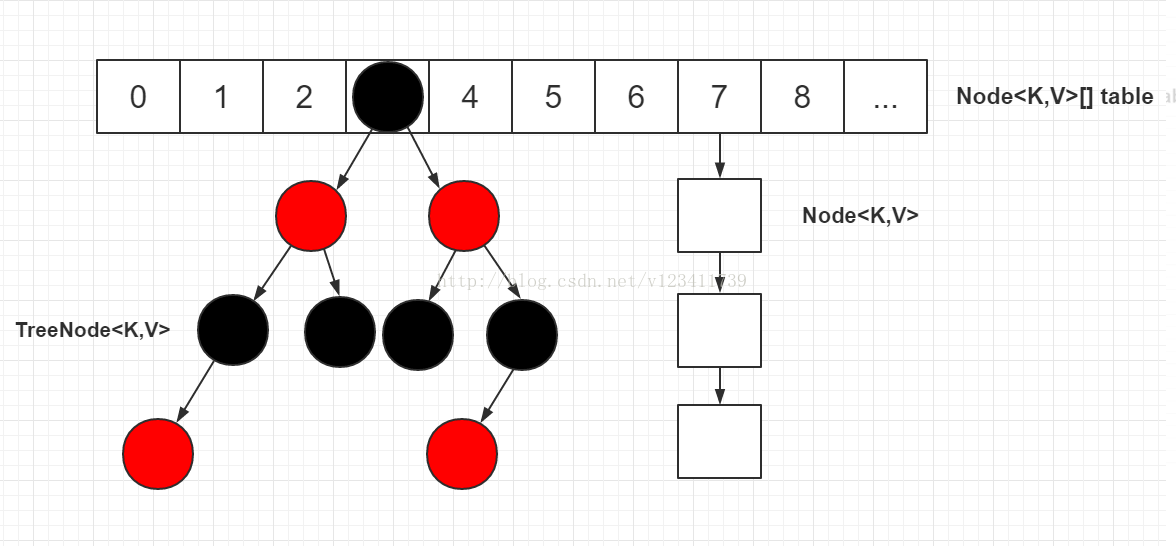
# Java学习笔记

## Java基础

#### 集合

##### HashMap

<https://blog.csdn.net/v123411739/article/details/78996181>



#### 多线程

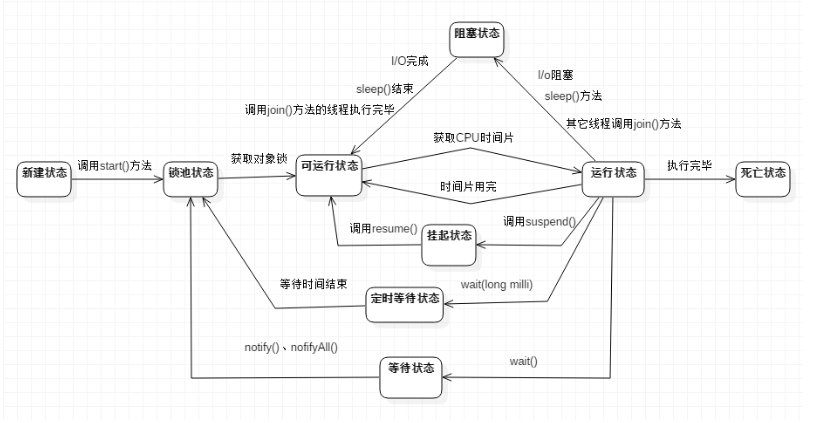
##### 线程状态

###### Jdk中的线程状态

* [NEW](eclipse-javadoc:%E2%98%82=study-java/D:%5C/professional%5C/jdk1.8.0_131%5C/jre%5C/lib%5C/rt.jar%3Cjava.lang(Thread$State.class%E2%98%83State%E2%98%82%E2%98%82NEW)  
  A thread that has not yet started is in this state.
* [RUNNABLE](eclipse-javadoc:%E2%98%82=study-java/D:%5C/professional%5C/jdk1.8.0_131%5C/jre%5C/lib%5C/rt.jar%3Cjava.lang(Thread$State.class%E2%98%83State%E2%98%82%E2%98%82RUNNABLE)  
  A thread executing in the Java virtual machine is in this state.
* [BLOCKED](eclipse-javadoc:%E2%98%82=study-java/D:%5C/professional%5C/jdk1.8.0_131%5C/jre%5C/lib%5C/rt.jar%3Cjava.lang(Thread$State.class%E2%98%83State%E2%98%82%E2%98%82BLOCKED)  
  A thread that is blocked waiting for a monitor lock is in this state.
* [WAITING](eclipse-javadoc:%E2%98%82=study-java/D:%5C/professional%5C/jdk1.8.0_131%5C/jre%5C/lib%5C/rt.jar%3Cjava.lang(Thread$State.class%E2%98%83State%E2%98%82%E2%98%82WAITING)  
  A thread that is waiting indefinitely for another thread to perform a particular action is in this state.
* [TIMED\_WAITING](eclipse-javadoc:%E2%98%82=study-java/D:%5C/professional%5C/jdk1.8.0_131%5C/jre%5C/lib%5C/rt.jar%3Cjava.lang(Thread$State.class%E2%98%83State%E2%98%82%E2%98%82TIMED_WAITING)  
  A thread that is waiting for another thread to perform an action for up to a specified waiting time is in this state.
* [TERMINATED](eclipse-javadoc:%E2%98%82=study-java/D:%5C/professional%5C/jdk1.8.0_131%5C/jre%5C/lib%5C/rt.jar%3Cjava.lang(Thread$State.class%E2%98%83State%E2%98%82%E2%98%82TERMINATED)  
  A thread that has exited is in this state.

