目录

[1. TCP与UDP 2](#_Toc11666)

[2. 在浏览器中输入URL到显示的过程 4](#_Toc4890)

[3. 常用熟知端口号 4](#_Toc2250)

[4. http与https的区别 4](#_Toc17686)

[5. http常见错误码 6](#_Toc18313)

[6. SSH 7](#_Toc11224)

[7. Cookie与Session的区别 8](#_Toc18153)

[8. Epoll 8](#_Toc15595)

[9. 常用排序算法 9](#_Toc26547)

[10. 数据库索引 10](#_Toc22944)

[11. 数据库引擎 10](#_Toc8955)

[12. Sql语句优化 11](#_Toc18356)

[13. 事务 12](#_Toc31897)

[14. 大量数据时的优化 12](#_Toc17515)

[15. Static关键字 13](#_Toc2661)

[16. Annotation 13](#_Toc11604)

[17. volatile 14](#_Toc15972)

[18. synchronized 14](#_Toc21577)

[19. Lock接口 15](#_Toc21320)

[20. 乐观锁与悲观锁 16](#_Toc116)

[21. JUC 17](#_Toc30184)

[22. AQS 17](#_Toc27731)

[23. GC 18](#_Toc23464)

[24. 四种引用 19](#_Toc20762)

[25. JVM内存模型 20](#_Toc13538)

[26. JVM类加载 21](#_Toc20961)

[27. 线程状态 22](#_Toc9166)

[28. 线程调度 23](#_Toc28029)

[29. JAVA线程池 23](#_Toc25470)

[30. 集合相关 25](#_Toc5117)

[31. 常见异常 27](#_Toc26602)

[32. 常用linux命令 27](#_Toc9496)

[33. 虚拟地址和物理地址 28](#_Toc18205)

[34. Swap分区 29](#_Toc1815)

[35. Redis 30](#_Toc31997)

[36. 消息中间件 32](#_Toc16893)

[37. Spring 32](#_Toc26196)

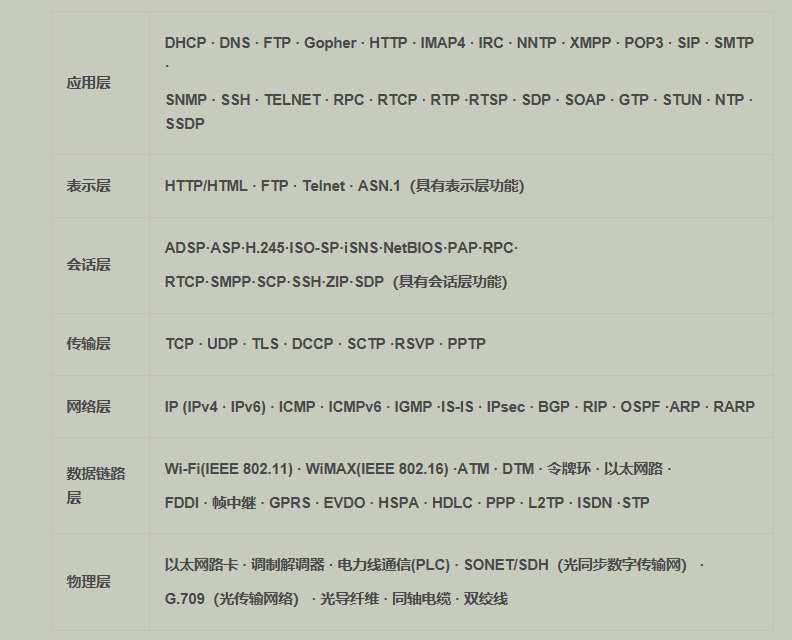
[38. SpringMVC 36](#_Toc10490)

[39. SpringCloud 36](#_Toc29942)

[40. JWT(JSON Web Token) 37](#_Toc11372)

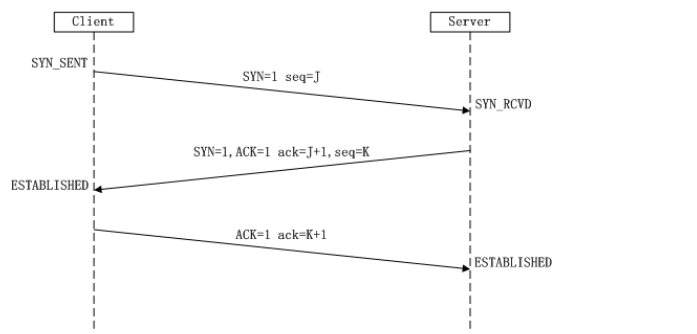
# TCP与UDP

1. TCP/IP协议栈只有4层，OSI模型的最下两层合为网络接口层，也有说法是前四层都是不变的，最上三层合为应用层，中间两层保留。



1. TCP面向连接的可靠的数据服务，面向字节流，通过TCP传输的数据保证不丢失、无错误、不重复并按序到达。提供全双工服务。有拥塞控制。TCP报头20个字节。一对一。

三次握手：



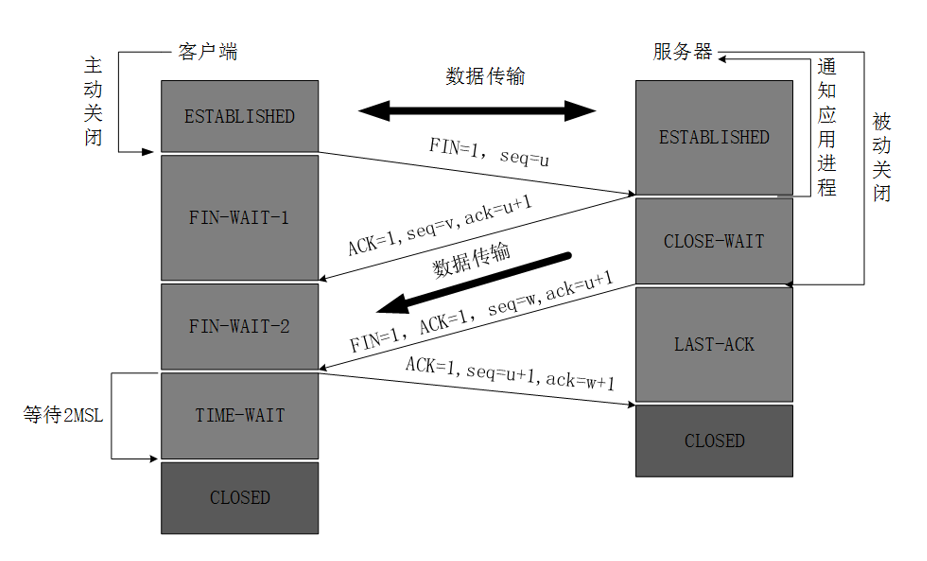
为什么要三次握手：为了建立可靠的通信信道，必须确认自己与对方的发送与接受都是正常的。如果只有一次握手，Client无法确认Server是否收到请求。如果只有两次握手，Server无法确认Client是否收到了请求。

SYN攻击：服务段在发送了SYN报文后，等待客户端返回ack报文，而客户端这时不仅不发送ack报文，反而再发送更多的tcp请求的报文，同样不给服务端的SYN报文发送确认报文。就这样通过发送大量的半连接请求，耗费服务端CPU和内存资源，甚至能危害路由器和防火墙等网络系统。

如何避免syn攻击：

1. 无效连接的监视与释放
2. 延缓分配TCB，使用syn cache和syn cookie技术
3. 使用syn proxy防火墙

四次挥手：

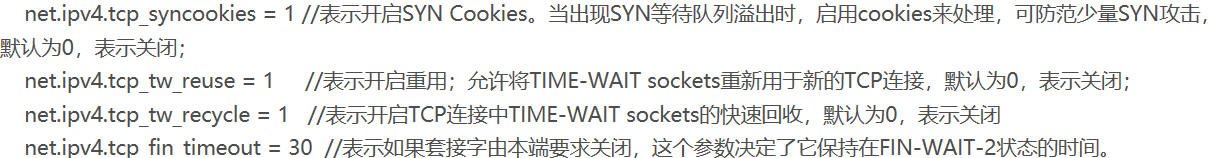


为什么Client进入TIME-WAIT之后还要等待2\*msl（最长报文段存活时间）：

1. 保证Client最后发送的ACK能顺利到达服务器。
2. 为了防止类似“三次握手”握手中提到的已经失效的报文出现在连接中。等到2\*msl可以保证本次连接的所有报文都从网络中消失了。

如何避免进入time-wait：

编辑/etc/sysctl.conf中的内容：

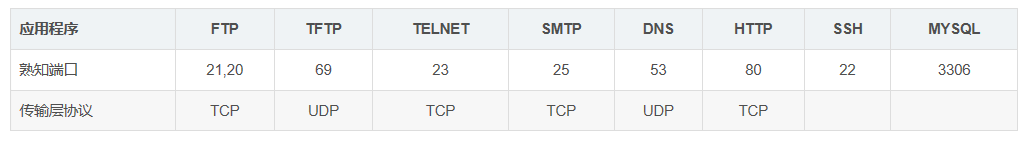


1. TCP如何保证可靠传输：
2. 将数据划分成合适大小的数据块。
3. 每个包都有编号，保证次序
4. 有校验和
5. 丢弃收到的重复数据
6. 使用滑动窗口来保证流量控制，接受方发送的确认报文中用窗口字段来控制发送窗口大小，该字段为0时，发送方不发送数据。
7. 有拥塞控制：慢启动、拥塞避免、快重传和快恢复。
8. 使用了停止-等待协议和超时重传机制
9. UDP提供无连接的服务，尽最大努力交付，面向报文。没有拥塞控制。报头8个字节。可一对一，一对多，多对多。

# 在浏览器中输入URL到显示的过程

1. DNS解析协议：一般采用递归和迭代查询，先询问本地域名服务器，看缓存里面有没有，没有的话问根域名服务器，再到顶级域名服务器，以此类推直到找到正确的IP地址。
2. 找到IP地址之后，建立TCP连接。
3. 发送HTTP/HTTPS请求。
4. 服务端收到请求，进行相应处理，然后进行HTTP/HTTPS响应
5. 浏览器收到响应并进行解析渲染。
6. 断开连接。

# 常用熟知端口号



# http与https的区别

HTTP1.1规定了默认保持长连接（HTTP persistent connection ，也有翻译为持久连接），数据传输完成了保持TCP连接不断开（不发RST包、不四次握手），等待在同域名下继续用这个通道传输数据，其头部有个和connection字段和keep-alive字段，分别表示连接的方式和超时时间；相反的就是短连接。

http协议传输的数据是明文传输，没有加密，信息不安全。因此网景公司设计了SSL协议用于对http传输的数据进行加密，诞生了https。https的主要作用就是1.建立信息安全的通道，保证数据的安全传输。2.确认网站的真实性。SSL现在已经不常用了，经修改变成了TLS。

