# 无名

## Http

附录 https://www.cnblogs.com/rickiyang/p/13138574.html

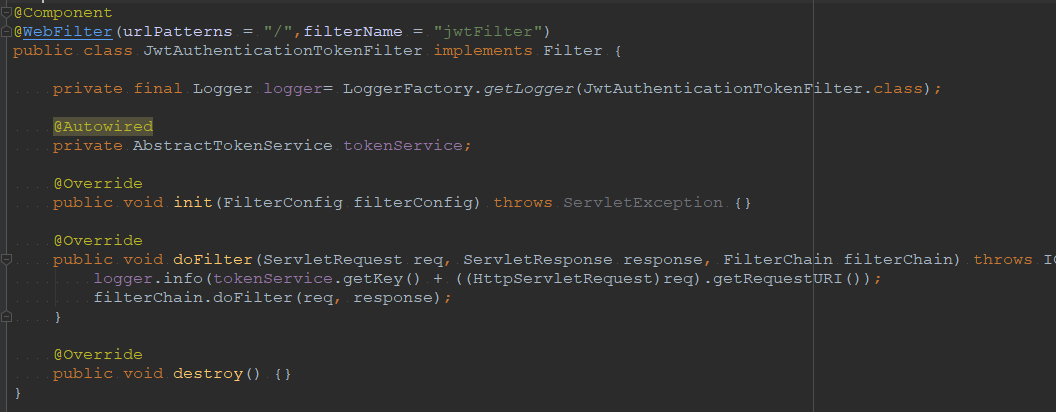
## Spring Boot

### 过滤器和拦截器

#### 过滤器

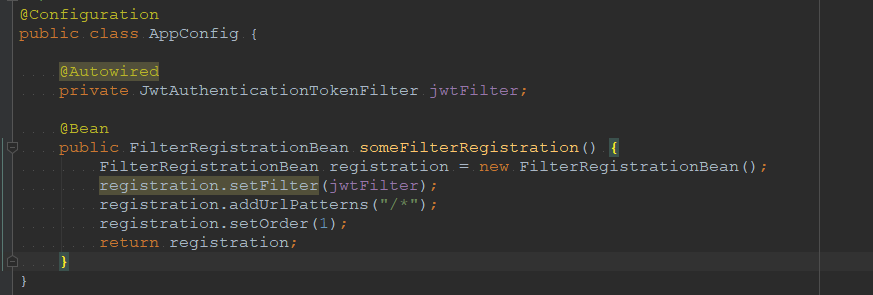
##### 方法1:

1. 实现Filter接口
2. 添加@WebFilter注解, 启动类使用@ServletComponentScan扫描@WebFilter、@WebServlet、@WebListener



##### 方法2:

1. 实现Filter接口
2. 注册该Filter (不使用@WebFilter注解和@Component注解)

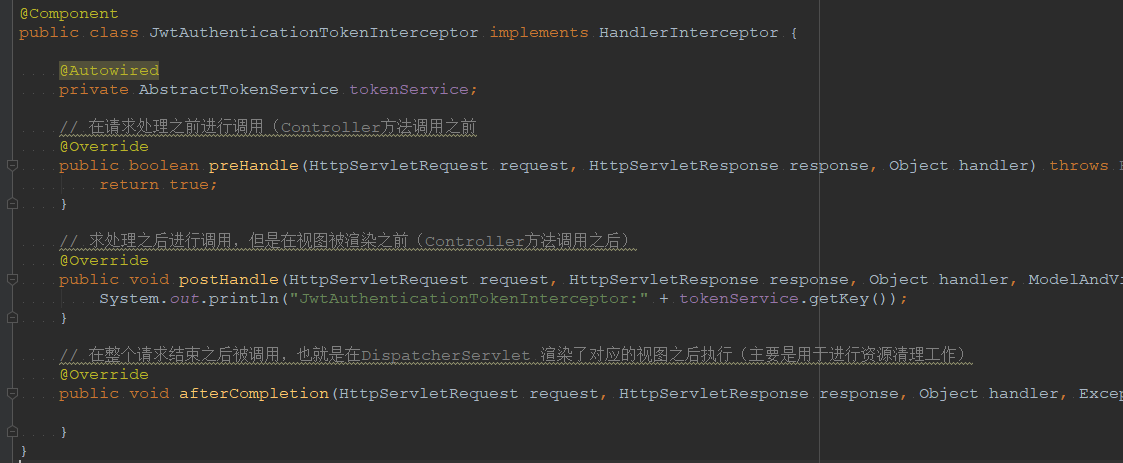


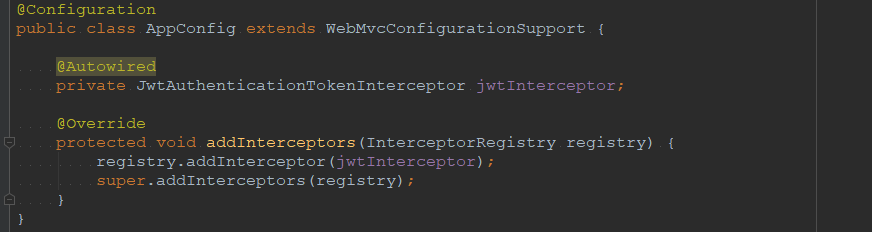
##### 方法3:

1. 实现Filter接口
2. 使用@Component注解即可, 默认过来/\* . 可以和@WebFilter一起使用. 此时也可以使用Ioc容器中的bean

#### 拦截器

1. 实现接口
2. 进行注册 (必须要注册了)





#### 区别

## Spring boot Security

Authentication: 鉴权

Authorization: 授权

UserDetails

Security中的用户信息.

UserDetailsService接口

UserDetailsServiceManager接口

InMemoryUserDetailsManager和JdbcUserDetailsManager实现类

用来操作UserDetails的。

WebSecurityConfigurerAdapter

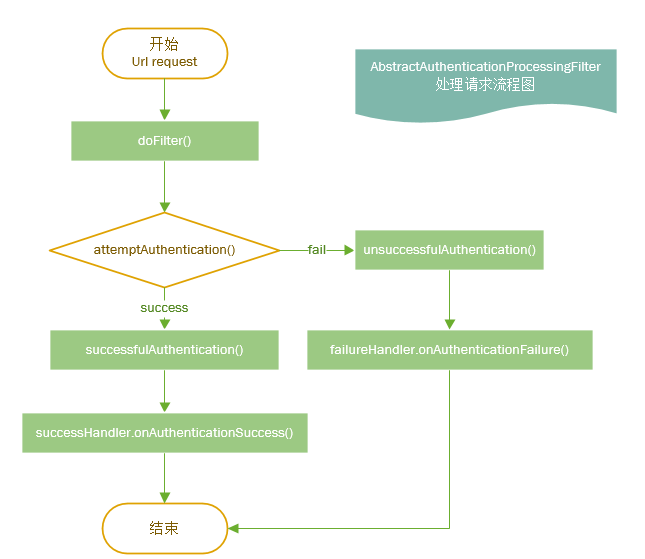
用来配置授权的

AbstractAuthenticationProcessingFilter

是处理 form 登陆的过滤器, 与 form 登陆有关的所有操作都是在该类及其子类中进行的。UsernamePasswordAuthenticationFilter就是它的一个子类.

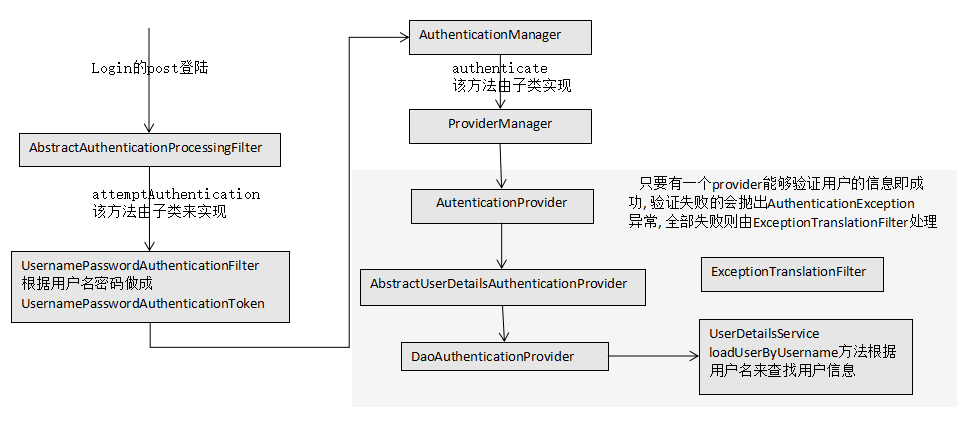
AuthenticationSuccessHandler为授权成功的处理类。

AuthenticationFailureHandler为授权失败的处理类。

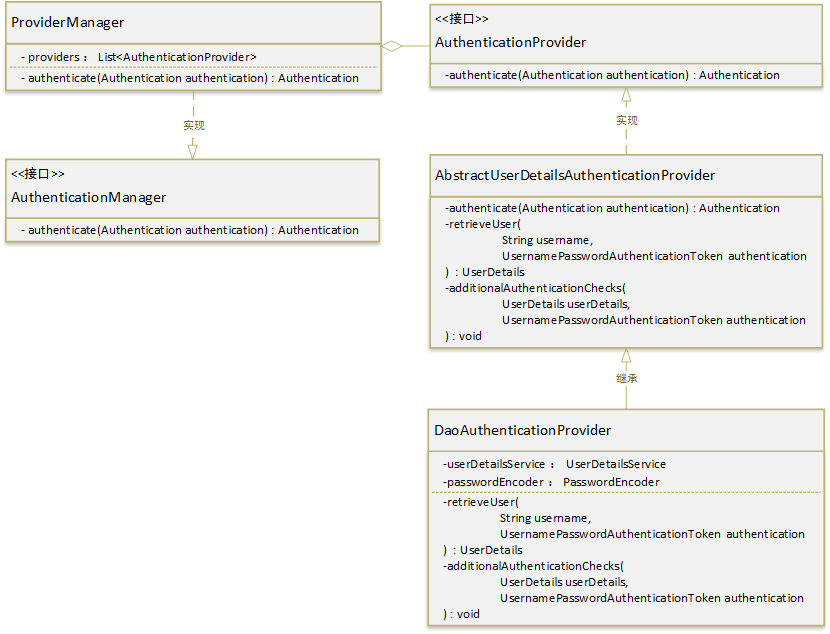
附件: <https://www.jianshu.com/nb/2455141>

https://blog.csdn.net/yuanlaijike/category\_9283872.html

认证的流程



类之间的关系



## MySql

### 字符编码

Unicode(UCS): 字符编码的一种方法(规约) , 由国际组织设计。

utf8: 与ISO兼容, UTF(UCS Transformation Format)

ucs2: 就是用两个字节编码

\* 字符带ci的 例如utf8\_general\_ci 即为忽略大小写的意思。

#### 默认编码

Mysql数据库的默认编码是latin1(ISO-8859-1的别名)