###### 参考状态转换

<https://www.cnblogs.com/GooPolaris/p/8079490.html>



## 多线程

#### 线程池

##### ThreadPoolExecutor

<https://segmentfault.com/a/1190000015368896>

<https://www.jianshu.com/p/ae67972d1156>

各个参数的作用说明

* int corePoolSize = 2; // 核心线程池大小
* int maximumPoolSize = 5; // 最大线程池大小
* long keepAliveTime = 10; // 线程池中超过corePoolSize数目的空闲线程最大存活时间；可以allowCoreThreadTimeOut(true) 设置核心线程[corePoolSize]有效时间
* TimeUnit unit = TimeUnit.SECONDS; // keepAliveTime时间单位
* BlockingQueue<Runnable> workQueue; // 阻塞任务队列，常用的有三种队列，SynchronousQueue, LinkedBlockingDeque, ArrayBlockingQueue。

<http://www.cnblogs.com/dolphin0520/p/3932906.html>

**上面的线程池，实际使用的线程数的最大值始终是 corePoolSize ，即便设置了 maximumPoolSize 也没有生效。 要用上 maximumPoolSize ，允许在核心线程满负荷下，继续创建新线程来工作 ，就需要选用有界任务队列。可以给 LinkedBlockingQueue 设置容量，比如 new LinkedBlockingQueue(128) ，也可以换成 SynchronousQueue。**

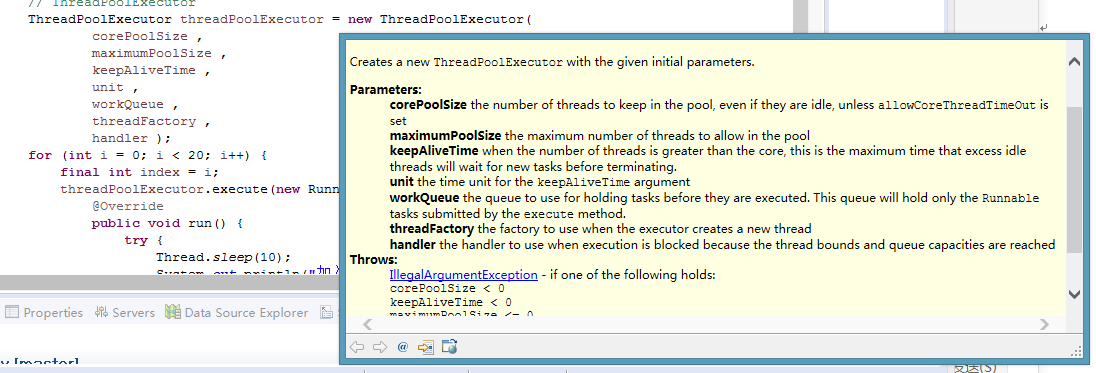
* ThreadFactory threadFactory; // 新建线程工厂
* RejectedExecutionHandler handler; // 当提交任务数超过maxmumPoolSize+workQueue之和时，任务会交给RejectedExecutionHandler来处理。

AbortPolicy（默认）：直接抛弃，并且抛出RejectedExecutionException异常

CallerRunsPolicy：用调用者的线程执行任务，如果添加到线程池失败，那么调用线程会自己去执行该任务，不会等待线程池中的线程去执行

DiscardOldestPolicy：抛弃队列中最久的任务

DiscardPolicy：抛弃当前任务，并且不会有任何异常



##### 常用4种实现

**可缓存线程池CachedThreadPool**

1. 这种线程池内部没有核心线程，线程的数量是有没限制的。
2. 在创建任务时，若有空闲的线程时则复用空闲的线程，若没有则新建线程。
3. 没有工作的线程（闲置状态）在超过了60S还不做事，就会销毁。