区别如下：

1. https协议需要一个CA证书（这个时候可以说说在公司的时候怎么弄的）
2. http是超文本传输协议，是明文传输的。https是使用SSL/TLS加密的。
3. 常用端口不一样，http是80端口，https是443端口
4. http协议是无状态的；https协议SSL/TLS+http构建的可进行加密传输、身份认证的网络协议。

数字签名：使用hash算法对传送内容进行hash值计算，得到摘要，然后使用私钥对摘要进行加密，得到数字签名。然后把签名附在传送内容的后面，一起发送给给接受方。

数字证书：证明公钥的真实性。数字证书包含了网站的信息和网站的公钥，接收方收到数字证书之后即可向权威机构发送信息（该过程通用使用公钥、私钥机制，接收方使用权威机构的公钥对证书进行解密）检查发送方的真实性，并取得发送方的公钥。

https连接过程：

（1）客户使用https的URL访问Web服务器，要求与Web服务器建立SSL连接。

（2）Web服务器收到客户端请求后，会将网站的证书信息（证书中包含公钥）传送一份给客户端。（非对称加密）

（3）客户端的浏览器与Web服务器开始协商SSL连接的安全等级，也就是信息加密的等级。

（4）客户端的浏览器根据双方同意的安全等级，建立会话密钥，然后利用网站的公钥将会话密钥加密，并传送给网站。

（5）Web服务器利用自己的私钥解密出会话密钥。

（6）Web服务器利用会话密钥加密与客户端之间的通信。（对称加密）

非对称加密算法：RSA，DSA/DSS

对称加密算法：AES，RC4，3DES

HASH算法：MD5，SHA1，SHA256

http请求：

请求方法 url 协议版本

头部字段名称：值

...

请求体

http响应：

协议版本 状态码 状态码说明

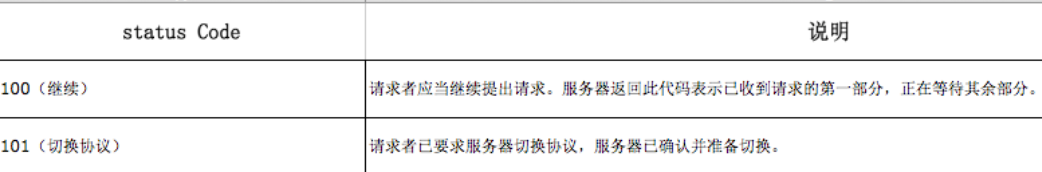
头部字段名称：值

...

响应体

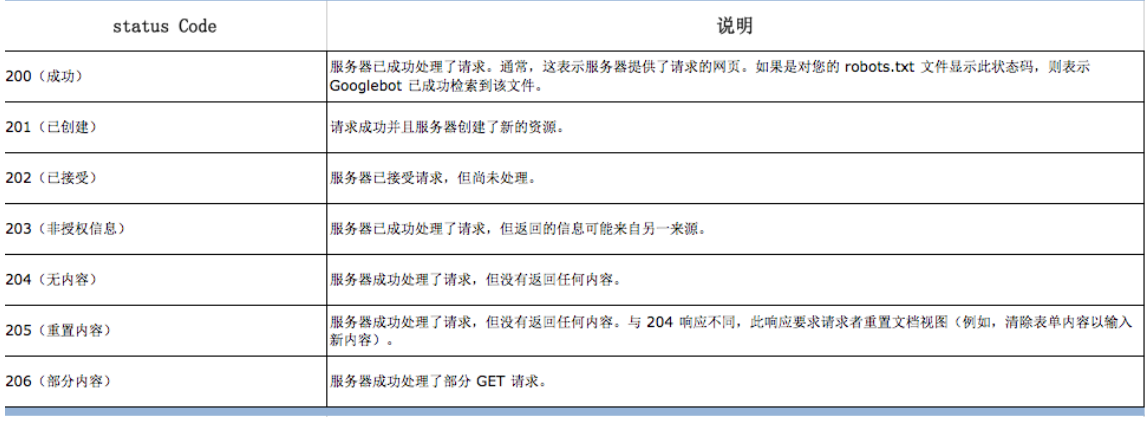
# http常见错误码

1xx（临时响应）表示临时响应并需要请求者继续执行操作的状态码



2xx （成功）

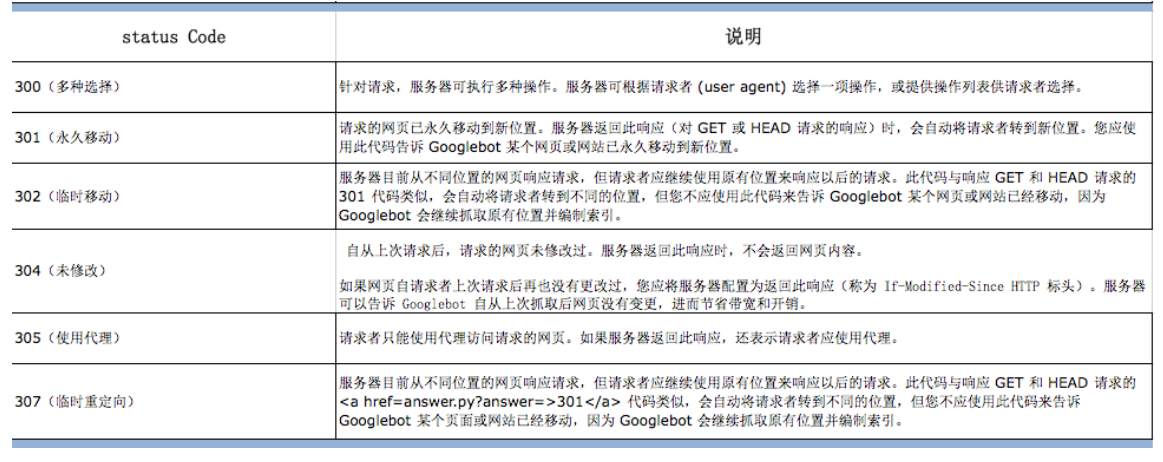
表示成功处理了请求的状态码。



3xx （重定向）

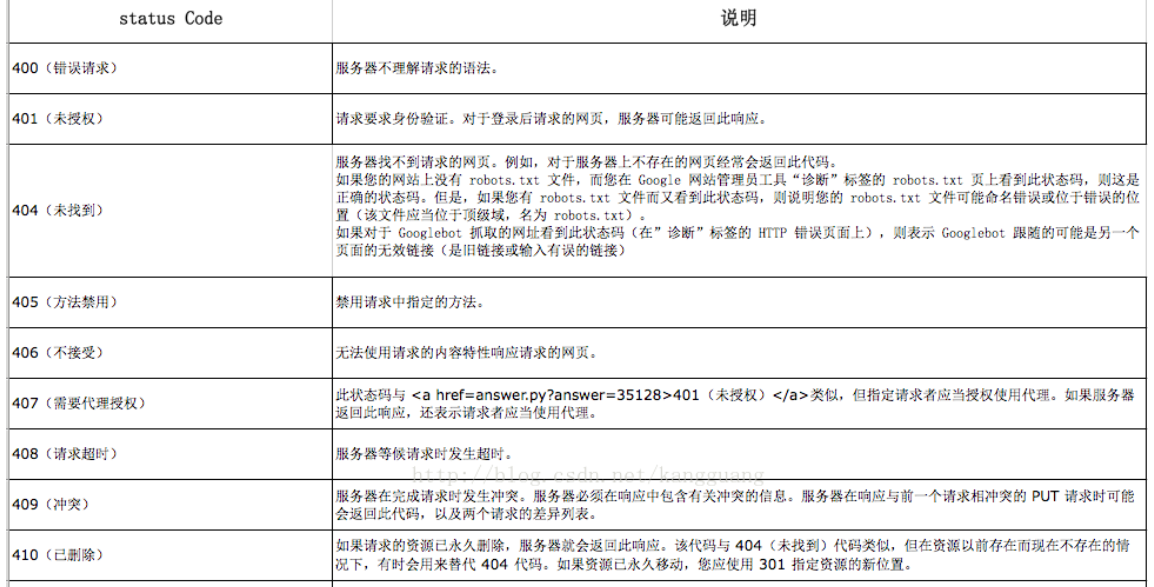
要完成请求，需要进一步操作。通常，这些状态码用来重定向。Google 建议您在每次请求中使用重定向不要超过 5 次。您可以使用网站管理员工具查看一下 Googlebot 在抓取重定向网页时是否遇到问题。诊断下的网络抓取页列出了由于重定向错误导致

Googlebot 无法抓取的网址。



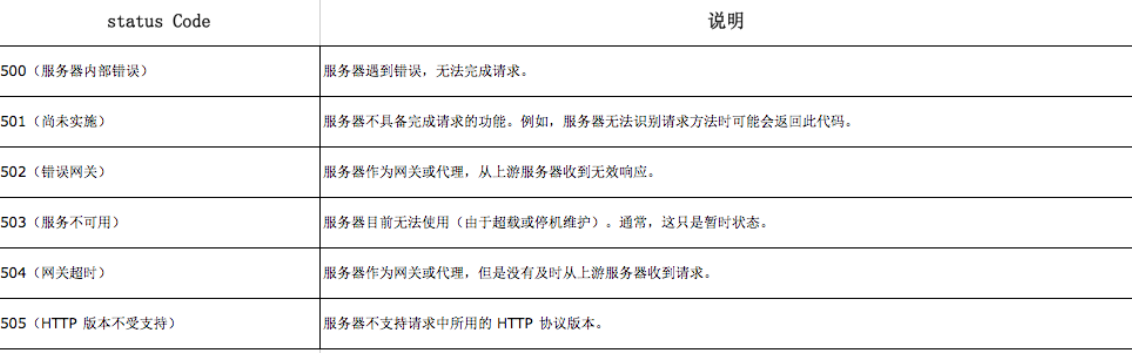
4xx（请求错误）

这些状态码表示请求可能出错，妨碍了服务器的处理。



5xx（服务器错误）

这些状态码表示服务器在处理请求时发生内部错误。这些错误可能是服务器本身的错误，而不是请求出错。



# SSH

SSH 是目前较可靠，专为远程登录会话和其他网络服务提供安全性的协议。可以抵御中间人攻击。OpenSSH为SSH协议的开源实现。

在整个通讯过程中，为实现SSH的安全连接，服务器端与客户端要经历如下五个阶段：

\*版本号协商阶段，SSH目前包括SSH1和SSH2两个版本，双方通过版本协商确定使用的版本

\*密钥和算法协商阶段，SSH支持多种加密算法，双方根据本端和对端支持的算法，协商出最终使用的算法

\*认证阶段，SSH客户端向服务器端发起认证请求，服务器端对客户端进行认证

\*会话请求阶段，认证通过后，客户端向服务器端发送会话请求

\*交互会话阶段，会话请求通过后，服务器端和客户端进行信息的交互

从客户端来看，SSH提供两种级别的安全验证：

1.第一种级别（基于口令的安全验证）

只要你知道自己帐号和口令，就可以登录到远程主机。所有传输的数据都会被加密，但是不能保证你正在连接的服务器就是你想连接的服务器。可能会有别的服务器在冒充真正的服务器，也就是受到“中间人”这种方式的攻击。

