

浏览器以协议头、域名、端口的组合标识一个页面的源。两个页面访问的协议头、域名及端口都相同，称两个页面具有相同的源。

下表列出与http://store.company.com/dir/page.html进行同源检测的结果：

| 页面地址 | 检测结果 | 原因 |
| --- | --- | --- |
| [http://store.company.com/dir2...](http://store.company.com/dir2/other.html) | 成功 |  |
| [http://store.company.com/dir/...](http://store.company.com/dir/inner/another.html) | 成功 |  |
| [https://store.company.com/sec...](https://store.company.com/secure.html) | 失败 | 协议差异（https/http） |
| [http://store.company.com](http://store.company.com/):81/dir/etc.html | 失败 | 端口差异（81/80） |
| [http://news.company.com/dir/o...](http://news.company.com/dir/other.html) | 失败 | 子域名差异（news/store） |

同源策略下的资源访问

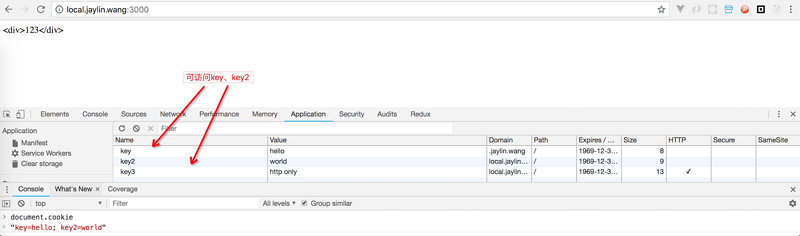
跨源页面之间使用 XMLHttpRequest 和 Fetch API 请求数据，通过<img>,<link>，<script>等 HTML 标签引用资源，这里统称为资源访问，都将受到浏览器的同源策略的控制。跨源资源访问时，不同的交互方式默认有不同的浏览器限制：

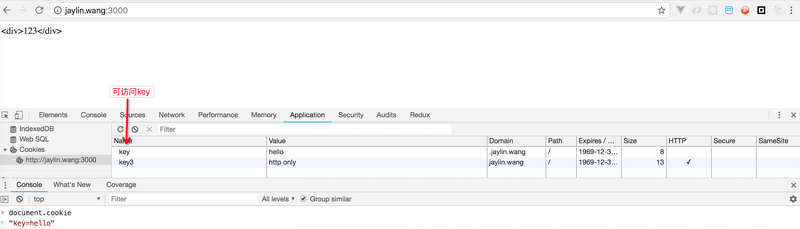
1. 跨域写，例如表单数据提交至其他源，浏览器默认允许。
2. 跨域嵌入，例如<img src="..." />,<iframe src="..."/>，<script src="..." />等标签引入其他源资源，浏览器默认是允许的。
3. 跨域读，例如通过 ajax 请求获取其他源返回数据，浏览器默认是禁止的。

同源策略下的存储访问

1. 浏览器最初使用 cookie 作为本地存储，H5新增 LocalStorage 和 IndexedDB 的存储方式。

浏览器通过检测 cookie 的 domain 属性控制源，例如有 cookie: key=hello 的 domain 设为.jaylin.wang，local.jaylin.wang,www.jaylin.wang,jaylin.wang均可访问，有 cookie: key2=world 的 domain 设置为local.jaylin.wang, 则只有local.jaylin.wang可访问。cookie 的 httpOnly 属性可控制cookie只能在服务端读取。





LocalStorage 和 IndexedDB 会根据不同源相互独立，数据的写入和读取都是针对当前源的存储，目前浏览器没有提供跨源读写的支持。

可改变的同源策略

浏览器默认的同源策略机制下，也为开发者提供了改变的机制，列出常用的几种方式：

domain setting

在一个页面中，通过设置 document.domain 的值，可以将子域名的源设置为父级域名。

CSP（Content Security Policy）

浏览器支持通过Content-Security-Policy响应头限制内容来源，确保加载内容的可信度。这个机制可以减少 XSS 攻击，

CORS（Cross-Origin Resource Sharing）

浏览器支持 CORS 配置可信的跨域读资源，以适应WEB应用扩展的需求。

恶意站点检测机制

用户在使用浏览器浏览网页的过程中，不乏有恶意网站给用户带来潜在的安全问题。常见的恶意网站有以下几类：

欺骗性网站  
欺骗性网站又被称为钓鱼网站，伪装成合法网站，诱导用户操作，从而窃取用户的隐私信息。

攻击性网站  
网站中包含恶意脚本，能够控制访问用户浏览器，窃取用户信息，引导用户进行高危操作。

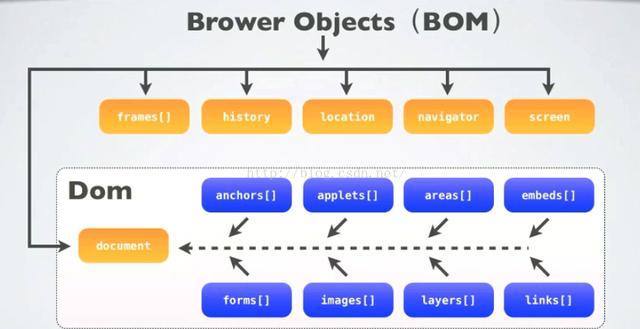
恶意软件  
软件携带病毒，可能导致计算机被攻击的危险软件。

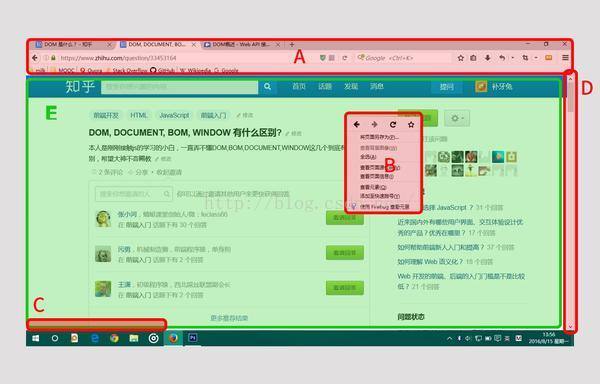
浏览器通常会维护一份被举报和恶意网站列表，检测用户将要访问的目标网站存在于站点列表中，则提示警告信息以阻止用户的继续访问。

## BOM 和 DOM

BOM缺乏标准，JavaScript语法的标准化组织是ECMA，DOM的标准化组织是W3C，BOM最初是Netscape浏览器标准的一部分。

BOM（浏览器对象模型）是对浏览器的抽象。DOM （文档对象模型）是对 html 页面的抽象，整个 BOM 模型结构如下：





归DOM管的：

E区（里面有index.html，CSS和JS，部署在服务器上，我们可以通过浏览器的地址栏输入URL然后回车将这个document加载到本地，浏览，右键查看源代码等）。

归BOM管的：

A区（浏览器的标签页，地址栏，搜索栏，书签栏，窗口放大还原关闭按钮，菜单栏等等）

B区（浏览器的右键菜单）

C区（document加载时的状态栏，显示http状态码等）

D区（滚动条scroll bar）

BOM怎么用？

window 对象是 BOM 模型的 js 实例，window 对象包含属性document、location、navigator、screen、history、frames，而 document 对象是 DOM 模型的实例。这些对象给 js 提供了一整套与浏览器相关的 API ，js 利用这些 API 可以控制浏览器的行为。因此 window 对象是浏览器环境下 js 代码的顶级对象，所有的对象和函数都是 window 对象的子孙对象。

### window 对象

window 对象是JavaScript层级中的顶层对象，它代表一个浏览器窗口、标签或一个框架。

很多浏览器对[新开](https://www.baidu.com/s?wd=%E6%96%B0%E5%BC%80&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd)弹出窗口进行了限制，弹窗在默认情况下会被禁止。这样可能会造成按钮/链接/脚本不工作的假象。

window.open("http:www.techbrood.com","\_blank")

以上JS语句是想新开一个窗口而不是转发，或者新标签。究竟是弹窗还是新建标签页是由浏览器的设置决定的，并不由第二个参数\_blank而确定。

如果用户浏览器的设置是弹出新的窗口，那么[JS脚本](https://www.baidu.com/s?wd=JS%E8%84%9A%E6%9C%AC&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd)无法影响这种情况下的行为。

如果用户浏览器的设置是新开一个标签页（最新的主流浏览器基本上默认情况下都是这个设置），

那么可以通过JS脚本来控制新建标签还是新建窗口的行为，必须是在[click事件](https://www.baidu.com/s?wd=click%E4%BA%8B%E4%BB%B6&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd)的处理过程中，代码如下：

[javascript] [view plain](http://blog.csdn.net/iefreer/article/details/22605181) [copy](http://blog.csdn.net/iefreer/article/details/22605181)

<a class="link">Link</a>

<script  type="text/javascript">

     $("a.link").on("click",function(){

         window.open('www.techbrood.com','\_blank');

     });

</script>

上述语句可以达到和如下静态html代码相同的效果：

<a href="www.techbrood.com" target='\_blank'>Link</a>

如果在click事件中使用了ajax请求，那么注意ajax默认的异步请求模式会让click事件失去时效，

所以如果要达到新建标签页的效果，务必保证async参数为false。

### location 对象

window对象的子对象，包含有关当前 URL 的信息。基本上就是http协议的一些东西。

## html页面加载顺序

加载顺序

|  |
| --- |
| <html>  <head>  <!-- 引用外部JS文件 -->  <script src="..........."></script>  <!--引用外部css样式 -->  <link href="............."/>  <style>  ..............  </style>  <script>  ..............  </script>  </head>  <body>  <script>  ..............  </script>  </body>  </html> |

从上到下运行，先解析head标签中的代码，

head标签中会包含一些引用外部文件的代码，从开始运行就会下载这些被引用的外部文件， 当遇到script标签的时候 浏览器暂停解析（不是暂停下载），将控制权交给JavaScript引擎（解释器），如果<script>标签引用了外部脚本，就下载该脚本，否则就直接执行，执行完毕后将控制权交给浏览器渲

当head中代码解析完毕，会开始解析body中的代码，如果此时head中引用的外部文件没有下载完，将会继续下载，浏览器解析body代码中的元素，会按照head中声明一部分样式去解析，如果此时遇到body标签中的<script>，同样会将控制权交给JavaScript引擎来解析JavaScript， 解析完毕后将控制权交还给浏览器渲染引擎。当body中的代码全部执行完毕、并且整个页面的css样式加载完毕后，css会重新渲染整个页面的html元素。

按照之前的描述，<script>写到body标签内靠后比较好， 因为JavaScript 会操作html元素， 如果在body加载完之前写JavaScript 会造成JavaScript 找不到页面元素，但是我们经常将<script>写到head中，body中不会有大量的js代码，body中的html代码结构会比较清晰

window.onload： 等待页面中的所有内容加载完毕之后才会执行

$(document).ready()： 页面中所有DOM结构绘制完毕之后就能够执行

可以这样理解：window.onload 和 $(document).ready()/$(function(){}); 相当于  写在body 内  最靠后的<script> 代码段

### 引入 Js 脚本

1. 直接写在 html 元素中

<标签开始 on+事件类型=“[js](https://www.baidu.com/s?wd=js&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd)代码”></标签结束>

1. 内部引入

<script>

alert("这是js的内部引入");

</script>

1. 外部引入

<script type="text/javascript" src="demo.js"/></script>

这种方式的脚本如果还有内容，是不会运行的。如

<script type="text/javascript" src="demo.js">

alert(“hello world!”);

</script>

这里面的 alert() 是不会运行的。

### 引入 css 脚本

CSS的引入方式共有三种：行内样式、内部样式表、外部样式表。

一、行内样式

使用style属性引入CSS样式。

示例：  
<h1 style="color:red;">style属性的应用</h1>  
<p  style="font-size:14px;color:green;">直接在HTML标签中设置的样式</p>  
实际在写页面时不提倡使用，在测试的时候可以使用。

例如：

[复制代码](javascript:void(0);)