查看数据库的字符编码:

show variables like "char%"

#### 设置编码

1 通过修改配置文件my.ini(windows下)或者/etc/my.cnf(linux下) 具体百度

2 创建数据库时设置编码

create database 数据库名 character set utf8;

3 创建表时设置编码

CREATE TABLE assign\_user (

yyuid int UNSIGNED NOT NULL COMMENT '用户的yyuid',

PRIMARY KEY (yyuid)

) ENGINE=InnoDB DEFAULT CHARSET=utf8;

4 设置字段编码

CREATE TABLE NewTable (

id varchar(255) CHARACTER SET utf8 NOT NULL,

PRIMARY KEY (id)

);

所以varchar(32) 这里的32是指的字符, 不是字节

### SQL执行顺序

FROM - JOIN - ON - WHERE - GROUP BY - HAVING - SELECT - DISTINCT - ORDER BY - LIMIT

#### 联合查询

对于多个表的联合查询并不是一定会按照固定的顺序去执行的, 而是会做一些优化, 以交叉处理的方式得到结果集.

具体: https://blog.csdn.net/qq\_27529917/article/details/78447882

### 常用SQL

#### 1.连接数据库

#### 2.存在插入不存在修改

mysql -hHost –u用户名 –p密码 –P端口号

#### 3.批量导出|导入

insert into table (字段1, 字段2) values(值1, 值2) on DUPLICATE key update 字段2 = values(字段2);

Mysql[dump] -hHost -u用户名 -p密码 -P端口 数据库名 [-d(仅导出表结构)] [表名] [>|<] 文件路径

#### 4.常用语句

DROP TABLE IF EXISTS B;

CREATE TABLE B LIKE A;

INSERT INTO B SELECT \* FROM A;

#### 5.日期格式

DATE\_SUB(curdate(), INTERVAL -1 DAY)

DATE\_FORMAT(now(), '%Y-%m-%d')

SELECT DATE\_FORMAT(now(), '%H-%i-%s')

默认值 CURRENT\_TIMESTAMP 一个表中不要出现两次

#### 插入查询的

INSERT INTO 表1(字段1, 字段2, 字段3) SELECT 字段1, 字段2, 字段3 FROM 表2 WHERE ...

#### 当前的链接信息

Show [full] processlist; 默认只显示100个

#### Count函数的使用

Count(\*) 包含了所有的列, 相当于行数在统计结果时不会忽略NULL

Count(1) 会统计表中的所有的记录数, 不会忽略NULL, 包含字段为null的记录

count(列名)会统计该列字段在表中出现的次数, 会忽略字段为null的情况, 即不统计字段为null的记录

若列名为主键, count(列名)会比count(1)快

若列名不为主键, count(1)会比count(列名)快

若表多个列并且没有主键, 则 count(1) 的执行效率优于 count(\*)

若表有主键, 则 select count(主键)的执行效率是最优的

若表只有一个字段, 则 select count(\*)最优

实际业务中一般用count(1)比较普遍, 但是如果需要聚合多个列, 则用count(列名)比较合适

### 索引

1. 一个表最多有16个索引, 最大索引长度是256个字符(可以在编译mysql被改变)
2. 对于CHAR和VARCHAR的列可以使用前置索引, BLOB和TEXT必须使用列的前缀
3. 组合索引最多由15个列组成
4. 索引名一般以 idx\_ 作为开头
5. Msql通常找最少行索引或者唯一行最多的索引(当同时多个索引可用时)

#### 查看一个表的索引

Show Index From 表名;

#### 结构

1. tree, R-tree, hash, full-text
2. Tree: myisam, bdb, innodb, ndb, archive仅支持geometry数据类型。

B-tree: myisam, innodb

Hash 适用于随机访问的场合, 查询每条数据时间几乎相同, 适用于, =, in, <=>, 不适用于范围查询, 不能排序。

Full-text: 仅archive支持, 只能用于char, char, text用来代替% \* 效率低问题。

附录: https://www.cnblogs.com/lijc1990/archive/2013/06/20/3146378.html

#### 索引表介绍

|  |  |
| --- | --- |
| **列名** | **描述** |
| Table | 表名 |
| Non\_unique | 0 索引不能重复 1 索引可以重复 |
| Key\_name | 索引的名称, 名称相同说明是同一个索引(联合索引) |
| Seq\_in\_index | 索引中的序列号, 一般用于知道在联合索引中的位置 |
| Column\_name | 索引的列名 |
| Collation | 排序规则 A 升序 Null无分类 |
| Cardinality | 表示索引中唯一值得数目的估计值(重复值越少该值越大), 值越大联合查询时使用该索引机会越大, 值小要考虑是否值得创建该索引。可以统计更新(百度)。 |
| Sub\_part | 前置索引,长数据类型会使用索引前缀(列被编入索引的数量), 全列通常是NULL |
| Packed | 关键字如何被压缩 没有压缩NULL |
| Null | 用于显示索引列中是否包含 NULL 若列含有 NULL, 该列的值为 YES, 若没有，则该列的值为 NO |
| Index\_type | 显示索引使用的类型和方法(BTREE、FULLTEXT、HASH、RTREE, SPATIAL) |
| Comment | 显示评注 |
| Index\_comment |  |

附录：

Index\_type(可以在创建索引时进行设置, 默认BTREE):

> BTREE：适合连续读取数据

> RTREE：适合根据一条数据找附近数据

> HASH：适合随机读取数据

> FULLTEXT

> SPATIAL

#### 索引的类型

聚集索引: 物理上连续, 一个表只能有一个, 一般是主键索引。 如果没有主键, 则会自动创建一个隐藏列, 索引的逻辑存储顺序决定了该行物理上的存储顺序。 聚集索引, 索引的叶子节点就是数据节点。

非聚集索引： 除了聚集索引的其它索引。 逻辑上连续, 物理上不连续。即不影响物理存储顺序且可以有多个。索引的叶子节点仍然是叶子节点, 它有一个指针指向对应的数据块。

--->

Index: 基本索引, 没有任何限制

Unique Index: 唯一索引, 索引的列必须唯一, 允许空值, 如果是组合索引, 则列的组合必须唯一

PRIMARY KEY: 主键索引(聚集索引), 特殊的唯一索引, 不允许有空值, 主键索引只允许有一个(创建表时创建, ALTER与其他不一样)

组合索引: 多个字段上创建索引, 查询中使用创建时索引的第一个字段, 索引才会被使用。 组合索引遵循最左前缀集合

FULLTEXT INDEX: 全文索引, 用来查询文本中的关键词, 不直接与索引中的关键词相比较。配合match against使用

#### 索引重复(冗余)

同一个索引类型, 如果有两个或者以上的索引包含了相同的索引信息, 那么存在索引冗余。(不同的索引类型, 索引值不同)

#### 添加 删除一个表的索引

CREATE INDEX 索引名 ON 表名(列名...)

ALTER TABLE 表名 ADD|Drop (PRIMARY KEY | UNIQUE | INDEX | FULLTEXT) 索引名 (列名...)

#### 索引优化

1. Where子句中出现的列, join子句中出现的列
2. LIKE 'mich%' 会用到索引, '%like' 不会用到索引
3. Where的多个and中, 必须都是一个多列索引的key\_part且必须包含key\_part1。如果都是各自的单一索引的话, 只使用遍历最少行的索引。
4. 在where中多个or的条件, 每一个条件都必须是一个有效索引。
5. Order by后面的条件必须都是同一索引的属性, 并且排序必须一致。
6. 所有GROUP BY列引用同一索引的属性, 并且索引必须是按顺序保存其关键字的。
7. 对智能的扫描全表使用FORCE INDEX告知MySQL, 使用索引效率更高。
8. 定期ANALYZE TABLE tbl\_name为扫描的表更新关键字分布
9. 定期使用慢日志检查语句，执行explain，分析可能改进的索引
10. 条件允许的话，设置较大的key\_buffer\_size和query\_cache\_size的值(全局参数)，和sort\_buffer\_size的值(session变量，建议不要超过4M)

#### (InnoDB)锁

##### 悲观锁

很悲观, 每次拿取数据时都认为有别的线程会对数据进行修改, 所以会给数据上锁. 核心就是不支持多并发, 是单线程操作. 通过抢占时间片的方式来抢锁的使用权, 把并发变成了串行.

适用于多写的场景. 保证线程安全和数据安全.

mysql的行锁, 表锁, 读锁, 写锁. java中的synchronized.

悲观锁是mysql自己实现了, 可以通过相关语法使用或者进行显示声明.

##### 乐观锁

很乐观, 拿数据的时候不会人为有其它线程对数据进行修改. 核心是支持多并发, 每个线程在不同的时间节点对数据做更新操作, 每次更新时候都会判断其他线程是否对数据做了更新.