2.第二种级别（基于密匙的安全验证）

需要依靠密匙，也就是你必须为自己创建一对密匙，并把公用密匙放在需要访问的服务器上。如果你要连接到SSH服务器上，客户端软件就会向服务器发出请求，请求用你的密匙进行安全验证。服务器收到请求之后，先在该服务器上你的主目录下寻找你的公用密匙，然后把它和你发送过来的公用密匙进行比较。如果两个密匙一致，服务器就用公用密匙加密“质询”（challenge）并把它发送给客户端软件。客户端软件收到“质询”之后就可以用你的私人密匙解密再把它发送给服务器。

# Cookie与Session的区别

1. cookie数据存放在客户端，session的数据存放在服务器端。
2. Cookie不是很安全，可以分析保存在本地的cookie，进行cookie欺骗，考虑到安全的时候应该用session。
3. Session会在服务器上保存一段时间，考虑到访问增多时，服务器上存放的session也会增多，会影响性能。
4. 单个cookie保存的数据不能超过4k，浏览器会限制一个站点保存的cookie数量，一般是20个。
5. Session是用session id来区别用户的，所以底层可能用到cookie（cookie被禁止时也可以加在URL后面）

# Epoll

常见IO模型：

1. 阻塞IO模型
2. 非阻塞IO模型
3. IO复用模型
4. 信号驱动IO模型
5. 异步IO

Select/poll的缺点:

1. 单个进程能够监视的文件描述符的数量存在最大限制，通常是1024，当然可以调，但是由于select采用了轮询的方式扫描文件描述符，文件描述符越多，性能越差。
2. 内核/用户空间内存拷贝问题，select需要复制大量的句柄数据结构，产生巨大的开销。
3. Select返回的是含有整个句柄的数组，应用程序需要遍历整个数组才能发现哪些句柄发生了事件。
4. Select的触发方式是水平触发，应用程序如果没有完成对一个已经就绪的文件描述符进行IO操作，那么之后每次select调用还是会将这些文件描述符通知进程。
5. Poll虽然使用了链表保存文件描述符，因此没有了监视文件数量的限制，但是还有其他三个缺点。

epoll通过在Linux内核中申请一个简易的文件系统（一般采用B+树），把原先的select/poll调用分成了三个部分：

1. 调用epoll\_create()建立一个epoll对象。（epoll在文件系统中为这个句柄对象分配资源）
2. 调用epoll\_ctl，向epll对象中添加流。
3. 调用epoll\_wait收集发生的事件的连接。epoll\_wait效率很高，并不会返回所有的连接的句柄数据，而且不需要遍历。

调用epoll\_create方法的时候，会创建一个eventpoll结构体，里面有两个很重要的成员，rb\_root（红黑树的根节点，保存了添加到epoll中的需要监控的时间）和list\_head（双向链表，存放着将要通过epoll\_wait返回给用户的满足条件的事件）。

两种工作模式：ET/LT

LT是默认的工作方式，相当于效率很高的poll模型；而ET是高效的工作方式。ET模式仅当状态发生变化的时候才获得通知,这里所谓的状态的变化并不包括缓冲区中还有未处理的数据,也就是说,如果要采用ET模式,需要一直read/write直到出错为止,很多人反映为什么采用ET模式只接收了一部分数据就再也得不到通知了,大多因为这样;而LT模式是只要有数据没有处理就会一直通知下去的。

# 常用排序算法



# 数据库索引

使用索引的优点：

1. 使用唯一索引的时候，可以保证数据库表中数据的唯一性。
2. 加快检索速度。
3. 加快表的连接。
4. 使用分组和排序子句检索的时候，可以减少检索时间。
5. 可以在查询的过程中，使用优化隐藏器，提高系统性能。

使用索引的缺点：

1. 创建和维护索引需要时间，而且数据越多花费越多。
2. 索引需要额外的物理存储地址。
3. 数据维护的时候，对应的索引也要维护。

最左前缀原则：

如果有一个2列的索引(col1,col2),则已经对(col1)、(col1,col2)上建立了索引；  
如果有一个3列索引(col1,col2,col3)，则已经对(col1)、(col1,col2)、(col1,col2,col3)上建立了索引；

当b+树的数据项是复合的数据结构，比如(name,age,sex)的时候，b+树是按照从左到右的顺序来建立搜索树的，比如当(张三,20,F)这样的数据来检索的时候，b+树会优先比较name来确定下一步的所搜方向，如果name相同再依次比较age和sex，最后得到检索的数据；但当(20,F)这样的没有name的数据来的时候，b+树就不知道下一步该查哪个节点，因为建立搜索树的时候name就是第一个比较因子，必须要先根据name来搜索才能知道下一步去哪里查询。比如当(张三,F)这样的数据来检索时，b+树可以用name来指定搜索方向，但下一个字段age的缺失，所以只能把名字等于张三的数据都找到，然后再匹配性别是F的数据了， 这个是非常重要的性质，即索引的最左匹配特性。

# 数据库引擎

Mysql底层用的数据引擎：MyISAM和InnoDB。MyISAM不支持行锁、事务、外键和安全恢复，优势在于读取速度快，占用资源较少，支持全文类型索引，保存表的总行数，适用于读多写少的场景。InnoDB支持行锁、事务、外键和快速恢复，没有全文类型索引，获取表的行数只有扫描。

数据库引擎中采用的数据结构主要是哈希表和B+树。区别在于哈希表对于单条记录的查询很快，但是对于一个范围内的查询还是B+树更好。

MyISAM与InnoDB中的B+树有区别，在于MyISAM中B+树的叶子节点保存的是数据记录的地址，也就是索引和数据是分开的，称为“非聚簇索引”，主索引与辅助索引区别不大；InnoDB数据文件本身就是索引文件。B+树的叶子中就保存了完整的数据。索引的key是表的主键，而辅助索引里面data域保存的是主键的值。所以一定要求表有一个主键，而且名字不能太长，而且不建议非单调字段，称为“聚簇索引”。

mvcc（multiversion concurrency control）：即多版本并发控制技术，它使得大部分支持行锁的事务引擎，不再单纯的使用行锁来进行数据库的并发控制，取而代之的是把数据库的行锁与行的多个版本结合起来，只需要很小的开销，就可以实现非锁定读，读写不冲突，从而大大提高数据库系统的并发性能。缺点是每行记录都需要额外的存储空间，需要做更多的行维护和检查工作。

实现原理：

1.每行数据都存在一个版本，每次数据更新时都更新该版本。

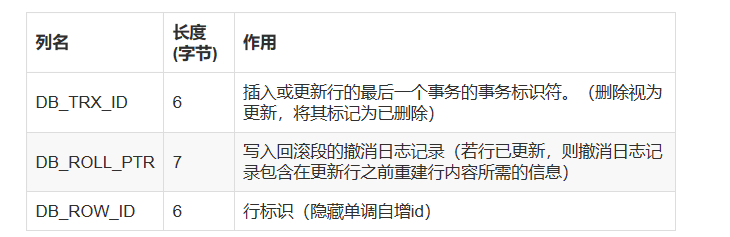
2.修改时Copy出当前版本随意修改，各个事务之间无干扰。

3.保存时比较版本号，如果成功（commit），则覆盖原记录；失败则放弃copy（rollback）

主要用到了一个uodo log，其简单的工作流程如下：



在InnoDB数据库引擎中，为每行数据增加了三个隐藏列：



# Sql语句优化

主要思想是避免全表扫描，合适的使用索引。经验表明，一张表上最多不要超过6个索引，优先考虑在where及order by上建立索引。

例子：

1. 避免使用 ！=或<>
2. 避免使用null，利用可用0代替null
3. 避免使用\*，将要查询的写完整，不需要的就不查询
4. 避免使用in或者not in，如果是连续的可以用between和and代替，或者exists代替。
5. 避免在where子句”=“的左边进行表达式操作、函数等等

# 事务

ACID

事务的5个特性：传播方式、隔离级别、是否只读、回滚规则、事务超时

并发事务的常见问题：脏读、不可重复读、幻读

隔离级别：read uncommitted、read committed、repeatable read、serializable

# 大量数据时的优化

1. 使用limit限定查询范围
2. 读写分离
3. 垂直拆分：

垂直分库：根据业务的耦合性，将关联度低的不同表存储在不同的数据库中，例如每个微服务单独使用一个数据库。

垂直分表：基于“列”进行，某个表中字段过多，可以新建一张扩展表，将不经常用或者字段长度较大的字段放到扩展表中。

优点：

1. 解决业务系统层面的耦合，使业务更清晰
2. 便于对不同业务的管理
3. 在一定程度下提升数据库性能

缺点：

1. 部分表不能join，只能通过接口聚会方式解决，提升开发复杂度
2. 分布式事务处理复杂
3. 依然存在单表数量过大的问题
4. 水平分区

分片规则：

1. 根据数值范围划分，例如时间区间、ID区间。问题在于热点数据可能成为性能瓶颈
2. 分局数值取模。问题在于扩容的时候，需要迁移旧数据，容易面临跨分片查询的问题。

库内分表：解决了单一表数据量过大的问题，但是没有将表分不到不同的机器上，没有减轻数据库压力。

分库分表

优点：

1. 不存在单库大数据和高并发的性能瓶颈
2. 应用端改造较少
3. 提供了系统的稳定性和负载能力

缺点：

1. 分片事务一致性难以解决
2. 跨节点join性能差，逻辑复杂
3. 数据多次扩展难度跟维护量极大

解决方案：

1. 客户端代理：分片逻辑在应用端，封装在jar包中，通过修改或者封装JDBC层来实现。
2. 中间件代理：在应用和数据中间加了一个代理层。分片逻辑统一维护在中间件**服务中。**

# **Static关键字**

修饰成员变量和成员方法

静态代码块

修饰类（只能修饰内部类）

静态导包：用来导入指定类中的静态资源，就可以不使用类名直接调用

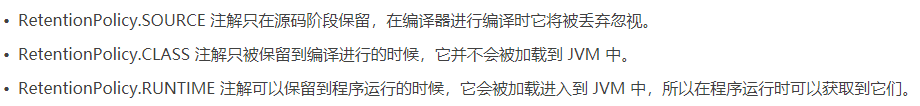
# **Annotation**

jdk内置三种标准注解：

1. @Override:子类重写父类中的方法
2. @Deprecated:表明该方法或者该元素已经过时
3. @SuppressWarnings：阻止编译器警告