**FixedThreadPool 定长线程池**

1. 该线程池的最大线程数等于核心线程数，所以在默认情况下，该线程池的线程不会因为闲置状态超时而被销毁。
2. 如果当前线程数小于核心线程数，并且也有闲置线程的时候提交了任务，这时也不会去复用之前的闲置线程，会创建新的线程去执行任务。如果当前执行任务数大于了核心线程数，大于的部分就会进入队列等待。等着有闲置的线程来执行这个任务。

**SingleThreadPool**

1. 有且仅有一个工作线程执行任务
2. 所有任务按照指定顺序执行，即遵循队列的入队出队规则

**ScheduledThreadPool**

1. 不仅设置了核心线程数，最大线程数也是Integer.MAX\_VALUE。
2. 这个线程池是上述4个中为唯一个有延迟执行和周期执行任务的线程池。

## Web

#### Cookie和Session

##### 区别

1、cookie数据存放在客户的浏览器上，session数据放在服务器上。

2、cookie不是很安全，别人可以分析存放在本地的COOKIE并进行COOKIE欺骗

考虑到安全应当使用session。

3、session会在一定时间内保存在服务器上。当访问增多，会比较占用你服务器的性能

考虑到减轻服务器性能方面，应当使用COOKIE。

4、单个cookie保存的数据不能超过4K，很多浏览器都限制一个站点最多保存20个cookie。

5、所以个人建议：

将登陆信息等重要信息存放为SESSION

其他信息如果需要保留，可以放在COOKIE中

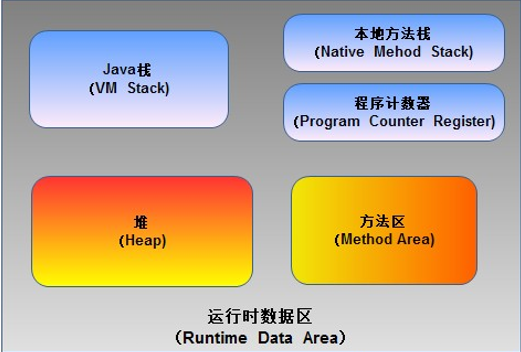
## JVM

#### 内存划分

<https://www.cnblogs.com/dolphin0520/p/3613043.html>

<https://www.cnblogs.com/wangjzh/p/5258254.html>

##### 结构图





##### 各个区域说明

###### 程序计数器（Program Counter Register）

它保存的是程序当前执行的指令的地址（也可以说保存下一条指令的所在存储单元的地址），当CPU需要执行指令时，需要从程序计数器中得到当前需要执行的指令所在存储单元的地址，然后根据得到的地址获取到指令，在得到指令之后，程序计数器便自动加1或者根据转移指针得到下一条指令的地址，如此循环，直至执行完所有的指令。

###### Java栈也称作虚拟机栈（Java Vitual Machine Stack）

当线程执行一个方法时，就会随之创建一个对应的栈帧，并将建立的栈帧压栈。当方法执行完毕之后，便会将栈帧出栈。因此可知，线程当前执行的方法所对应的栈帧必定位于Java栈的顶部。

###### 本地方法栈

本地方法栈与Java栈的作用和原理非常相似。区别只不过是Java栈是为执行Java方法服务的，而本地方法栈则是为执行本地方法（Native Method）服务的。

###### 堆

Java中的堆是用来存储对象本身的以及数组（当然，数组引用是存放在Java栈中的）。Java的垃圾回收机制会自动进行处理。因此这部分空间也是Java垃圾收集器管理的主要区域。另外，堆是被所有线程共享的，在JVM中只有一个堆。

###### 方法区

方法区在JVM中也是一个非常重要的区域，它与堆一样，是被线程共享的区域。在方法区中，存储了每个类的信息（包括类的名称、方法信息、字段信息）、静态变量、常量以及编译器编译后的代码等

#### GC回收

##### 对象存活判断方法

###### 引用计数算法

给对象添加一个引用计数器，每当有一个地方引用它时，计数器的值就+1；

当引用失效时，值就-1；任何时刻技术器为0的对象就是不能再被使用的。

###### 可达性分析算法

在主流的商用程序语言[java，C#]的主流实现中都是通过可达性分析来判定对象是否存在的。这个算法的基本思路就是通过一系列的成为”GC Roots”的对象作为起始点，从这些结点开始向下搜索，搜索所经过的路径成为“引用链”,当一个对象到GC Roots没有任何引用链时，则证明此对象是不可用的。