<!DOCTYPE>

<html>

<head>

<meta charset="utf-8" />

<title>行内样式</title>

</head>

<body>

<!--使用行内样式引入CSS-->

<h1 style="color:red;">Leaping Above The Water</h1>

<p style="color:red;font-size:30px;">我是p标签</p>

</body>

</html>

[复制代码](javascript:void(0);)

二、内部样式表  
在style标签中书写CSS代码。style标签写在head标签中。  
示例：  
<head>  
   <style type="text/css">  
      h3{  
            color:red;  
         }  
   </style>  
</head>

例如：

[复制代码](javascript:void(0);)

<!DOCTYPE>

<html>

<head>

<meta charset="utf-8" />

<title>内部样式表</title>

<!--使用内部样式表引入CSS-->

<style type="text/css">

div{

background: green;

}

</style>

</head>

<body>

<div>我是DIV</div>

</body>

</html>

[复制代码](javascript:void(0);)

三、外部样式表  
CSS代码保存在扩展名为.css的样式表中  
HTML文件引用扩展名为.css的样式表，有两种方式：链接式、导入式。  
语法：  
1、链接式  
<link type="text/css" rel="styleSheet"  href="CSS文件路径" />  
2、导入式  
<style type="text/css">  
  @import url("css文件路径");  
</style>

例如：

[复制代码](javascript:void(0);)

<!DOCTYPE>

<html>

<head>

<meta charset="utf-8" />

<title>外部样式表</title>

<!--链接式:推荐使用-->

<link rel="stylesheet" type="text/css" href="css/style.css" />

<!--导入式-->

<style type="text/css">

@import url("css/style.css");

</style>

</head>

<body>

<ol>

<li>1111</li>

<li>2222</li>

</ol>

</html>

[复制代码](javascript:void(0);)

链接式和导入式的区别  
<link>  
1、属于XHTML  
2、优先加载CSS文件到页面  
@import  
1、属于CSS2.1  
2、先加载HTML结构在加载CSS文件。

四、CSS中的优先级

1、样式优先级

行内样式>内部样式>外部样式(后两者是就近原则)

例如：

行内样式和内部样式比较优先级：

[复制代码](javascript:void(0);)

<!DOCTYPE>

<html>

<head>

<meta charset="utf-8" />

<title>行内样式和内部样式表的优先级</title>

<!--内部部样式表-->

<style type="text/css">

p{

color: blue;

}

</style>

</head>

<body>

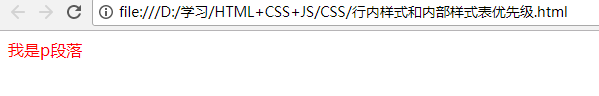
<!--行内样式-->

<p style="color: red;">我是p段落</p>

</html>

[复制代码](javascript:void(0);)

浏览器运行效果：



结论：行内样式优先级高于内部样式表。

内部样式表和外部样式表比较优先级：

a、内部样式表在外部样式表上面

[复制代码](javascript:void(0);)

<!DOCTYPE>

<html>

<head>

<meta charset="utf-8" />

<title>内部样式表和外部样式表的优先级</title>

<!--内部部样式表-->

<style type="text/css">

p{

color: blue;

}

</style>

<!--外部样式表-->

<link rel="stylesheet" type="text/css" href="css/style.css" />

</head>

<body>

<p>我是p段落</p>

<div>我是div</div>

<ol>

<li>1111</li>

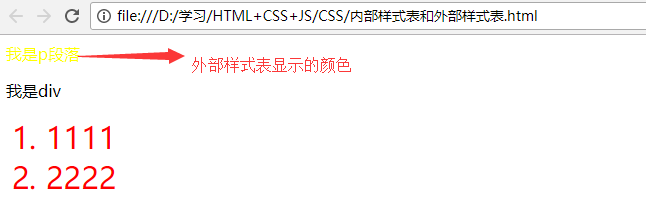
<li>2222</li>

</ol>

</html>

[复制代码](javascript:void(0);)

浏览器运行效果：



b、外部样式表在内部样式表上面

[复制代码](javascript:void(0);)

<!DOCTYPE>

<html>

<head>

<meta charset="utf-8" />

<title>内部样式表和外部样式表的优先级</title>

<!--外部样式表-->

<link rel="stylesheet" type="text/css" href="css/style.css" />

<!--内部部样式表-->

<style type="text/css">

p{

color: blue;

}

</style>

</head>

<body>

<p>我是p段落</p>

<div>我是div</div>

<ol>

<li>1111</li>

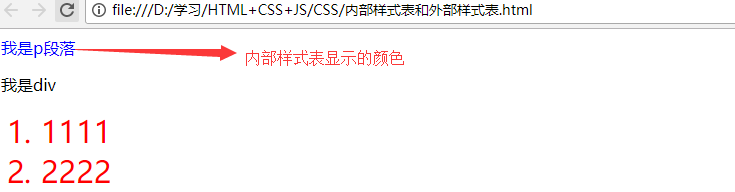
<li>2222</li>

</ol>

</html>

[复制代码](javascript:void(0);)

浏览器运行效果：



结论：内部样式表和外部样式表使用就近原则，即谁写在下面以谁为准。

注意：导入式和链接式的优先级也是使用就近原则。

2、选择器优先级

优先级：ID选择器>类选择器>标签选择器

[复制代码](javascript:void(0);)

<!DOCTYPE>

<html>

<head>

<meta charset="utf-8" />

<title>选择器的优先级</title>

<style type="text/css">

#a{

color: green;

}

.b{

color: yellow;

}

h2{

color: red;

}

</style>

</head>

<body>

<h2>111</h2> <!--红色-->

<h2 id="a" class="b">222</h2> <!--绿色-->

<h2 class="b">333</h2><!--黄色-->

</html>

## 事件

事件是可以被 JavaScript 侦测到的行为。在事件发生之前，要把与事件相关的响应函数全部注册到位，这样才能响应事件。

## jQuery

### jQuery对象

$ 是 jQuery 别名， $( ) 和 jQuery( ) 是同一个函数。$(document).ready() 函数与 $ ( ) 等价，但是不是同一个函数。jQuery( ) 内的函数，必须要等到页面 ready 之后才能执行。这三个函数表达的是同一个意思。

$(document).ready和window.onload都是在都是在页面加载完执行的函数，大多数情况下差别不大，但也是有区别的。

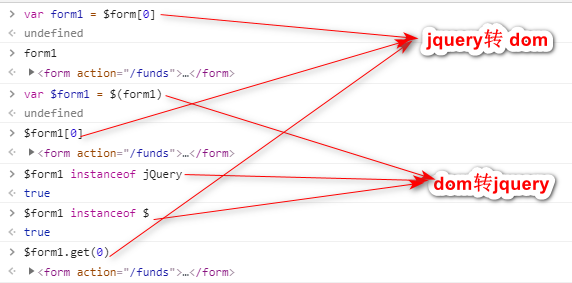
$(document).ready:是DOM结构绘制完毕后就执行，不必等到加载完毕。 意思就是DOM树加载完毕，就执行，不必等到页面中图片或其他外部文件都加载完毕。并且可以写多个.ready。

window.onload:是页面所有元素都加载完毕，包括图片等所有元素。只能执行一次。

ready 事件要早于 window.onload 事件。

### jQuery 对象转 dom 对象

使用 jQuery 选择器获取的对象，都是被 jQuery 封装过的对象，不再是 document 对象，即 dom 对象。可以通过下标方式获取到被封装的 dom 对象。



### 过滤器

#### hasClass()

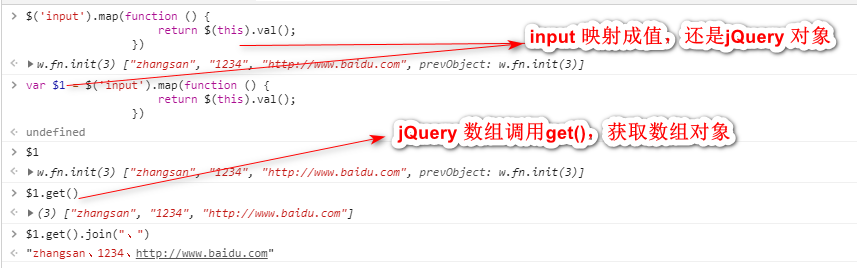
判断jQuery选择的标签列表中，有没有该类型标签，有返回 true，否则返回 false。



#### eq() 索引过滤

#### eq() 是子元素得索引

#### map() 映射



#### not()

#### slice()

截取 [start, end)

### 查找

#### children(expr)

children(expr) 所有子标签的 jQuery 对象

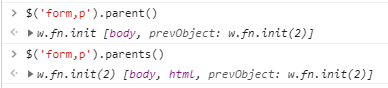
#### contents()

子孙标签的 jQuery 对象

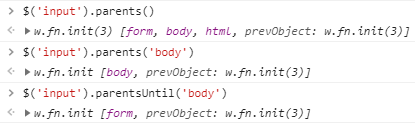
#### find(expr)

查找对应标签 jQuery 对象

#### parents()/parent()/parentsUntil()/closest()



parent() 返回 form 和 p 元素的父亲，body元素的 jQuery 对象。而 parents() 返回的是祖先元素， body 和 html 元素



parents() 一共有三个元素 form, body, html。parentsUntil() 返回了 form元素，不包括 body 元素自身 [ ) 右边是开区间。

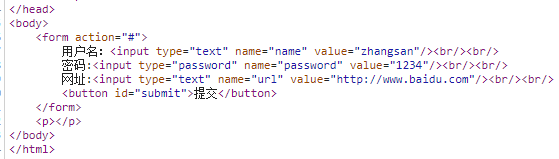


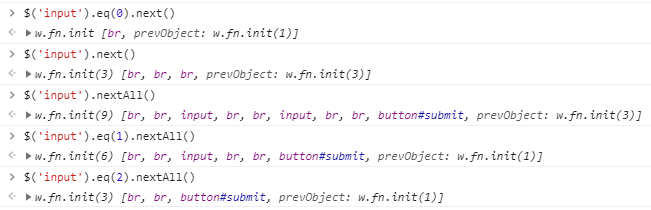
closest() 最近的父元素，从下面往上递归。

#### prev()， prevAll(), prevUntil()

参考 next()

#### next()， nextAll()， nextUntil()



第一个input 后面是 br元素。每个input后面都是 br ，一共三个 br 元素，$('input').next()就获得了这三个 br。 nextAll 是从第一个匹配的元素开始算的，都不包括 body 及其以上的元素。

#### siblings() 查全部兄弟

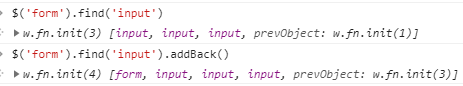
#### add(expr)

往 jQuery 选中的数组中加元素



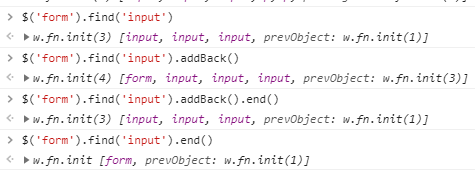
往 input 数组中，加入了 p 元素。

#### addBack()



将元素串联。

#### end()链的前一个



### Document操作

#### append() 向元素内部追加内容。

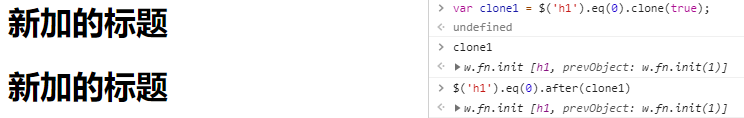


#### after()

元素后面加入标题

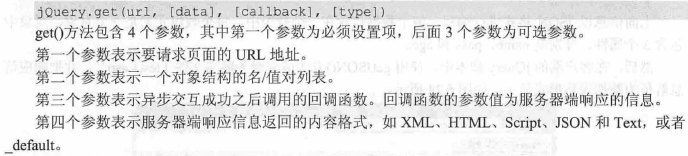


#### clone()



### jQuery Ajax

#### get()



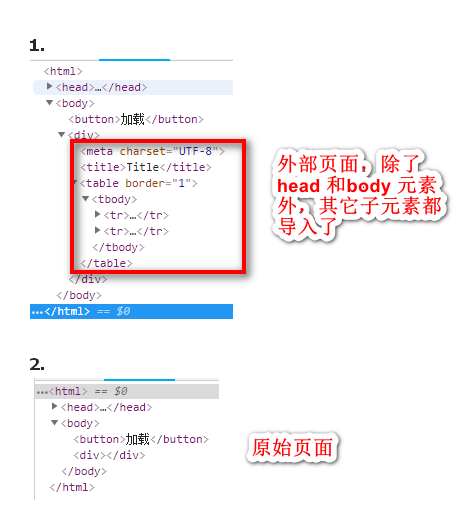
#### post()