1. version版本号机制

数据表中添加数据版本号, 更新数据后会使其版本号+1, A线程进行更新数据时, 读取数据时读取到版本号, 更新时会比较版本信息是否一致, 否则从新执行整个更新操作.

1. CAS算法机制

判断当前的内存值和之前取到的值是否相等, 若相等则用新值更新, 若不相等则重试, 即不断重试(自旋操作).

适用于多读的操作, 加大数据的吞吐量. 需要自己去实现.

##### 悲观锁·共享锁

一把锁有很多个钥匙.

##### 悲观锁·排它锁

同一个资源只能有一把锁.

##### 行锁

给一条记录加锁.

##### 表锁

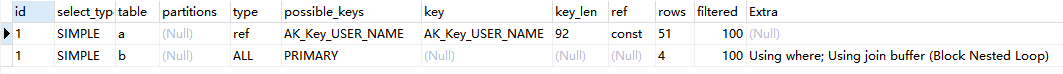
给一个表加锁.

#### 其它

1. 字段需要设置默认值
2. 不同表相同属性 应保持一致性
3. SELECT COUNT(\*) 在没有where的语句中执行效率小于SELECT count(col\_name), 但有where它的效率快。

### 执行计划Explain

查询语句前使用explain即可



执行计划表介绍:

|  |  |
| --- | --- |
| **列名** | **描述** |
| id | 选择标识符, sql的执行顺序(相同时由上到下执行, 不同时由大到小执行) |
| select\_type | 查询类型(SIMPLE, PRIMARY, UNION, DEPENDENT UNION, UNION RESULT, SUBQUERY, DEPENDENT SUBQUERY, DERIVED, UNCACHEABLE SUBQUERY) |
| table | 输出结果集的表(表名或者简称) |
| partitions | 匹配的分区 |
| type | 表的访问方式(ALL, index, range, ref, eq\_ref, const, system, null 性能差->好) |
| possible\_keys | 查询时可能使用到的索引 没有是null |
| key | 实际所使用的索引 |
| key\_len | 索引字段的长度(索引的最大可能长度 越短越好) |
| ref | 列与索引的比较 |
| rows | 扫描(估算)出的行数 |
| filtered | 按条件过滤的行的百分比 |
| Extra | 执行情况的描述和说明 (Using where, Using temporary, Using filesort, Using join buffer, Impossible where, Select tables optimized away, No tables used) |

\* 通过收集统计不可能存在的结果, 执行计划是没用的。

**附录:**

Select\_type:

SIMPLE: 简单的select, 不使用 union或子查询等

PRIMARY: 查询中包含子查询, 最外层的select被标记成PRIMARY

UNION: UNION中的第二个或后面的select语句

type:

all: 全表扫描

index: 从索引中全部扫描(索引文件比较小, all是从硬盘读取, 它们都是度全表)

range: 使用一个索引来检索给定范围的行, between < > in

ref: 非唯一索引扫描

eq\_ref: 唯一性索引扫描, 对于每个索引表中只有一行与之匹配

const: 通过索引一次找到, 相当于查一行

system: 表只有一行记录(等于系统表), const类型特例

Extra:

Using Index: 不用读取表中所有信息，仅通过索引就可以获取所需数据, 索引用于读取

Using Where: 不用读取表中所有信息，仅通过索引就可以获取所需数据, 索引用于查询

Using temporary: 使用临时表来存储结果集, 常见 group by; order by 没使用到索引(要优化)

Using filesort: 包含 order by 操作, 无法利用索引完成的排序操作称为"文件排序"(要优化)

Using join buffer: 表明使用了连接缓存, 比如说在查询的时候, 多表join的次数非常多，那么将配置文件中的缓冲区的join buffer调大一些(要优化)

Impossible where: where子句的值总是false (where语句不正确一般)

Select tables optimized away: 没有group By通过索引进行聚合函数的优化

No tables used: Query语句中使用from dual 或不含任何from子句

Distinct: 优化distinct操作, 找到第一个匹配的后停止寻找同样值得动作

#### 附件

**执行计划Explain详解: <https://blog.csdn.net/why15732625998/article/details/80388236>**

### Bin log

二进制日志, 记录用户对数据库的操作(select除外), 主要用于数据恢复和主从

没有造成数据修改的语句也不会被记录

> 在my.cnf中进行配置, 开启, 日志路径(包含日志文件, 索引文件)等一些配置

#### 三种模式

##### Row

日志中记录每一行数据被修改后的形式, 只有value不会有sql多表关联。Slave端根据数据进行相同的修改。

优点: 记录详细明确 缺点: 产生大量的日志, 并且也不知道具体做了什么操作

##### Statement

每一条会修改数据的sql都会被记录在master的binlog中, salve复制的时候sql进程会解析成和原来master端执行相同的sql进行执行。

优点: 减少日志量 缺点: 需要上下文信息, 某些特定函数功能导致复制出错

##### Mixed

row和statement的结合。

### Redo log

Redo log是基于innoDB引擎的物理修改记录, 只能回滚到最后一次事务提交的节点上.并且它的记录是在bin log之后的. 主要用于事务回滚。

### 事务

> 事务是数据库管理系统执行过程中的一个逻辑单位(把多件事情当一件事情做)

> 数据库底层的事务和回滚是通过bin log(归档日志)和redo log(重做日志)来实现的

> mysql默认的事务隔离级别是REPEATABLE-READ(可重复读)

> 基本属性(ACID)

原子性(Atomicity): 要么全部成功, 要么全失败

一致性(Consistency): 事务必须是数据库从一个一致性状态变到另一个一致性状态

隔离性(isolation): 一个事务内部的操作及使用的数据对并发的其他事务是隔离的

持久性(Durability): 事务结束后 对数据的改变时永久性的

> InnoDB支持事务 MyISAM不支持事务 事务支撑是在引擎层上实现的

#### 简单操作

SELECT @@tx\_isolation # 查询当前事务的隔离级别

set session transaction isolation level read uncommitted #设置事务的隔离级别

start transaction #开启事务

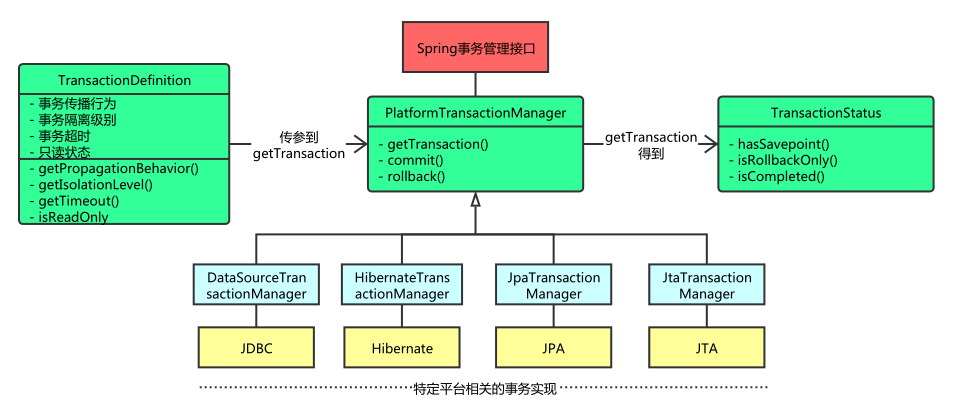
Rollback #回滚 commit #提交

#### 事务的隔离级别

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **事务** | **脏读** | **不可重复读** | **幻读** | **描述** |
| read uncommitted (未提交读) | √ | √ | √ |  |
| read committed (提交读) | × | √ | √ | 事务提交后才能被读取 |
| repeatable read( 可重复读) | × | × | √ | 事务执行期间 多次读取同一数据返回第一次快照 |
| serialization (可串行化) | × | × | × | 写加写锁 读加读锁, 读写冲突需前一个事务完成 |
| **脏读** | 事务2使用事务1未提交的数据 | | | |
| **不可重复读** | 事务1读取数据(未提交), 事务2对此数据进行修改提交, 事务1再次读取该数据 | | | |
| **幻读** | 事务1读取数据的条数, 事务2插入一条数据到该范围, 事务1再次读取数据的条数 | | | |

附件: <https://blog.csdn.net/qq_35433593/article/details/86094028>

#### Spring Boot中事务的实现



spring不直接管理事务, 而是提供了一个PlaformTransactionManager接口, 由各平台来实现事务管理。

##### 事务的属性定义

事务的隔离级别: 事务执行期间可否被其它事务访问的配置

事务超时: 事务必须在规定时间内完成, 否则回滚

回滚规则: 指定那种情况下进行回滚, 默认是RuntimeException和Error, 可指定异常

事务是否只读: 确保读取多条数据的一致性(确保数据整体的一致性, 从开始时间a开始, a之后被提交的数据不可见)

事务的传播行为: 当一个事务方法被另方法调用时的传播行为。

方法A调用事务方法B, B是否继续在A的事务中执行, 取决于B事务的传播行为。

|  |  |
| --- | --- |
| **事务的传播行为** | **方法A 调用 事务方法B 时 事务的传播行为** |
| required | 继续使用A的事务, 若A没有事务, 则B自己创建一个 |
| supports | 继续使用A的事务, 若A没有事务, 则B也不使用事务 |
| mandatory | A必须有事务, B使用A的事务, 若A没有则异常 |
| required\_new | B总使用自己的事, A(事务方法时)和B是两个独立的事务, A挂起, B完成自己的事务, A继续 |
| not\_supported | A无论有没有事务 B都不会有事务 即B的运行是独立的 |
| never | A不能时事务方法, 否则抛异常 且B本身也不有事务行为 |
| nested | B总使用自己的事务, A(事务方法时)的事务嵌套B的事务, B随A的提交提交, 回滚而回滚 |

##### 使用事务

1. 启动类上添加@EnableTransactionManagement来开启事务(可通过其它配置使用不同的事务管理器)
2. 在service方法上添加@Transactional来使用事务(指定事务的属性)

该注解以下条件可能会失效 即不回滚

> 用于非public方法上, aop反射调用的时候会检查是不是public修饰的

> propagation的配置从而导致事务不会发生回滚

> 异常未捕获 跑出其它异常导致未回滚

> **同一个类中方法调用(A调用B A方法没有事务 B有事务 则B方法的事务失效)**

> 引擎不支持(很少有不支持的)

\* dbop中有demo可参照

### MVCC

#### 简介

Mutil-Version Concurrenty Control多版本并发控制, 一般用在数据库管理系统中, 实现对数据库的并发访问.