在Java语言中， 可作为GC Roots的对象包括下面几种：

虚拟机栈中引用的对象、

方法区中类静态属性引用的对象、

方法区中常量引用的对象、

本地方法栈中JNI[即一般说的Native]引用的对象

##### GC回收算法

<https://www.cnblogs.com/wanhua-wu/p/6582994.html>

###### 标记-清除算法

定义：先标记要回收的对象，然后统一回收；

适用：存活对象较多的垃圾回收

缺点：

* 效率低； 标记和清除的过程效率不高；
* 空间问题； 标记清除后产生大量不连续的内存碎片，给大对象分配内存时没有足够连续的内存空间，导致提前出发垃圾回收动作。



###### 复制算法

定义：将可用内存划分成相等大小两块，每次只使用其中一块，当这一块用完后将还存活的对象复制到另一块，然后将已使用过的内存一次清理。

适用：存活对象较少的垃圾回收

优点：

* 每次对整个半区进行内存回收，不用考虑内存碎片问题，只要移动堆顶指针，按顺序分配内存即可；
* 实现简单，运行高效

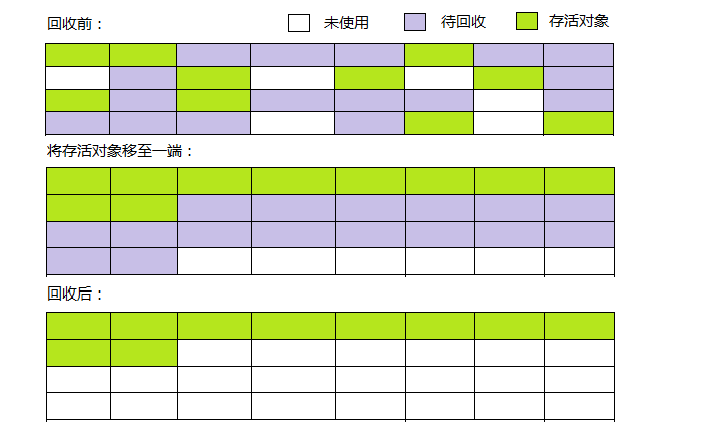
缺点：将内存缩小了一半

其他：

* 将新生代内存按照8:1:1分为Eden，From Survivor，To Survivor三个空间，每次使用Eden和From Survivor两个空间给对象分配内存，
* 当内存不足垃圾回收时，将存活对象复制到To Survivor空间，然后清理Eden和From Survivor空间；这样相当于内存指浪费了10%；
* 如果10%的To Survivor空间不够存放存活对象时需要老年代进行分配担保（将存活对象通过分配担保机制直接进入老年代）

###### 标记-整理算法

定义：先标记要回收的对象，将存活对象移至一端，最后清理端边界以外的内存



###### 分代收集算法

定义：根据对象存活周期将内存划分为新生代和老年代，然后根据每个年代的特点使用合适的回收算法；如：新生代存活对象少可以采用复制算法； 老年代存活对象多并且没有分配担保必须使用标记清理或标记整理回收算法

##### 回收时机

<https://blog.csdn.net/yhyr_ycy/article/details/52566105>

**Minor GC触发条件**

当Eden区满时，触发Minor GC。

**Full GC触发条件**

（1）调用System.gc时，系统建议执行Full GC，但是不必然执行

（2）老年代空间不足

（3）方法区空间不足

（4）通过Minor GC后进入老年代的平均大小大于老年代的可用内存

（5）由Eden区、From Space区向To Space区复制时，对象大小大于To Space可用内存

##### GC收集器

<https://www.jianshu.com/p/9051bd062980>

**serial收集器**

serial是一个单线程的垃圾收集器，它只会使用一个cpu、一个收集线程工作；它在进行gc的时候，必须暂停其他所有线程到它工作结束

**CMS收集器**

JAVA垃圾回收器回收垃圾的时间尽可能短；

应用运行在多CPU的机器上，有足够的CPU资源；

有比较多生命周期长的对象；

应用的响应时间短。

**G1收集器**

利用多CPU来缩短STW的时间

可以独立管理整个堆（使用分代算法）

整体是基于标记-整理，局部使用复制算法，不会产生碎片空间

可以预测停顿：G1把整个堆分成多个Region，然后计算每个Region里面的垃圾大小（根据回收所获得的空间大小和回收所需要的时间的经验值），在后台维护一个优先列表，每次根据允许的收集时间，优先回收价值最大的Region。