这个跟get 是一样的。这个提交的是 formdata

#### ajax()

ajax() 是 jQuery 对 AJAX 的实现，get 和 post 均以此为基础。ajaxStart()、ajaxSend()、ajaxSuccess()、ajaxCoplate()、ajaxStop()、ajaxError() 是与 ajax 请求事件对应得方法。

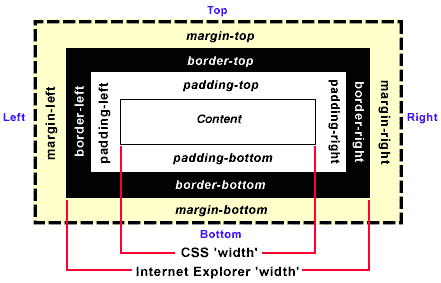
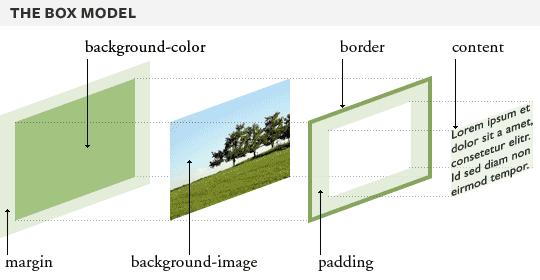
### 导入外部页面



#### 

# Css

## [CSS padding margin border属性详解](https://www.cnblogs.com/linjiqin/p/3556497.html)

图解CSS padding、margin、border属性  
  
W3C组织建议把所有网页上的对像都放在一个盒(box)中，设计师可以通过创建定义来控制这个盒的属性，这些对像包括段落、列表、标题、图片以及层。盒模型主要定义四个区域：内容(content)、内边距(padding)、边框(border)和外边距(margin)。对于初学者，经常会搞不清楚margin，background-color，background-image，padding，content，border之间的层次、关系和相互影响。这里提供一张盒模型的3D示意图，希望便于你的理解和记忆。  


margin：层的边框以外留的空白

background-color：背景颜色

background-image：背景图片

padding：层的边框到层的内容之间的空白

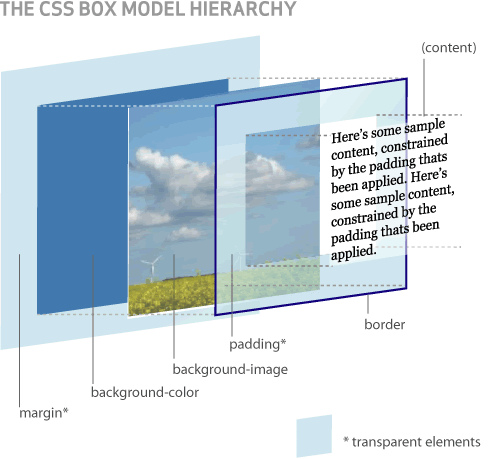
border：边框

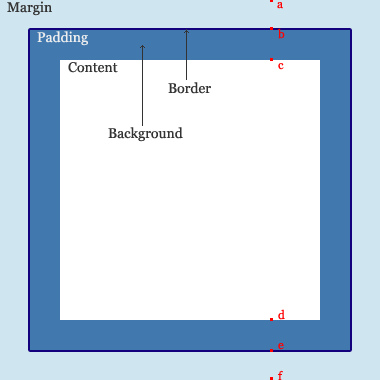
content：内容

接下来将讲述HTML和CSS的关键——盒子模型(Box model)。理解Box model的关键便是margin和padding属性，而正确理解这两个属性也是学习用css布局的关键。

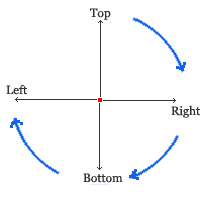
注: 为什么不翻译margin和padding?   
原因一: 在汉语中并没有与之相对应的词语；   
原因二: 即使有这样的词语, 由于在编写css代码时, 必须使用margin和padding, 如果我们总用汉语词语代替其来解释的话, 到了实际应用时容易混淆margin和padding的概念。

如果有一点Html基础的话，就应该了解一些基本元素(Element)，如p、h1~h6、br、div、li、ul、img等。如果将这些元素细分，又可以分别为顶级(top-level)元素、块级(block-level)元素和内联(inline)元素。

块级元素是构成一个html的主要和关键元素，而任意一个块级元素均可以用Box model来解释说明。  
Box Model: 任意一个块级元素均由content(内容)、padding、background(包括背景颜色和图片)、border(边框)、margin五个部分组成。  
立体图如下:  


平面图如下:  


根据以上两图，相信大家对于Box model会有个直观的认识。

**以下说明margin和padding属性:**  
1、margin: 包括margin-top、margin-right、margin-bottom、margin-left，**控制块级元素之间的距离**，它们是透明不可见的。根据上、 右、下、左的顺时针规则，可以写为 margin: 40px 40px 40px 40px;  
为便于记忆，请参考下图:  


当上下、左右margin值分别一致, 可简写为:

margin: 40px 40px;

前一个40px代表上下margin值，后一个40px代表左右margin值。  
当上下左右margin值均一致，可简写为:

margin: 40px;

2、Padding: 包括padding-top、padding-right、padding-bottom、padding-left，**控制块级元素内部**，content与border之间的距离，其代码，简写请参考margin属性的写法。

至此，我们已经基本了解margin和padding属性的基本用法。但是，在实际应用中，却总是发生一些让你琢磨不透的事，而它们又或多或少的与margin有关。

注: 当你想让两个元素的content在垂直方向(vertically)分隔时，既可以选择padding-top/bottom，也可以选择margin-top/bottom，再此[**Ruthless**](http://www.cnblogs.com/linjiqin/)建议你尽量使用padding-top/bottom来达到你的目的，这是因为css中存在Collapsing margins(折叠的margins)的现象。

Collapsing margins: margins折叠现象只存在于临近或有从属关系的元素，垂直方向的margin中。

详细说明如下：   
如果只提供一个，将用于全部的四条边；  
如果提供两个，第一个用于上－下，第二个用于左－右；   
如果提供三个，第一个用于上，第二个用于左－右，第三个用于下；   
如果提供全部四个参数值，将按上－右－下－左的顺序作用于四边。

body { padding: 36px;} //对象四边的补丁边距均为36px

body { padding: 36px 24px; } //上下两边的补丁边距为36px，左右两边的补丁边距为24px

body { padding: 36px 24px 18px; } //上、下两边的补丁边距分别为36px、18px，左右两边的补丁边距为24px

body { padding: 36px 24px 18px 12px; } //上、右、下、左补丁边距分别为36px、24px、18px、12px

# Web

## 加密

常见的加密[算法](http://lib.csdn.net/base/31)可以分成三类，对称加密算法，非对称加密算法和Hash算法。

对称加密

    指加密和解密使用相同密钥的加密算法。对称加密算法的优点在于加解密的高速度和使用长密钥时的难破解性。假设两个用户需要使用对称加密方法加密然后交换数据，则用户最少需要2个密钥并交换使用，如果企业内用户有n个，则整个企业共需要n×(n-1) 个密钥，密钥的生成和分发将成为企业信息部门的恶梦。对称加密算法的安全性取决于加密密钥的保存情况，但要求企业中每一个持有密钥的人都保守秘密是不可能的，他们通常会有意无意的把密钥泄漏出去——如果一个用户使用的密钥被入侵者所获得，入侵者便可以读取该用户密钥加密的所有文档，如果整个企业共用一个加密密钥，那整个企业文档的保密性便无从谈起。

常见的对称加密算法：DES、3DES、DESX、Blowfish、IDEA、RC4、RC5、RC6和AES

非对称加密

    指加密和解密使用不同密钥的加密算法，也称为公私钥加密。假设两个用户要加密交换数据，双方交换公钥，使用时一方用对方的公钥加密，另一方即可用自己的私钥解密。如果企业中有n个用户，企业需要生成n对密钥，并分发n个公钥。由于公钥是可以公开的，用户只要保管好自己的私钥即可，因此加密密钥的分发将变得十分简单。同时，由于每个用户的私钥是唯一的，其他用户除了可以可以通过信息发送者的公钥来验证信息的来源是否真实，还可以确保发送者无法否认曾发送过该信息。非对称加密的缺点是加解密速度要远远慢于对称加密，在某些极端情况下，甚至能比非对称加密慢上1000倍。

常见的非对称加密算法：RSA、ECC（移动设备用）、Diffie-Hellman、El Gamal、DSA（数字签名用）

Hash算法

    Hash算法特别的地方在于它是一种单向算法，用户可以通过Hash算法对目标信息生成一段特定长度的唯一的Hash值，却不能通过这个Hash值重新获得目标信息。因此Hash算法常用在不可还原的密码存储、信息完整性校验等。

常见的Hash算法：MD2、MD4、MD5、HAVAL、SHA、SHA-1、HMAC、HMAC-MD5、HMAC-SHA1

    加密算法的效能通常可以按照算法本身的复杂程度、密钥长度（密钥越长越安全）、加解密速度等来衡量。上述的算法中，除了DES密钥长度不够、MD2速度较慢已逐渐被淘汰外，其他算法仍在目前的加密系统产品中使用。

加密算法的选择

    前面的章节已经介绍了对称解密算法和非对称加密算法，有很多人疑惑：那我们在实际使用的过程中究竟该使用哪一种比较好呢？

    我们应该根据自己的使用特点来确定，由于非对称加密算法的运行速度比对称加密算法的速度慢很多，当我们需要加密大量的数据时，建议采用对称加密算法，提高加解密速度。

对称加密算法不能实现签名，因此签名只能非对称算法。

由于对称加密算法的密钥管理是一个复杂的过程，密钥的管理直接决定着他的安全性，因此当数据量很小时，我们可以考虑采用非对称加密算法。

    在实际的操作过程中，我们通常采用的方式是：采用非对称加密算法管理对称算法的密钥，然后用对称加密算法加密数据，这样我们就集成了两类加密算法的优点，既实现了加密速度快的优点，又实现了安全方便管理密钥的优点。

    如果在选定了加密算法后，那采用多少位的密钥呢？一般来说，密钥越长，运行的速度就越慢，应该根据的我们实际需要的安全级别来选择，一般来说，RSA建议采用1024位的数字，ECC建议采用160位，AES采用128为即可。

二.          加密[算法](http://lib.csdn.net/base/31)介绍

对称加密算法

对称加密算法用来对敏感数据等信息进行加密，常用的算法包括：

DES（Data Encryption Standard）：数据加密标准，速度较快，适用于加密大量数据的场合。

3DES（Triple DES）：是基于DES，对一块数据用三个不同的密钥进行三次加密，强度更高。

AES（Advanced Encryption Standard）：高级加密标准，是下一代的加密算法标准，速度快，安全级别高；

AES与3DES的比较

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 算法名称 | 算法类型 | 密钥长度 | 速度 | 解密时间（建设机器每秒尝试255个密钥） | 资源消耗 |
| AES | 对称block密码 | 128、192、256位 | 高 | 1490000亿年 | 低 |
| 3DES | 对称feistel密码 | 112位或168位 | 低 | 46亿年 | 中 |