InnoDb引擎的RC, RR这两个隔离级别使用了MVCC(乐观锁的实现)

#### MVCC的使用

##### 版本链

索引表中聚集索引会有两个隐藏的列, trx\_id事务id, roll\_pointer上一个版本记录地址.

|  |  |  |  |
| --- | --- | --- | --- |
| **id** | **data** | **trx\_id** | **Roll\_point** |
| 主键索引(聚集索引) | 数据信息 | 事务ID | 上一个版本地址 |

##### RC RR中的使用

当进行读取的时候会取得一个ReadView 该ReadView中记录着未提交事务的版本连信息

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **操作** | **id** | **data** | **trx\_id** | **Roll\_point** |
| 修改（未提交） | 1 | 小明3 | 40 | 上一个版本地址  上一个版本地址 |
| 修改（未提交） | 1 | 小明2 | 30 | 上一个版本地址  上一个版本地址 |
| 修改（已提交） | 1 | 小明1 | 20 | 上一个版本地址  上一个版本地址 |
| 添加（已提交） | 1 | 小明 | 10 | 上一个版本地址 |

###### RC中

1. 取数据时, 会取得当前的ReadView

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 修改（未提交） | 1 | 小明2 | 30 | 上一个版本地址  上一个版本地址 |
| 修改（未提交） | 1 | 小明1 | 20 | 上一个版本地址  上一个版本地址 |

1. 当前访问事务id如果是40则无法访问, 如果是10不存在版本链中, 则可以访问, 如果是20存在版本链中不可访问, 如果是25不存在版本链中, 可以访问。
2. 读取数据时会从当前事务id向下取, 进行对比, 取到一个可以访问的版本的数据信息。

###### RR中

1. 第一次取得的readview是

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 修改（未提交） | 1 | 小明2 | 30 | 上一个版本地址  上一个版本地址 |
| 修改（已提交） | 1 | 小明1 | 20 | 上一个版本地址  上一个版本地址 |

1. 第二次读取数据时仍会使用开始取得的readview, 而RC中第二次读取数据时可能取到一个新的readview

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 修改（未提交） | 1 | 小明3 | 40 | 上一个版本地址  上一个版本地址 |
| 修改（已提交） | 1 | 小明2 | 30 | 上一个版本地址  上一个版本地址 |
| 修改（已提交） | 1 | 小明1 | 20 | 上一个版本地址  上一个版本地址 |

所以对于RC来说第一次读的是事务id是20的版本数据, 第二次读取的是事务为30的版本数据. RR会使用一个readview所以不存在不可重复读的问题.

## Redis

### Spring Boot中应用redis

#### 常用的客户端

|  |  |  |  |
| --- | --- | --- | --- |
| 名称 | 原理 | 线程安全 | 其它 |
| Jedis | 与redis server直连 | 否 | 和redis指令一一对应, 同步阻塞, 不支持异步 |
| Letture | 基于Netty事件驱动通讯 | 是 | 适用分布式缓存 springboot2.0默认使用 连接池需要添加commons-pool2依赖 |
| Redisson | 基于Netty事件驱动通讯 |  |  |

RedisTemplate默认使用的是JdkSerializationRedisSerializer，StringRedisTemplate默认使用的是StringRedisSerializer

GenericToStringSerializer、Jackson2JsonRedisSerializer、JacksonJsonRedisSerializer、JdkSerializationRedisSerializer、OxmSerializer、StringRedisSerializer。

Spring boor 缓存

https://www.jianshu.com/p/e9b40acb2993

## Java

### 基础

#### 数组的copy

System.arraycopy： 指定数组, 指定副本进行数据copy. 底层使用c语言

Arrays.copy： 里面使用的是System.arraycopy, 它返回一个新的数组. arraycopy没有返回值, 必须传入副本.

#### 不常用关键字

**Const**: 关键字保留, 但是不允许使用.

**Native**: 使用native关键字说明这个方法是原生函数, 也就是这个方法是用C/C++语言实现的, 并且被编译成了DLL, 由java去调用.

**Strictfp**: 修饰类, 接口, 方法. 修饰的作用域内所有浮点运算都是精确的, 符合IEEE-754的规范.

**Volatitle**: 使变量具有可见性, volatile修饰的变量不允许线程内部缓存和重排序，即直接修改内存。

**Transient**: 被修饰的关键字不可被序列化. 原理: 生命周期只在内存中. 静态变量不会被序列化. 瞬态变量也不可以.

Serializable: 实现该接口 添加uuid.

Exteranlizable: 实现该接口的writeExternal和readExternal效率高于Serializable可以决定哪些属性需要序列化(即使是transient修饰的), 但是对大量对象, 或者重复对象, 则效率低.

**Goto**: 关键字保留, 但是不允许使用.

#### 变量

**整形:** 0123 八进制 0xadf 十六进制 0b1010 二进制 1\_2\_323和1\_\_2\_\_32视觉分组

**字符:** '/n' '/'' '/"' '/ddd' '/uxxxx'

**整形转换成更小的类型:** 对更小的类型进行取模, 然后取余数。

**浮点变整形:** 对小数点及后面内容进行截断。

**类型提升规则:**

> byte, char, short 作为操作数会被提升为int

> 如果一个操作数是long, float则整个表达式提升为long, float

> 如果任何一个操作数为double则结果为double类型.

#### 其它

**Break:** 通常用来跳出循环, 设计初衷是具有go的作用, 调到标记位, 该标记代码块必须包含break否则编译失败.

**Continue：** 跳出本次循环, 也可以指定标记调到包含continue的标记位.

**可变参数(...):** 在一定的条件下具有模糊性, 编译器也无法理解, 需要自己另辟蹊径.

例: a(int a, int... b)和a(int a)当方法调用是a()或者a(1)的话, 是具有模糊性的。

### 数据结构

#### 阻塞队列

##### 概论

满足两个附加操作的队列即为阻塞队列. 附加操作: 在队列为空时获取元素的线程会等待元素变为非空, 当队列满时, 存储元素的线程会等待队列可用. 阻塞队列常用于生产者消费者模式.

##### 方法

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **方法/处理方式** | **抛出异常** | **返回特殊值** | **一直阻塞** | **超时退出** |
| IllegalStateException/NoSuchElementException | True/False/返回值/Null | 一直阻塞或者响应中断退出 | 超过时间就退出 |
| 插入方法 | add(e) | offer(e) | put(e) | offer(e, timr, unit) |
| 移除方法 | remove() | poll() | take() | poll(time, unit) |
| 检查方法 | element() | peek() | 不可用 | 不可用 |

boolean add(E e) : 将给定元素设置到队列中, 如果设置成功返回true, 否则抛出异常. 如果是往限定了长度的队列中设置值, 推荐使用offer()方法.

boolean offer(E e) : 将给定的元素设置到队列中, 如果设置成功返回true, 否则返回false. e的值不能为空, 否则抛出空指针异常.

void put(E e) throws InterruptedException : 将元素设置到队列中, 如果队列中没有多余的空间, 该方法会一直阻塞, 直到队列中有多余的空间.

boolean offer(E e, long timeout, TimeUnit unit)throws InterruptedException : 将给定元素在给定的时间内设置到队列中, 如果设置成功返回true, 否则返回false.

E take() throws InterruptedException : 从队列中获取值, 如果队列中没有值, 线程会一直阻塞, 直到队列中有值, 并且该方法取得了该值.

E poll(long timeout, TimeUnit unit) throws InterruptedException : 在给定的时间里, 从队列中获取值, 如果没有取到会抛出异常.

int remainingCapacity() : 获取队列中剩余的空间.

boolean remove(Object o) : 从队列中移除指定的值.

public boolean contains(Object o) : 判断队列中是否拥有该值.

int drainTo(Collection<? super E> c) : 将队列中值, 全部移除, 并发设置到给定的集合中.

int drainTo(Collection<? super E> c, int maxElements) : 指定最多数量限制将队列中值, 全部移除, 并发设置到给定的集合中.

##### 家庭成员

|  |  |  |  |
| --- | --- | --- | --- |
| **队列** | **有界性** | **锁** | **数据结构** |
| ArrayBlockingQueue | Bounded(有界) | 加锁 | ArrayList |
| 支持公平锁和非公平锁, 按照FIFO的原则对元素进行排序 | | | |
| LinkedBlockingQueue | Optionally-bounded | 加锁 | linkedList |
| 默认长度为Integer.MAX\_VALUE | | | |
| PriorityBlockingQueue | unbounded | 加锁 | Heap |
| 支持优先级排序, 默认自然序进行排序, 可以实现compareTo()接口, 不能保证同优先级元素的顺序. | | | |
| DelayQueue | unbounded | 加锁 | heap |
| 一个实现PriorityBlockingQueue实现延迟获取的无界队列, 在创建元素时, 可以指定多久才能从队列中获取当前元素. 只有延时期满后才能从队列中获取元素. 例: TimeQueue是使用DelayQueue实现的 | | | |
| SynchronousQueue | bounded | 加锁 | 无 |
| 一个不存储元素的阻塞队列, 每一个put操作必须等待take操作, 否则不能添加元素. 支持公平锁和非公平锁 | | | |
| LinkedTransferQueue | unbounded | 加锁 | Heap |
| 由链表结构组成的无界阻塞队列, 相当于其它队列, LinkedTransferQueue队列多了transfer和tryTransfer方法 | | | |
| LinkedBlockingDeque | unbounded | 无锁 | Heap |
| 由链表结构组成的双向阻塞队列, 队列头部和尾部都可以添加和移除元素, 多线程并发时, 可以将锁的竞争最多降到一半 | | | |

附录: https://www.cnblogs.com/bjxq-cs88/p/9759571.html

### 接口

**default方法**

在接口中被default修饰的方法为普通方法. 只能被接口的实现类的对象调用, 可以被重写.