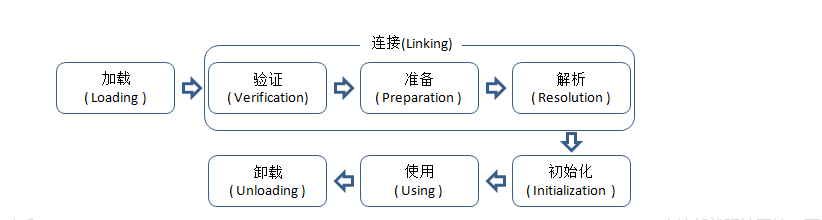
#### Class加载过程

<https://blog.csdn.net/u011109589/article/details/80320562>

<https://blog.csdn.net/uniquewonderq/article/details/79996643>

##### 总体过程

加载（Loading）、验证（Verification）、准备(Preparation)、解析(Resolution)、初始化(Initialization)、使用(Using)和卸载(Unloading)7个阶段。其中准备、验证、解析3个部分统称为连接（Linking）。如图所示。



加载、验证、准备、初始化和卸载这5个阶段的顺序是确定的，类的加载过程必须按照这种顺序按部就班地开始，而解析阶段则不一定：它在某些情况下可以在初始化阶段之后再开始，这是为了支持Java语言的运行时绑定（也称为动态绑定或晚期绑定）

##### 各个过程

###### 加载

在装载阶段，虚拟机需要完成以下3件事情

(1) 通过一个类的全限定名来获取定义此类的二进制字节流，虚拟机规范中并没有准确说明二进制字节流应该从哪里获取以及怎样获取,这里可以通过定义自己的类加载器去控制字节流的获取方式

(2) 将这个字节流所代表的静态存储结构转化为方法区的运行时数据结构

(3) 在Java堆中生成一个代表这个类的java.lang.Class对象，作为方法区这些数据的访问入口。

###### 验证

如果我们是从自己本地的class文件加载类信息肯定不会出错，但是我们上面讲到了类的加载只是加载了一系列的二进制字节码，无法保证字节码的正确性，所以需要验证

###### 准备

准备阶段是正式为类变量分配并设置类变量初始值的阶段，这些内存都将在方法区中进行分配（因为这里的变量都是类变量，实例变量在堆，类变量在方法区）如: public static int value = 123; value在准备阶段过后的初始值为0而不是123,而把value赋值的putstatic指令将在初始化阶段才会被执行

###### 解析

解析阶段是虚拟机将常量池内的符号引用替换为直接引用的过程。解析动作主要针对类或接口、字段、类方法、接口方法、方法类型、方法句柄和调用点限定符7类符号引用进行。类装入器装入类所引用的其他所有类。可以用许多方式引用类，如超类、接口、字段、方法签名、方法中使用的本地变量

###### 初始化

1. 类初始化阶段是类加载过程的最后一步，到了初始化阶段，才真正开始执行类中定义的java程序代码。在准备阶段变量已经付过一次系统要求的初始值，而在初始化阶段，则根据程序猿通过程序制定的主管计划去初始化类变量和其他资源，或者说：初始化阶段是执行类构造器<clinit>()方法的过程.
2. <clinit>()方法是由编译器自动收集类中的所有类变量的赋值动作和静态语句块static{}中的语句合并产生的，编译器收集的顺序是由语句在源文件中出现的顺序所决定的
3. <clinit>()方法与实例构造器<init>()方法不同，它不需要显示地调用父类构造器，虚拟机会保证在子类<init>()方法执行之前，父类的<clinit>()方法方法已经执行完毕
4. 就是由于父类的<clinit>（）方法先与子类执行，所以在父类的static语句先于子类执行，然后是父类的非static语句块和构造方法，接下来是子类的非static语句块和构造方法

## Spring

#### 请求过程

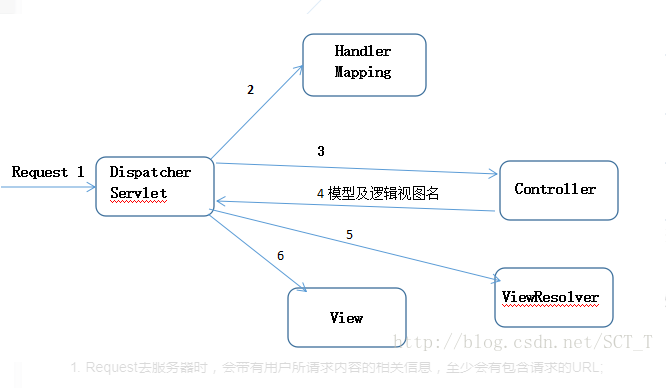
<https://blog.csdn.net/SCT_T/article/details/53998216?utm_source=blogxgwz0>