自定义注解：有五个元注解可以使用

@Retention：表明注解的生命周期，有以下三个值可选



@Document：是否将注解中的元素加入JavaDoc中

@Target：表明注解可以修饰的对象，有八个值可选

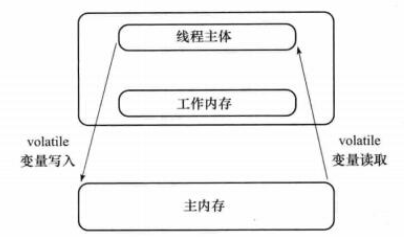
@Inherited：表明注解是否可以继承

@Repeatable：JDK1.8之后的新注解，表明注解的值可以取多个

//以上5个注解写在自定义注解之前

Public @interface MyAnnotation{}

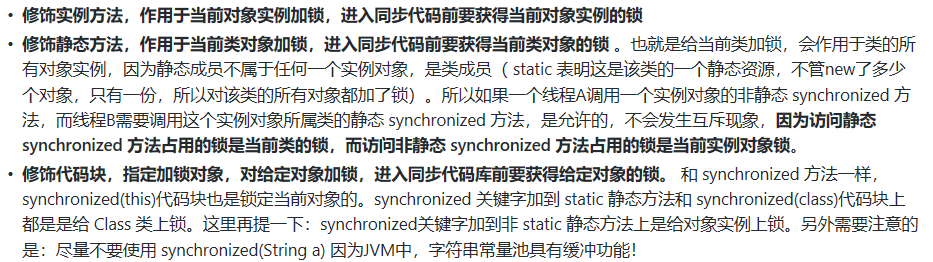
# volatile



使用volatile修饰的变量在每次被线程访问的时候，都强迫从主内存中重读该成员变量的值。而且当成员变量发生变化时，强迫线程将变化值协会到主存。JVM会尽量保持变量的可见性，意思就是只要CPU有时间就会尽量保证数据的更新，但是如果CPU一直没时间，就需要用volatile关键字保证。

volatile保证变量的可见性，但是不保证原子性。多线程访问volatile修饰的变量不会发生阻塞

# synchronized



使用双重校验实现的单例模式：



在底层原理：

1. synchronized修饰代码块的时候，使用了monitorenter和monitorexit指令，monitorenter指向同步代码块开始的地方，monitorexit指向同步代码块结束的地方。
2. Synchronized修饰方法的时候，使用了ACC\_SYNCHROZIED标志位

JDK1.6之后对synchronized的优化：

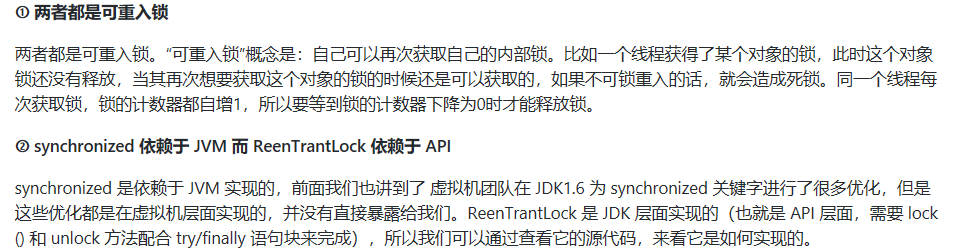
1. 将锁的状态分为：无锁状态、偏向锁、轻量级锁、重量级锁
2. 采用自旋锁、适应性自旋锁、锁消除、锁粗化等技术

# Lock接口

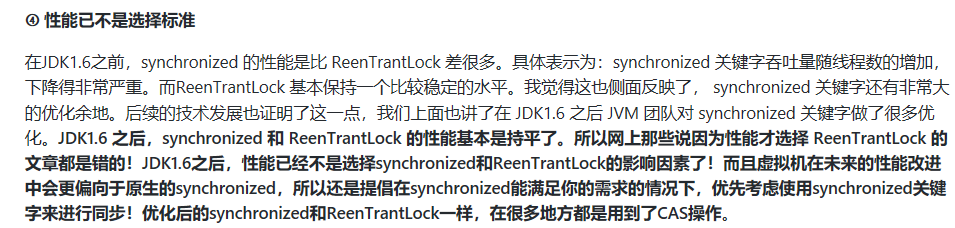
主要实现类有：

ReentranLock、ReentrantReadWriteLock.ReadLock、ReentrantReadWriteLock.WriteLock

Lock与synchronized的对比：







# 乐观锁与悲观锁

悲观锁：总是假设最坏的情况，认为每次拿数据都会对数据进行修改，synchronized与Lock都是悲观锁，数据库中的表锁、行锁也是。

乐观锁：总是假设最好的情况，认为每次拿到数据后不会进行修改，适合读多写少的场景，常用实现方式：版本号机制和CAS

版本号机制：数据中多加一个version字段，当数据被修改的时候，version值会加一。读取数据的时候，也会读入当时的版本号，当一个线程修改数据并提交的时候，会比较自己读取时候的版本号和当前的版本号，只有提交版本号大于当前记录的版本时才会执行数据更新。

CAS（compare and swap）算法：常用的无锁算法，非阻塞同步。只有读写的内存值与进行比较的值相等时，才会用新值进行更新。一般情况下是用自旋锁实现。

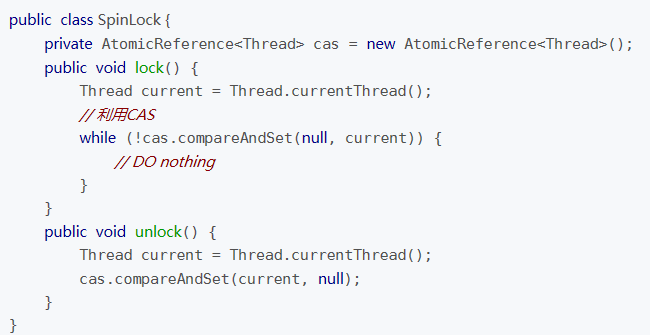
乐观锁的缺点

1. 不能解决ABA问题
2. 循环时间长开销大
3. 只能保证一个共享变量的原子操作。但是JDK1.5之后JUC包下的atomic类等可以解决这个问题

补充：自旋锁

自旋锁的问题：如果某个线程持有锁的时间过长，会导致等待该锁的线程进入循环等待，消耗CPU；非公平的

最简单的实现：



常用变种：

1. TicketLock，解决公平问题
2. CLHLock，申请线程只在本地变量上自旋，不断轮询前驱状态，基于隐式链表，没有实际的链表存在
3. MCSLock，对本地变量的节点进行循环。使用显示链表，有真正的链表存在。

# JUC

常用Atomic原子类：

1. 基本类型，eg: AtmoicInteger
2. 数组类型，eg: AtmoicIntegerArray
3. 引用类型，eg: AtmoicReference
4. 引用属性修改类型，eg: AtmoicIntegerFiledUpdater

# AQS

AQS(abstractQueuedSynchronizer)，用来构建锁和同步器的框架。

原理：如果被请求的共享资源空闲，那么将当前请求资源的线程作为有效的工作线程，并且将共享资源设定为锁定状态。如果被请求的资源已经被占用，那么就需要一套线程阻塞等待以及被唤醒的机制，AQS使用CLH队列实现。

资源的两种共享方式：

1. 独占
2. 共享

常用同步器：

1. Semaphore（信号量），运行多个线程同时访问某个资源

Semaphore semaphore = new Semaphore(int m, boolean fair)

semaphore.acquire(int n) 可以有个参数，默认为1，实际允许线程数量为 m/n

semaphore.release(int n)

1. CountDownLatch（倒计时器），允许一个或者多个线程一直等待，直到其他线程的操作执行完之后才执行。三种常用场景：某一线程在开始运行前等待n个线程执行完毕；实现多个线程开始执行任务的最大并行；死锁检测

CountDownLatch cdl = new CountDownLatch(int count) 为倒计时器设置初始值

cdl.await()

cdl.countDown() 每运行一次count减一

CountDownLatch是一次性的，只能在初始化的时候设置一次count的值，所以用完之后就不能再用了

1. CyclicBarrier（循环栅栏）让一组线程到达一个屏障然后等待，直到等待的线程数量达到设定值才继续往下执行。

CyclicBarrier cb = new CyclicBarrier(int parties, Runnable barrierAction)，第二个参数为可选参数，表示在线程到达屏障时优先执行的方法

cb.await()

CyclicBarrier可以reset，因此可以多次使用。



# GC

可达性分析：

1. 引用计数法
2. 可达性分析。可作为GC root的有：虚拟机栈中引用的对象、方法区中的静态属性引用对象、方法区中的常量引用对象、本地方法栈中的对象

4种引用：强引用、软引用、弱引用、虚引用

对象死亡前的两次扫描

垃圾回收算法：

标记-清除

标记-整理

复制算法

分代收集算法，新生代由于每次回收都有大量对象死去，所以常用复制算法，老年代常用两外两种

常用收集器：

CMS收集器，一种以获取最短回收停顿时间为目标的收集器，基于标记-清除算法。过程有：初始标记、并发标记、重新标记、并发清除

G1收集器：适合多颗CPU和多核环境下。总体上看基于标记-整理算法。可以预测停顿。过程有：初始标记、并发标记、最终标记、筛选回收

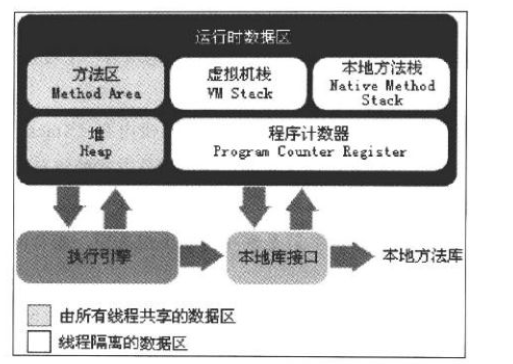
对象分配策略：

对象优先在eden分配，大对象直接进入老年代，长期存活对象进入老年代，采用动态年龄判定，采用空间担保策略。

# 四种引用

1. 强引用：代码中最普遍的存在，类似“Object obj = new Object（）”，只要强引用还在，JVM绝对不会回收，宁愿抛出OOM
2. 软引用：用于描述一些还有用但非必需的对象。只有在内存不足的时候才会被回收，用java.lang.ref.SoftReference类表示。常用来表示页面缓存、图片缓存等。
3. 弱引用：被弱引用关联的对象只能生存道下一次垃圾收集之前，用java.lang.ref.WeakReference类表示。用以构造弱集合，例如添加到类库中的WeakHashMap，使用弱引用关联的对象作为key，如果该对象已经被回收，那么get()方法返回null，有一个私有方法expungeStaleEntries用于回收过时对象。
4. 虚引用：不影响被关联对象的生命周期，设置虚引用的唯一目的就是能在这个对象被收集器回收时收到一个系统通知。