> 该方法可以被子类继承使用

> 当子类实现两个接口, 且接口中都有default A 那么A必须被重写, 否则编译失败.

> 当子类继承一个类实现一个接口, 父类中有A, 接口中有default A那么实际继承类父类的A方法.

**static方法**

在接口中用static修饰的方法为接口的类方法, 只能被接口调用, 不能被实现类或者实现类的对象调用.

### JVM底层

#### 对象头

在HostSpot VM(Sun JDK和OpenJDK中所带的虚拟机, 也是目前使用范围最广的Java虚拟机)中对象头主要包含两部分, Mark Word(标记字段), Klass Pointer(类型指针)。

其中类型指针时对象指向它的类元数据的指针, 虚拟机通过这个指针来确定这个对象是那个类的实例, 标记字段是存储对象自身的运行时数据。

对象头一般是2个机器码(一个机器码4个字节), 如果是数组类型则需要3个, 其中一个记录数据长度。Mark Word会根据对象的状态来复用自己的存储空间达到空间效率(在极小的空间内存储尽量多的数据)。

下图是32位的, 64位和32位基本类似。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **锁状态** | **25bit** | | **4bit** | **1bit** | **2bit** |
| **23bit** | **2bit** | **是否是偏向锁** | **锁标志位** |
| 无状态锁 | 对象hashcode | | 对象分代年龄 | 0 | 01 |
| 轻量级锁 | 指向锁记录的指针 | | | | 00 |
| 重量级锁 | 指向重量级锁的指针 | | | | 10 |
| GC标记 | 空 | | | | 11 |
| 偏向锁 | 线程ID | Epoch | 对象分代年龄 | 1 | 01 |

#### Jmap 命令

Visualvm用来分析内存使用情况, jdk自带的在bin下执行jvisualvm.exe

代理有问题的解决: https://blog.csdn.net/qq\_23747821/article/details/79684001

jmap -dump:live,format=b,file=myjmapfile.txt 19570

## 并发

### 线程安全

#### 造成线程安全的原因

1. 存在共享数据(也称临界资源: 各进程采取互斥的方式, 实现共享的资源称作临界资源)
2. 存在多个线程共通操作临界资源

### Thread

#### 状态

1. 初始(NEW): 新创建了一个线程对象. 但还没有调用start()方法.

2. 运行(RUNNABLE): Java线程中将就绪(ready)和运行中(running)两种状态笼统的成为"运行".

线程对象创建后. 其他线程(比如main线程)调用了该对象的start()方法. 该状态的线程位于可运行线程池中. 等待被线程调度选中. 获取cpu 的使用权. 此时处于就绪状态(ready). 就绪状态的线程在获得cpu 时间片后变为运行中状态(running).

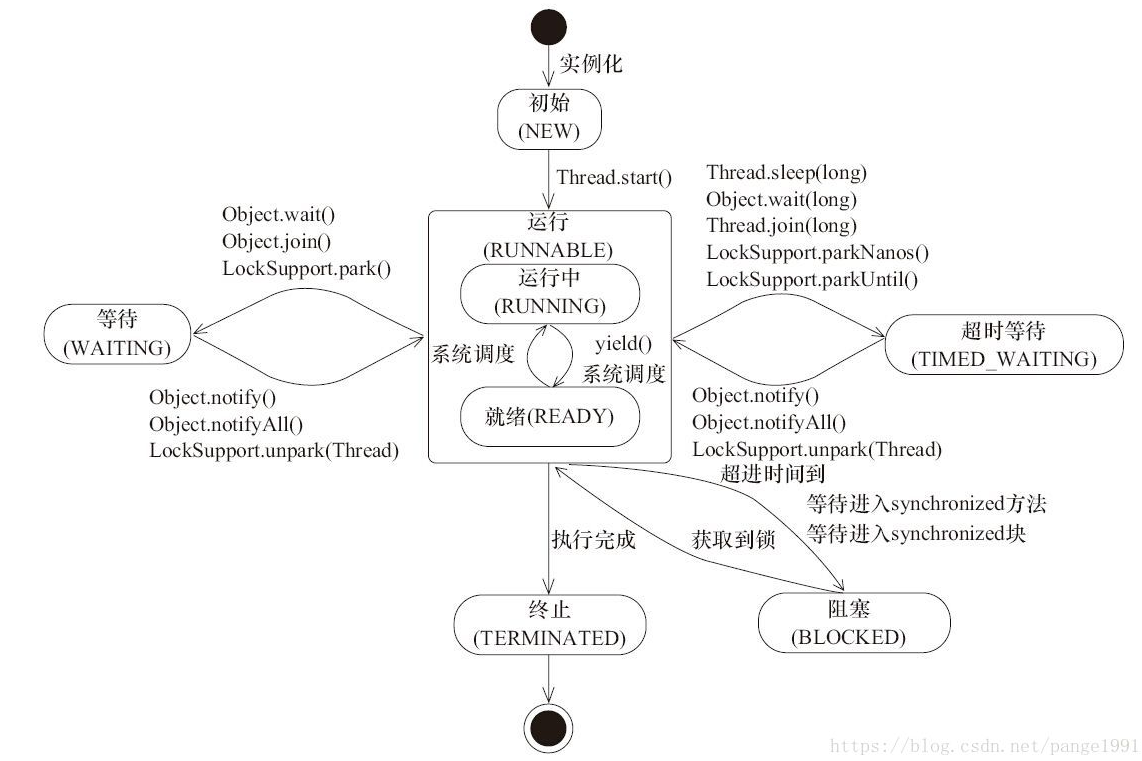
3. 阻塞(BLOCKED): 表线程阻塞于锁.

4. 等待(WAITING): 进入该状态的线程需要等待其他线程做出一些特定动作(通知或中断).

5. 超时等待(TIME\_WAITING): 该状态不同于WAITING. 它可以在指定的时间内自行返回.

6. 终止(TERMINATED): 表示该线程已经执行完毕.

附录: https://www.cnblogs.com/cowboys/p/9315331.html



#### 线程相关方法

interrupt(): Thread的实例方法, 调用后修改该线程标记为中断状态. 该线程仍会继续执行. 若使用的线程处于阻塞状态(wait sleep join), 调用后会唤起线程, 直接抛出InterruptedException异常, 并清除中断状态. 特例: 如果是因为synchronized获取锁时进入阻塞状态, 调用该方法并不能抛出异常, 只是修改状态, 它还会一直等待获取锁.

isInterrupted(): Thread的实例方法, 调用后返回该线程是否是中断状态.

interrupted(): Thread的类方法, 返回该方法所处当前环境的线程的中断状态, 之后清除中断状态.

sleep(millSecond) : thread的类方法, 线程进入Waiting状态, 不释放锁. 给其它线程执行创造了机会.

yield() : thread的类方法, 放弃获取的cpu时间片, 进入就绪状态, 不会导致阻塞.

join() : Thread的实例方法, 当前线程里调用其它线程t的join方法, 当前线程进入WAITING/TIME\_WAITING状态, 当前线程不释放已经持有的对象锁. 线程t执行完毕或者millis时间到, 当前线程进入就绪状态。

**当使用synchronized的时候锁对象会拥有关联的monitor对象, 即此时该锁对象会维护两个集合, Entry Set和Wait Set, 即锁池和等待池. {**

wait() : 拥有锁时由锁对象调用, 当前线程进入WAITING状态, 且进入到wait set中, 需要notify和notifyall唤醒. wait之后会立即放掉锁, 即释放资源.

notify() : 拥有锁时由锁对象调用, 唤醒wait set中的随机一个线程加入到Entry Set中, WAITING->RUNNABLE.

notifyAll() : 拥有锁时由锁对象调用, 唤醒wait set中的全部线程加入到Entry Set中, 全部由WAITING->RUNNABLE. **}**

### ThreadPool

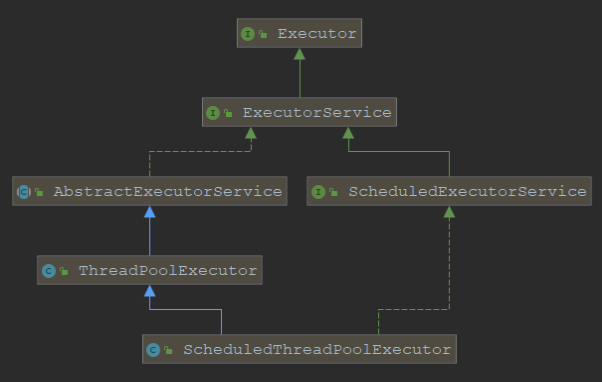
线程池, jdk1.5之后通过框架的方式来控制线程的生命周期, 并且可以避免this指针逃逸(构造函数返回之前, 其他线程已经就持有了该对象的应用, 产生的结果自然和预期可能会产生差异, 例NullPointException. 常见的this指针逃逸: 在构造函数中注册事件监听, 在构造函数中启动新线程)问题.