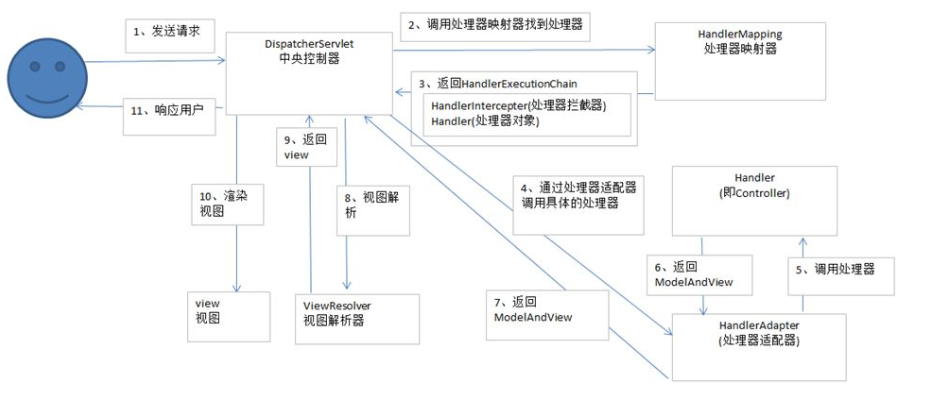
<https://www.cnblogs.com/dreamworlds/p/5396112.html>

<https://www.cnblogs.com/java1024/p/8556519.html>

<https://www.cnblogs.com/glorywzm/p/6503141.html>

##### 结构图







##### 各个过程

* Request去服务器时，会带有用户所请求内容的相关信息，至少会有包含请求的URL;
* DispatcherServlet 查询HandlerMapping来确定将Request发送给哪个Controller处理；
* 选定了合适的Controller，DispatcherServlet会将Request发送过去，等待Controller对请求的处理。
* Controller在完成逻辑处理后，会产生一些信息，需要返回给用户并且要在浏览器上显示的。这些信息称为模型（Model） ,而这些信息的加工格式化者就是HTML,信息需要发送一个视图（通常是JSP）。
* Controller所做的最后一件事就是将模型数据打包，并且标出用于渲染输出的视图名称，发还给DispatcherServlet.
* ViewResolver根据DispatcherServlet的逻辑名查找真正的视图。

视图将数据渲染输出，并且通过这个输出将响应对象传递给客户端。

⑴ 用户发送请求至前端控制器DispatcherServlet

⑵ DispatcherServlet收到请求调用HandlerMapping处理器映射器。

⑶ 处理器映射器根据请求url找到具体的处理器，生成处理器对象及处理器拦截器(如果有则生成)一并返回给DispatcherServlet。

⑷ DispatcherServlet通过HandlerAdapter处理器适配器调用处理器

⑸ 执行处理器(Controller，也叫后端控制器)。

⑹ Controller执行完成返回ModelAndView

⑺ HandlerAdapter将controller执行结果ModelAndView返回给DispatcherServlet

⑻ DispatcherServlet将ModelAndView传给ViewReslover视图解析器

⑼ ViewReslover解析后返回具体View

⑽ DispatcherServlet对View进行渲染视图（即将模型数据填充至视图中）。

⑾ DispatcherServlet响应用户。

1.用户发送请求到DispatchServlet

2.DispatchServlet根据请求路径查询具体的Handler

3.HandlerMapping返回一个HandlerExcutionChain给DispatchServlet

　HandlerExcutionChain：Handler和Interceptor集合

4.DispatchServlet调用HandlerAdapter适配器

5.HandlerAdapter调用具体的Handler处理业务

6.Handler处理结束返回一个具体的ModelAndView给适配器

ModelAndView:model-->数据模型，view-->视图名称

7.适配器将ModelAndView给DispatchServlet

8.DispatchServlet把视图名称给ViewResolver视图解析器

9.ViewResolver返回一个具体的视图给DispatchServlet

10.渲染视图

11.展示给用户

#### 事务

<https://www.cnblogs.com/yixianyixian/p/8372832.html>

<https://www.jdon.com/concurrent/acid-database.html>

##### 四大特性

事务有四个特性：ACID

**原子性（Atomicity）**

事务是一个原子操作，由一系列动作组成。事务的原子性确保动作要么全部完成，要么完全不起作用。

**一致性（Consistency）**

一旦事务完成（不管成功还是失败），系统必须确保它所建模的业务处于一致的状态，在现实中的数据不应该被破坏。如果一个操作触发辅助操作（级联，触发器），这些也必须成功，否则交易失败。如果系统是由多个节点组成，一致性规定所有的变化必须传播到所有节点。