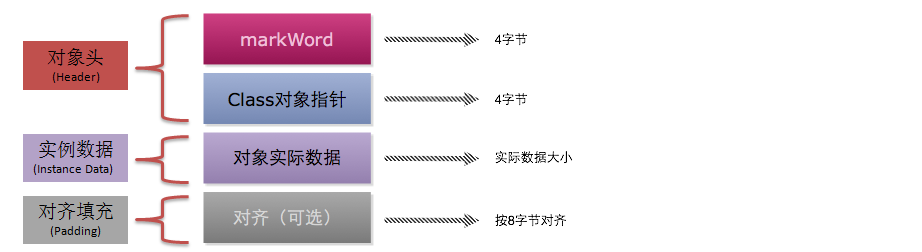
# JVM内存模型



对象的创建：

类加载检查-分配内存空间-初始化设置-设置对象头-调用初始化方法

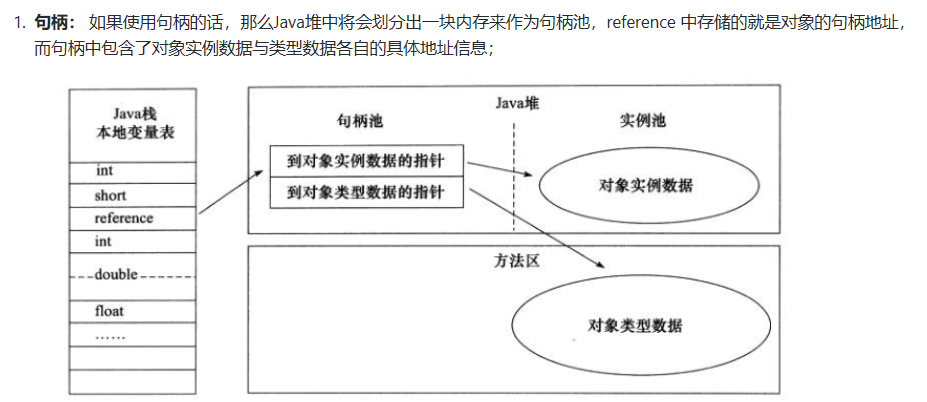
对象的结构：



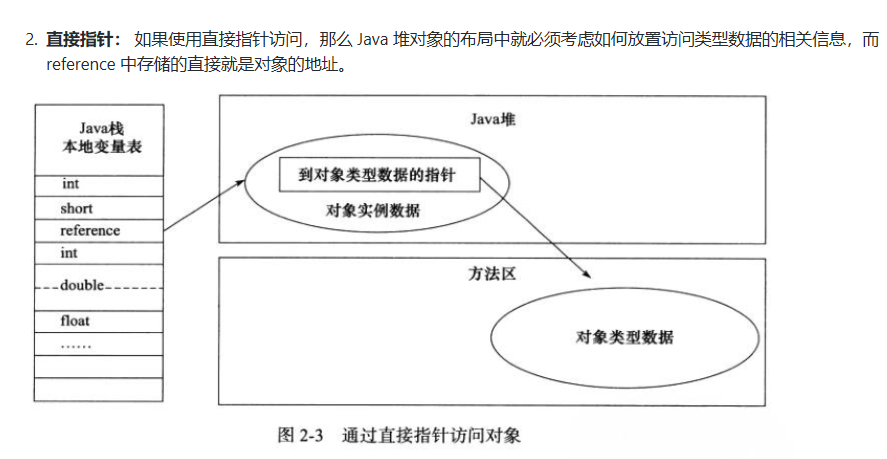
Mark word用于存储对象自身运行时数据，主要有：哈希码、GC分代年龄、锁状态标志、线程持有的锁、偏向线程ID、偏向时间戳等

对象的访问定位：

1. 句柄

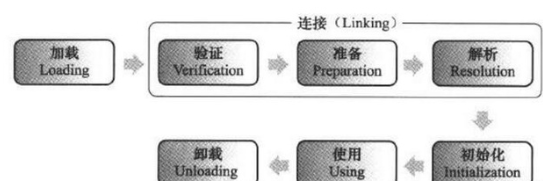


1. 直接指针



# JVM类加载

类加载大体过程：



第一步加载的过程：

1. 通过类的权限定名，获取对应的二进制字节流。
2. 根据二进制字节流对应的静态数据结构建立方法区中的运行时数据结构。
3. 声明一个该类对应的java.lang.Class对象，作为方法去中这个类的各种数据的访问接口

验证：确保class文件中包含的字节流符合当前虚拟机的需求，而且不会危害虚拟机安全。验证的东西主要有：文件格式、元数据、字节码、符号引用

准备：在方法区中为类变量分配内存空间，并设置类变量初始值（static变量）。

解析：将符号引用替换为直接引用

初始化：根据程序员定制的主观方法初始化类变量或者其他资源，或者说执行类构造器的<clinit>（）方法。<clinit>（）是自动收集所有类变量的赋值动作和静态语句块中的代码合并成的，收集顺序由源文件中出现的顺序决定。该方法是非必需的。

类加载器：

Bootstrap ClassLoader(启动类加载器)，最上层的加载器，么有父类加载器

Extension ClassLoader(扩展类加载器)

Application ClassLoader(应用程序加载器)

双亲委派模型：除了最上层的加载器外，每个加载器都有自己的父类加载器。当一个类加载器收到加载请求的时候，他不会自己区尝试加载这个类，而是交给父类去处理，然后以此类推，直到父类加载器不能加载的时候，子类加载器才会尝试自己去加载。

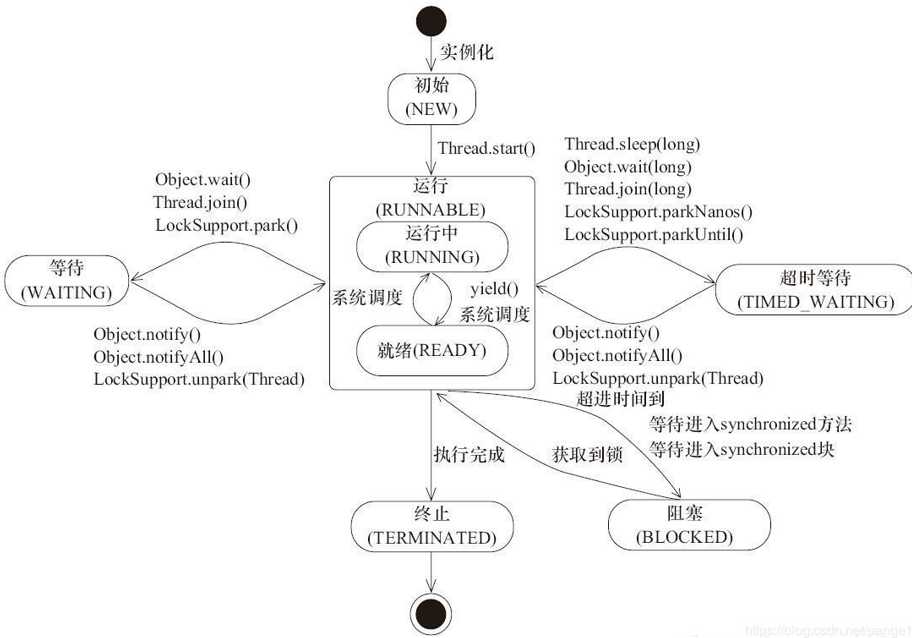
双亲委派模型的三次破坏：

1. JDK版本的升级，1.0——>1.2
2. 当基础类要调用用户自己写的代码的时候，例如JNDI服务
3. 代码热替换、模块热部署等，例如OSGi

无用类：

1. 该类所有的对象实例都已经被回收。
2. 该类的ClassLoader已经被回收。
3. 改类对应的java.lang.Class对象没有在任何地方被引用，没有反射用到该类的地方。

# 线程状态



# 线程调度

JVM的线程调度：采用抢占式调度，由系统分配线程的执行时间，线程的切换不由线程本身决定。如果想某个线程多一点执行时间可以设置线程优先级，分为1-10个等级，默认为5，级别越高越容易被系统选中。

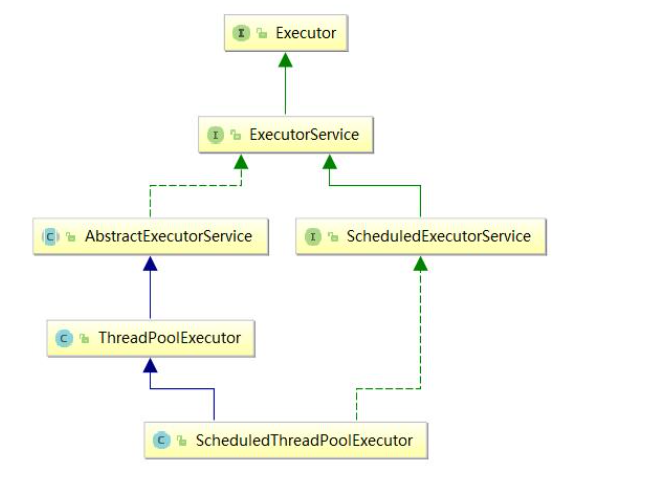
Linux线程调度策略：

1. SCHED\_OTHER:分时调度策略
2. SCHED\_FIFO实时调度策略，先到先得，一直运行直到结束或者更高优先级的线程到达
3. SCHED\_RR实时调度策略，时间片轮转。

# JAVA线程池

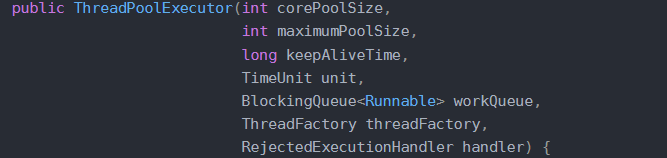
创建线程的三种方式的区别：

1. 使用Runnable和Callable接口方式还可以继承其他类
2. 使用Runnable和Callable接口方式可以共享一个Target
3. 使用Callable接口方式可以有返回值，而且可以抛出异常，返回值是Future类型的，它的实现类是FutureTask。
4. 使用继承Thread类的方式的话，获取当前线程比起另外两种方式要方便一点。



线程池常用实现类有两个，ThreadPoolExecutor和ScheduledThreadPoolExecutor

ThreadPoolExecutor的构造方法，牢记参数的含义。



阻塞队列有三种可选：

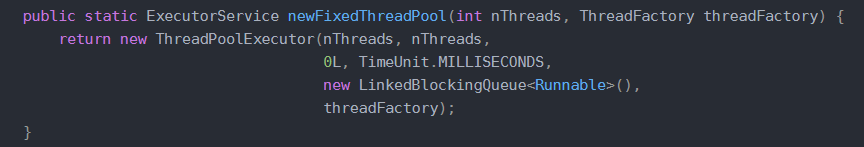
1. ArrayBlockQueue
2. LinkedBlockQueue
3. SynchronousQueue：适用于CachedThreadPool，不会保存提交的任务，而是新建线程