非对称算法

RSA：由 RSA 公司发明，是一个支持变长密钥的公共密钥算法，需要加密的文件块的长度也是可变的；

DSA（Digital Signature Algorithm）：数字签名算法，是一种标准的 DSS（数字签名标准）；

ECC（Elliptic Curves Cryptography）：椭圆曲线密码编码学。

ECC和RSA相比，在许多方面都有对绝对的优势，主要体现在以下方面：

抗攻击性强。相同的密钥长度，其抗攻击性要强很多倍。

计算量小，处理速度快。ECC总的速度比RSA、DSA要快得多。

存储空间占用小。ECC的密钥尺寸和系统参数与RSA、DSA相比要小得多，意味着它所占的存贮空间要小得多。这对于加密算法在IC卡上的应用具有特别重要的意义。

带宽要求低。当对长消息进行加解密时，三类密码系统有相同的带宽要求，但应用于短消息时ECC带宽要求却低得多。带宽要求低使ECC在无线网络领域具有广泛的应用前景。

下面两张表示是RSA和ECC的安全性和速度的比较。

|  |  |  |  |
| --- | --- | --- | --- |
| 攻破时间(MIPS年) | RSA/DSA(密钥长度) | ECC密钥长度 | RSA/ECC密钥长度比 |
| 104 | 512 | 106 | 5：1 |
| 108 | 768 | 132 | 6：1 |
| 1011 | 1024 | 160 | 7：1 |
| 1020 | 2048 | 210 | 10：1 |
| 1078 | 21000 | 600 | 35：1 |
| 攻破时间(MIPS年) | RSA/DSA(密钥长度) | ECC密钥长度 | RSA/ECC密钥长度比 |
| 104 | 512 | 106 | 5：1 |
| 108 | 768 | 132 | 6：1 |
| 1011 | 1024 | 160 | 7：1 |
| 1020 | 2048 | 210 | 10：1 |
| 1078 | 21000 | 600 | 35：1 |

RSA和ECC安全模长得比较

|  |  |  |
| --- | --- | --- |
| 功能 | Security Builder 1.2 | BSAFE 3.0 |
| 163位ECC(ms) | 1,023位RSA(ms) |
| 密钥对生成 | 3.8 | 4,708.3 |
| 签名 | 2.1(ECNRA) | 228.4 |
| 3.0(ECDSA) |
| 认证 | 9.9(ECNRA) | 12.7 |
| 10.7(ECDSA) |
| Diffie—Hellman密钥交换 | 7.3 | 1,654.0 |

RSA和ECC速度比较

散列算法

散列是信息的提炼，通常其长度要比信息小得多，且为一个固定长度。加密性强的散列一定是不可逆的，这就意味着通过散列结果，无法推出任何部分的原始信息。任何输入信息的变化，哪怕仅一位，都将导致散列结果的明显变化，这称之为雪崩效应。散列还应该是防冲突的，即找不出具有相同散列结果的两条信息。具有这些特性的散列结果就可以用于验证信息是否被修改。

单向散列函数一般用于产生消息摘要，密钥加密等，常见的有：

l         MD5（Message Digest Algorithm 5）：是RSA数据安全公司开发的一种单向散列算法，非可逆，相同的明文产生相同的密文。

l         SHA（Secure Hash Algorithm）：可以对任意长度的数据运算生成一个160位的数值；

SHA-1与MD5的比较

因为二者均由MD4导出，SHA-1和MD5彼此很相似。相应的，他们的强度和其他特性也是相似，但还有以下几点不同：

l         对强行供给的安全性：最显著和最重要的区别是SHA-1摘要比MD5摘要长32 位。使用强行技术，产生任何一个报文使其摘要等于给定报摘要的难度对MD5是2128数量级的操作，而对SHA-1则是2160数量级的操作。这样，SHA-1对强行攻击有更大的强度。

l         对密码分析的安全性：由于MD5的设计，易受密码分析的攻击，SHA-1显得不易受这样的攻击。

l         速度：在相同的硬件上，SHA-1的运行速度比MD5慢。

对称与非对称算法比较

    以上综述了两种加密方法的原理，总体来说主要有下面几个方面的不同：

l         在管理方面：公钥密码算法只需要较少的资源就可以实现目的，在密钥的分配上，两者之间相差一个指数级别（一个是n一个是n2）。所以私钥密码算法不适应广域网的使用，而且更重要的一点是它不支持数字签名。

l         在安全方面：由于公钥密码算法基于未解决的数学难题，在破解上几乎不可能。对于私钥密码算法，到了AES虽说从理论来说是不可能破解的，但从计算机的发展角度来看。公钥更具有优越性。

l         从速度上来看：AES的软件实现速度已经达到了每秒数兆或数十兆比特。是公钥的100倍，如果用硬件来实现的话这个比值将扩大到1000倍。

三.            加密算法的选择

由于非对称加密算法的运行速度比对称加密算法的速度慢很多，当我们需要加密大量的数据时，建议采用对称加密算法，提高加解密速度。

对称加密算法不能实现签名，因此签名只能非对称算法。

由于对称加密算法的密钥管理是一个复杂的过程，密钥的管理直接决定着他的安全性，因此当数据量很小时，我们可以考虑采用非对称加密算法。

在实际的操作过程中，我们通常采用的方式是：采用非对称加密算法管理对称算法的密钥，然后用对称加密算法加密数据，这样我们就集成了两类加密算法的优点，既实现了加密速度快的优点，又实现了安全方便管理密钥的优点。

那采用多少位的密钥呢？ RSA建议采用1024位的数字，ECC建议采用160位，AES采用128为即可。

四.            密码学在现代的应用

保密通信：保密通信是密码学产生的动因。使用公私钥密码体制进行保密通信时，信息接收者只有知道对应的密钥才可以解密该信息。

数字签名：数字签名技术可以代替传统的手写签名，而且从安全的角度考虑，数字签名具有很好的防伪造功能。在政府机关、军事领域、商业领域有广泛的应用环境。

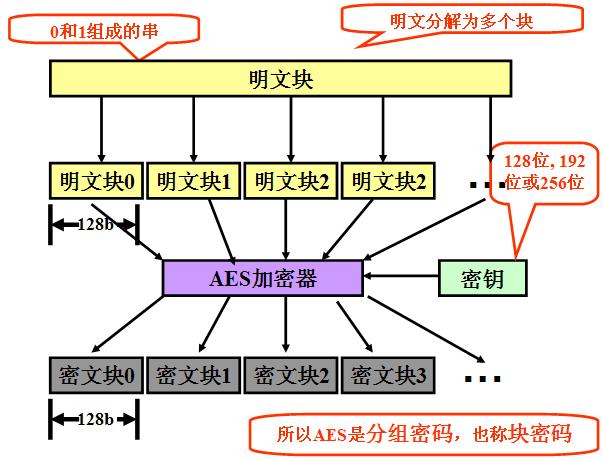
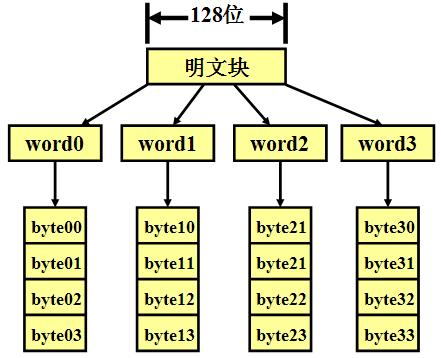
秘密共享：秘密共享技术是指将一个秘密信息利用密码技术分拆成n个称为共享因子的信息，分发给n个成员，只有k(k≤n)个合法成员的共享因子才可以恢复该秘密信息，其中任何一个或m(m≤k)个成员合作都不知道该秘密信息。利用秘密共享技术可以控制任何需要多个人共同控制的秘密信息、命令等。

认证功能：在公开的信道上进行敏感信息的传输，采用签名技术实现对消息的真实性、完整性进行验证，通过验证公钥证书实现对通信主体的身份验证。

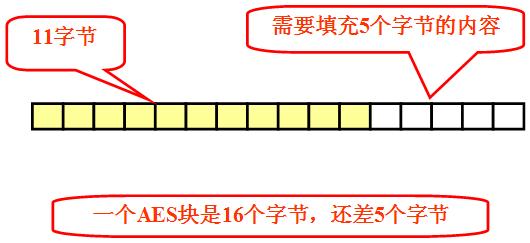
密钥管理：密钥是保密系统中更为脆弱而重要的环节，公钥密码体制是解决密钥管理工作的有力工具；利用公钥密码体制进行密钥协商和产生，保密通信双方不需要事先共享秘密信息；利用公钥密码体制进行密钥分发、保护、密钥托管、密钥恢复等。

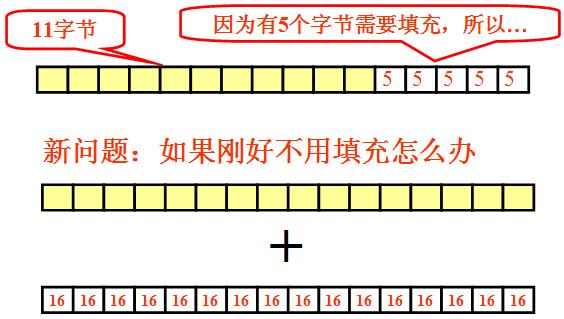
基于公钥密码体制可以实现以上通用功能以外，还可以设计实现以下的系统：安全电子商务系统、电子现金系统、电子选举系统、电子招投标系统、电子彩票系统等。

一. AES对称加密:

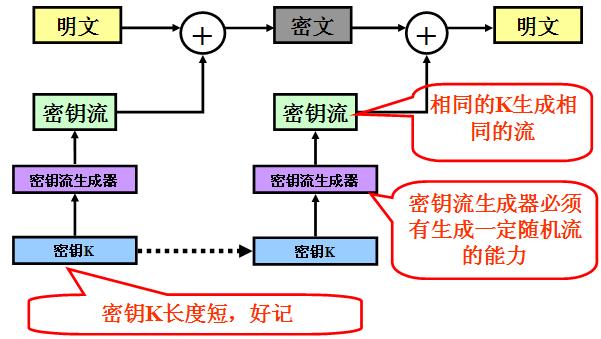
  
                                                      AES加密  
  
  
  
                         分组

二. 分组密码的填充

  
                                                   分组密码的填充

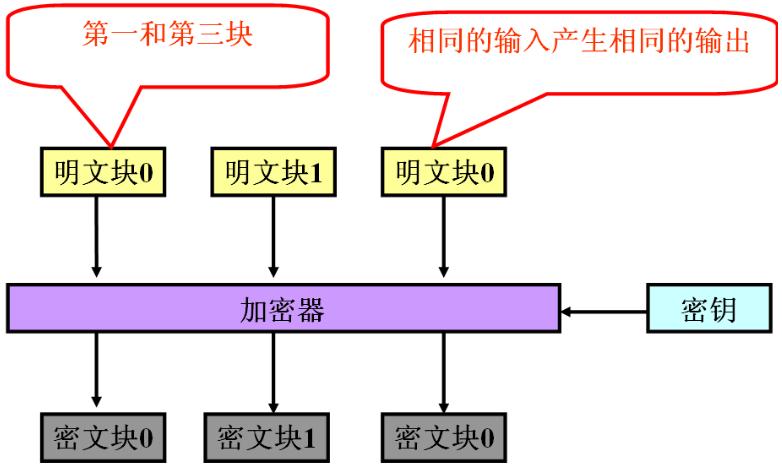
e.g.:  
  
                                                         PKCS#5填充方式

三. 流密码:



四. 分组密码加密中的四种模式:

3.1 ECB模式



优点:

1.简单；

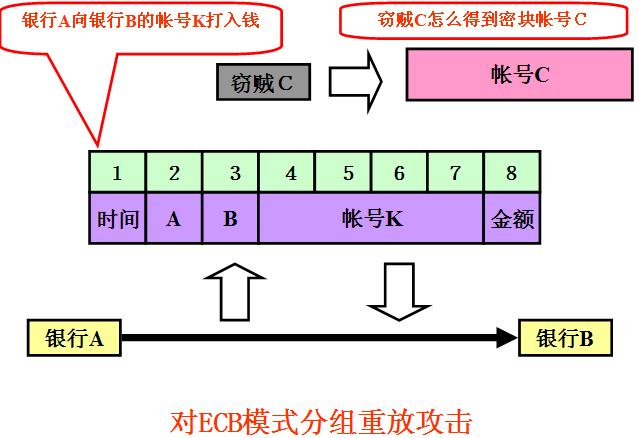
2.有利于并行计算；

3.误差不会被传送；

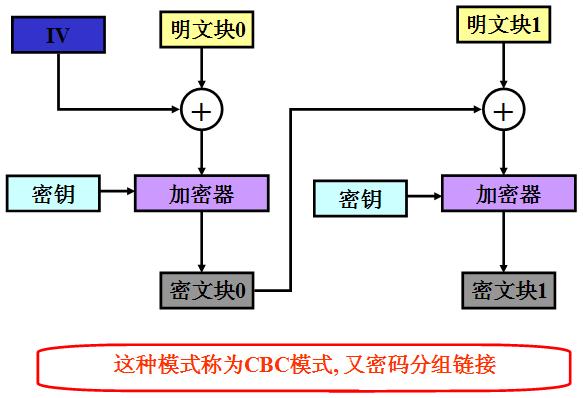
缺点:

1.不能隐藏明文的模式；

2.可能对明文进行主动攻击；



3.2 CBC模式：



优点：

1.不容易主动攻击,安全性好于ECB,适合传输长度长的报文,是SSL、IPSec的标准。

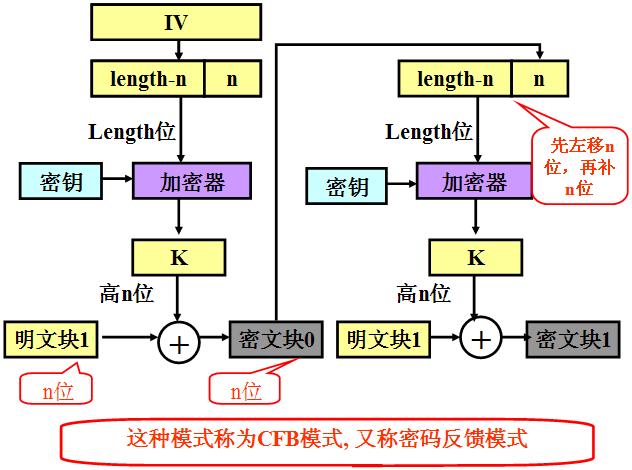
缺点：

1.不利于并行计算；

2.误差传递；

3.需要初始化向量IV

3.3 CFB模式：

  
 优点：

1.隐藏了明文模式;

2.分组密码转化为流模式;

3.可以及时加密传送小于分组的数据;

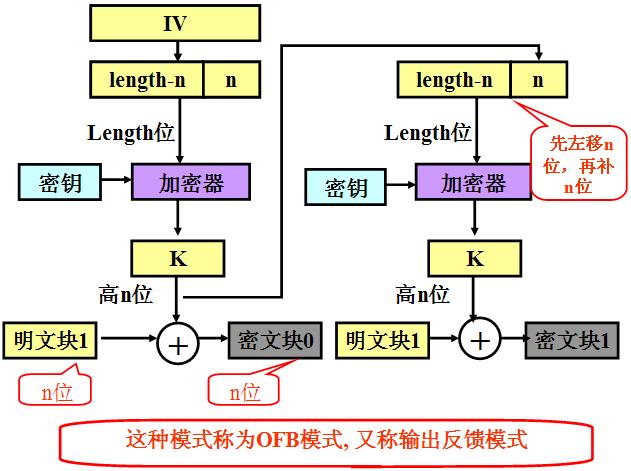
缺点:

1.不利于并行计算;

2.误差传送：一个明文单元损坏影响多个单元;

3.唯一的IV;

3.4 OFB模式：

  
 优点:

1.隐藏了明文模式;

2.分组密码转化为流模式;

3.可以及时加密传送小于分组的数据;

缺点:

1.不利于并行计算;

2.对明文的主动攻击是可能的;

3.误差传送：一个明文单元损坏影响多个单元;

## 端口连通性测试

1. telnet

命令格式：

telnet [参数] [主机] [port]

命令功能：

执行telnet指令开启终端机阶段作业，并登入远端主机。

命令参数：

-8 允许使用8位字符资料，包括输入与输出。

-a 尝试自动登入远端系统。

-b<主机别名> 使用别名指定远端主机名称。

-c 不读取用户专属目录里的.telnetrc文件。

-d 启动排错模式。

-e<脱离字符> 设置脱离字符。

-E 滤除脱离字符。

-f 此参数的效果和指定"-F"参数相同。

-F 使用Kerberos V5认证时，加上此参数可把本地主机的认证数据上传到远端主机。

-k<域名> 使用Kerberos认证时，加上此参数让远端主机采用指定的领域名，而非该主机的域名。

-K 不自动登入远端主机。

-l<用户名称> 指定要登入远端主机的用户名称。

-L 允许输出8位字符资料。

-n<记录文件> 指定文件记录相关信息。

-r 使用类似rlogin指令的用户界面。

-S<服务类型> 设置telnet连线所需的IP TOS信息。

-x 假设主机有支持数据加密的功能，就使用它。

-X<认证形态> 关闭指定的认证形态。

1. ssh法，具体用法格式：ssh -v -p port username@ip
2. curl法，具体用法格式：curl IP：port。IP是测试主机的IP地址，port是端口。
3. wget法，具体用法格式：wget IP：port。IP是测试主机的IP地址，port是端口。

## 编码

http://www.w3school.com.cn/tags/html\_ref\_urlencode.html

## Cookie

Cookie 是浏览器本地存储的一种，可以由本地产生，也可以由客户端送回本地。正常来说，浏览器首次访问一个域名，request 中是没有 cookie 的。如果我们在服务端添加了 cookie ，那么 response 的头中有一个 set-cookie 的属性，浏览器收到这个属性后，会自动在本地存储中添加这个 cookie。当然也可能是更新操作。 Js 脚本操作本地存储那不是应该的嘛？

Cookie的几个属性如下：

Name、Value、 domain、path、expires/Max-Age、size、http、Secure、SameSite

Domain属性指定浏览器发出 HTTP 请求时，哪些域名要附带这个 Cookie。如果没有指定该属性，浏览器会默认将其设为当前 URL 的一级域名(localhost就不一定了)，比如www.example.com会设为example.com，而且以后如果访问example.com的任何子域名，HTTP 请求也会带上这个 Cookie。如果服务器在Set-Cookie字段指定的域名，不属于当前 url 的子域名，浏览器会拒绝这个 Cookie。

Path属性指定浏览器发出 HTTP 请求时，哪些路径要附带这个 Cookie。前提是域名要跟这个 cookie相同。比如域名为 [www.badiu.com](http://www.badiu.com) ，path为 /person 。那么当 url 为 <http://www.baidu.com/person> 的请求会自动带上这个 cookie。

Expires属性指定一个具体的到期时间，是 UTC 格式。

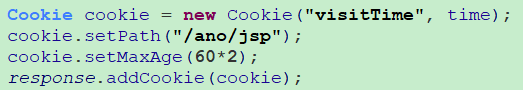
Max-Age属性指定从现在开始 Cookie 存在的秒数，比如60 \* 60 \* 24 \* 365（即一年）。

如果同时指定了Expires和Max-Age，那么Max-Age的值将优先生效。

在服务端，如果设置 Max-Age 大于 0 有效。小于0 只存在内存中，不放到本地存储。等于 0 表示删除 cookie。

HttpOnly属性，表示这个 cookie只有在发送 http 请求的时候，才会带上该 cookie。任何通过脚本语言或者其它手段想或者这个 cookie 都不成功。

Secure属性是 https 的开关，设置为 true 表示只有 https 协议才能操作的 cookie，想通过 http 协议来操作这个 cookie 是没法成功的。





## Session

对Tomcat而言，Session是一块在服务器开辟的内存空间，其存储结构为ConcurrentHashMap；也支持持久化。

服务器第一次接收到请求时，开辟了一块Session空间（创建了Session对象），同时生成一个Session id，并通过响应头的Set-Cookie：“JSESSIONID=XXXXXXX”命令，向客户端发送要求设置cookie的响应；

客户端收到响应后，在本机客户端设置了一个JSESSIONID=XXXXXXX的cookie信息，该cookie的过期时间为浏览器会话结束；

接下来客户端每次向同一个网站发送请求时，请求头都会带上该cookie信息（包含Session id）；

然后，服务器通过读取请求头中的Cookie信息，获取名称为JSESSIONID的值，得到此次请求的Session id；

ps：服务器只会在客户端第一次请求响应的时候，在响应头上添加Set-Cookie：“JSESSIONID=XXXXXXX”信息。

而客户端是会在每次请求头的cookie中带上JSESSIONID信息；

只要浏览器未关闭，在访问同一个站点的时候，其请求头Cookie中的JSESSIONID都是同一个值，被服务器认为是同一个会话。关闭之后，就重新生成一个JSESSIONID，就是另外一个会话了。

并且这个Session是封装到了 request 里面，我们可以从 request 里面取。

## https

### 证书申请流程

HTTPS证书是数字证书中的一种，由 CA用户要携带有关证件到各地的证书受理点，或者直接到证书发放机构即CA中心填写申请表并进行身份审核，审核通过后交纳一定费用就可以得到装有证书的相关介质（磁盘或Key）和一个写有密码口令的密码信封。域名型的证书申请的时候，无须递交书面审查资料，仅需进行域名有效性验证，网上申请。而企业型证书需要进行严格的网站所有权的真实身份验证，证书标示企业组织机构详情，强化信任度。增强型证书除了进行严格的网站所有权的真实身份验证之外，还加入第三方验证，证书标示增强组织机构详情，强化信任度。

【HTTPS证书申请的3个主要步骤】

1、制作CSR文件

所谓CSR就是由申请人制作的Certificate Secure Request证书请求文件。制作过程中，系统会产生2个密钥，一个是公钥就是这个CSR文件，另外一个是私钥，存放在服务器上。要制作CSR文件，申请人可以参考WEB SERVER的文档，一般APACHE等，使用OPENssl命令行来生成KEY+CSR2个文件，Tomcat，JBoss，Resin等使用KEYTOOL来生成JKS和CSR文件，IIS通过向导建立一个挂起的请求和一个CSR文件。

2、CA认证

将CSR提交给CA，CA一般有2种认证方式：

1)域名认证：一般通过对管理员邮箱认证的方式，这种方式认证速度快，但是签发的证书中没有企业的名称;

2)企业文档认证：需要提供企业的营业执照。

### 2.1  客户端验证流程

1 客户端发起一个https的请求，把自身支持的一系列Cipher Suite（密钥算法套件，简称Cipher）发送给服务端

2  服务端，接收到客户端所有的Cipher后与自身支持的对比，如果不支持则连接断开，反之则会从中选出一种加密算法和HASH算法

 以证书的形式返回给客户端 证书中还包含了 公钥 颁证机构 网址 失效日期等等。

3 客户端收到服务端响应后会做以下几件事

 3.1 验证证书的合法性

颁发证书的机构是否合法与是否过期，证书中包含的网站地址是否与正在访问的地址一致等

 证书验证通过后，在浏览器的地址栏会加上一把小锁(每家浏览器验证通过后的提示不一样 不做讨论)

 3.2 生成随机密码

 如果证书验证通过，或者用户接受了不授信的证书，此时浏览器会生成一串随机数，然后用证书中的公钥加密。

 3.3 HASH握手信息

 用最开始约定好的HASH方式，把握手消息取HASH值，  然后用 随机数加密 “握手消息+握手消息HASH值(签名)”  并一起发送给服务端

 在这里之所以要取握手消息的HASH值，主要是把握手消息做一个签名，用于验证握手消息在传输过程中没有被篡改过。

4  服务端拿到客户端传来的密文，用自己的私钥来解密握手消息取出随机数密码，再用随机数密码 解密 握手消息与HASH值，并与传过来的HASH值做对比确认是否一致。