> 降低资源消耗：重复利用已创建的线程降低线程创建和销毁造成的消耗

> 提高响应速度：任务到达时，任务不需要等待线程创建

> 提高线程的可管理性：可以对线程统一分配、调优和监控。

#### 接口相关描述



##### Executor

一个接口, 其定义了一个接收Runnable对象的方法executor. 其方法签名为executor(Runnable command). 该方法接收一个Runable实例, 它用来执行一个任务, 任务即一个实现了Runnable接口的类.

##### ExecutorService

是一个比Executor使用更广泛的子类接口, 其提供了生命周期管理的方法, 返回 Future 对象以及可跟踪一个或多个异步任务执行状况返回Future的方法.

##### Executors

提供了一系列工厂方法用于创建线程池, 返回的线程池都实现了ExecutorService接口.

> public static ExecutorService newFiexedThreadPool(int Threads) 创建固定数目线程的线程池

> public static ExecutorService newCachedThreadPool() 创建一个可缓存的线程池, 调用execute 将重用以前构造的线程(如果线程可用). 如果没有可用的线程, 则创建一个新线程并添加到池中. 终止并从缓存中移除那些已有 60 秒钟未被使用的线程。

> public static ExecutorService newSingleThreadExecutor()创建一个单线程化的Executor

> public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize)

以上的都存在缺点, 或者是任务数没有限制, 或者是线程数没有做限制.

##### ThreadPoolExecutor

|  |  |  |
| --- | --- | --- |
| **字段** | **类型** | **描述** |
| corePoolSize | int | 线程池中所保存的核心线程数, 包括空闲线程 |
| maximumPoolSize | int | 池中允许的最大线程数 |
| keepAliveTime | long | 线程池中的空闲线程所能持续的最长时间 |
| unit | TimeUnit | 持续时间的单位 |
| workQueue | BlockimgQueue<Runnable> | 任务执行前保存任务的队列, 仅保存由execute方法提交的Runnable任务 |
| threadFactory | ThreadFactory | 线程创建工厂 |
| handler | RejectedExecutionHandler | 拒绝策略 |

**corePoolSeize, maximumPoolSize**

Runtime.getRuntime().availableProcessors(); 取得当前系统的cpu核数.

对于CPU密集型(大部分时间用来做计算, 逻辑判断等动作的程序称之为CPU bound, 例计算圆周率)的任务, 若最大线程数大于cpu内核数(cpu内核数==同时最大执行线程数), 那么根据cpu的调度算法有的可能会频繁的切换而带来很大的开销。所以推荐设置为 CPU核心数 或者 CPU核心数+1

对于IO密集型(涉及到网络, 磁盘IO的任务都是IO bound[对于IO密集型, 任务越多cpu效率越高], 例web应用)一个线程对应一个方法栈, 线程的生命周期与方法栈相同, 由于IO需要耗费时间是的该线程一直再等待IO所以可以扩大线程数。2倍CPU个数+1

计算方式: https://www.cnblogs.com/warehouse/p/10810338.html

**当新提交一个任务时：**

1. 如果poolSize<corePoolSize, 新增加一个线程处理新的任务。
2. 如果poolSize=corePoolSize, 新任务会被放入阻塞队列等待。
3. 如果阻塞队列的容量达到上限, 且这时poolSize<maximumPoolSize, 新增线程来处理任务。
4. 如果阻塞队列满了, 且poolSize=maximumPoolSize, 那么线程池已经达到极限, 会根据饱和策略RejectedExecutionHandler拒绝新的任务。

#### ExecutorService

shutdown()

平滑地关闭 ExecutorService,导致ExecutorService停止接受任何新的任务且等待已经提交的任务执行完成(已经提交的任务会分两类：一类是已经在执行的，另一类是还没有开始执行的)，当所有已经提交的任务执行完毕后将会关闭ExecutorService.

submit()

方法返回的 Future 对象，可以调用isDone（）方法查询Future是否已经完成。当任务完成时，它具有一个结果，你可以调用get()方法来获取该结果。你也可以不用isDone（）进行检查就直接调用get()获取结果，在这种情况下，get()将阻塞，直至结果准备就绪，还可以取消任务的执行。Future 提供了 cancel() 方法用来取消执行 pending 中的任务

### CAS

Compare and Swap 比较并设置, 用于在硬件层面上提供原子性操作, 在 Intel 处理器中, 比较并交换通过指令cmpxchg实现, 比较是否和给定的数值一致, 如果一致则修改, 不一致则不修改.

do{

备份旧数据；

基于旧数据构造新数据；

}while(!CAS(内存地址, 备份的旧数据, 新数据))

在java中CAS主要由sun.misc.Unsafe这个类通过JNI调用CPU底层指令实现.

### Synchronized

#### 概要

Synchronized可以保证方法或者代码块在被访问时, 同一时刻只有一个线程对该资源进行访问, 且可以保证共享内存变量的可见性(完全可以替代Volatile功能)。 可以用在代码块上, 方法上, 类方法上. 非公平锁, 可重入锁.

> synchronized关键字不能被继承.

> 定义接口方法时, 不能使用synchronized关键字.

> 构造方法不能使用synchronized关键字, 但可以使用代码块进行同步.

> synchronized使用对象锁或者作用于方法上, 那么同一个对象使用同一把锁. 若作用于类方法上, 或者使用类对象锁, 那么该类的所有对象使用同一把锁。

#### 底层实现

开始是重量级锁即底层使用对象关联的monitor辅助锁对象(互斥量对象)来进行锁机制, 1.6后进行了优化, 引入了偏向锁, 轻量级锁, 自旋锁, 消除锁. 这些优化手段并没有使用到monitor对象.

同步代码块使用monitorenter和monitorexit指令实现, 而方法同步是由方法调用指令读取运行时常量池中方法的 ACC\_SYNCHRONIZED 标志来隐式实现的.

#### 重点

1. 因为synchronized是可重入的所以A对象的同步方法里面可以调用它的另一个同步方法, 当然也可以调用继承的父类种的同步方法, 只是每次重入monitor中的计数器仍会继续加1.
2. Synchronized获取锁进入阻塞后不能被阻断, 只能一直去等锁, 然后执行, 等不到就继续等.
3. 由于wait notify 和notifyAll的实现和monitor对象有关, 所以这些方法必须放在synchronizzed块中.

详细: https://blog.csdn.net/javazejian/article/details/72828483

### ReentrantLock

#### 概要

是java.util.concurrent包下提供的一套互斥锁. 它提供了等待可中断, 公平锁, 非公平锁(默认), 锁绑定多个条件这些特性.

#### 接口

##### Lock

void lock() : 获取锁, 可重入, 取不到就一直阻塞, 不能被中断。

void lockInterruptibly() : 获取锁, 可重入, 优先相应中断, 中断后抛出InterruptedException异常并清除中断标记, 终止当前线程。

boolean tryLock : 拿到锁就返回true, 拿不到就false. 可重入, 可以设置等待时间, 也检测中断。

unlock() : 释放锁

Condition newCondition() : 返回当前线程的condition可多次调用.

##### ReentrantLock

实现了Lock的接口.

构造函数, ReentranLock(boolean fair) : true创建一个公平锁, false为非公平锁.

Public方法:

int GetQueueLength() 查看有多少线程等待锁

boolean hasQueuedThreads() 是否有线程等待抢锁

boolean hasQueuedThread(Thread thread) 是否有指定线程等待锁

int getHoldCount() 当前线程是否抢到锁, 0没有.

boolean isLocked() 查询此锁是否由任何线程持有

Boolean isFair() 是否为公平锁

**Condition**

Condition是Lock上的一个条件, 可用于线程间通信, 在粒度和性能上都优于Object的通信方法(wait, notify, notifyAll)

void await() : condition线程进入阻塞状态, 释放锁即资源, 允许中断抛异常, 必须在lock域中使用。

void awaitUninterruptibly() : 不允许中断, 若中断还会继续等待唤醒

Long awaitNanos(long nanosTimeout) 设置阻塞时间, 返回时间大于0表示被唤醒, 小于等于0超时, 其它等于await.

boolean awaitUntil(Date deadline) 和awaitNanos类似只是等待由参数指定的某一时刻

boolean await(long time, TimeUnit unit) 类似awaitNanos, 被唤醒true, 超时false

void signal() 唤醒指定线程.

void signalAll() 唤醒全部线程.

### Synchronized和ReentrantLock

|  |  |  |
| --- | --- | --- |
|  | **Synchronized** | **ReentrantLock** |
| 实现方式 | 依赖JVM实现 | 依赖JDK实现 |
| 使用 | 使用方便 | 手动加锁放锁 |
| 特性 | 非公平锁, 可重入 | 公平锁, 非公平锁, 可重入 |
| 粒度 | 粗粒度 | 细粒度, 更灵活 |
| 性能 | 1.6之后也很快 官方推荐 | 很快 api实现级别的 |

### 锁

根据锁的特性, 对锁进行分类.

#### 乐观锁悲观锁

体现在线程是否要同步, 数据库中对此概有念实际应用, 参照Mysql数据库章节。

**悲观锁**

对于同一数据的并发操作, 悲观锁总是认为在自己使用数据时有别的线程来修改数据. 常见的java的synchronized和lock以及数据库的事务都是采用的悲观锁. 常用于写操作.

**乐观锁**

对于同一数据的并发操作, 乐观锁认为在自己使用数据时没有别的线程来修改数据. java原子类的递增操作采用的是CAS自旋实现的, 以及数据库的MVCC也是乐观锁的一种实现.