最后一个参数拒绝策略有4种：

1. 丢弃任务，并抛出RejectedExecutionExecption
2. 丢弃任务，不抛出异常
3. 丢弃队列最前面的任务，并尝试重新执行任务
4. 交给调用的线程处理

创建ThreadPoolExecutor线程池的两种方式

1. 使用ThreadPoolExecutor的构造函数
2. 使用Exectors工具类提供的三种常用线程池：
3. FixedThreadPool：其底层实现是调用ThreadPoolExecutor的构造函数，参数如下，ThreadFactory可以缺省，使用默认的工厂。



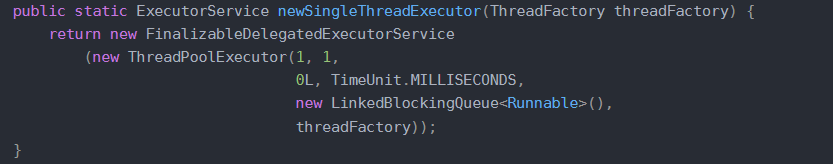
如果当前运行的线程数小于corePoolSize，则创建新的线程执行任务；

如果当前运行的线程数等于corePoolSize，则加入到无解链表阻塞队列（最大长度Integer.MAX\_VALUE）中；

线程执行完任务后从队列中取任务执行。

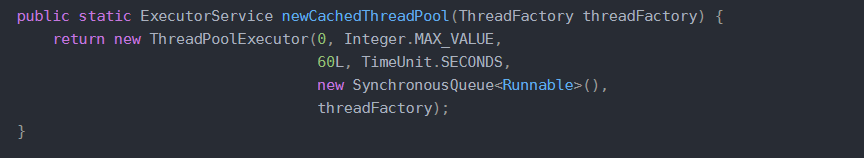
参数3，4无效

1. SingleThreadExctor：实现原理如下，线程工厂参数同样可以缺省。



线程池方式的单线程模式

1. CachedThreadPool：线程工厂参数同样可以缺省。

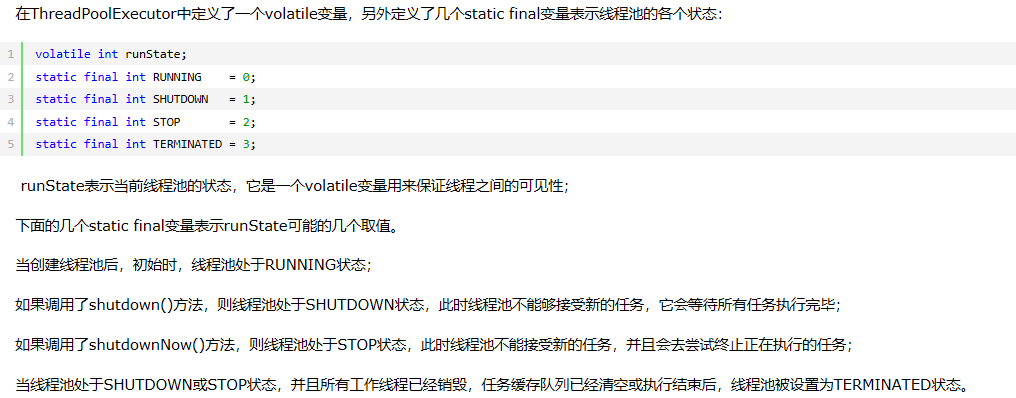


当新的任务到来的时候，如果没有空闲线程，则会创建新的线程，所以极端情况下会耗尽CPU资源。如果线程60秒内没有任务可执行则销毁该线程。

重要函数

shutdown()和shutdownNow()：shutdown()不会再向队列中添加新的任务，但是已经在队列中的任务会被完成。shutdownNow()会尝试停止正在执行的线程，然后关闭线程池。

IsSHutdown()和isTerminated()：跟线程池的状态有关系，状态如下：



ScheduledThreadPoolExecutor：主要用来在给定的延迟后运行任务或者定期执行任务。使用的队列为DelayedQueue

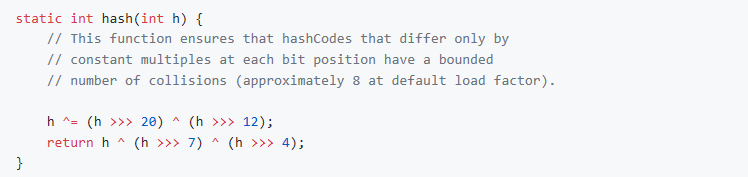
SingleThreadScheduledExector

# 集合相关

计算数组位置的办法是 hash&(n-1)

HashMap：

在JDK1.8以前，底层是使用数组和链表结合，而且hash方法也不同，JDK1.7的hash方法如下：



在JDK1.8以后，当链表长度大于阈值（默认为8）时，将链表转化为红黑树，且hash方法有变化：



HashMap的扩容机制：

默认初始化长度length为16，负载因子为load factor为0.75，threshold=length \* load factor是hashmap所能容纳的最大数据量的个数，实际容纳元素多于这个数后就开始扩容。前提是length不等于Integer.MAX\_VALUE，如果超过这个值，就不会再扩容了。如果没有超过最大值，扩容后的容量为以前的两倍。扩容后需要重新计算每个元素的位置，直接根据hashcode值的下一位来判断，要么等于原位置，要么等于原位置+oldcap。Threshold也会重新计算。

HashMap与Hashtable的区别：

1. HashMap非线程安全，Hashtable线程安全
2. HashMap比Hashtable的效率高
3. HashMap中null可以作为key，但这样的key只能有一个，而value可以为null，也可有多个。Hashtable中key和value都不可以为null
4. HashMap默认初始化大小为16，每次扩容变为以前的两倍；Hashtable默认初始化大小为11，容量为2\*n+1。
5. Hashmap底层使用数组+链表+红黑树，而hashtable不是。

ConcurrentHashMap与Hashtable的区别：

1. 底层数据结构不同
2. 实现线程安全的方式不同：Hashtable使用的是全表锁。ConcurrentHashMap在JDK1.8后对每个数组节点分别加锁（链表的头节点或者红黑树的根节点）。

HashMap为何是线程不安全的：

在并发情况下，发生扩容时，会产生循环链表，在执行get的时候，会触发死循环，引起CPU占用率100%的问题。

ArrayList扩容机制：

初始默认容量为10。加入元素之后用调用一个函数得到最小扩容量，然后再调用另一个函数判断是否需要扩容。如果需要扩容，newCapacity=oldCapacity + (oldCapacity >>1)。让背后再判断newCapacity是否大于可最小需要容量，如果还是小于最小需要容量，那么就把最小需要容量作为newCapacity。然后检查新容量是否大于MAX\_ARRAY\_SIZE，如果大于了，则newCapacity = Integer.MAX\_VALUE.最后把原数组中的数据拷贝到新的数据中。

# 常见异常

java.lang.ClassNotFoundException

找不到类异常。当应用试图根据字符串形式的类名构造类，而在遍历CLASSPAH之后找不到对应名称的class文件时，抛出该异常。

ConcurrentModificationException

在删除容器中的元素的时候，例如使用list.remove()而不是迭代器的remove方法删除这个元素的时候就会报错。而如果在多线程的情况下，则要对迭代器加synchronized关键字，如果是list的话可以用CopyOnWriteArrayList代替。

Java.lang.IllegalMonitorExecption

在使用wait()和notify()方法时，没有适当的锁，会报出该异常

# 常用linux命令

1. 进程相关命令：

ps命令：

ps -aux，查看所有系统运行的程序，会列出内存和CPU 的使用情况

ps -ef，显示所有进程，连同命令行

top命令：动态查看进程的变化，会列出系统的总体运行情况和每个进程具体的情况

top -d 1 刷新间隔

kill命令：杀死进程

kill -1 pid 重新启动被中止的进程

kill -9 pid 强行中断一个进程的运行

kill -15 pid 以正常的结束进程来终止该进程

1. 系统占用：

内存占用：free

IO占用；iostat

1. 管道相关：

cut 分割一段信息，以行为单位

cut -d ‘分隔符’ -f filed：以分隔符作为分割，并取第filed段

grep 可以使用正则表达式搜索文本，并把匹配的行打出来

sort 排序命令

uniq 可用于排序后去除重复

wc 统计字符数目

数据流向：

标准输入：代码为0， 使用<或者<<

标准输出：代码为1， 使用>或者>>，

使用>时：如果目标文件不存在则创建，如果存在会先清空

如果不想清空，可使用>>

标准错误输出：代码为2，使用2>或者2>>

1. 文件

cat 从第一行起显示文件内容

cat -b 列出行号，但是空行没有行号

cat -n 列出行号，空行也有行号

tac 反向显示

more 可以一页一页的翻阅查看，但是不能往前翻页

less 比more更好用，可以前后翻页

head -n number 显示头number行

tail -n number 显示倒数number行

touch 创建或者修改文件日期等属性

1. 压缩： tar -zcvf

解压： tar -xvf

1. 磁盘：
   1. df：列出文件系统的整体磁盘使用量（默认显示的东西很多）
   2. du：评估文件系统的磁盘使用量（常用于评估目录所占容量）
2. 网络：

ping

netstat： 显示网络连接，路由表，接口统计数据，伪装连接，多播成员关系。

traceroot：发送icmp包

tcpdump：抓包命令

# 虚拟地址和物理地址

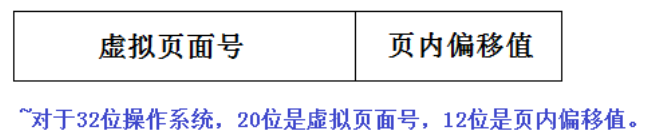
虚拟内存：虚拟内存是一种逻辑上扩充物理内存的技术。基本思想是用软、硬件技术把内存与外存这两级存储器当做一级存储器来用。虚拟内存技术的实现利用了自动覆盖和交换技术。简单的说就是将硬盘的一部分作为内存来使用。

虚拟地址空间：在32位的i386 CPU的地址总线的是32位的，也就是说可以寻找到4G的地址空间。我们的程序被CPU执行，就是在0x00000000到0xFFFFFFFF这一段地址中。高1G的空间为内核空间，由操作系统调用，低3G的空间为用户空间，由用户使用。这样是为了解决物理内存稀缺性问题。