    然后用随机密码加密一段握手消息(握手消息+握手消息的HASH值 )给客户端

5  客户端用随机数解密并计算握手消息的HASH，如果与服务端发来的HASH一致，此时握手过程结束，之后所有的通信数据将由之前浏览器生成的随机密码并利用对称加密算法进行加密

     因为这串密钥只有客户端和服务端知道，所以即使中间请求被拦截也是没法解密数据的，以此保证了通信的安全

非对称加密算法：RSA，DSA/DSS     在客户端与服务端相互验证的过程中用的是对称加密

对称加密算法：AES，RC4，3DES     客户端与服务端相互验证通过后，以随机数作为密钥时，就是对称加密

HASH算法：MD5，SHA1，SHA256  在确认握手消息没有被篡改时

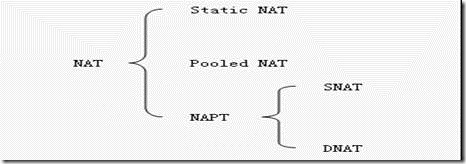
### 证书有效性校验

1. **package** com.longhine.demo;
3. **import** java.io.File;
4. **import** java.io.FileInputStream;
5. **import** java.security.PublicKey;
6. **import** java.security.SignatureException;
7. **import** java.security.cert.Certificate;
8. **import** java.security.cert.CertificateFactory;
10. **public** **class** Verify\_cert
11. {
12. **public** **static** **void** main(String[] args) **throws** Exception
13. {
15. String server\_cert\_path = "C:\\Users\\Administrator\\Desktop\\ttt\\Cert\_test\\ca\_52.cer";
16. String client\_cert\_path = "C:\\Users\\Administrator\\Desktop\\ttt\\Cert\_test\\user\_52.cer";
17. String client\_cert\_path\_fake = "C:\\Users\\Administrator\\Desktop\\ttt\\Cert\_test\\user\_csdn.cer";
19. verify\_cert(server\_cert\_path, client\_cert\_path, client\_cert\_path\_fake);
20. }
22. /\*\*
23. \* @param server\_cert\_path      //服务器证书地址
24. \* @param client\_cert\_path      //本地证书地址
25. \* @param client\_cert\_path\_fake //本地假证书地址
26. \*/
27. **public** **static** **void** verify\_cert(String server\_cert\_path, String client\_cert\_path, String client\_cert\_path\_fake) **throws** Exception
28. {
29. CertificateFactory certfactory = CertificateFactory.getInstance("X.509");
31. FileInputStream server\_fin = **new** FileInputStream(**new** File(server\_cert\_path));
32. Certificate server\_cert = certfactory.generateCertificate(server\_fin);
33. PublicKey server\_cert\_publicKey = server\_cert.getPublicKey();
35. FileInputStream client\_fin = **new** FileInputStream(**new** File(client\_cert\_path));
36. FileInputStream client\_fin\_fake = **new** FileInputStream(**new** File(client\_cert\_path\_fake));
37. Certificate client\_cert = certfactory.generateCertificate(client\_fin);
38. Certificate client\_cert\_fake = certfactory.generateCertificate(client\_fin\_fake);
40. **try**
41. {
42. client\_cert.verify(server\_cert\_publicKey);
43. } **catch** (SignatureException e)
44. {
45. System.out.println("该证书不属于 此CA签发.");
46. }
48. System.out.println("该证书 属于 此CA签发");
50. **try**
51. {
52. client\_cert\_fake.verify(server\_cert\_publicKey);
53. } **catch** (SignatureException e)
54. {
55. System.out.println("该证书不属于 此CA签发.");
56. }
57. }
58. }

## NAT

定义:网络地址转换(NAT,Network Address Translation)是一种将私有（保留）地址转化为合法IP地址的转换技术。

下面是 NAT 的分类



静态NAT

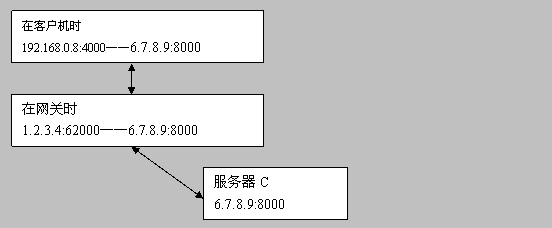
如果一个内部主机映射到确定的公网IP。

动态NAT (pooled nat)

外部IP地址可变的NAT，它为每一个内部的IP地址分配一个临时的外部IP地址，主要应用于拨号，对于频繁的远程联接也可以采用动态NAT。当远程用户联接上之后，动态地址NAT就会分配给他一个IP地址，用户断开时，这个IP地址就会被释放而留待以后使用。

动态NAT方式适合于 当机构申请到的全局IP地址较少，而内部网络主机较多的情况。内网主机IP与全局IP地址是多对一的关系。当数据包进出内网时，具有NAT功能的设备对IP数据包的处理与静态NAT的一样，只是NAT table表中的记录是动态的，若内网主机在一定时间内没有和外部网络通信，有关它的IP地址映射关系将会被删除，并且会把该全局IP地址分配给新的IP数据包使用，形成新的NAT table映射记录。

下面介绍两类不同方式实现的：

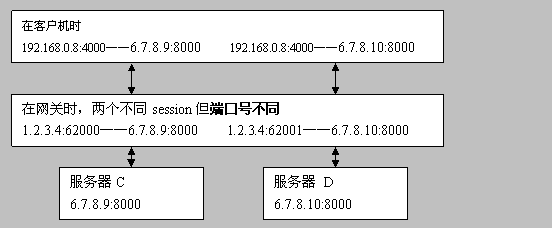


在客户机时           192.168.0.8:4000——6.7.8.9:8000

在网关时              1.2.3.4:62000——6.7.8.9:8000

服务器C               6.7.8.9:8000

         Symmetric NAT型 (对称型)



在客户机时              192.168.0.8:4000——6.7.8.9:8000 192.168.0.8:4000——6.7.8.10:8000

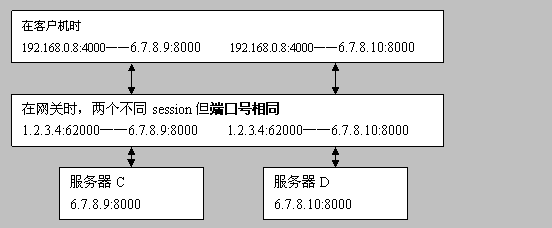
在网关时，两个不同session并且端口号不同      1.2.3.4:62000——6.7.8.9:8000 1.2.3.4:62001——6.7.8.10:8000

服务器C      6.7.8.9:8000

服务器 D     6.7.8.10:8000

这种形式会让很多p2p软件失灵。

        2.2、Cone NAT型（圆锥型）端口复用技术



在客户机时              192.168.0.8:4000——6.7.8.9:8000 192.168.0.8:4000——6.7.8.10:8000

在网关时，两个不同session但端口号相同      1.2.3.4:62000——6.7.8.9:8000 1.2.3.4:62000——6.7.8.10:8000

服务器C           6.7.8.9:8000

服务器D           6.7.8.10:8000

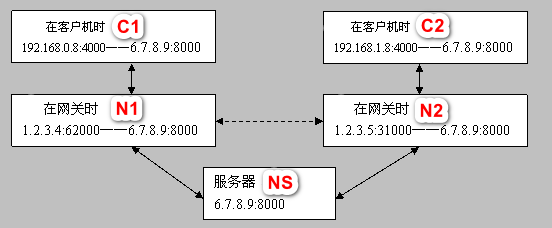
目前绝大多数属于这种。Cone NAT又分了3种类型，这种圆锥形的实际上就是端口复用技术：

a)Full Cone NAT（完全圆锥型）：从同一私网地址端口192.168.0.8:4000发至公网的所有请求都映射成同一个公网地址端口1.2.3.4:62000 ，192.168.0.8可以收到任意外部主机发到1.2.3.4:62000的数据报。就是说1.2.3.4：62000代表了整个局域网。往来随意

b)Address Restricted Cone NAT （地址限制圆锥型）：从同一私网地址端口192.168.0.8:4000发至公网的所有请求都映射成同一个公网地址端口1.2.3.4:62000，只有当内部主机192.168.0.8先给服务器C 6.7.8.9发送一个数据报后，192.168.0.8才能收到6.7.8.9发送到1.2.3.4:62000的数据报。

c)Port Restricted Cone NAT（端口限制圆锥型）：从同一私网地址端口192.168.0.8:4000发至公网的所有请求都映射成同一个公网地址端口1.2.3.4:62000，只有当内部主机192.168.0.8先向外部主机地址端口6.7.8.9：8000发送一个数据报后，192.168.0.8才能收到6.7.8.9：8000发送到1.2.3.4:62000的数据报，端口 8000 是不能变的，要不然收不到。

穿越NAT的实现



C1在客户机时                192.168.0.8:4000——6.7.8.9:8000

N1在网关时                   1.2.3.4:62000——6.7.8.9:8000

服务器NS                       6.7.8.9:8000

C2在客户机时                192.168.1.8:4000——6.7.8.9:8000

N2在网关时                   1.2.3.5:31000——6.7.8.9:8000

两内网用户要实现通过各自网关的直接呼叫，需要以下过程：

1、 客户机C1、C2顺利通过格子网关访问服务器NS ，均没有问题（类似于登录）

2、 服务器NS保存了 C1、C2各自在其网关的信息（1.2.3.4:62000、1.2.3.5:31000）没有问题。并可将该信息告知C1、B2。

3、 此时C1发送给C2网关的1.2.3.5:31000是否会被C2收到？答案是基本上不行（除非N2设置为完全圆锥型，但这种设置非常少），因为N2上检测到其存活的会话中没有一个的目的IP或端口于1.2.3.4:62000有关而将数据包全部丢弃！

4、 此时要实现C1、C2通过N1、N2来互访，需要服务器NS告诉它们各自在自己的网关上建立“UDP隧道”，即命令C1发送一个 192.168.0.8:4000——1.2.3.5:31000的 UDP 数据报，C2发送一个192.168.1.8:4000——1.2.3.4:62000 UDP数据报。这样N1、N2上均存在了IP端口相同的两个不同会话（一个与NS通信，一个 C1 到 N2 或者 C2 到 N1， N1 和 N2 上都是两个会话。这里必须用端口复用，否则 NS 不知道 C1 到 N2 或者 C2 到 N1 的端口）。

5、 此时C1发给N2，或者C2发给N1的数据报将不会被丢弃且正确的被对方收到.