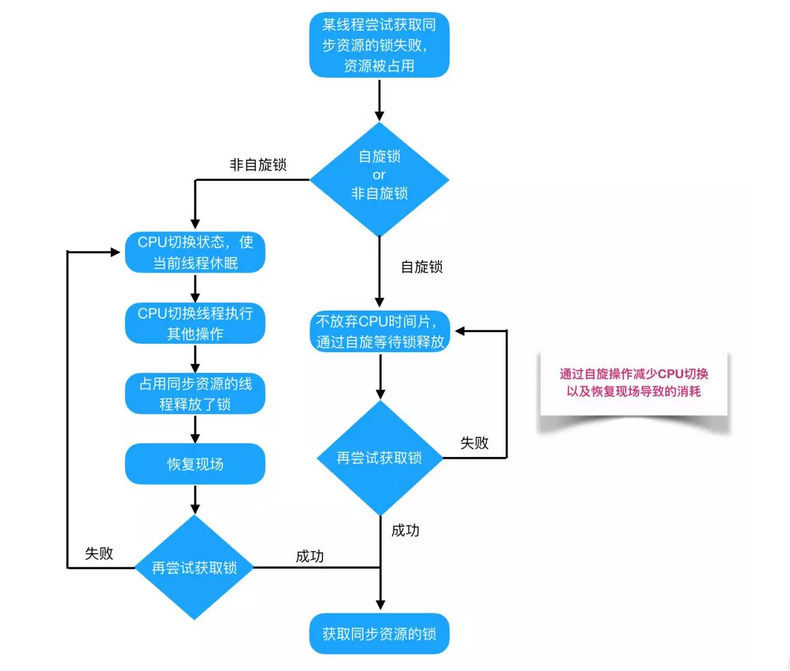
#### 自旋锁和适应性自旋锁

**自旋锁**

线程获取一个正被其它线程使用中的资源时, 会进入阻塞状态, 而进入阻塞状态需要cpu修改状态等一系列操作. 若该资源占用的时间非常短暂, 比cpu修改状态更快, 那么通过自旋无疑更好。Jdk6中默认开启, 并引入自适应自旋锁.

**自适应自旋锁**

根据上一次相同对象的锁的自旋情况来决定下次自旋是否继续。



#### 无锁, 偏向锁, 轻量级锁, 重量级锁

Synchronized在jdk1.5之前使用的就是重量级锁. 1.6之后实现下面的几种锁机制。

无锁, 不加锁, 对共享资源进行不断的循环尝试修改(类似于CAS), 直至修改成功, 适用于并发很少的情况。

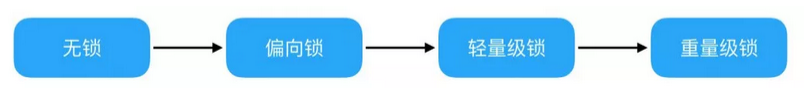
偏向锁, 在对象头和栈帧中的锁记录里存储锁偏向的线程 ID, 以后该线程在进入和退出同步块时不需要花费 CAS 操作来加锁和解锁. 适用于不存在多线程竞争，而且总是由同一线程多次获得的情况.

轻量级锁, 当多个线程通过自旋的方式获取偏向锁时, 升级为轻量级锁.

重量级锁, 当多个线程通过自选的方式获取不到锁时膨胀为重量级锁。依赖于操作系统 Mutex Lock 互斥锁实现, 比较消耗cpu.

|  |  |  |  |
| --- | --- | --- | --- |
| 锁 | 优点 | 缺点 | 适用场景 |
| 偏向锁 | 加锁和解锁不需要额外的消耗, 和执行非同步方法比仅存在纳秒级的差距 | 如果线程间存在锁竞争, 会带来额外的锁撤销的消耗 | 适用于只有一个线程访问同步块场景 |
| 轻量级锁 | 竞争的线程不会阻塞, 提高了程序的响应速度 | 如果始终得不到锁竞争的线程使用自旋会消耗CPU | 追求响应时间  同步块执行速度非常快 |
| 重量级锁 | 线程竞争不使用自旋, 不会消耗CPU | 线程阻塞, 响应时间缓慢 | 追求吞吐量  同步块执行速度较长 |

锁只能升级不能降级.



附录: https://www.cnblogs.com/cxuanBlog/p/11684390.html

#### 公平锁和非公平锁

公平锁是指多个线程按照申请锁的顺序来获取锁, 线程直接进入队列中排队, 队列中的第一个线程才能获得锁. 公平锁的优点是等待锁的线程不会饿死. 缺点是整体吞吐效率相对非公平锁要低, 等待队列中除第一个线程以外的所有线程都会阻塞, CPU唤醒阻塞线程的开销比非公平锁大.

非公平锁是多个线程加锁时直接尝试获取锁, 获取不到才会到等待队列的队尾等待. 但如果此时锁刚好可用, 那么这个线程可以无需阻塞直接获取到锁, 所以非公平锁有可能出现后申请锁的线程先获取锁的场景. 非公平锁的优点是可以减少唤起线程的开销, 整体的吞吐效率高, 因为线程有几率不阻塞直接获得锁, CPU不必唤醒所有线程. 缺点是处于等待队列中的线程可能会饿死, 或者等很久才会获得锁. Synchronized是非公平锁。

#### 可重入锁和非可重入锁

可重入锁(递归锁), 当一个线程获取锁没有释放后, 在接下来个的业务中还需要获取同样的一个锁时, 它可以直接获取锁, 锁标记继续+1. Java中synchronized和ReentrantLock都是可重入锁.

非可重入锁与之相反.

#### 共享锁和排它锁(独享锁)

共享锁, 该锁可以被多个线程共享持有, 若数据A加上共享锁后不能再加排他锁, 且其它线程只能加共享锁, 获取共享锁的线程只能读数据.

独享锁, 一次只能被一个线程所有, 可以进行读修改. 不能被其它线程加以其它锁.

## Window

### 常用操作

#### 进程

**> 查看80端口占用情况**

netstat -ano |findstr "80"

**> 查看对应进程id的应用程序**

tasklist |findstr "665120"

**> 杀死该进程**

taskkill /f /t /im 665120

## Linux

### 常用命令

#### 系统命令

lsb\_release -a # 查看操作系统的版本信息

java -version # 查看java的版本信息

mvn -v # 查看maven的版本信息

ps -ef | grep ngnix # 查询ngnix的信息

#### 端口相关命令

netstat -tunlp # 查看端口的占用情况

#### 文件操作

Chown 修改的所有者 文件名

Chgrp 所属组名 文件名

#### 发送WebSocket请求

curl --include \

--no-buffer \

--header "Connection: Upgrade" \

--header "Upgrade: websocket" \

--header "Host: 127.0.0.1:9092" \

--header "Origin: https://echo.websocket.org" \

--header "Sec-WebSocket-Key: NVwjmQUcWCenfWu98asDmg==" \

--header "Sec-WebSocket-Version: 13" \

http://127.0.0.1:9092/socket/xxxx

## Docker

### 介绍

#### 什么是容器

虚拟机(虚拟机代表: VMWare, OpenStack), 模拟运行一整套操作系统(包括内核, 应用运行态环境和其它系统环境)和跑在上面的应用.

容器(container), 独立运行的一个或者一组应用, 以及他们必需的运行环境。 容器是Pass(platform-as-a-service)的一种体现。 通俗的说容器是一种沙盒技术或者说是个轻量级的虚拟机。

详细: https://blog.csdn.net/gui951753/article/details/81148067

#### 什么是Docker

源于百度 - Docker 是一个开源的应用容器引擎, 让开发者可以打包他们的应用以及依赖包到一个可移植的镜像中, 然后发布到任何流行的 Linux或Windows 机器上, 也可以实现虚拟化。容器是完全使用沙箱机制, 相互之间不会有任何接口。

通俗的讲, 容器可以看成是一个轻量级的虚拟机, Docker或是用于管理容器的一种引擎技术。 曾经的Java是提出的是"Write Once, Run Anywhere", 而现在的Docker提出的是"Build, Skip and Run Any APP, Anywhere"

优点: Docker解决了运行环境和配置问题, 方便发布, 也就方便做持续集成。更轻量的虚拟化, 节省了虚拟机的性能损耗。

缺点: 缺乏成熟的管理工具。

其它:

\* go语言实现,

\* 打出来的包叫做镜像, 镜像运行时叫做容器。Docker解决了运行环境和配置问题, 方便发布, 也就方便做持续集成。

\* Docker是基于Linux 64bit的, 在32位机器上无法运行

详细: https://baijiahao.baidu.com/s?id=1614385619899407709&wfr=spider&for=pc

### 安装

**1. 查看linux系统版本**

Docker运行在CentOS7以上, 要求系统为64位, 系统内核版本为3.10以上。

查看linux内核版本的命令:

uname -r 或者 cat /proc/version

**2. 安装**

1) yum update // yum更新到最新

2) yum -y install docker-io

3) 安装完成查看docker版本: docker version

或者直接使用, 一键安装命令

curl -sSL https://get.daocloud.io/docker | sh

**3. 启动**

systemctl start docker

设置开机启动: systemctl enable docker

**4. 卸载docker**

yum remove docker-ce

rm -rf /var/lib/docker

**5. 常用命令**

docker images // 查看本地的镜像有哪些

docker search 镜像命令 // 搜索镜像

docker pull 镜像名称 // 下载docker镜像

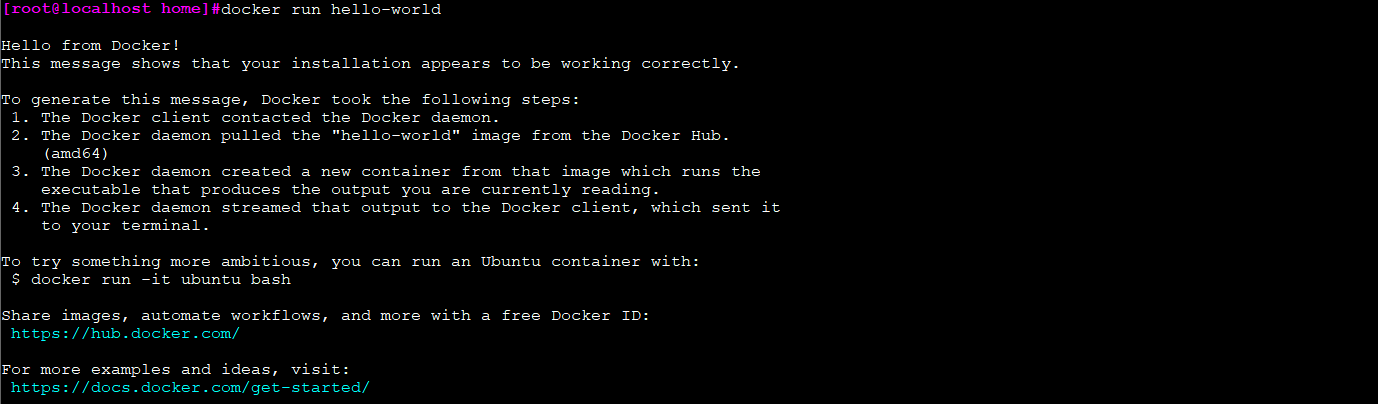
docker run 镜像名称 [启动容器里的执行命令] // 会根据镜像创建一个容器