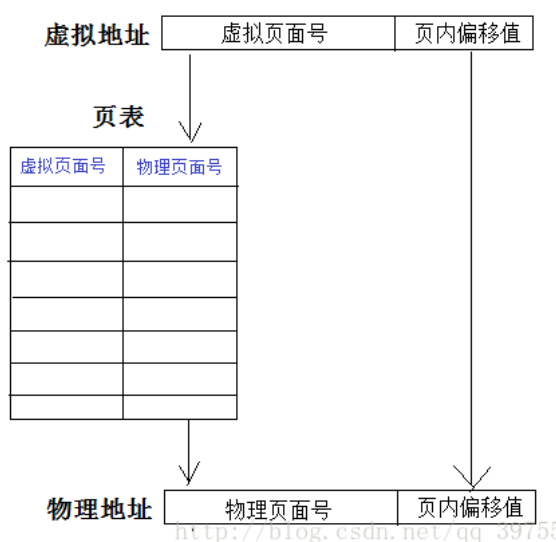
CPU在寻址的时候，是按照虚拟地址来寻址，然后通过MMU(内存管理单元)将虚拟地址转换为物理地址。因为只有程序的一部分加入到内存中，所以会出现所寻找的地址不在内存中的情况（CPU产生缺页异常），如果在内存不足的情况下，就会通过页面调度算法来将内存中的页面置换出来，然后将在外存中的页面加入到内存中，使程序继续正常运行。进程要知道哪些内存地址上的数据在物理内存上，哪些不在，还有在物理内存上的哪里，需要用页表来记录。

分页内存管理：

将虚拟内存空间和物理内存空间皆划分成大小相同的页面，例如4KB、8KB和16KB等。并将页作为内存空间的最小分配单位，一个程序的一个页面(虚拟页面)可以存放在任何一个物理页面中，会存在页内碎片。



转换过程如下：



还有段页式内存管理，会存在外部碎片。

# Swap分区

Linux系统的Swap分区，即交换区，Swap空间的作用可简单描述为：当系统的物理内存不够用的时候，就需要将物理内存中的一部分空间释放出来，以供当前运行的程序使用。那些被释放的空间可能来自一些很长时间没有什么操作的程序，这些被释放的空间被临时保存到Swap空间中，等到那些程序要运行时，再从Swap中恢复保存的数据到内存中。这样，系统总是在物理内存不够时，才进行Swap交换。其实，Swap的调整对Linux服务器，特别是Web服务器的性能至关重要。通过调整Swap，有时可以越过系统性能瓶颈，节省系统升级费用。

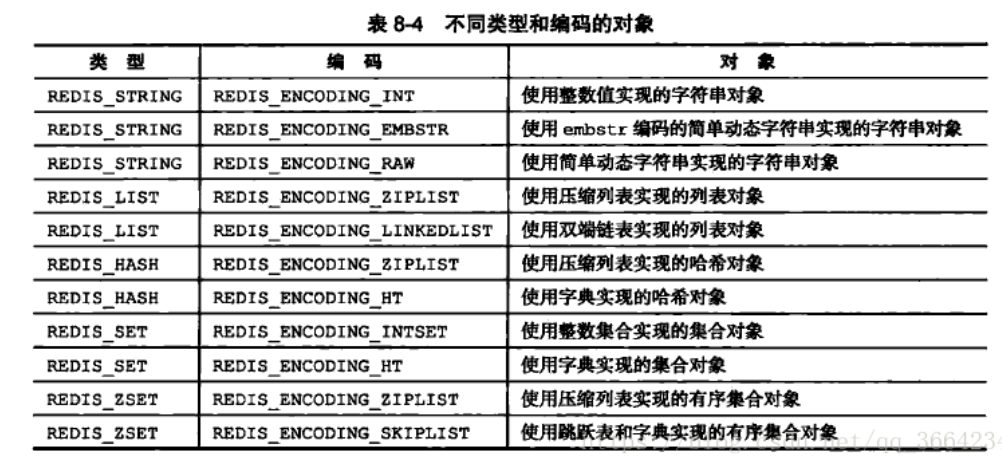
# Redis

1. 五种数据类型：String、list、hash、set、sorted set
2. 底层数据结构：

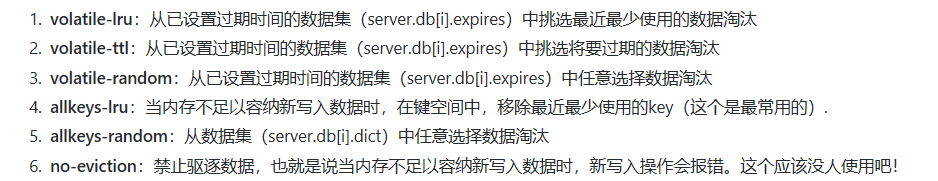
每一种数据类型在底层都是一个封装好的对象如下所示：







1. 两种持久化方式：RDB、AOF
2. 主从模式、哨兵模式
3. 事务：MULTI、EXEC、DISCARD、WATHC、UNWATCH
4. 分布式锁的redis实现：使用SETNX（set if not exist），只有key不存在的时候才会进行set操作，value值通常为一个UUID，可以用于获取一个锁，一定记得设置一个过期时间，以免占有锁的客户端突然崩掉，然后使用DEL命令释放这个锁，在删除的时候通过UUID判断是不是该锁，需要设置锁的那个线程进行删除。
5. Redis与memcached的区别：
   1. Redis支持更丰富的数据类型，memcached只支持简单的数据类型，String
   2. Redis支持数据的持久化，memcached中的数据全在内存中
   3. Redis有原生的集群模式，即主从模式等
   4. Memcached是多线程非阻塞IO服用；Redis是单线程多路IO复用
6. Redis的内存淘汰策略：



1. 缓存雪崩：缓存同一时间大面积失效，后续请求落到了数据库上，导致数据库在短时间内承受大量请求而崩掉。解决策略：
   1. 事前：保证缓存服务高可用性，例如使用集群
   2. 事中：对缓存访问进行资源隔离（熔断）;使用ehcache本地缓存；使用限流组件hystrix
   3. 事后：redis持久化机制快速恢复数据
2. 缓存穿透：大量请求不在缓存中的数据，导致大量请求落到了数据库上，使数据库崩掉。

解决办法：布隆过滤器，将所有可能的数据hash到一个bitmap中，一定不存在的数据会被拦截；缓存空对象，设置过期时间不超过5分钟。

1. 短时间大量数据插入：创建一个包含插入命令的文件，使用pipe mode模式执行命令

cat date.txt | redis-cli --pipe

1. 缓存更新策略：主要宗旨，给缓存设置过期时间，写数据以数据库为准，对缓存的操作只做最大努力即可，有以下三中策略：
2. 更新数据库，再更新缓存：

不采用，因为存会导致脏数据，A更新数据库-B更新数据库-B更新缓存-A更新缓存

1. 先删除缓存，再更新数据库：

会有不一致性，A删缓存-B查询缓存没有则查询数据库-B以旧值更新缓存-A更新数据库。可以采用双延时策略，删除和更新操作之后，sleep一段时间，再删除缓存。

1. 先更新数据库，再删除缓存：

可能会删除缓存失败，这时候用一个队列解决，自己订阅自己的删除请求，删除失败则把这个请求放进队列里面，不断重试。

# 消息中间件

为什么要使用消息队列：

1. 通过异步处理提高系统性能
2. 降低系统耦合性

使用到过MQTT协议，服务端用的百度云的，客户端用的eclipse paho

一种基于发布/订阅模式的轻量级协议，构建在TCP/IP协议上。

特性：

1. 使用发布/订阅模式，提供一对多的消息发布。
2. 对负载内容屏蔽的消息传输。
3. 使用tcp/ip提供网络连接。
4. 有三种消息发布质量：至多一次、至少一次、只有一次。
5. 小型传输、开销很小。

# Spring

核心组件：bean、context、core

1. AOP：面向切面编程。同通俗的说，将那些封装好的对象剖开，找出其中对多个对象产生影响的公共行为，并将其封装为一个可重用的模块（Aspect），切面将那些与业务无关，却被业务模块共同调用的逻辑封装起来，减少代码重复，降低模块间的耦合度。

相关术语：

连接点（Joinpoint）：可以被拦截到的点（理解为可以被增强的方法）

切入点（Pointcut）：真正被拦截到的点

通知/增强（advice）：对被拦截到的点进行增强的方法

引介（Introduction）：类层面的增强

目标（Target）：被增强的对象

织入（Weaving）：将通知运用到目标的过程

代理（Proxy）：被增强后的代理对象

切面（Aspect）：多个通知和多个切入点的组合

通知类型：

Before：在目标方法被调用之前做增强，可以指定一个JoinPoint类型参数，获得切入点信息。

AfterReturning：在目标正常完成后增强，可以指定一个JoinPoint类型参数，获得切入点信息，可以指定一个Object的形参用于获取返回值，参数名字需与配置中一样。

AfterThrowing：主要用来处理程序中未处理的异常，可以指定一个throwing的形参用于获取抛出的异常

After：在目标完成后增强，不论执行是否成功

Aroud：环绕通知，在目标完成前后都增强，这个方法返回类型必须是Object，有个ProceedingJoinPoint类型参数，用于调用切入点方法。

Aop的使用：

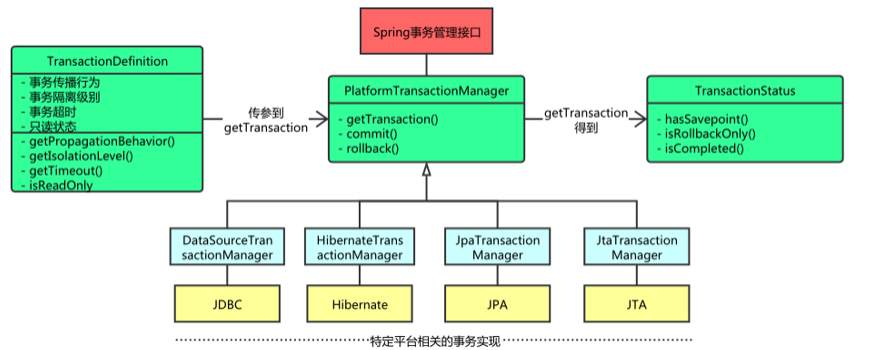
1. 基于AspectJ的XML方式：
2. 引入jar包，4个
3. 编写一个切面类
4. 将切面类交给spring
5. 在xml中配置aop：
   1. 使用aop：config标签
   2. 配置切点aop：pointcut，注意execution的使用（返回值 包名.类名.方法名（参数），\*代表任意，参数为..代表任意参数）
   3. 配置切面aop：aspect，里面有子标签用于选择通知类型
6. 基于AspectJ的注解方式：
7. 引入jar包，4个
8. 编写一个切面类
9. 将切面类交给spring，并在配置文件中，开启注解开发<aop:aspectj-autoproxy>
10. 在切面类上使用注解@Aspect，在类中的各个方法中使用@Before，@After等注解