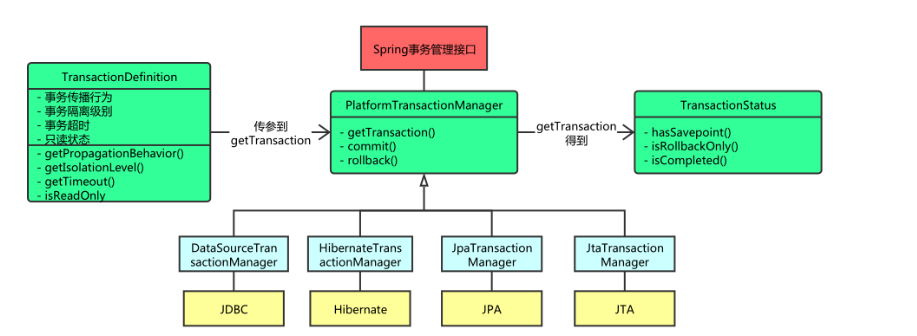
**隔离性（Isolation）**

可能有许多事务会同时处理相同的数据，因此每个事务都应该与其他事务隔离开来，防止数据损坏。

**持久性（Durability）**

一旦事务完成，无论发生什么系统错误，它的结果都不应该受到影响，这样就能从任何系统崩溃中恢复过来。通常情况下，事务的结果被写到持久化存储器中。

##### 相关接口



##### 传播行为

<https://segmentfault.com/a/1190000013341344>

Propagation：宣传; 传播，传输，蔓延

| **传播行为** | **含义** |
| --- | --- |
| PROPAGATION\_REQUIRED  propagation\_required | 表示当前方法必须运行在事务中。如果当前事务存在，方法将会在该事务中运行。否则，会启动一个新的事务 |
| PROPAGATION\_SUPPORTS  propagation\_supports | 表示当前方法不需要事务上下文，但是如果存在当前事务的话，那么该方法会在这个事务中运行 |
| PROPAGATION\_MANDATORY  propagation\_mandatory(强制的; 命令的) | 表示该方法必须在事务中运行，如果当前事务不存在，则会抛出一个异常 |
| PROPAGATION\_REQUIRED\_NEW  propagation\_required\_new | 表示当前方法必须运行在它自己的事务中。一个新的事务将被启动。如果存在当前事务，在该方法执行期间，当前事务会被挂起。如果使用JTATransactionManager的话，则需要访问TransactionManager |
| PROPAGATION\_NOT\_SUPPORTED  propagation\_not\_supported | 表示该方法不应该运行在事务中。如果存在当前事务，在该方法运行期间，当前事务将被挂起。如果使用JTATransactionManager的话，则需要访问TransactionManager |
| PROPAGATION\_NEVER  propagation\_never | 表示当前方法不应该运行在事务上下文中。如果当前正有一个事务在运行，则会抛出异常 |
| PROPAGATION\_NESTED  propagation\_nested | 表示如果当前已经存在一个事务，那么该方法将会在嵌套事务中运行。嵌套的事务可以独立于当前事务进行单独地提交或回滚。如果当前事务不存在，那么其行为与PROPAGATION\_REQUIRED一样。注意各厂商对这种传播行为的支持是有所差异的。可以参考资源管理器的文档来确认它们是否支持嵌套事务 |

##### 隔离级别

<http://www.cnblogs.com/s-b-b/p/5845096.html>

**READ COMMITTED（提交读）**

大多数数据库系统的默认隔离级别是READ COMMITTED，这种隔离级别就是一个事务的开始，只能看到已经完成的事务的结果，正在执行的，是无法被其他事务看到的。这种级别会出现读取旧数据的现象

**READ UNCIMMITTED（未提交读）/ 脏读（Dirty reads）**

脏读发生在一个事务读取了另一个事务改写但尚未提交的数据时。如果改写在稍后被回滚了，那么第一个事务获取的数据就是无效的。这就是事务还没提交，而别的事务可以看到他其中修改的数据的后果，也就是脏读。

**REPEATABLE READ（可重复读）**

REPEATABLE READ解决了脏读的问题，该级别保证了每行的记录的结果是一致的，也就是上面说的读了旧数据的问题，但是却无法解决另一个问题，幻行，顾名思义就是突然蹦出来的行数据。指的就是某个事务在读取某个范围的数据，但是另一个事务又向这个范围的数据去插入数据，导致多次读取的时候，数据的行数不一致。

**SERIALIZABLE（可串行化）**

SERIALIZABLE是最高的隔离级别，它通过强制事务串行执行（注意是串行），避免了前面的幻读情况，由于他大量加上锁，导致大量的请求超时，因此性能会比较底下，再特别需要数据一致性且并发量不需要那么大的时候才可能考虑这个隔离级别