docker ps [-a] // 查看当前运行的容器 -a是所有的包括停止的死亡的等

docker start|stop|restart|rm 容器ID或name

docker exec -it 容器ID或name bash // 进入容器交互界面 exit退出或者 ctrl+P ctrl+q

**安装成功:**



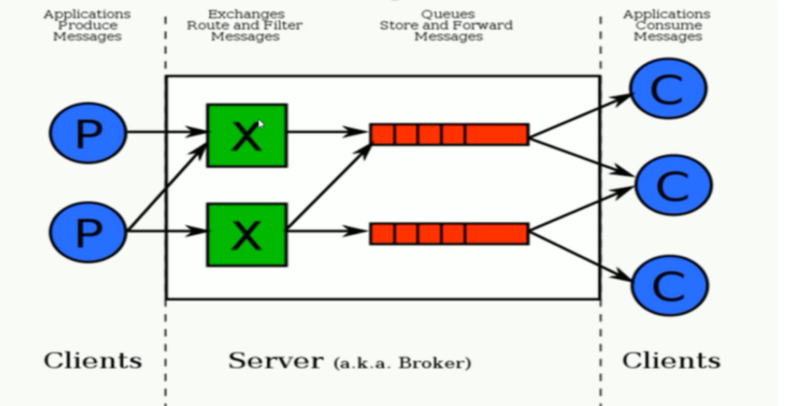
## RabbitMQ

**作用**: 解耦 削峰 异步

### 简介

#### AMQP

Advanced Message Queuing Protocol 应用层标准高级消息队列协议

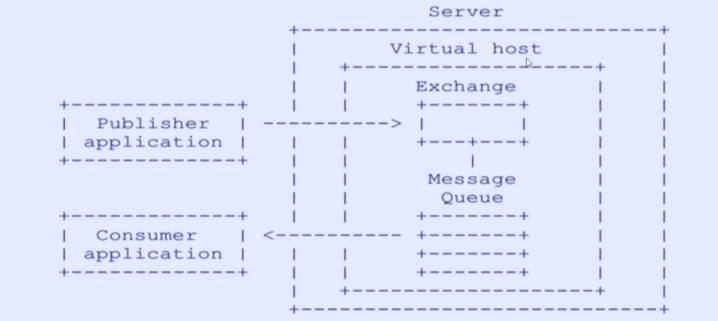


**Server:** 接收客户端的连接, 实现AMQP实体服务.

**Connection:** 连接, 应用程序与Server的网络连接, TCP连接.

**Channel:** 信道, 消息读写等操作在信道中进行.客户端可以建立多个信道, 每个信道代表一个会话任务.

**Message:** 消息, 应用程序和服务器之间传送的数据, 消息可以非常简单, 也可以很复杂.有Properties和Body组成.Properties为外包装, 可以对消息进行修饰, 比如消息的优先级、延迟等高级特性; Body就是消息体内容.

**Virtual Host:** 虚拟主机, 用于逻辑隔离.一个虚拟主机里面可以有若干个Exchange和Queue, 同一个虚拟主机里面不能有相同名称的Exchange或Queue.

**Exchange:** 交换器, 接收消息, 按照路由规则将消息路由到一个或者多个队列.如果路由不到, 或者返回给生产者, 或者直接丢弃.RabbitMQ常用的交换器常用类型有direct、topic、fanout、headers四种

**Binding:** 绑定, 交换器和消息队列之间的虚拟连接, 绑定中可以包含一个或者多个RoutingKey.

**RoutingKey:** 路由键, 生产者将消息发送给交换器的时候, 会发送一个RoutingKey, 用来指定路由规则, 这样交换器就知道把消息发送到哪个队列.路由键通常为一个"."分割的字符串, 例如"com.rabbitmq".

**Queue:** 消息队列, 用来保存消息, 供消费者消费.

#### 安装

安装和简单使用: https://blog.csdn.net/qq\_40241957/article/details/95064848

安装好的默认路径是: http://localhost:15672/

默认用户和密码: guest

mq的web界面描述: https://www.jianshu.com/p/7b6e575fd451

mq的学习: https://github.com/mingyang66/spring-parent/tree/master/spring-boot-control-rabbitmq-service

#### Web界面

##### QueueMessage:

显示的是消息堆积后的数据.

ready是有未被消费的消息时才会有数量.

unacked是投递给消费者没有收到ACK才会有数量, 消费者接收了消息, 但是未发送ACK.

##### Ack Mode

Nack message requeue true: 获取消息不做ack应答确认, 消息重新入队

Ack message requeue false: 获取消息, 应答确认, 消息不重新入队, 将会从列表删除

Reject requeue true: 拒绝获取消息, 消息重新入队

Reject requeue false: 拒绝获取消息, 消息不重新入队, 将会被删除.

##### 常用的交换机类型

将消息转发到routing key所匹配的队列中

Direct：绝对匹配 , 即: routing key = binding key

Topic: 模糊匹配, 即: com.# 可以匹配到com.a com.a.em等

# 匹配一个或多个词 \* 匹配一个词

Fanout: 不需要key匹配, 广播给与该交换机绑定的所有队列.

Headers: 根据消息中的header属性进行匹配, 性能差, 不推荐.

特殊的: Header Exchange, Default Exchange, 死信交换机

## 云

### 概念

### 云

提供资源的网络被称为"云", 云中的资源在使用者看来是可以无限扩展的, 并且可以随时获取, 按需使用, 随时扩展, 按使用付费。通俗的说它类似于公共的水电设施的IT基础设施。

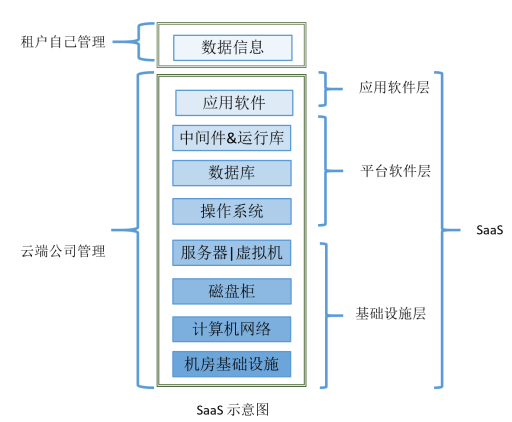
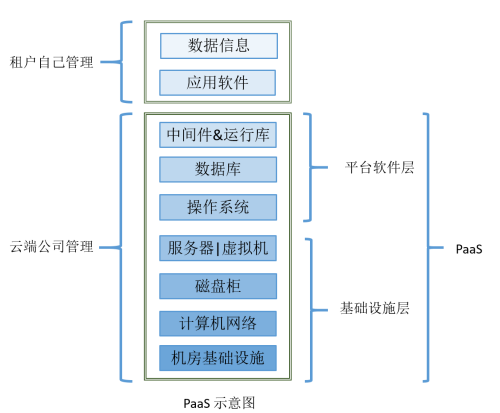
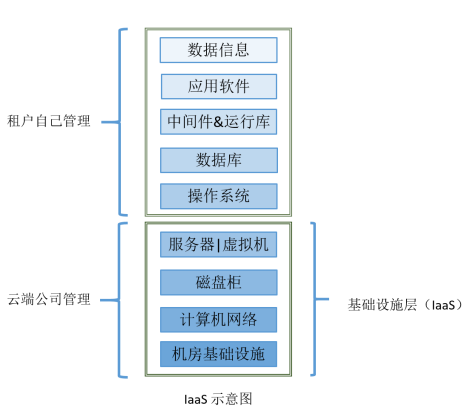
#### 云计算

云计算是指IT基础设施的交付和使用模式, 通过网络以按需、易扩展的方式获得所需的服务, 这种服务可以是IT和软件(硬件、平台、软件)、互联网相关的, 也可以使任意其他的服务.

#### 云服务

云服务就是指云提供的四个层面服务(基础架构即服务、平台即服务、软件即服务, 数据即服务)中的某类服务。 云服务其实是运行在云计算之上的, 算是云计算的一个具体化抽象.

#### 云的一些概念



##### IMG_256IaaS

IaaS是Infrastructure as a server的缩写, 即基础设施即服务。

##### PaaS

PaaS是Platform as a Server的缩写, 意思是平台即服务, 即把运行用户所需的软件平台作为服务出租.

##### SaaS

SaaS是Software as a Server的缩写, 意思是软件即是服务.

##### DaaS

DaaS是Data as a Server的缩写, 意思数据即服务。

RBAC

基于角色的权限控制.

SCMS

SCMS是一个基于MVC的安全内容管理系统, 它的目的是从根本上抵御常见Web应用安全漏洞如：SQL注入、XSS、CSRF、Session固定/劫持等. 除了支持MySQL数据库外, 还支持PostgreSQL.

## WebSocket

#开启Mybatis下划线命名转驼峰命名

mybatis.configuration.map-underscore-to-camel-case=true