综合P2P可实现的条件需要：

1、 中间服务器保存信息、并能发出建立UDP隧道的命令

2、 网关均要求为Cone NAT类型这种端口复用。Symmetric NAT不适合。

3、 完全圆锥型网关可以无需建立udp隧道，但这种情况非常少，要求双方均为这种类型网关的更少。

4、 假如N1网关为Symmetric NAT， N2为Address Restricted Cone NAT 或Full Cone NAT型网关，各自建立隧道后。N2 被动的情况下没法收消息。

5、 假如双方均为Symmetric NAT的情形，新开了端口，对方可以在不知道的情况下尝试猜解，也可以达到目的，但这种情形成功率很低，且带来额外的系统开支，不是个好的解决办法。

6、 不同网关型设置的差异在于，对内会采用替换IP的方式、使用不同端口不同会话的方式，使用相同端口不同会话的方式；对外会采用什么都不限制、限制IP地址、限制IP地址及端口。

7、 这里还没有考虑同一内网不同用户同时访问同一服务器的情形，如果此时网关采用AddressRestricted Cone NAT 或Full Cone NAT型，有可能导致不同用户客户端可收到别人的数据包，这显然是不合适的。

 UDP和TCP打洞

为什么网上讲到的P2P打洞基本上都是基于UDP协议的打洞？难道TCP不可能打洞？还是TCP打洞难于实现？

假设现在有内网客户端A和内网客户端B，有公网服务端S。如果A和B想要进行UDP通信，则必须穿透双方的NAT路由。假设为NAT-A和NAT-B。

    A发送数据包到公网S,B发送数据包到公网S,则S分别得到了A和B的公网IP，S也和A B 分别建立了会话，由S发到NAT-A的数据包会被NAT-A直接转发给A，由S发到NAT-B的数据包会被NAT-B直接转发给B，除了S发出的数据包之外的则会被丢弃。

所以：现在A B 都能分别和S进行全双工通讯了，但是A B之间还不能直接通讯。 解决办法是：A向B的公网IP发送一个数据包，则NAT-A能接收来自NAT-B的数据包并转发给A了（即B现在能访问A了）；再由S命令B向A的公网IP发送一个数据包，则NAT-B能接收来自NAT-A的数据包并转发给B了（即A现在能访问B了）。

以上就是“打洞”的原理。

为了保证A的路由器有与B的session，A要定时与B做心跳包，同样，B也要定时与A做心跳，这样，双方的通信通道都是通的，就可以进行任意的通信了。

    但是TCP和UDP在打洞上却有点不同。这是因为伯克利socket（标准socket规范）的API造成的。    UDP的socket允许多个socket绑定到同一个本地端口，而TCP的socket则不允许。    这是这样一个意思：A B要连接到S，肯定首先A B双方都会在本地创建一个socket，去连接S上的socket。创建一个socket必然会绑定一个本地端口（就算应用程序里面没写端口，实际上也是绑定了的，至少java确实如此），假设为8888，这样A和B才分别建立了到S的通信信道。接下来就需要打洞了，打洞则需要A和B分别发送数据包到对方的公网IP。但是问题就在这里：因为NAT设备是根据端口号来确定session，如果是UDP的socket，A B可以分别再创建socket，然后将socket绑定到8888，这样打洞就成功了。但是如果是TCP的socket，则不能再创建socket并绑定到8888了，这样打洞就无法成功。

UDP打洞的过程大致如此：

1、双方都通过UDP与服务器通讯后，网关默认就是做了一个外网IP和端口号 与你内网IP与端口号的映射，这个无需设置的，服务器也不需要知道客户的真正内网IP

2、用户A先通过服务器知道用户B的外网地址与端口

3、用户A向用户B的外网地址与端口发送消息，

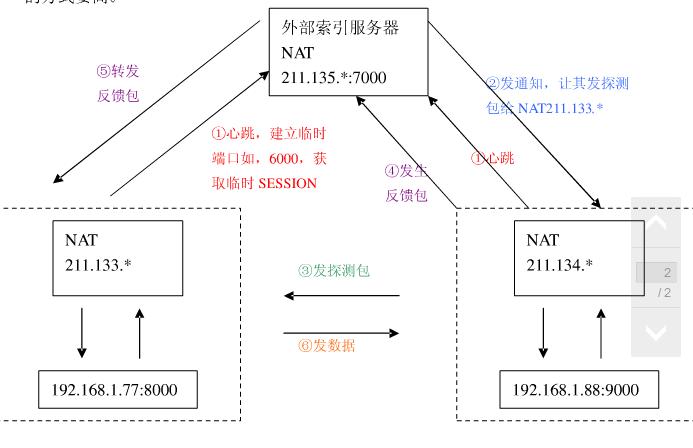
4、在这一次发送中，用户B的网关会拒收这条消息，因为它的映射中并没有这条规则。

5、但是用户A的网关就会增加了一条允许规则，允许接收从B发送过来的消息

6、服务器要求用户B发送一个消息到用户A的外网IP与端口号

7、用户B发送一条消息，这时用户A就可以接收到B的消息，而且网关B也增加了允许规则

8、之后，由于网关A与网关B都增加了允许规则，所以A与B都可以向对方的外网IP和端口号发送消息。



TCP打洞技术：  
1：启动服务器，监听端口8877  
2：第一次启动客户端（称为client1），连上服务器，服务器将返回字符串first，标识这个是client1，同时，服务器将记录下这个客户端的（经过转换之后的）IP和端口。  
3：第二次启动客户端（称为client2），连上服务器，服务器将向其返回自身的发送端口（称为port2)，以及client1的（经过转换之后的）IP和端口。  
4：然后服务器再发client1返回client2（经过转换之后的）IP和端口，然后断开与这两个客户端的连接（此时，服务器的工作已经全部完成了）  
5：client2尝试连接client1，这次肯定会失败，但它会在路由器上留下记录，以帮忙client1成功穿透，连接上自己，然后设置port2端口为可重用端口，并监听端口port2。  
6：client1尝试去连接client2，前几次可能会失败，因为穿透还没成功，如果连接10次都失败，就证明穿透失败了（可能是硬件不支持）,如果成功，则每秒向client2发送一次hello, world  
7：如果client1不断出现send message: Hello, world，client2不断出现recv message: Hello, world，则证明实验成功了，否则就是失败了。

另外的问题：  
1、Windows XP SP2操作系统之前的主机，这些主机不能正确处理TCP同时开启，或者TCP套接字不支持SO\_REUSEADDR的参数。需要让AB有序的发起连接才可能完成。  
  
上述tcp连接过程，仅对NAT1、2、3有效，对NAT4（对称型）无效。  
由于对称型nat通常采用规律的外部端口分配方法，对于nat4的打洞，可以采用端口预测的方式进行尝试。

一些现在常用的技术：

ALG（应用层网关）：它可以是一个设备或插件，用于支持SIP协议，主要类似与在网关上专门开辟一个通道，用于建立内网与外网的连接，也就是说，这是一种定制的网关。更多只适用于使用他们的应用群体内部之间。

UpnP：它是让网关设备在进行工作时寻找一个全球共享的可路由IP来作为通道，这样避免端口造成的影响。要求设备支持且开启upnp功能，但大部分时候，这些功能处于安全考虑，是被关闭的。即时开启，实际应用效果还没经过测试。

STUN（Simple Traversalof UDP Through Network）：这种方式即是类似于我们上面举例中服务器C的处理方式。也是目前普遍采用的方式。但具体实现要比我们描述的复杂许多，光是做网关Nat类型判断就由许多工作，RFC3489中详细描述了。

TURN(Traveral Using Relay NAT)：该方式是将所有的数据交换都经由服务器来完成，这样NAT将没有障碍，但服务器的负载、丢包、延迟性就是很大的问题。目前很多游戏均采用该方式避开NAT的问题。这种方式不叫p2p。

ICE(Interactive Connectivity Establishment)：是对上述各种技术的综合，但明显带来了复杂性。

## Wireshark

## Nginx

Nginx 由主进程和多个工作进程，主进程负责处理配置文件并管理工作进程，工作进程负责处理请求。工作进程采用基于事件的模型和依赖操作系统机制实现对请求的高效分发，工作进程的数量可以配置或者根据内核数量自动调整。

配置文件一般位于/usr/local/nginx/conf, /etc/nginx, or/usr/local/etc/nginx.

### 启动，关闭，重载配置，查看日志

格式：nginx -s signal

Where signal may be one of the following:

stop — fast shutdown

quit — graceful shutdown

reload — reloading the configuration file

reopen — reopening the log files

主线程接到reload信号后，重建新的工作进程，向原工作进程发送退出信号；否则回退，并且发送停止信号，工作进程在处理完原有工作后退出。

在logs/nginx.pid文件中是主进程的 PID，可以通过

kill -s QUIT 1628

### Cas 单点登陆

Ticket Granting ticket (TGT) ：可以认为是CAS Server根据用户名密码生成的一张票，存在Server端

Ticket-granting cookie (TGC) ：其实就是一个Cookie，存放用户身份信息，由Server发给Client端

Service ticket (ST) ：由TGT生成的一次性票据，用于验证，只能用一次。相当于Server发给Client一张票，然后Client拿着这个票再来找Server验证，看看是不是Server签发的。

1、基于Cookie的单点登录

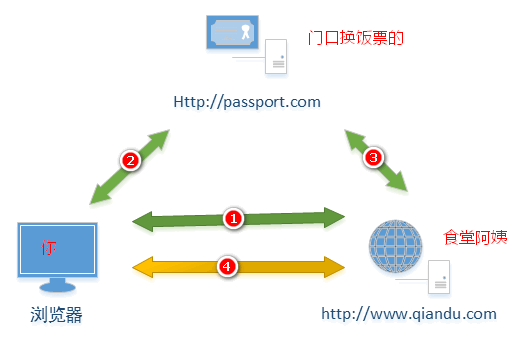
基于Cookie的单点登录核心原理：将用户名密码加密之后存于Cookie中，之后访问网站时在过滤器（filter）中校验用户权限，如果没有权限则从Cookie中取出用户名密码进行登录，让用户从某种意义上觉得只登录了一次。

      该方式缺点就是多次传送用户名密码，增加被盗风险，以及不能跨域。同时[www.qiandu.com与mail.qiandu.com](http://www.qiandu.com与mail.qiandu.com)要分别处理登陆逻辑，代码是重复的。

2、统一认证中心方案原理

      在生活中我们也有类似的相关生活经验，例如你去食堂吃饭，食堂打饭的阿姨（www.qiandu.com）告诉你，不收现金。并且告诉你，你去门口找换票的（passport.com）换小票。于是你换完票之后，再去找食堂阿姨，食堂阿姨拿着你的票，问门口换票的，这个票是真的吗？换票的说，是真的，于是给你打饭了。

      基于上述生活中的场景，我们将基于Cookie的单点登录改良以后的方案如下：



     为了避免上述缺陷，我们将认证统一化，形成一个独立的服务。当我们需要登录操作时，则重定向到统一认证中心http://passport.com。于是乎整个流程大致如下：

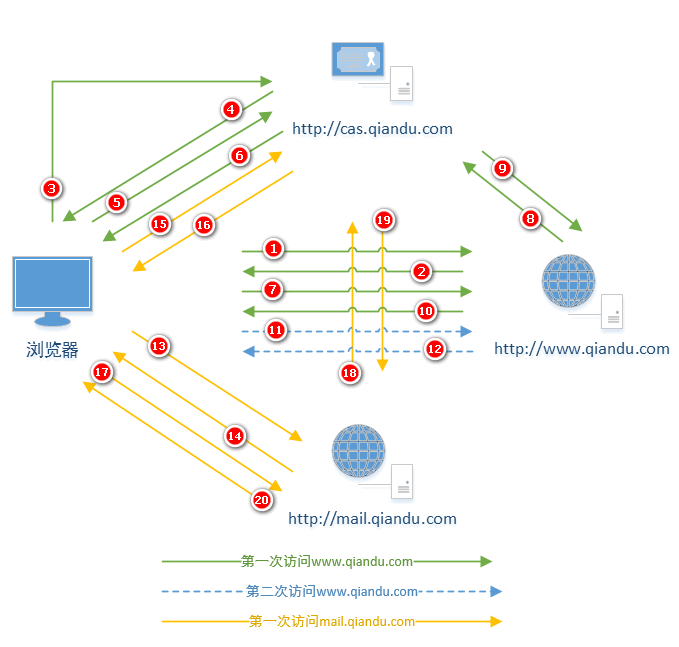
      第一步：用户访问www.qiandu.com。过滤器判断用户是否登录，没有登录，则重定向（302）到网站<http://passport.com> 的界面。

      第二步：在 passport.com登陆界面，输入用户名密码。passport.com将用户登录的信息记录到[服务器](https://www.baidu.com/s?wd=%E6%9C%8D%E5%8A%A1%E5%99%A8&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd)的session中。

      第三步：passport.com给浏览器发送一个特殊的凭证 ST 和 STC，浏览器将凭证交给www.qiandu.com，www.qiandu.com则拿着浏览器交给他的凭证去passport.com验证凭证是否有效，从而判断用户是否登录成功。