#### Bean生命周期

<https://www.cnblogs.com/kenshinobiy/p/4652008.html>

<https://www.cnblogs.com/redcool/p/6397398.html>

实现：/study-spring/src/main/java/com/bage/lifecircle/LifeCircleBean.java

1. 实例化一个Bean，也就是我们通常说的new

2. 按照Spring上下文对实例化的Bean进行配置，也就是IOC注入，使用依赖注入，Spring按照Bean定义信息配置Bean所有属性

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

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

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

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

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

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

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

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

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

#### AOP实现原理

<https://www.cnblogs.com/lcngu/p/5339555.html>

## 基础

### ****Java基础****

* 面向对象的特征：继承、封装和多态
* final, finally, finalize 的区别
* Exception、Error、运行时异常与一般异常有何异同
* 请写出5种常见到的runtime exception
* int 和 Integer 有什么区别，Integer的值缓存范围
* 包装类，装箱和拆箱
* String、StringBuilder、StringBuffer
* 重载和重写的区别
* 抽象类和接口有什么区别
* 说说反射的用途及实现
* 说说自定义注解的场景及实现
* HTTP请求的GET与POST方式的区别
* Session与Cookie区别
* 列出自己常用的JDK包
* MVC设计思想
* equals与==的区别
* hashCode和equals方法的区别与联系
* 什么是Java序列化和反序列化，如何实现Java序列化？或者请解释Serializable 接口的作用
* Object类中常见的方法，为什么wait  notify会放在Object里边？

因为synchronized中的这把锁可以是任意对象，所以任意对象都可以调用wait()和notify()；所以wait和notify属于Object。

* Java的平台无关性如何体现出来的

最主要的是Java平台本身。Java平台扮演Java程序和所在的硬件与操作系统之间的缓冲角色。这样Java程序只需要与Java平台打交道，而不用管具体的操作系统。

Java语言保证了基本数据类型的值域和行为都是由语言自己定义的。而C/C++中，基本数据类是由它的占位宽度决定的，占位宽度由所在平台决定的。不同平台编译同一个C++程序会出现不同的行为。通过保证基本数据类型在所有平台的一致性，Java语言为平台无关性提供强有力的支持。

Java class文件。Java程序最终会被编译成二进制class文件。class文件可以在任何平台创建，也可以被任何平台的Java虚拟机装载运行。它的格式有着严格的定义，是平台无关的。

可伸缩性。Sun通过改变API的方式得到三个基础API集合，表现为Java平台不同的伸缩性：J2EE,J2SE,J2ME。

* JDK和JRE的区别
* Java 8有哪些新特性

Lambda 表达式 − Lambda允许把函数作为一个方法的参数（函数作为参数传递进方法中。

Java8 内置的四大核心函数式接口

|  |  |  |  |
| --- | --- | --- | --- |
| 函数式接口 | 参数类型 | 返回类型 | 用途 |
| Consumer<T>  消费型接口 | T | void | 对类型为T的对象应用操作，包含方法 :  void accept(T t) |
| Supplier<T>  供给型接口 | 无 | T | 返回类型为T的对象，包含方法 :  T get() |
| Function<T, R>  函数型接口 | T | R | 对类型为T的对象应用操作，并返回结果。结果是R类型的对象，包含方法 :  R apply(T t) |
| Predicate<T>  断定型接口 | T | boolean | 确定类型为T的对象是否满足某约束，并返回boolean 值。包含方法 :  boolean test(T t) |

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

**四、四种方法引用类型**

方法引用的标准形式是：类名::方法名。（**注意：只需要写方法名，不需要写括号**）

有以下四种形式的方法引用：

|  |  |
| --- | --- |
| **类型** | **示例** |
| 引用静态方法 | ContainingClass::staticMethodName |
| 引用某个对象的实例方法 | containingObject::instanceMethodName |
| 引用某个类型的任意对象的实例方法 | ContainingType::methodName |
| 引用构造方法 | ClassName::new |

默认方法 − 默认方法就是一个在接口里面有了一个实现的方法。

新工具 − 新的编译工具，如：Nashorn引擎 jjs、 类依赖分析器jdeps。

Stream API −新添加的Stream API（java.util.stream） 把真正的函数式编程风格引入到Java中。

Date Time API − 加强对日期与时间的处理。

Optional 类 − Optional 类已经成为 Java 8 类库的一部分，用来解决空指针异常。

Nashorn, JavaScript 引擎 − Java 8提供了一个新的Nashorn javascript引擎，它允许我们在JVM上运行特定的javascript应用。

## 设计模式

### 设计模式设计原则

* + 1. 找出应用中可能需要变化之处，把它们独立出来，不要和那些不需要变化的代码混在一起。
    2. 针对接口编程，而不是针对实现编程
    3. 多用组合，少用继承。
    4. 为交互对象之间的松耦合设计而努力