动态代理：

1. JDK：
2. 定义一个接口
3. 被代理对象必须实现这个接口
4. 代理对象实现InvocationHandler接口，然后重写invoke方法
5. Cglib（流程如下图）
6. 被代理类不用强制实现接口
7. 代理类实现MethodInterceptor接口，重写interceptor方法。
8. 使用的时候创建一个Ehancer对象，代理对象实际上是被代理对象的子类



1. 事务管理：



Spring并不直接管理事务，而是提供了事务管理器接口PlatformTransactionManager，通过这个接口为各个平台提供对应的事务管理器，将事务的管理职责委托给持久层框架的事务来实现。

PlatformTransactionManager通过getTransaction()获取事务，获取到的时候包含事务的五个属性：传播方式、隔离界别、是否只读、事务超时、回滚规则。

事务的传播行为：主要是为了解决业务层方法互相调用的问题。有7种传播行为，大体分为三类：1.保证多个操作在同一个事务中，常用PROPAGATION\_REQUIRED（默认值，如果A中有事务，就是用A中的事务，如果没有就创建新的事务）；2。保证多个操作不在同一个事务中，常用PROPAGATION\_REQUIRES\_NEW（如果A中有事务，则将A中的事务挂起，创建新的任务包含自身的任务）；3.如果当前事务存在，则嵌套事务执行，PROPAGATION\_NESTED（如果A中有事务，按照A的事务执行，执行完成后，设置保存点，执行B的操作，如果没有异常，则执行通过，如果有异常可选择回到最初值或者博爱存点）。

提供了编程式事务和声明式事务。

1. 编程式事务侵入到了业务代码里面，提供了更加详细的事务管理；
   1. 在xml中配置平台事务管理器
   2. 配置事务管理的模板（简化代码）
   3. 在业务层注入事务管理的模板，编写代码
2. 声明式事务通过配置实现，基于AOP，有XML方式和注解式
   1. XML方式：
      1. 引入AOP相关jar
      2. Xml中配置事务管理器
      3. Xml中配置增强，标签<tx:advice>及其子标签
      4. Xml配置aop
   2. 注解方式：
      1. 引入AOP相关jar
      2. Xml中配置事务管理器
      3. 开启注解事务，<tx:annotation-driven>
      4. 在业务层上添加注解@Transaction
3. IOC：由spring容器控制对象的生命后期和对象间的关系

两个重要接口：

1. BeanFactory：提供了ioc容器最基本的API。
2. ApplicationContext：ioc容器的高级形态，提供了更高级的功能。现在更常用。两个实现类：ClassPathXmlApplicationContenxt和FileSystemXmlApplicationContext

用到的技术：

使用XML配置文件

使用dom4j解析xml文件

使用工厂模式

使用反射创建对象

循环依赖注入（A中注入B，B中注入C，C中注入A）：三种情况

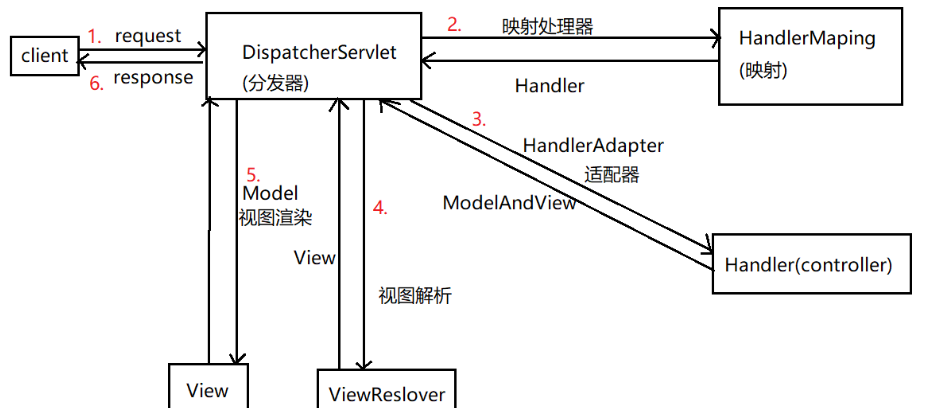
1. 构造器循环依赖：即在构造方法中有相应对象的参数传入，此种依赖没有办法解决，只有抛出BeanCurrentlyInCreationExecption异常。
2. Setter方法循环注入（单例）：使用三级缓存，SingletonFactories、earlySingletonObjects、singletonObject。先调用A的无参构造函数，然后提前曝光到SingletonFactories中，然后A发现B没有creat，然后就执行B的create，B也是同样的流程，然后C也是执行无参构造函数，然后由于A已经在缓存中了（虽然没有构造完成），能够顺利完成创建，并把自己放入缓存，然后B就能从缓存中拿到C对象的引用成功创建，然后再创建A。
3. Setter方法循环注入（非单例）：此种依赖没有办法解决，只有抛出BeanCurrentlyInCreationExecption异常。

依赖注入的四种方式：

1. Set注入
2. 构造函数注入
3. 静态工厂注入
4. 工厂方法注入

注入枚举类型时使用org.springframework.beans.factory.config.FieldRetrievingFactoryBean类将枚举类型进行转换

# SpringMVC



数据校验：

使用Validator接口对该类实体类尽心校验，需要提供该接口的实现类，并实现supports和validate方法，supports方法用于判断当前的Validator实现类是否支持校验当前需要检验的实体类，只有当supports方法的返回结果为true是，该validator接口实现类的validate方法才会被调用对当前校验的实体类进行校验。

# SpringCloud

Spring Cloud是一系列框架的有序集合。它利用Spring Boot的开发便利性巧妙地简化了分布式系统基础设施的开发，如服务发现注册、配置中心、消息总线、负载均衡、断路器、数据监控等，都可以用Spring Boot的开发风格做到一键启动和部署。

核心组件：

1. Spring Cloud Netflix
2. Eureka：服务中心，云端服务发现，一个基于 REST 的服务，用于定位服务，以实现云端中间层服务发现和故障转移。Eureka 采用了 C-S 的设计架构。Eureka Server 作为服务注册功能的服务器，它是服务注册中心。而系统中的其他微服务，使用 Eureka 的客户端连接到 Eureka Server，并维持心跳连接。这样系统的维护人员就可以通过 Eureka Server 来监控系统中各个微服务是否正常运行。Spring Cloud 的一些其他模块（比如Zuul）就可以通过 Eureka Server 来发现系统中的其他微服务，并执行相关的逻辑。
3. Hystrix：熔断器，容错管理工具，旨在通过熔断机制控制服务和第三方库的节点,从而对延迟和故障提供更强大的容错能力。特性：断路器机制，请求后端服务失败数量超过一定比例（默认50%），断路器会切换到开路状态，这时候所用请求会直接失败，在保持断路状态一段时间后（默认5秒），自动切换到半自动状态，会判断下一次请求，如果成功则切回闭路状态，否则继续回到开路；Fallback，相当于降级操作，对于查询操作，当请求后端服务失败后，可以返回一个默认值或来自缓存的值；资源隔离，主要是通过线程池实现资源隔离。Hystrix-dashboard是一款针对Hystrix进行实时监控的工具，通过Hystrix Dashboard我们可以在直观地看到各Hystrix Command的请求响应时间, 请求成功率等数据。但是只使用Hystrix Dashboard的话, 你只能看到单个应用内的服务信息, 这明显不够. 我们需要一个工具能让我们汇总系统内多个服务的数据并显示到Hystrix Dashboard上, 这个工具就是Turbine.
4. Zuul：是在云平台上提供动态路由,监控,弹性,安全等边缘服务的框架。Zuul 相当于是设备和 Netflix 流应用的 Web 网站后端所有请求的前门。Filter是Zuul的核心，用来实现对外服务的控制。Zuul同样提供路由熔断。
5. Feign

Feign是一个声明式的伪Http客户端，它使得写Http客户端变得更简单。使用Feign，只需要创建一个接口并注解。它具有可插拔的注解特性，可使用Feign 注解和JAX-RS注解。Feign支持可插拔的编码器和解码器。Feign默认集成了Ribbon，并和Eureka结合，默认实现了负载均衡的效果。

1. Ribbon

ribbon是一个负载均衡客户端，可以很好的控制htt和tcp的一些行为。

1. Spring cloud config
2. 俗称的配置中心，配置管理工具包，让你可以把配置放到远程服务器，集中化管理集群配置，目前支持本地存储、Git以及Subversion。Spring cloud config包含了Client和Server两个部分，server提供配置文件的存储、以接口的形式将配置文件的内容提供出去，client通过接口获取数据、并依据此数据初始化自己的应用。
3. Spring Cloud Bus

事件、消息总线，用于在集群（例如，配置变化事件）中传播状态变化，可与Spring Cloud Config联合实现热部署。其实本质是利用了MQ的广播机制在分布式的系统中传播消息，目前常用的有Kafka和RabbitMQ。

1. Spring Cloud Cluster

Spring Cloud Cluster将取代Spring Integration。提供在分布式系统中的集群所需要的基础功能支持，如：选举、集群的状态一致性、全局锁、tokens等常见状态模式的抽象和实现。

1. Spring Cloud Consul

Consul 是一个支持多数据中心分布式高可用的服务发现和配置共享的服务软件,由 HashiCorp 公司用 Go 语言开发, 基于 Mozilla Public License 2.0 的协议进行开源. Consul 支持健康检查,并允许 HTTP 和 DNS 协议调用 API 存储键值对。Spring Cloud Consul 封装了Consul操作，consul是一个服务发现与配置工具，与Docker容器可以无缝集成。

# JWT(JSON Web Token)

JWT是为了网络应用环境间传递声明而执行的一种基于JSON的开发标准，该token被设计为紧凑且安全的，特别适用于分布式站点的单点登陆（SSO）场景。JWT的声明一般被用来在身份提供者和服务者间传递被认证的用户身份信息，以便于从资源服务器中获取资源，可以增加一些额外的其他业务逻辑所必须的声明信息，该token也可直接被用于认证，也可被加密。

区别与传统的基于session的认证方式。传统session认证方式，要求服务器集群session数据共享，每台服务器都能读取session，不容易扩展。

JWT的流程：

1. 用户使用用户名密码请求服务器
2. 服务器验证用户信息
3. 服务器通过验证，发送给用户一个token
4. 客户端存储token，并在每次请求时附加这个token值
5. 服务器验证token，并返回数据。

JWT由三部分构成（用‘.’分隔）：

1. 头部：声明类型，声明加密算法
2. 载荷：存放有效信息的地方，包括标准中注册的声明、公共的声明、私有的声明
3. 签证：header、payload、secret

优点：

1. JSON跨语言的特性
2. 有载荷部分，所以可以存放业务逻辑有关的非敏感信息
3. 便于传输
4. 不需要在服务端保存会话信息，易于应用的扩展