      第四步：登录成功，网站创建Seesion，浏览器与网站之间进行正常的访问。

4、CAS的详细登录流程



上图是3个登录场景，分别为：第一次访问www.qiandu.com、第二次访问、以及登录状态下第一次访问mail.qiandu.com。

下面就详细说明上图中每个数字标号做了什么，以及相关的请求内容，响应内容。

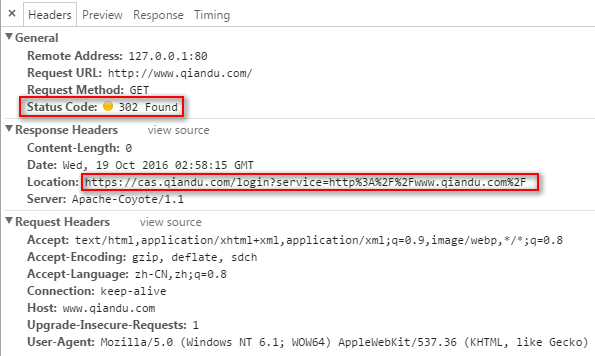
4.1、第一次访问www.qiandu.com

标号1：用户访问http://www.qiandu.com，经过他的第一个过滤器（cas提供，在web.xml中配置）AuthenticationFilter。

      过滤器全称：org.jasig.cas.client.authentication.AuthenticationFilter

      主要作用：判断是否登录，如果没有登录则重定向到认证中心。

标号2：www.qiandu.com发现用户没有登录，则返回浏览器重定向地址。



     应用将请求重定向到认证中心了，并且在重定向地址中加上了我们想要访问的应用地址。

标号3：浏览器接收到重定向之后发起重定向，请求cas.qiandu.com。

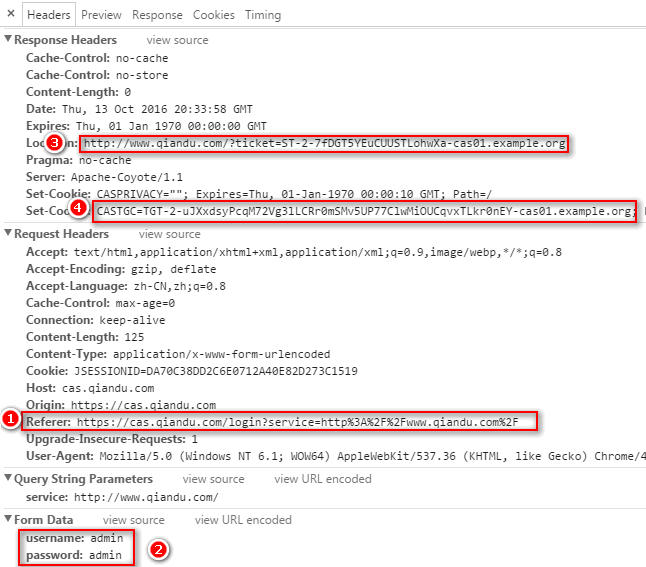
标号4：认证中心cas.qiandu.com接收到登录请求，返回登陆页面。



      上图就是标号3的请求，以及标号4的响应。请求的URL是标号2返回的URL。之后认证中心就展示登录的页面，等待用户输入用户名密码。

标号5：用户在cas.qiandu.com的login页面输入用户名密码，提交。

标号6：认证中心接收到用户名密码，则验证是否有效，验证逻辑可以使用cas-server提供现成的，也可以自己写代码实现。



     csa-server认证通过之后，会返回给浏览器302，重定向的地址就是Referer中的service参数对应的值。后边一个ticket令牌参数，这个ticket就是ST（数字3处）。同时会在Set-Cookie中设置一个CASTGC (TGC)，该cookie是网站cas.qiandu.com的cookie，只有访问这个网站才会携带这个cookie过去。

认证中心校验通过后，重定向到真正的地址，并且携带了两个标识，一个是 ST，一个是 TGC

ST 是要给 应用的，直接放到了重定向参数里面

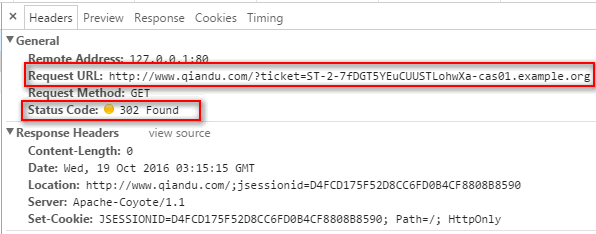
TGC 是认证中心域的 cookie，再次访问认证中心会用，根据这个TGC，查找与之对应的TGT。从而判断用户是否登录过了，是否需要展示登录页面。TGT与TGC的关系就像SESSION与Cookie中SESSIONID的关系。

      TGT：Ticket Granted Ticket（俗称大令牌，或者说票根，他可以签发ST）

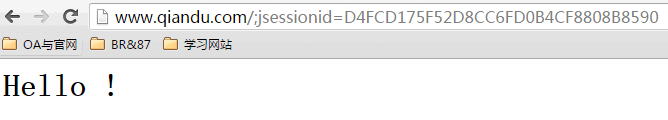
      TGC：Ticket Granted Cookie（cookie中的value），存在Cookie中，根据他可以找到TGT。

      ST：Service Ticket （小令牌），是TGT生成的，默认是用一次就生效了。也就是上面数字3处的ticket值。

标号7：浏览器从cas.qiandu.com哪里拿到ticket之后，就根据指示重定向到www.qiandu.com，请求的url就是上面返回的url。



www.qiandu.com在过滤器中会取到ticket的值，（内部逻辑是请求认证中心对 ST 校验，这些校验是通过 http 工具获取结果的，没截图）然后重定向到[www.qiandu.com](http://www.qiandu.com) 上的某一个页面或者出错页面。



      至此，第一次访问的整个流程结束。

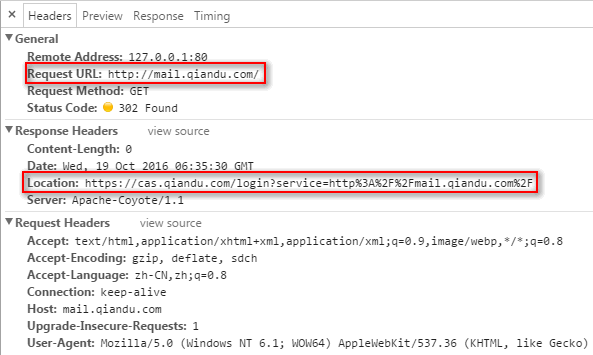
4.2、第二次访问应用，www.qiandu.com

第一次访问时，在应用中已经有 Seesion 了，不用再通过认证中心，直接通过。

4.3、访问中心管理的另外一个应用mail.qiandu.com

标号13：用户在www.qiandu.com正常上网，突然想访问mail.qiandu.com，于是发起访问mail.qiandu.com的请求。

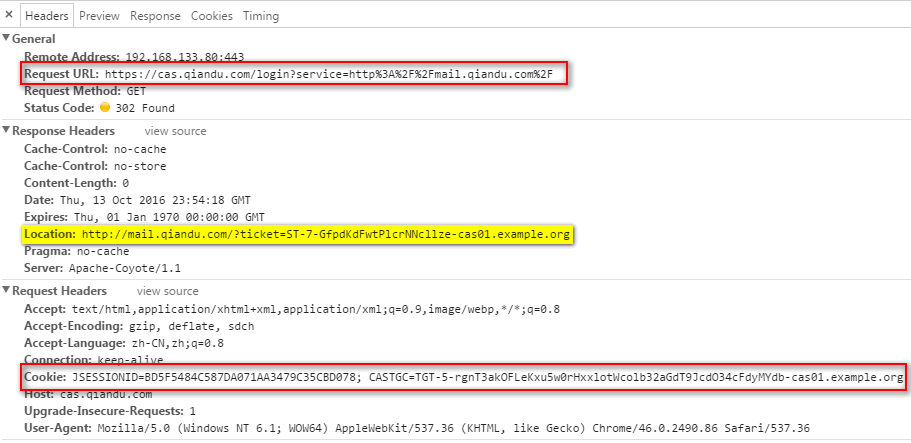
标号14：mail.qiandu.com接收到请求，发现第一次访问，于是给他一个重定向的地址，让他去找认证中心登录。



      上图可以看到，用户请求mail.qiandu.com，然后返回给他一个网址，状态302重定向，service参数就是回来的地址。

标号15：浏览器根据14返回的地址，发起重定向，因为之前访问过一次了，因此这次会携带上次返回的Cookie：TGC到认证中心。

标号16：认证中心收到请求，发现TGC对应了一个TGT，于是用TGT签发一个ST，并且返回给浏览器，让他重定向到mail.qiandu.com



      可以发现请求的时候是携带Cookie：CASTGC的，响应的就是一个地址加上TGT签发的ST也就是ticket。

标号17：浏览器根据16返回的网址发起重定向。

标号18：mail.qiandu.com获取ticket去认证中心验证是否有效。

标号19：认证成功，返回在mail.qiandu.com的session中设置登录状态，下次就直接登录。

标号20：认证成功之后就反正用想要访问的资源了。



# Java工具

## java

### java 命令列表

1. C:\Users\Administrator>java>javaInfo.txt
2. 执行 class 文件或者 jar 文件
3. 用法: java [-options] **类名字** [args...]  或者 java [-options] -jar jarfile [args...]
4. 其中选项包括:
5. -server       以 "server" 模式运行，  默认 VM 是 server 模式
7. -cp <目录和 zip/jar 文件的类搜索路径>
8. -classpath <目录和 zip/jar 文件的类搜索路径>
9. 用 ; 分隔的目录, JAR 档案
10. 和 ZIP 档案列表, 用于搜索类文件。
11. -D<名称>=<值>  设置系统属性
12. -verbose:[**class**|gc|jni]  启用详细输出
13. -version      输出产品版本并退出
14. -version:<值>
15. 警告: 此功能已过时, 将在
16. 未来发行版中删除。
17. 需要指定的版本才能运行
18. -showversion  输出产品版本并继续
19. -jre-restrict-search | -no-jre-restrict-search
20. 警告: 此功能已过时, 将在
21. 未来发行版中删除。
22. 在版本搜索中包括/排除用户专用 JRE
23. -? -help      输出此帮助消息
24. -X            输出非标准选项的帮助
25. -ea[:<packagename>...|:<classname>]
26. -enableassertions[:<packagename>...|:<classname>]
27. 按指定的粒度启用断言
28. -da[:<packagename>...|:<classname>]
29. -disableassertions[:<packagename>...|:<classname>]
30. 禁用具有指定粒度的断言
31. -esa | -enablesystemassertions
32. 启用系统断言
33. -dsa | -disablesystemassertions
34. 禁用系统断言
35. -agentlib:<libname>[=<选项>]
36. 加载本机代理库 <libname>, 例如 -agentlib:hprof
37. 另请参阅 -agentlib:jdwp=help 和 -agentlib:hprof=help
38. -agentpath:<pathname>[=<选项>]
39. 按完整路径名加载本机代理库
40. -javaagent:<jarpath>[=<选项>]
41. 加载 Java 编程语言代理, 请参阅 java.lang.instrument
42. -splash:<imagepath>
43. 使用指定的图像显示启动屏幕
44. 有关详细信息, 请参阅 http://www.oracle.com/technetwork/java/javase/documentation/index.html。

### –javaagent

首先被debug程序的虚拟机在启动时要开启debug模式，启动debug监听程序。jdwp是Java Debug Wire Protocol的缩写。

java -Xdebug -Xrunjdwp:server=y,transport=dt\_socket,address=8000,suspend=n zhc\_application

这是jdk1.7版本之前的方法，1.7之后可以这样用：

java -agentlib:jdwp=transport=dt\_socket,address=8000,server=y,suspend=n zhc\_application

zhc\_application是main程序，server=y表示是监听其他debugclient端的请求。address=8000表示端口是8000

suspend表示是否在调试客户端建立连接之后启动 VM。如果为y，那么当前的VM就是suspend直到有debug client连接进来才开始执行程序。如果你的程序不是服务器监听模式并且很快就执行完毕的，那么可以选择在y来阻塞它的启动。